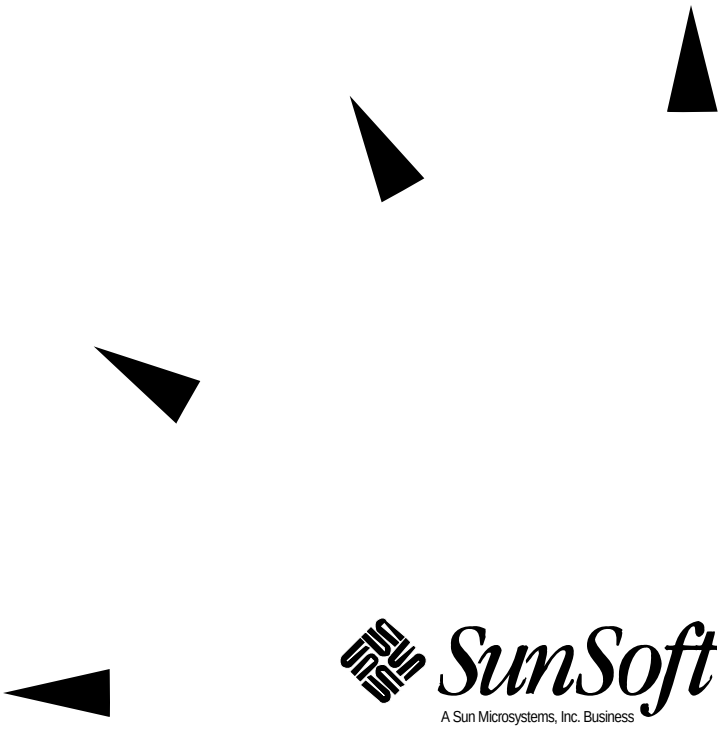
Desktop Integration Guide



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

SunSoft
A Sun Microsystems, Inc. Business

Please
Recycle

Adobe PostScript

Contents

Figures

Tables

Preface

This manual describes the various technologies available for integrating window applications running the Solaris™ System Software, which consists of two parts: the operating system and windowing environment. Desktop Integration is defined as the ability to exchange and process data using the special features of the Solaris graphical user interface.

Who Should Use This Book

This guide is written for independent software vendors (ISVs) with previous experience developing in the X11 windowing environment, who wish to integrate their applications with other applications and tools on the Solaris™ desktop.

How This Book Is Organized

Chapter 1, “Desktop Integration,“provides an overall introduction to desktop integration and lists the available technologies.

Chapter 2, “The Selection Mechanism,” describes the selections mechanism for copying or moving data between applications.

Chapter 3, “Drag and Drop,” describes the application program interface and provides working XView™ code examples.

Chapter 4, "Implementing Drag and Drop with DeskSet," tells how to drag and drop between DeskSet Applications.

Chapter 5, "Classing Engine," describes the principles, organization, and operation of the Classing Engine, and working example code is supplied.

Chapter 6, "The ToolTalk Service," provides the background and a short tutorial for process-oriented messaging capabilities of the ToolTalk™ service.

Chapter 7, "The ToolTalk Service and DeskSet Integration," discusses how the ToolTalk protocol is used with Solaris DeskSet applications.

Appendix A, "Drag and Drop User Interface Specification," provides the drag and drop user interface specification.

Appendix B, "Examining a Classing Engine Database," provides a procedure for producing an ASCII printout of an existing Classing Engine database.

Appendix C, "Vendor Data Type Registration," describes the process used to reserve a vendor-unique data type designator.

Appendix D, "ToolTalk Example Program for XView Toolkit," shows a ToolTalk program.

Appendix E, "Drag and Drop Programming Example for XView Toolkit," shows a program integrated with the DeskSet.

Appendix F, "The ToolTalk Desktop Services Message Set," details all ToolTalk messages for Desktop Services valid for this release.

Appendix G, "The ToolTalk Document and Media Exchange Message Set," details all ToolTalk messages for Media Exchange valid for this release.

Related Books

The X Window System Programming and Applications with Xt, OPEN LOOK Edition, Prentice Hall, 1992

OLIT Reference Manual, SunSoft, 1994

X Window System Programming and Applications with Xt – OPEN LOOK Edition, by John Pew, published by Prentice Hall, 1992.

OLIT Quick Start Programmer's Guide, Part Number 801-5317-10, Sun Microsystems, Inc., 1994.

X Window System Toolkit, The Complete Programmer's Guide and Specification, Digital Press, 1992.

X Window System, The Complete Guide to Xlib, X Protocol, ICCCM, XLFD, Digital Press, 1992.

X Protocol Reference Manual, O'Reilly & Associates, Inc., 1990

XView Programming Manual, O'Reilly & Associates, Inc., 1991

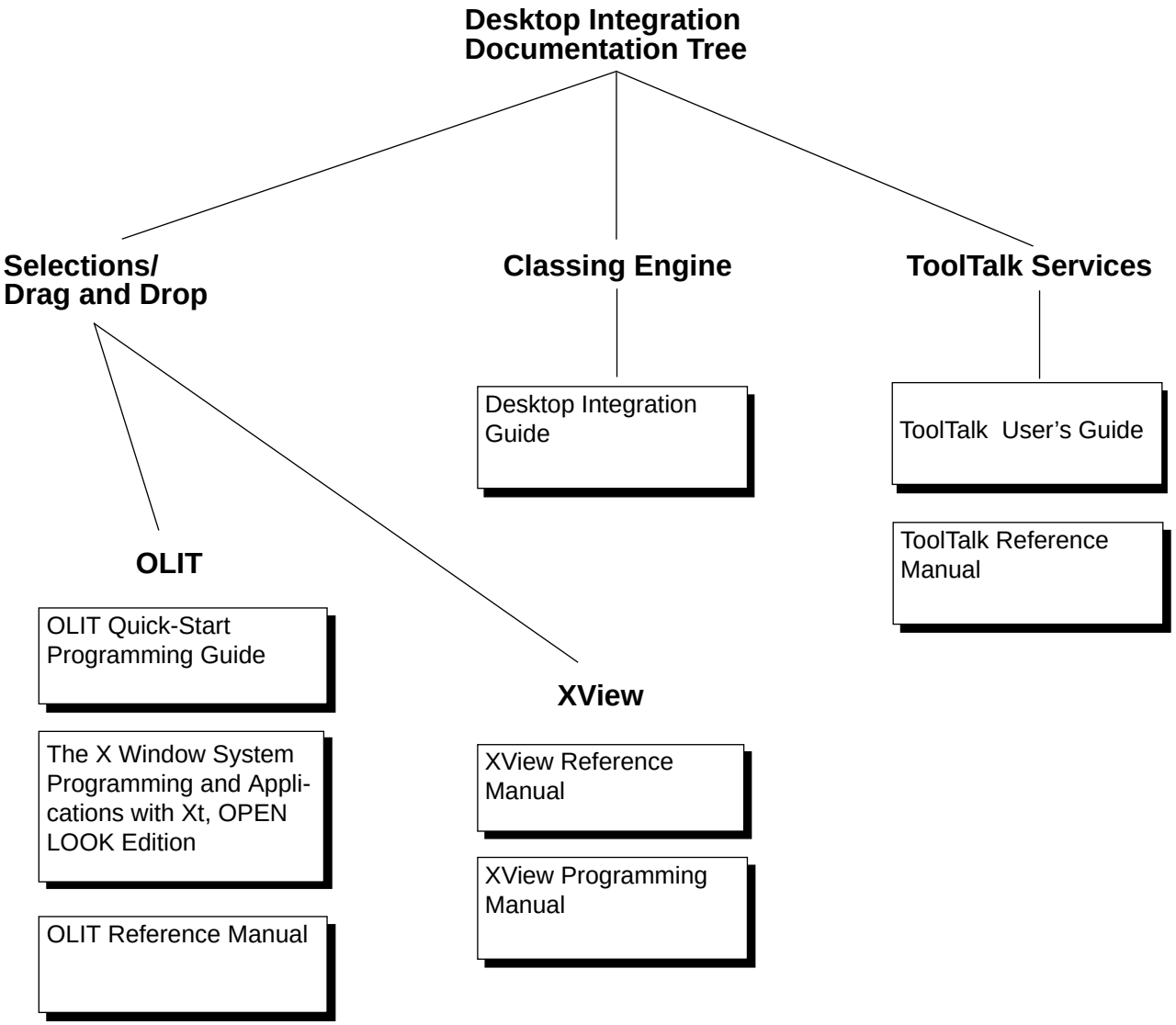
XView Reference Manual, O'Reilly & Associates, Inc., 1991

ToolTalk User's Guide, SunSoft, 1994

ToolTalk Reference Guide, SunSoft, 1994

Writing Applications for Sun Systems, Volume I, A Guide for Macintosh® Programmers, Addison Wesley, 1991

Applications achieve varying degrees of integration with the three integration technologies described in this manual. These technologies are: selections and drag and drop, the Classing Engine, and ToolTalk services. The diagram below directs you to further information sources.



What Typographic Changes and Symbols Mean

The following table describes the type changes and symbols used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	**Meaning**	**Example**
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output, code samples	Edit your `.login` file. Use `ls -a` to list all files. `system% You have mail.`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.
Code samples are included in boxes and may display the following:		
`%`	UNIX C shell prompt	`system%`
`$`	UNIX Bourne shell prompt	`system$`
`#`	Superuser prompt, either shell	`system#`

Desktop Integration 1

Desktop integration is the name given to a suite of technologies that allow seamless cooperation and interoperability between applications on the desktop. Desktop integration allows the following:

- Users can select data from one application, and drop it into a different application without regard to format.
- A data object on the desktop can be dragged to the Print Tool where it will be printed in the appropriate format.
- A user can attach an icon representing a desktop publishing file to a mail message. The message receiver could then open the document into the desktop publishing application by simply double-clicking on the icon.
- The development of *groupware*, or applications that allow several people to work simultaneously on a document or program, while the system automatically performs the various housekeeping chores (such as updating files and informing other users of file changes).

Desktop integration lets applications share information and processes with other applications. This sharing results in a higher degree of communication, cooperation, and software productivity.

The guide is written for software developers who have previous window programming experience, and who wish to integrate their applications with other applications on the Solaris desktop. This guide presents an overview of

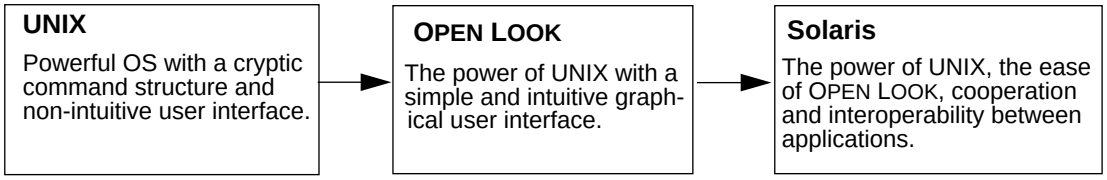
desktop integration and its constituent technologies: selections / drag and drop, the Classing Engine, and ToolTalk services. Refer to the technology-specific manuals listed in the Preface for further sources of programming information.

1.1 UNIX Evolution

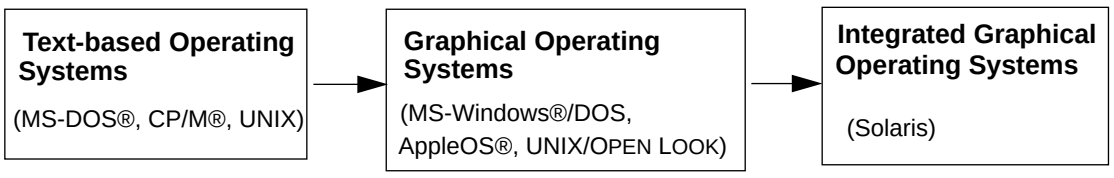
OpenWindows desktop integration represents an evolutionary step in the growth and maturation of the UNIX® operating system. Although UNIX itself is an extremely powerful operating system, its command line interface is non-intuitive, cryptic, and difficult for most users to master. In recent years the UNIX command line interface has been supplanted by windowing interfaces such as the OPEN LOOK™ graphical user interface, which provides a simpler and more intuitive way to control the system. By the end of 1992, there were more than two thousand OPEN LOOK application programs offered by more than a thousand vendors.

Software Evolution

UNIX:



Operating Systems:



Although OPEN LOOK greatly increases the usability of UNIX, there are some limitations: occasionally pathnames and other elements of command line UNIX need to be manually entered. Also, data exchange is severely limited by

incompatible file formats, and cannot always be performed on files, folders, or other large units of data. Solaris addresses these limitations by combining UNIX, the OPEN LOOK GUI, and the application interoperability tools described in this manual.

1.1.1 SunSoft Terms

OPEN LOOK is the graphical user interface specification adopted by UNIX International and used by SunSoft. OpenWindows, Solaris's window system, implements the OPEN LOOK specification. Solaris refers to the entire computer operating environment—the UNIX operating system, the OpenWindows window system, and the desktop environment—running on SPARC and Intel systems. SunSoft is the developer of Solaris and is a wholly-owned subsidiary of Sun Microsystems, Inc.

1.2 Solaris Desktop Integration Technologies

To overcome the limitations of the OPEN LOOK windowed user interface, and to move towards a fully integrated desktop, Solaris provides the following enhancements:

- An enhancement to the X Window System selections mechanism to provide a user-directed flow of information from one application to another (drag and drop)
- A method for applications to determine the identity and operating characteristics of objects on the desktop (the Classing Engine)
- A mechanism for passing messages and commands between applications (the ToolTalk service)

These enhancements represent three distinct technologies. Together these Solaris technologies provide the desktop with a powerful cohesiveness and data interchangeability between applications.

1.2.1 Drag and Drop

This mechanism allows users to use the mouse to exchange data between or within applications. This is done by selecting the graphical representation of the data with the Select and Adjust mouse buttons, clicking the representation (object) with the Select mouse button, keeping the mouse button down while

dragging the object to a destination, and dropping it (releasing the mouse button). The user need not be concerned with the subtleties of moving data between applications, such as the data's format, or whether data translation is required. All of this is handled by the application programmer using the drag and drop API.

Another application of Selection Services has been in use for some time. This involves using the mouse to select data, as outlined above, then using the copy and paste menu functions to move the data to another client.

Selection Services are covered in Chapter 2. The drag and drop API is presented later in this manual, starting with Chapter 3.

1.2.1.1 Application

Drag and drop can move data within an application. For example, a user can rearrange a text file by selecting text, dragging the text to a new position, and dropping (inserting) the text at that position.

Drag and drop can also move data between applications. An example of this would be copying data from Mail Tool and dropping it into a desktop publisher. Another example is copying an appointment from a mail message and dropping it onto Calendar Manager where it is properly entered.

1.2.1.2 Drag and Drop and the X Server

Selections and drag and drop provide a communications link between an *owner* client (the client that owns the data) and a *requestor* client (the client that receives the data). All data transferred through Selection Services and drag and drop is transferred through the X server. Each toolkit (OLIT or XView) provides a selections and drag and drop API. Though the APIs for each toolkit are somewhat different, selection between the toolkits is seamless and invisible.

1.2.2 Classing Engine

The Classing Engine is a database that identifies the characteristics, or *attributes*, of desktop objects. In other words, the Classing Engine stores attributes of desktop objects, such as print method, icons, and file opening commands.

If an application is to interoperate with other objects, the application must be able to identify those objects and determine their various operating characteristics. That is, if the object is another application, can they intercommunicate? If the object is a file, can the application read it? Every object on the desktop must be readily identifiable—is it an ASCII file, a SunSoft DeskSet tool, a spreadsheet program, a data file from a desktop publishing program, or something else? The Classing Engine provides a database for storing this information and an API to access the information. Applications query the Classing Engine database to determine an object's type and the attributes associated with the object. Chapter 5, "Classing Engine" discusses this technology in detail.

1.2.2.1 Application

File Manager, a DeskSet application shipped with OpenWindows, provides the best example of using the Classing Engine. File Manager graphically displays a UNIX file system as a set of folders (directories) and documents (files). Users can move, copy, and rearrange files by dragging and dropping file icons into directory icons. Files may be deleted by dropping icons into the waste basket icon. File Manager uses startup information stored in the Classing Engine to allow users to double click on a file icon and open the file with its associated application. For example, double clicking on a spreadsheet data file icon opens the file into the spreadsheet program. Double clicking a desktop publishing file opens the file into the correct desktop publishing application. Using print instructions from the Classing Engine, File Manager also allows users to print a data file by dragging and dropping it on the Print Tool.

Another feature of File Manager is that different file types are represented by different icons. Thus, one application's file will have one type of icon, and another application will have a different icon. Again, the Classing Engine provides File Manager with the icon display information.

1.2.3 ToolTalk Service

The ToolTalk service is used by 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.

To use ToolTalk in your application, you must first decide what message protocol your application will implement. A message protocol is a set of ToolTalk messages that describe operations that applications agree to perform. By adopting a message protocol, applications can speak the same ToolTalk language. The message protocol specification includes the set of messages, as well as how applications will behave when they receive the messages. Refer to Chapter 6, “The ToolTalk Service” for further details on ToolTalk. Refer to Chapter 7, “The ToolTalk Service and DeskSet Integration” for a discussion on the DeskSet message protocol.

1.2.3.1 Application Example

Calendar Manager, another DeskSet application shipped with OpenWindows, provides an interesting example of how the ToolTalk service is used on the Solaris desktop. Calendar Manager (or CM) has the ability to browse multiple users’ calendars, and schedule a meeting in a time slot that is open for all of them. CM needs to be able to send an electronic mail message to the relevant users informing them of the new meeting. There are several ways CM could do this.

One way would be for CM to open a text window to allow the user to enter the body of the mail message, after which CM would invoke the system’s mail program to send the message. However, this would mean the CM has to implement its own facility for composing electronic mail messages — a facility not likely to be similar to the user’s usual mail composition tool. The result is duplication of effort by developers and inconsistent facilities for users.

Another way would be for CM to create a temporary mail message file, look up the user’s preferred mail composition tool, and invoke that tool on that file. The problem here is that communication between CM and the mail tool is limited to command line arguments and the process environment. (See the command `environ(5)`). For example,

- The mail tool has no standard way of finding out where on the screen it should appear.
- CM has no way of knowing if there already exists a running instance of the mail tool that could respond more quickly, or bring to bear on the job some state information that the user considers valuable.
- CM has no way of (for example) iconifying the composition window if the user wants to temporarily put away CM and its associated windows.

- CM has no way of gracefully aborting the composition operation if the user suddenly decides to quit.

The Solaris Desktop does not suffer from these constraints. In Solaris, CM can issue a ToolTalk request asking that the indicated mail message be edited. The ToolTalk service routes the request to the running mail editing tool that is best prepared to handle it, or starts the user's favorite mail tool if no instance is already running. The handling tool is placed in communication with the requesting CM, and is free to inquire about where on the screen to appear or what internationalization locale to assume, and to send back periodic status reports or data checkpoints. In turn, CM is able to monitor and control the operation, and manipulate the composition window almost as if it were one of its own. The result is that the user concentrates on his work, and not on his tools.

1.3 ISV Registration

Maximum desktop integration requires public notice of application data types, naming conventions, custom icon design and ToolTalk message protocols. By making this information public, ISVs can be sure that their applications and data files are recognized by other applications. SunSoft provides a vehicle for making this information public through the Developer Integration Format Registration program. Independent software vendors can register the data type information for their applications with SunSoft. This information will be made available to other ISVs through SunSoft. Refer to Appendix C, "Vendor Data Type Registration" for details.

1

The Selection Mechanism 2

2.1 Overview

The X11 selection mechanism is a means of copying data between or within applications running under the OpenWindows Desktop. The term *selection* is taken from the paradigm in graphic user interfaces by which the user selects an object (such as a block of text or a file icon) by highlighting it, before moving or copying it.

Here are two examples of how the selection mechanism is used to *copy* data:

- drag and drop (see Chapter 3, “Drag and Drop)
- the Copy and Paste command keys

Besides copying, the selection mechanism is also used to *move* data. For instance, here is how an OpenWindows Desktop application would allow you to move a sentence from one location to another location:

1. Make the selection. Use the mouse to place the insertion point at the start of the sentence and momentarily press the Select mouse button. Move the pointer to the end of the sentence and press the Adjust mouse button; the *selection* will be highlighted in reverse video.[1]

2. Store the selection. Press the Cut key on the keyboard to temporarily store the selection in the clipboard.

1. Another selection method is to place the insert point at the beginning of the sentence, press the Select mouse button and hold it down as you “wipe” across the text to the end of the sentence, then release the button.

3. Insert the data. Use the mouse to place the insertion point at the desired location. Insert the text by pressing the Paste key.

For the remainder of this manual, the selection mechanism is referred to as *selections*, a common usage in the X11 developer community. Selections provide a communications link between a *holder* client (the client which owns the data) and a *requestor* client (the client that receives the data). All data transferred through selections is transferred through the X server. Each toolkit (XView or OLIT) provides a selections API. Although the API for each toolkit are somewhat different, selections between the toolkits is seamless and invisible.

User interface conventions for selections are outlined in the *OPEN LOOK Functional Specification*. For further selections programming instructions, refer to the *XView Programming Manual* and *XView Reference Manual* from O'Reilly and Associates, and *The X Window System Programming and Applications with Xt, OPEN LOOK Edition* from Prentice Hall.

XView and OLIT selection examples are at
`$OPENWINHOME/share/src/dig_samples/`

as

`selection_olit/olit_sel.c`

`selection_xview/xview_sel.c`.

2.2 Selections Outline

Selections provide a well-defined method of implementing the Copy and Paste keys. The following outline describes the generic steps for implementing Copy and Paste with any of the OpenWindows toolkits.

Selections communicate between an *owner* client and a *requestor* client. The owner client has the data representing the value of the selection. The requestor client desires the value that the selection provides. Selection code is required for both the owner and requestor clients. Refer to the Inter-Client Communications Conventions Manual (ICCCM) for a detailed discussion of the selections protocol. The ICCCM can be found in Appendix L of X Protocol Reference Manual from O'Reilly and Associates.

2.2.1 Selection Owner

1. Mark Selection

Visual feedback of the selected object should be provided to the user. For example, the selection can be shown by displaying the selected text in reverse video.

2. Make Selection

When the user presses the Copy key, create a selection holder and set the other attributes required by the application.

Note that a conversion procedure must be written to handle conversion requests from the selection requestor. The request for text is handled automatically.

3. Associate Data

Associate selection (highlighted text) with the owner client. If the selection is currently owned, the owner receives an event and is expected to do the following:

- Convert the contents of the selection to the requested data type
- Place this data in the named property on the named window
- Send the requestor an event to let it know the property is available

2.2.2 Selection Requestor

1. Paste Event

The event handler must detect the `Paste` event, so that the Paste operation (selection request) can be initiated.

2. Request Data

Request data from the owner client. Post a request to get data from the selection owner. The owner has the data representing the value of its selection, and the requestor client wishing to obtain the value of a selection provides:

- The name of the selection
- The name of a property
- A window

- An atom representing the data type required

2.3 Implementing Selections with DeskSet

Call the SunSoft Catalyst Information Center (see Appendix C, "Vendor Data Type Registration") for information on the selection protocol for DeskSet. Note, however, that ICCCM currently does not specify the protocol supported by DeskSet. The current DeskSet selection protocol may change to comply with future ICCCM specifications.

Drag and Drop 3

3.1 Overview

Drag and drop is an implementation of selections, which allows users to select a data object (text block, graphic, audio object, file icon, etc.) with the mouse, “drag” it across the screen (by keeping the mouse button down), and “drop” the data into another application for use there. For example, a text file icon can be selected, dragged to the Print Tool, and dropped, where it is printed. Another example: a spreadsheet data file icon can be dragged and dropped onto a spreadsheet application icon, causing the data to be loaded and displayed on the screen. Note that drag and drop differs from moving data with the Cut, Copy, and Paste command keys: it is not limited to moving only text blocks or images in a drawing program, but can move objects of many data types.

Applications that implement drag and drop can exchange data with other applications. Drag and drop, like selections, has a different API for the XView and OLIT toolkits. Once implemented in a client however, drag and drop works invisibly and seamlessly with both toolkits.

This chapter discusses the following:

- Drag and drop user interface
- The steps required to implement drag and drop in one of the toolkits.
- A detailed example of drag and drop as implemented in the OLIT environment.

For further drag and drop programming instructions, refer to the *XView Programming Manual* and *XView Reference Manual* published by O'Reilly and Associates, and *The X Window System Programming and Applications with Xt, OPEN LOOK Edition* from Prentice Hall.

3.2 Drag and Drop User Interface

To implement drag and drop, you must understand the drag and drop user interface. This section briefly describes this interface for purposes of terminology. Appendix A, "Drag and Drop User Interface Specification," describes:

- the kinds of objects that can be dragged
- the meanings of dropping objects on specific locations (such as on a window header, on a pane in a window, or on a drag and drop target)
- the differences between dragging with and without the Duplicate modifier key held down
- the visual feedback associated with the stages of a drag and drop operation
- how the process of data translation appears to users
- how users can cancel drag operations in progress, and undo completed drag operations
- how error messages are presented to users

3.2.1 Overview

Drag and drop allows users to transfer data objects, using the mouse, between or within applications. A drag and drop action consists of a *source* (object to be transferred), and a *destination* (the place where the source will be dropped). Before an object can be dragged or dropped, it must be *selected*. There are two types of objects that can be selected: a *data span* or *glyph*. A data span is a segment of on-screen data. It can be a segment of text, digitized audio, video, and so forth. A glyph is an on-screen representation of some object, such as a file, application, or directory.

A data span can be selected in three ways: the wipe method, the select-adjust method, and the multi-click method. With the wipe method you place the pointer at the beginning of the data span, press the Select button, drag the

mouse to the end of the selection, and release the mouse. In the select-adjust method, you place the pointer at the beginning of the selection and click the Select button to select the starting point. Then you move the pointer to the end of the desired span and click the Adjust button on the mouse to make the selection. With the multi-click method you rapidly press the Select button to select increasing larger segments of the segment. For example, two rapid clicks selects a word, three a line, and four a paragraph. A selected data span is displayed in reverse video.

Glyphs are selected by clicking the Select button on the glyph. To select additional glyphs, click Adjust on additional glyphs.

3.2.2 Initiating the Drag

Drag and drop can be initiated as either a *cut-and-paste* or a *copy-and-paste* operation. In a cut-and-paste operation, the original object is deleted after it is dropped. In a copy-and-paste operation, the original object remains after the object is dropped on a destination; the original object is not deleted.

3.2.3 Visual Feedback

Drag and drop requires visual feedback to display the status of the drag. At a minimum, once an object is selected, the pointer should change appearance and a representation of the object should follow the pointer as the mouse is moved. In addition, the pointer should indicate the receptivity of potential drop sites. The drag and drop specification includes details about changes in pointer appearance and other visual feedback associated with drag operations.

3.2.4 The Drop

The final action in the drag and drop gesture is to drag the selection over the destination object and to release the Select mouse button. The destination is determined by the position of the pointer's hot spot at the time the user releases the Select button.

Applications supporting drops other than a simple cut or copy sometimes require a specific drop site, referred to as a *drag and drop target*. A drag and drop target is a graphical element located in the control area of an open

window. In addition to serving as the destination in drag and drop operations, drag and drop targets sometimes contain a glyph that can be used as the source in a drag and drop operation.

3.3 Implementing Drag and Drop

The following sections summarize the toolkit independent processes required for sourcing drags and receiving drops.

3.3.1 Sourcing a Drag

To adapt an application to source a drag, the following steps are required:

1. Define a drag and drop object and associate a drag pointer with it. The window manager will use the drag pointer to provide visual feedback to the user when the object is selected.
2. Associate a selection with your drag and drop object, which will contain the data you want to make available to the target.
3. Provide an event callback procedure for your drag and drop object that will detect when it has been dragged. Set the actual data in the data object for the source to retrieve, and wait for a source response or error condition.

Depending on the application, you may also want to perform the following:

4. Define a selection conversion procedure for your own data types.
5. Provide the data through an alternate transport mechanism (ATM), such as sockets or the ToolTalk service.

Handshake protocol for when drag occurs is shown in the pseudo code below.

```
/* Set default values that the owner app provides to a requestor that doesn't make */
/* any specific requests */
set item = default item/* Generally set to the first item */
set type = default_type/* Set the data type you would like to send the
       /* default item in */
set transport = X_SERVER/* Default transport method is the X-Server*/
set length = length of 1st item/* Length of default item */

/* make available the types you support for the default item */
```

```
set avail_types = types for item or types you support (array of types)

/* set up the Convert proc for the drag-n-drop process */
In a registered callback, get ready to provide information:

if filename/* if they ask for a filename */
    set filename/* fill in the filename atom info */
    return true/* return the information */

if data label/* if they ask for the data label */
    set data label/* fill in the data label atom info */
    return true/* return the information */

/* Advertise the targets supported by the owner app. (i.e., this is the list of */
/* atoms returned when the TARGETS is converted */
if targets
    set targets
    create array of:
    _SUN_ENUMERATION_COUNT/ITEM
    _SUN_AVAILABLE_TYPES
    LENGTH = length
    return true

if _SUN_ENUMERATION_ITEM/* only if handling multiple selections and */
        /* enumeration count > 1 is the case */

set item = _SUN_ENUMERATION_ITEM
set length = length of item
set name = name of item/* or data label if needed */
specify _SUN_AVAILABLE_TYPES for that item

/* Additional information can be set or simply returned as false */
```

3.3.2 Receiving a Drop

To adapt your application to receive drops, the following steps are required:

1. Define a drop site, and associate an event procedure with it.
2. (Optional) Provide an image for drop site previewing that will provide visual feedback when the pointer is over the drop site.
3. In your event callback procedure, determine the event type, and obtain data from the source selection.

Depending on the application, you may also need to do the following:

4. Provide drop site feedback when pointer enters and leaves the drop site.
5. Use an alternate transport method (ATM), such as sockets or the ToolTalk service, to transfer data if your application design requires it.

Handshake protocol for when drop occurs is shown in the pseudo code below.

```
get enumeration_count /* gets number of selection objects */
if no count
    count = 1
if count > 1 and application can only handle one
    end dragdrop
    return False
else
get ready to handle multiple selections

/* obtain drop information from owner application This applies both when handling single */
/* or multiple selections*/

For each item in count:

    /* Note: a special atom corresponds to all the information below. So using */
    /* selections, request the information by supplying the pre-defined atom */

Specify item being requested/* specify item number */
Request host name/* Get owners host name */
Request filename /* Request pathname of item */
Request data label/* Data Label is an identifier for the
        /* item (usually the name for items items that don't
        /* have real files). For example, an attachment in
        /* mailtool may have a data label without a filename
        /* attached to it */
Request available types /* List of data types in which the
        /* owner may send the data */

    /* Now that we have information on data, let source know that we are ready toload */
    /* the data by converting the load atom */

Request Load

specify item, and data type/* Specify item and data type in which the */
        /* requestor would like the data */
```

```
if delete then
    delete/* If the action is a move, send owner
    done
```

3.4 Drag and Drop Programming Example: OLIT Toolkit

The source files for this program can be found online in the Solaris 2.2 distribution at `$OPENWINHOME/share/src/dig_samples/DnD`. This example is a simple implementation of drag and drop using the OLIT toolkit. The directory contains the header file, C source files, resource file, and the Makefile for compiling the executable, `dnd`.

When the program is executed, it opens a text window with a drag and drop target. Users may drag any text file from the file manager and drop it on the window's drop site. The text is displayed in the text pane, and the filename path appears in the window header. The document can be exported by dragging the drag and drop target to another window. A portion of the text can be moved by selecting the desired text and dropping it at a specific insert point.

Appendix E, "Drag and Drop Programming Example for XView Toolkit" gives an example of drag and drop implemented under Xview.

3.5 Summary of Files and Functions

Here is a summary of the files and important functions that comprise the example source.

Table 3-1 Overview of the Modules

`dnd.h`	Contains global variable declaration
`main.c`	Contains the OLIT header includes, initial declarations, main function declaration, and other important function definitions
`owner.c`	Contains function definitions for drag and drop owner handling

Table 3-1 Overview of the Modules

`requestor.c`	Contains function definitions for drag and drop requestor handling
`Resources`	Contains X resources needed to make up the windows and controls in the sample application
`Makefile`	Allows the program to be compiled and linked with the `make` command

Table 3-2 Overview of the Functions

`main()`	Contains the main program loop and creates the necessary widgets
`DropTargetCB()`	Called when some action happens on the drop target
`text_modified()`	Called when user edits the text

3.6 Module dnd.h

This module contains the global variable declarations that are not defined in the included OLIT and Xt header files. Of particular interest are:

Table 3-3 Global Variable Declarations

target_type	Enumerated type declaring the target atoms this application supports
targets_t	Structure definition for atom information
Dnd_t	Structure defining the data that may be transferred in a drag and drop
*extern **	Ansi C prototypes for functions and global variables

3.7 Module main.c

Contains the OLIT header includes, initial declarations, main function definition, and other important function definitions.

The main function and other important functions are discussed below. Here are the contents of `main.c`:

```
/* main.c (continued) */
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <Xol/OpenLook.h>
#include <Xol/TextEdit.h>
#include <Xol/ScrolledWi.h>
#include <Xol/RubberTile.h>
#include <Xol/MenuButton.h>
#include <Xol/OblongButt.h>
#include <Xol/StaticText.h>
#include <Xol/DropTarget.h>
#include <Xol/Exclusives.h>
#include <Xol/RectButton.h>
#include <sys/systeminfo.h>
#include “dnd.h”

/* array of targets we want to use */
targets_tmytargets[] =
{
    {TARGETS,“TARGETS”,0},
    {FILE_NAME,“FILE_NAME”,0},
    {STRING,“STRING”,0},
    {LENGTH,“LENGTH”,0},
    {SUN_AVAILABLE_TYPES,“_SUN_AVAILABLE_TYPES”,0},
    {SUN_LOAD,“_SUN_LOAD”,0},
    {SUN_DATA_LABEL,“_SUN_DATA_LABEL”,0},
    {SUN_DRAGDROP_DONE,“_SUN_DRAGDROP_DONE”,0},
    {TEXT,  “TEXT”, 0},
    {SUN_SELECTION_END,“_SUN_SELECTION_END”,0},
    {NAME,  “NAME”, 0},
    {SUN_FILE_HOST_NAME,“_SUN_FILE_HOST_NAME”,0},
    {SUN_ENUMERATION_COUNT,“_SUN_ENUMERATION_COUNT”,0},
};

/* the number of targets */
int num_targets = sizeof(mytargets)/sizeof(targets_t);

/* widgets we interact with */
Widgetdrop_target;
Widgettextedit;
```

```
/* main.c (continued) */
Widgetfile;

char*arg0;/* program name */

/*
 * handle source & destination sides of dnd
 */
static void
DropTargetCB(Widget w, XtPointer clientData, XtPointer callData)
{
    OlDropTargetCallbackStruct*cd;

    DP fprintf(stderr, “calling DropTargetCB\n”);

    /* initalize the atom names */
    if(mytargets[0].atom == 0)
    {
    XrmValuesource, dest;
    int    i;

    dest.size = sizeof(Atom);

    for(i = 0; i < num_targets; i++)
    {
        source.size = strlen(mytargets[i].name)+1;
        source.addr = mytargets[i].name;
        dest.addr = (char *)&mytargets[i].atom;
        XtConvertAndStore(drop_target, XtRString,
            &source, XtRAtom, &dest);
    }
    }

    /* call the appropriate owner/requestor routines */
    cd = (OlDropTargetCallbackStruct *) callData;
    switch (cd->reason)
    {
    case OL_REASON_DND_OWNSELECTION: /* case when we do a drag */
    owner(cd->widget, cd->time);
    break;
    case OL_REASON_DND_TRIGGER: /* case when we detect a drop */
    requestor(cd->widget, cd->selection, cd->time);
    }
}
```

```
/* main.c (continued) */
/* mark drop target as full if text is typed */
static void
text_modified(Widget w, XtPointer clientData, XtPointer
callData)
{
    XtVaSetValues(drop_target, XtNfull, (XtArgVal)TRUE, NULL);
}

/* main initalization routine */
main(int argc, char **argv)
{
    XtAppContext    appContext;
    Widgettoplevel, base;
    Widgetcontrol, scrolledwin;
    Widgetblank;

    /* save program name */
    arg0 = argv[0];

    /* initialize and build the GUI widgets */
    OlToolkitInitialize((XtPointer) NULL);
    toplevel = XtAppInitialize(&appContext, “AsciiEdit”,
           (XrmOptionDescList) NULL,
           0, &argc, argv, (String *) NULL,
           (ArgList) NULL, 0);

    base = XtVaCreateManagedWidget(“base”,
        rubberTileWidgetClass,
        toplevel,
        NULL);

    control = XtVaCreateManagedWidget(“control”,
        rubberTileWidgetClass,
        base,
        NULL);

    blank = XtVaCreateManagedWidget(“blank”,
        staticTextWidgetClass,
        control,
        NULL);

    /* build the drop target and set its initial properties */
    drop_target = XtVaCreateManagedWidget(“drop_target”,
        dropTargetWidgetClass,
```

```
/* main.c (continued) */
        control,
        XtNfull,
            (XtArgVal)FALSE,
        XtNdndPreviewHints,
            (XtArgVal)OlDnDSitePreviewDefaultSite,
        XtNdndMoveCursor,
            (XtArgVal)OlGetMoveDocDragCursor(toplevel),
        XtNdndCopyCursor,
            (XtArgVal)OlGetDupeDocDragCursor(toplevel),
        XtNdndAcceptCursor,
            (XtArgVal)OlGetDupeDocDropCursor(toplevel),
        XtNdndRejectCursor,
            (XtArgVal)OlGetDupeDocNoDropCursor(toplevel),
        NULL);


    file = XtVaCreateManagedWidget("file",
        staticTextWidgetClass,
        control,
        NULL);

    scrolledwin = XtVaCreateManagedWidget("scrolledwin",
        scrolledWindowWidgetClass,
        base,
        NULL);

    textedit = XtVaCreateManagedWidget("textedit",
        textEditWidgetClass,
        scrolledwin,
        NULL);

    /* let us know if we have text to drag-n-drop */
    XtAddCallback(textedit, XtNpostModifyNotification,
text_modified, NULL);

    /* notifiy us of a drag or drop operation */
    XtAddCallback(drop_target, XtNownSelectionCallback,
DropTargetCB, NULL);
    XtAddCallback(drop_target, XtNdndTriggerCallback,
DropTargetCB, NULL);

    /* realize the widget and start the notification loop */
    XtRealizeWidget(toplevel);
    XtAppMainLoop(appContext);
```

```
/* main.c (continued) */

}

/* return the ascii name of a X Atom */
char *
get_atom_name(Atom atom)
{
    return(XGetAtomName(XtDisplay(drop_target), atom));
}

/* return the data and length of the text displayed */
Boolean
get_data(char **data, unsigned long *length)
{
    staticchar*content = NULL;

    if(content)
    {
    XtFree(content);
    }
    if (!OlTextEditCopyBuffer((TextEditWidget) textedit,
&content))
    {
    OlWarning("get_data: error trying to copy textedit
buffer\n");
    return(FALSE);
    }
    *data = content;
    *length = strlen(content)+1;
    return(TRUE);
}

/* return a data label for the data */
char *
get_data_label()
{
    staticcharbuff[200];
    char*c;
    Stringfilename;

    XtVaGetValues(file, XtNstring, &filename, NULL);
    c = strrchr(filename, '/');
    if(c)
    {
```

```
/* main.c (continued) */
    strcpy(buff, c);
    }
    else
    {
    strcpy(buff, filename);
    }
    XtFree(filename);

    return(buff);
}

/* return the name of the application */
char *
get_name()
{
    char*label = strrchr(arg0, '/');

    if(label)
    {
    return(label);
    }
    else
    {
    return(arg0);
    }
}

/* return the name of the file currently being edited if any */
char *
get_file_name()
{
    staticcharbuff[200];
    Stringfilename;

    XtVaGetValues(file, XtNstring, &filename, NULL);
    strcpy(buff, filename);
    XtFree(filename);

    return(buff);
}

/* callback with the data from the selection */
dnd_load(Dnd_t *dnd)
{
```

```
/* main.c (continued) */
    staticchar*sys = 0;

    /* fprintfs just to see ALL the data that came back */
    fprintf(stderr, “filename\t= %s\n”,
    dnd->filename?dnd->filename:”(NULL)”);

    fprintf(stderr, “data\t\t= %.50s%s\n”,
    dnd->data?dnd->data:”(NULL)”,
    ((int)strlen(dnd->data)>50)?” . . .”:””);

    fprintf(stderr, “length\t\t= %d\n”,
    dnd->length);

    fprintf(stderr, “data_label\t= %s\n”,
    dnd->data_label?dnd->data_label:”(NULL)”);

    fprintf(stderr, “app_name\t= %s\n”,
    dnd->app_name?dnd->app_name:”(NULL)”);

    fprintf(stderr, “host_name\t= %s\n”,
    dnd->host_name?dnd->host_name:”(NULL)”);

    fprintf(stderr, “enum_count\t= %d\n”,
    dnd->enum_count);

    /* displays the data returned */
    if(!sys)
    {
    char buff[100];

    if(sysinfo(SI_HOSTNAME, buff, 100) == -1)
    {
        return(FALSE);
    }
    sys = strdup(buff);
    }
    if(dnd->host_name && strcmp(dnd->host_name, sys) == 0 && dnd-
>filename)
    {
    XtVaSetValues(drop_target, XtNfull, (XtArgVal)TRUE, NULL);

    XtVaSetValues(textedit,
        XtNsourceType,OL_DISK_SOURCE,
        XtNsource, dnd->filename,
```

```
/* main.c (continued) */
        NULL);

    XtVaSetValues(file,
        XtNstring, dnd->filename,
        NULL);
    }
    else
    {
    XtVaSetValues(drop_target, XtNfull, (XtArgVal)TRUE, NULL);

    XtVaSetValues(textedit,
        XtNsourceType,OL_STRING_SOURCE,
        XtNsource, dnd->data,
        NULL);

    XtVaSetValues(file,
        XtNstring, "",
        NULL);
    }
}
```

3.7.1 Function main()

The program begins by calling functions for OLIT and for an Xt application to set up the widgets.

Then managed widgets are defined by calls to `XtVaCreateManagedWidget()`. Next, callback routines are added by calls to `XtAddCallback()`. The widgets are then realized by calls to `XtRealizeWidget()`.

Finally, `XtAppMainLoop()` is called, establishing the main event handling loop.

3.7.2 Function DropTargetCB()

`DropTargetCB()` is a callback function to handle both the source and destination sides of the drag and drop.

3.7.3 Other Important Functions

`get_atom_name()` is a function used in debugging to return a printable name for an atom.

`get_data()` returns the text of the TextEdit widget.

`get_data_label()` returns the last component of the file name to be used as a data label.

`get_name()` returns the name of this application.

`get_file_name()` returns the file path and filename.

`dnd_load()` is called after the drop is completed, so we can display the data we received.

These functions are called by routines in the `owner.c` and `requestor.c` files when that data is needed in the drag and drop process.

3.8 Module requestor.c

Contains the definition of functions `requestor()`, `GetSelection()`, and other important functions. These functions are discussed below. Here are the contents of `requestor.c`:

```
/* requestor.c (continued) */
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <Xol/OpenLook.h>
#include <Xol/DropTarget.h>
#include <stdio.h>
#include “dnd.h”

/* forward pointers */
staticvoidcheck_state();
staticvoidmake_request();

#defineSET_FLAG(i, flag)(flag |= 1<<i)
#defineFLAG_SET(i, flag)(flag & 1<<i)

structload/* structure to keep track of the current state */
```

```
/* requestor.c (continued) */
{
    unsigned longreq_flag;/* current request */
    unsigned longrec_flag;/* values received back */
    unsigned longerr_flag;/* errors received back */
    unsigned longseen_flag;/* targets seen so far */
    unsigned longtargets;/* targets suppored by other tool */
    char*filename;/* contents of filename selection */
    char*data;/* contents of string/text selection */
    intlength;/* contents of length selection */
    intnum_avail_types;/* Number of available types */
    Atom*avail_types;/* contents of available types sel. */
    char*data_label;/* contents of data label selection */
    char*app_name;/* contents of name selection */
    char*host_name;/* contents of host name selection */
    intenum_count;/* contents of the enumerate count sel*/
    Widgetwidget;/* drop target widget */
    Atomselection;/* current selection item */
    Timetime;/* time stamp */
} state;

/* strdup type of routine that takes care of the length of the
string */
static char*
save_str(char *str, int len)
{
    char*new;

    if(!str || len <= 0)
    {
    return(NULL);
    }
    new = (char *)malloc((len+1));
    strncpy(new, str, len);
    new[len] = '\0';
}

/* the next set of 13 routines each take the data passed by a
 * selection request and saves the data in the state structure.
*/
static Boolean
load_sun_file_host_name(
    Atom *type,
    XtPointer value,
    unsigned long *length,
```

```
/* requestor.c (continued) */
    int *format)
{
    DP fprintf(stderr, “calling load_sun_file_host_name\n”);
    state.host_name = save_str(value, *length);
    return(TRUE);
}

static Boolean
load_string(
    Atom *type,
    XtPointer value,
    unsigned long *length,
    int *format)
{
    DP fprintf(stderr, “calling load_string\n”);
    state.data = save_str(value, *length);
    return(TRUE);
}

static Boolean
load_name(
    Atom *type,
    XtPointer value,
    unsigned long *length,
    int *format)
{
    DP fprintf(stderr, “calling load_name\n”);
    state.app_name = save_str(value, *length);
    return(TRUE);
}

static Boolean
load_sun_available_types(
    Atom *type,
    XtPointer value,
    unsigned long *length,
    int *format)
{
    DP fprintf(stderr, “calling load_sun_available_types\n”);
    state.avail_types = (Atom *)malloc(*length*sizeof(Atom));
    state.num_avail_types = (int)*length;
    memcpy(state.avail_types, value,
(int)(*length*sizeof(Atom)));
    return(TRUE);
```

```
/* requestor.c (continued) */
}

static Boolean
load_length(
    Atom *type,
    XtPointer value,
    unsigned long *length,
    int *format)
{
    unsigned long*len;

    DP fprintf(stderr, “calling load_length\n”);
    len = (unsigned long *)value;
    state.length = (int)len[0];
    return(TRUE);
}

static Boolean
load_text(
    Atom *type,
    XtPointer value,
    unsigned long *length,
    int *format)
{
    DP fprintf(stderr, “calling load_text\n”);
    state.data = save_str(value, *length);
    return(TRUE);
}

static Boolean
load_sun_dragdrop_done(
    Atom *type,
    XtPointer value,
    unsigned long *length,
    int *format)
{
    DP fprintf(stderr, “calling load_sun_dragdrop_done\n”);
    return(TRUE);
}

static Boolean
load_targets(
    Atom *type,
    XtPointer value,
```

```
/* requestor.c (continued) */
    unsigned long *length,
    int *format)
{
    inti,j;
    Atom*list;

    DP fprintf(stderr, “calling load_targets\n”);
    list = (Atom *)value;
    while(list && *list)
    {
    for(i = 0; i < num_targets; i++)
    {
        if(*list == mytargets[i].atom)
        {
            SET_FLAG(mytargets[i].type, state.targets);
            break;
        }
    }
    list++;
    }
    return(TRUE);
}

static Boolean
load_sun_selection_end(
    Atom *type,
    XtPointer value,
    unsigned long *length,
    int *format)
{
    DP fprintf(stderr, “calling load_sun_selection_end\n”);
    OlDnDDragNDropDone(state.widget,
             state.selection,
             state.time,
             NULL, NULL);
    return(TRUE);
}

static Boolean
load_sun_enumeration_count(
    Atom *type,
    XtPointer value,
    unsigned long *length,
    int *format)
```

```
/* requestor.c (continued) */
{
    unsigned long*len;

    DP fprintf(stderr, “calling load_sun_enumeration_count\n”);
    len = (unsigned long *)value;
    state.enum_count = (int)len[0];
    return(TRUE);
}

static Boolean
load_file_name(
    Atom *type,
    XtPointer value,
    unsigned long *length,
    int *format)
{
    DP fprintf(stderr, “calling load_file_name\n”);
    state.filename = save_str(value, *length);
    return(TRUE);
}

static Boolean
load_sun_data_label(
    Atom *type,
    XtPointer value,
    unsigned long *length,
    int *format)
{
    DP fprintf(stderr, “calling load_sun_data_label\n”);
    state.data_label = save_str(value, *length);
    return(TRUE);
}

static Boolean
load_sun_load(
    Atom *type,
    XtPointer value,
    unsigned long *length,
    int *format)
{
    DP fprintf(stderr, “calling load_sun_load\n”);
    return(TRUE);
}
```

```
/* requestor.c (continued) */
/* a dubugging routine to list out the names of a list of atoms */
static char *
list_types(Atom *list, int num)
{
    staticcharbuff[1000];
    inti;

    buff[0] = '\0';
    for(i = 0; i < num; i++)
    {
    strcat(buff, (char *)get_atom_name(list[i]));
    strcat(buff, ",");
    }
    if(buff[0] == '\0')
    {
    strcat(buff, "NULL");
    }
    else
    {
    buff[strlen(buff)-1] = '\0';
    }
    return(buff);
}

/* debbugging routine to list out the names of atoms marked in a
flag */
static char *
list_flags(unsigned long flags)
{
    staticcharbuff[1000];
    inti;

    buff[0] = '\0';
    for(i = 0; i < num_targets; i++)
    {
    if(FLAG_SET(mytargets[i].type, flags))
    {
        strcat(buff, mytargets[i].name);
        strcat(buff, ",");
    }
    }
    if(buff[0] == '\0')
    {
    strcat(buff, "NULL");
```

```
/* requestor.c (continued) */
    }
    else
    {
    buff[strlen(buff)-1] = '\0';
    }
    return(buff);
}

/* debugging routine to print out the state structure */
static void
print_state()
{
    fprintf(stderr, "\t--------------------\n");
    fprintf(stderr, "\treq_flag\t= 0x%04X\n\t\t%s\n",
    state.req_flag,
    list_flags(state.req_flag));

    fprintf(stderr, "\trec_flag\t= 0x%04X\n\t\t%s\n",
    state.rec_flag,
    list_flags(state.rec_flag));

    fprintf(stderr, "\terr_flag\t= 0x%04X\n\t\t%s\n",
    state.err_flag,
    list_flags(state.err_flag));

    fprintf(stderr, "\tseen_flag\t= 0x%04X\n\t\t%s\n",
    state.seen_flag,
    list_flags(state.seen_flag));

    fprintf(stderr, "\ttargets\t\t= 0x%04X\n\t\t%s\n",
    state.targets,
    list_flags(state.targets));

    fprintf(stderr, "\tfilename\t= %s\n",
    state.filename?state.filename:"(NULL)");

    fprintf(stderr, "\tdata\t\t= %s\n",
    state.data?state.data:"(NULL)");

    fprintf(stderr, "\tlength\t\t= %d\n",
    state.length);

    fprintf(stderr, "\tnum_avail_types\t= %d\n",
    state.num_avail_types);
```

```
/* requestor.c (continued) */

    fprintf(stderr, “\tavail_types\t= %s\n”,
    list_types(state.avail_types, state.num_avail_types));

    fprintf(stderr, “\tdata_label\t= %s\n”,
    state.data_label?state.data_label:”(NULL)”);

    fprintf(stderr, “\tapp_name\t= %s\n”,
    state.app_name?state.app_name:”(NULL)”);

    fprintf(stderr, “\thost_name\t= %s\n”,
    state.host_name?state.host_name:”(NULL)”);

    fprintf(stderr, “\tenum_count\t= %d\n”,
    state.enum_count);
    fprintf(stderr, “\t--------------------\n”);
}

/* called each time we get a selection back */
static void
GetSelection(
    Widget w,
    XtPointer clientData,
    Atom *selection,
    Atom *type,
    XtPointer value,
    unsigned long *length,
    int *format)
{
    atom_tthis_atom;
    intresults = FALSE;
    Atomtarget = (Atom) clientData;
    inti;

    DP fprintf(stderr, “calling GetSelection\n”);

    /* find out which selection came back */
    for(i = 0; i < num_targets; i++)
    {
    if(target == mytargets[i].atom)
    {
        this_atom = mytargets[i].type;
        break;
    }
```

```
/* requestor.c (continued) */
    }

    /* check for errors */
    if (length == 0)
    {
    DP fprintf(stderr, "ERROR\n");
    SET_FLAG((int)this_atom, state.err_flag);
    return;
    }
    if(i == num_targets)
    {
    this_atom = UNKNOWN;
    fprintf(stderr, "atom requested 0x%X\n", target);
    fprintf(stderr, "unknown atom requested '%s'\n",
        get_atom_name(target));
    return;
    }
    else
    {
    DP fprintf(stderr, "ConvertSelection called for %s\n",
           mytargets[i].name);
    }

    /* call the appropriate procedure to load the info */
    switch(this_atom)
    {
    caseTARGETS:
    results = load_targets(type, value, length, format);
    break;
    caseFILE_NAME:
    results = load_file_name(type, value, length, format);
    break;
    caseSTRING:
    results = load_string(type, value, length, format);
    break;
    caseLENGTH:
    results = load_length(type, value, length, format);
    break;
    caseSUN_AVAILABLE_TYPES:
    results = load_sun_available_types(type, value,
           length, format);
    break;
    caseSUN_LOAD:
    results = load_sun_load(type, value, length, format);
```

```
/* requestor.c (continued) */
    break;
    caseSUN_DATA_LABEL:
    results = load_sun_data_label(type, value, length, format);
    break;
    caseSUN_DRAGDROP_DONE:
    results = load_sun_dragdrop_done(type, value,
           length, format);
    break;
    caseTEXT:
    results = load_text(type, value, length, format);
    break;
    caseSUN_SELECTION_END:
    results = load_sun_selection_end(type, value,
           length, format);
    break;
    caseNAME:
    results = load_name(type, value, length, format);
    break;
    caseSUN_FILE_HOST_NAME:
    results = load_sun_file_host_name(type, value,
           length, format);
    break;
    caseSUN_ENUMERATION_COUNT:
    results = load_sun_enumeration_count(type, value,
           length, format);
    break;
    }

    /* if the load was succesfull */
    if(results)
    {
    /* mark the received flag */
    SET_FLAG((int)this_atom, state.rec_flag);
    }
    else
    {
    /* mark the error flag */
    SET_FLAG((int)this_atom, state.err_flag);
    }
    DP fprintf(stderr, "GetSelection call\n%04X\n%04X\n%04X\n",
        state.req_flag,
        state.rec_flag,
        state.err_flag);
```

```
/* requestor.c (continued) */
    /* if we got all our requests back check the state to
     * find out what to do next
     */

    if(state.req_flag == state.rec_flag|state.err_flag)
    {
    check_state();
    }
}

/* initialize the state structure (We're begining a new drop */
static void
init_state()
{
    state.req_flag = 0;
    state.rec_flag = 0;
    state.err_flag = 0;
    state.seen_flag = 0;

    state.targets = 0;

    if(state.filename)
    {
    free(state.filename);
    }
    state.filename = 0;

    if(state.data)
    {
    free(state.data);
    }
    state.data = 0;

    state.length = -1;

    if(state.avail_types)
    {
    free(state.avail_types);
    }
    state.avail_types = 0;
    state.num_avail_types = 0;

    if(state.data_label)
    {
```

```
/* requestor.c (continued) */
    free(state.data_label);
    }
    state.data_label = 0;

    if(state.app_name)
    {
    free(state.app_name);
    }
    state.app_name = 0;

    if(state.host_name)
    {
    free(state.host_name);
    }
    state.host_name = 0;

    state.enum_count = -1;
}

/* request the target specified in the request flag */
staticvoid
make_request()
{
    staticAtom*requests = 0;
    intnum_request = 0;
    inti;

    /* make an array that will hold the requests */
    if(!requests)
    {
    requests = (Atom *)malloc(num_targets*sizeof(Atom));
    }

    /* check each target to see if we request it */
    for(i = 0; i < num_targets; i++)
    {
    if(FLAG_SET(mytargets[i].type, state.req_flag))
    {
        requests[num_request] = mytargets[i].atom;
        num_request++;
    }
    }
    requests[num_request] = 0;
```

```
/* requestor.c (continued) */
    /* ask for the list of targets */
    XtGetSelectionValues(state.widget,
        state.selection,
        requests,
        num_request,
        GetSelection,
        requests,
        state.time);
}

/* we've been droped on */
void
requestor(Widget widget, Atom selection, Time time)
{
    /*
     * put into an array a series of questions in the
     * form of atoms that source understands.  Then ask
     * selection to deliver the questions.  We also
     * register a function to handle the answers
     */

    DP fprintf(stderr, "calling requestor\n");

    /* initialize the state and save the calling info */
    init_state();
    state.widget = widget;
    state.selection = selection;
    state.time = time;

    /* request the target list */
    SET_FLAG((int)TARGETS, state.req_flag);

    make_request();
}

/* part of the request have been completed so lets see if there
 * is anything else we should do
*/
static void
check_state()
{
    inttmp = 0;
```

```
/* requestor.c (continued) */
    DP fprintf(stderr, “Before\n”);

    /* save those targets we’ve seen */
    state.seen_flag |= state.rec_flag;

    DP print_state();

    /* clear the request, received and error flags */
    state.req_flag = 0;
    state.rec_flag = 0;
    state.err_flag = 0;

    /* check if this is the first request */
    SET_FLAG((int)TARGETS, tmp);
    if(tmp == state.seen_flag)
    {
    /* request the info for those we know about */
    if(FLAG_SET(FILE_NAME, state.targets))
        SET_FLAG((int)FILE_NAME, state.req_flag);

    if(FLAG_SET(LENGTH, state.targets))
        SET_FLAG((int)LENGTH, state.req_flag);

    if(FLAG_SET(SUN_AVAILABLE_TYPES, state.targets))
        SET_FLAG((int)SUN_AVAILABLE_TYPES, state.req_flag);

    if(FLAG_SET(SUN_DATA_LABEL, state.targets))
        SET_FLAG((int)SUN_DATA_LABEL, state.req_flag);

    if(FLAG_SET(NAME, state.targets))
        SET_FLAG((int)NAME, state.req_flag);

    if(FLAG_SET(SUN_FILE_HOST_NAME, state.targets))
        SET_FLAG((int)SUN_FILE_HOST_NAME, state.req_flag);

    if(FLAG_SET(SUN_ENUMERATION_COUNT, state.targets))
        SET_FLAG((int)SUN_ENUMERATION_COUNT, state.req_flag);


    if(FLAG_SET(STRING, state.targets))
    {
        SET_FLAG((int)STRING, state.req_flag);
    }
    else if(FLAG_SET(TEXT, state.targets))
```

```
/* requestor.c (continued) */
    {
        SET_FLAG((int)TEXT, state.req_flag);
    }
    }
    else if(!FLAG_SET(SUN_DRAGDROP_DONE, state.seen_flag))
    {
    /* since we haven't seen the end info then request it */
    SET_FLAG((int)SUN_DRAGDROP_DONE, state.req_flag);
    SET_FLAG((int)SUN_SELECTION_END, state.req_flag);
    }
    else
    {
    Dnd_tuser_data;

    /* fill in the user info structure and display it */
    user_data.filename = state.filename;
    user_data.data = state.data;
    user_data.length = state.length;
    user_data.data_label = state.data_label;
    user_data.app_name = state.app_name;
    user_data.host_name = state.host_name;
    user_data.enum_count = state.enum_count;

    dnd_load(&user_data);
    init_state();
    return;
    }
    DP fprintf(stderr, "After\n");
    DP print_state();
    make_request();
}
```

3.8.1 Function requestor()

The requestor side begins by calling `requestor()`, which initializes the state of the drop and starts things off by setting the request flag (*state.req_flag*) to request the targets. From this point we never know what order things are going to be coming back to us, so we have to handle them as they come.

3.8.2 Function GetSelection()

`GetSelection()` checks which atom came back, and calls the appropriate load function. It then goes on to set the *state* structure, so we know which atom came back and what its state is.

3.8.3 Function init_state()

This function initializes the *state* structure for a new drop. The atom parameters in this structure are defined by *state.*

3.8.4 Function make_request()

`make_request()` checks the request flag (*req_flag*) and requests that the appropriate atoms be converted. It also registers a callback (`GetSelection`) to handle the return of the data.

3.8.5 Load Functions

`load_sun_file_host_name()`, `load_name()`, `load_sun_selection_end()` and similar functions load the *state* structure with the data from the corresponding X atom.

3.8.6 Debugging Functions

The functions `list_flags()` and `print_state()` print out information used for debugging.

3.9 Module owner.c

Contains the definition of functions `owner()`, `TransactionState()`, and a series of “convert” function definitions. These functions are discussed below. Here are the contents of `owner.c`:

```
/* owner.c (continued) */
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <X11/Xatom.h>
#include <Xol/OpenLook.h>
#include <Xol/DropTarget.h>
#include <sys/systeminfo.h>
#include <stdio.h>
#include “dnd.h”

/* these 13 routines just get the requested data and gives
 * it to the selection service when requested.
*/
static Boolean
convert_sun_file_host_name(
    Atom *type,
    XtPointer *value,
    unsigned long *length,
    int *format)
{
    staticchar*sys = 0;

    DP fprintf(stderr, “calling convert_sun_file_host_name\n”);
    if(!sys)
    {
    char buff[100];

    if(sysinfo(SI_HOSTNAME, buff, 100) == -1)
    {
        return(FALSE);
    }
    sys = strdup(buff);
    }
    *type = XA_STRING;
    *value = sys;
    *length = strlen(sys)+1;
    *format = 8;
    return(TRUE);
}

static Boolean
convert_string(
    Atom *type,
```

```
/* owner.c (continued) */
    XtPointer *value,
    unsigned long *length,
    int *format)
{
    DP fprintf(stderr, “calling convert_string\n”);
    *type = XA_STRING;
    get_data(value, length);
    *format = 8;
    return(TRUE);
}

static Boolean
convert_name(
    Atom *type,
    XtPointer *value,
    unsigned long *length,
    int *format)
{
    DP fprintf(stderr, “calling convert_name\n”);
    *type = XA_STRING;
    *value = (XtPointer)get_name();
    *length = (unsigned long)strlen(*value)+1;
    *format = 8;
    return(TRUE);
}

static Boolean
convert_sun_available_types(
    Atom *type,
    XtPointer *value,
    unsigned long *length,
    int *format)
{
    staticAtomav_type = XA_STRING;

    DP fprintf(stderr, “calling convert_sun_available_types\n”);
    *type = XA_ATOM;
    *value = (XtPointer)&av_type;
    *length = 1;
    *format = 32;
    return(TRUE);
}

static Boolean
```

```
/* owner.c (continued) */
convert_length(
    Atom *type,
    XtPointer *value,
    unsigned long *length,
    int *format)
{
    char*val;
    unsigned longlen;

    DP fprintf(stderr, “calling convert_length\n”);
    *type = XA_INTEGER;
    get_data((char **)&val, &len);
    *length = 1;
    *value = (XtPointer)&len;
    *format = 32;
    return(TRUE);
}

static Boolean
convert_text(
    Atom *type,
    XtPointer *value,
    unsigned long *length,
    int *format)
{
    DP fprintf(stderr, “calling convert_text\n”);
    *type = XA_STRING;
    get_data(value, length);
    *format = 8;
    return(TRUE);
}

static Boolean
convert_sun_dragdrop_done(
    Atom *type,
    XtPointer *value,
    unsigned long *length,
    int *format)
{
    DP fprintf(stderr, “calling convert_sun_dragdrop_done\n”);
    *type = XA_INTEGER;
    *length = (unsigned long)0;
    *value = (XtPointer)0;
    *format = 32;
```

```
/* owner.c (continued) */
    return(TRUE);
}

static Boolean
convert_targets(
    Atom *type,
    XtPointer *value,
    unsigned long *length,
    int *format)
{
    staticAtom*targets;

    DP fprintf(stderr, “calling convert_targets\n”);
    if(!targets)
    {
    int i;

    targets = (Atom *)malloc(num_targets*sizeof(Atom));
    for(i = 0; i < num_targets; i++)
    {
        targets[i] = mytargets[i].atom;
    }
    }
    *type = XA_ATOM;
    *value = (XtPointer)targets;
    *length = num_targets;
    *format = 32;
    return(TRUE);
}

static Boolean
convert_sun_selection_end(
    Atom *type,
    XtPointer *value,
    unsigned long *length,
    int *format)
{
    DP fprintf(stderr, “calling convert_sun_selection_end\n”);
    *type = XA_INTEGER;
    *length = (unsigned long)0;
    *value = (XtPointer)0;
    *format = 32;
    return(TRUE);
}
```

```
/* owner.c (continued) */

static Boolean
convert_sun_enumeration_count(
    Atom *type,
    XtPointer *value,
    unsigned long *length,
    int *format)
{
    staticintcount = 1;

    DP fprintf(stderr, “calling
convert_sun_enumeration_count\n”);
    *type = XA_INTEGER;
    *length = 1;
    *value = (XtPointer)&count;
    *format = 32;
    return(TRUE);

}

static Boolean
convert_file_name(
    Atom *type,
    XtPointer *value,
    unsigned long *length,
    int *format)
{
    DP fprintf(stderr, “calling convert_file_name\n”);
    *type = XA_STRING;
    *value = (XtPointer)get_file_name();
    *length = (unsigned long )strlen(*value)+1;
    *format = 8;
    return(TRUE);
}

static Boolean
convert_sun_data_label(
    Atom *type,
    XtPointer *value,
    unsigned long *length,
    int *format)
{
    DP fprintf(stderr, “calling convert_sun_data_label\n”);
    *type = XA_STRING;
```

```
/* owner.c (continued) */
    *value = (XtPointer)get_data_label();
    *length = strlen(*value)+1;
    *format = 8;
    return(TRUE);
}

static Boolean
convert_sun_load(
    Atom *type,
    XtPointer *value,
    unsigned long *length,
    int *format)
{
    DP fprintf(stderr, “calling convert_sun_load\n”);
    *type = XA_INTEGER;
    *length = (unsigned long)0;
    *value = (XtPointer)0;
    *format = 32;
    return(TRUE);
}

/* this routine gets called when ever a conversion is requested */
static Boolean
ConvertSelection(
    Widget w,
    Atom *selection,
    Atom *atom,
    Atom *type,
    XtPointer *value,
    unsigned long *length,
    int *format)
{
    atom_tthis_atom;
    Booleanresults = FALSE;
    inti;

    DP fprintf(stderr, “calling ConvertSelection\n”);

    /* find out which atom is requested */
    for(i = 0; i < num_targets; i++)
    {
    if(*atom == mytargets[i].atom)
    {
        this_atom = mytargets[i].type;
```

```
/* owner.c (continued) */
        break;
    }
    }
    if(i == num_targets)
    {
    this_atom = UNKNOWN;
    DP fprintf(stderr, "atom requested 0x%X\n", *atom);
    DP fprintf(stderr, "unknown atom requested '%s'\n",
        get_atom_name(*atom));
    }
    else
    {
    DP fprintf(stderr, "ConvertSelection called for %s\n",
        mytargets[i].name);
    }

    /* call the appropriate convert proc */
    switch(this_atom)
    {
    caseTARGETS:
    results = convert_targets(type, value, length, format);
    break;
    caseFILE_NAME:
    results = convert_file_name(type, value, length, format);
    break;
    caseSTRING:
    results = convert_string(type, value, length, format);
    break;
    caseLENGTH:
    results = convert_length(type, value, length, format);
    break;
    caseSUN_AVAILABLE_TYPES:
    results = convert_sun_available_types(type, value,
           length, format);
    break;
    caseSUN_LOAD:
    results = convert_sun_load(type, value, length, format);
    break;
    caseSUN_DATA_LABEL:
    results = convert_sun_data_label(type, value, length,
format);
    break;
    caseSUN_DRAGDROP_DONE:
    results = convert_sun_dragdrop_done(type, value,
```

```
/* owner.c (continued) */
            length, format);
    break;
    caseTEXT:
    results = convert_text(type, value, length, format);
    break;
    caseSUN_SELECTION_END:
    results = convert_sun_selection_end(type, value,
            length, format);
    break;
    caseNAME:
    results = convert_name(type, value, length, format);
    break;
    caseSUN_FILE_HOST_NAME:
    results = convert_sun_file_host_name(type, value,
            length, format);
    break;
    caseSUN_ENUMERATION_COUNT:
    results = convert_sun_enumeration_count(type, value,
            length, format);
    break;
    default:
    return(FALSE);
    }
    return(results);
}

/* check the state of the request etc. */
static void
TransactionState(
    Widget w,
    Atom selection,
    OlDnDTransactionState state,
    Time timestamp,
    XtPointer clientData)
{
    DP fprintf(stderr, "calling TransactionState\n");
    switch (state)
    {
    case OlDnDTransactionDone:
    case OlDnDTransactionRequestorError:
    case OlDnDTransactionRequestorWindowDeath:
    /*
     * some sort of failure occured or we are done, give up
     * selection we own. Note: we could have done the disowning
```

```
/* owner.c (continued) */
     * of selection when we got SELECTION_END, but we chose to do
     * it here...
     */
    OlDnDDisownSelection(w,
                selection,
                XtLastTimestampProcessed(XtDisplay(w)));
    OlDnDFreeTransientAtom(w, selection);
    break;
    case OlDnDTransactionBegins:
    case OlDnDTransactionEnds:
    break;
    }
}

/* starts the drag operation */
void
owner(Widget widget, Time time)
{
    Atomatom;

    DP fprintf(stderr, "calling owner\n");

    /* allocate and own selection.  Register a convert proc */
    atom = OlDnDAllocTransientAtom(widget);

    XtVaSetValues(widget, XtNselectionAtom, atom, NULL);

    OlDnDOwnSelection(widget, atom, time,
          ConvertSelection,
          (XtLoseSelectionProc) NULL,
          (XtSelectionDoneProc) NULL,
          TransactionState,
          NULL);
}
```

3.9.1 Function owner()

This function takes ownership of a selection and sets up a set of callbacks to deal with requests made of that selection. In this example we register only a "convert" procedure, which we call `ConvertSelection()` and a

"transaction" procedure, which we call `TransactionState()`. The "lose" and "done" procedures could also be registered here for more complex applications.

3.9.2 Function ConvertSelection()

`ConvertSelection()` is called each time a request is made of the selection. It will then check for errors and decide which conversion function needs to be called.

3.9.3 Function TransactionState()

`TransactionState()` is called when the selection mechanism transitions state. In this example it only looks for the *WindowDeath* transaction so that it can disown the selection.

3.9.4 Conversion Functions

This series of functions is used by `ConvertSelection()` to parse the selection request from the requestor process. Which function is used depends on the type of atom.

3.10 Resource File

This file contains X resources needed to make up the windows and controls in the sample application.

Here is the file called `Resources`:

```
AsciiEdit.base.orientation: vertical
AsciiEdit.base.control.orientation: horizontal
AsciiEdit.base.control.weight: 0

AsciiEdit.base.control.file.weight: 1
AsciiEdit.base.control.drop_target.weight: 0
AsciiEdit.base.control.blank.weight: 0

AsciiEdit.base.control.blank.strip: false
AsciiEdit.base.control.blank.string: \    \
```

```
AsciiEdit.base.scrolledwin.weight: 1

AsciiEdit.base.scrolledwin.forceHorizontalSB: False
AsciiEdit.base.scrolledwin.forceVerticalSB: True
AsciiEdit.base.scrolledwin.textedit.charsVisible: 80
AsciiEdit.base.scrolledwin.textedit.linesVisible: 60
AsciiEdit.title: Drag and Drop sample
AsciiEdit*font: lucidasans
```

3.11 Makefile

This file contains the instructions to compile and link the sample into an executable application, using the command `make`.

Here is the `Makefile`:

```
#
#########################################################################
#

SRC += main.c owner.c requestor.c
HDR += dnd.h
OBJ += $(SRC:%.c=%.o)

INCLUDE+= -I${OPENWINHOME}/include

#CFLAGS+= -g -DDEBUG
CFLAGS+= ${INCLUDE}

LDFLAGS+= -L${OPENWINHOME}/lib -R${OPENWINHOME}/lib

LIBS+= -lXol -lXt -lX11 -ltt

PROGRAM+= dnd

.KEEP_STATE:

$(PROGRAM):$(OBJ)
    $(CC) -o $(PROGRAM) $(OBJ) $(CFLAGS) $(LDFLAGS) $(LIBS)

clean:
    rm -f core $(PROGRAM) $(OBJ)
```

```
#

.INIT: $(SRC) $(HDR)

# End makefile
########################################################################
```

3.12 Data Type Registration

If a receiving application is to receive a drop from a source application, the source application must send the data in a format readable by the receiving application. (In this discussion, we use data format and data type interchangeably.) For example, if Text Editor wishes to drop data into Mail Tool, Text Editor must be able to convert the data to a format that Mail Tool can read. Conversely, if Mail Tool wishes to drop data into Text Editor, Mail Tool must be able to convert the data to a format Text Editor can read.

Although the source application is responsible for converting data to a format readable by the receiving application, it also behooves receiving application to be able to receive data in some of the more common data formats like ASCII, Sun raster imaging, or POSTSCRIPT® page description language.

Programmatically, drag and drop handshaking works as follows:

- data is selected from the source application
- data is sent (dropped) on the receiving application
- receiving application requests a list of the data formats in which the source application can send the drop
- source application replies with a list of data formats
- receiving application tells the source application which format it would like the data sent
- data is transferred.

A source application must have data conversion routines for each application into which it wishes to drop data. Creating conversion routines consists of finding out the data format of the desired drop applications, and writing

conversion routines specifically for those formats.[1] Again, if you wish your application to be able to receive drops from other applications, ensure that your application can receive data in some of the more common data formats.

SunSoft has undertaken a data type registration program to help standardize the data format names by which applications request data formats from each other. SunSoft encourages all companies that wish to share their data with other applications to register data format names for their application's data. This name will be used by other applications to reference desired data formats. Refer to Appendix C, "Vendor Data Type Registration" for more information on data type registration.

SunSoft will provide a public repository for data format names as well as additional format information. This information will be made available to all software developers.

1. Refer to the receiving application's manuals or call the company that produces the receiving application for details of the data format.

Implementing Drag and Drop with DeskSet

4

DeskSet drag and drop is implemented using X selections to negotiate formats and transfer information. The techniques described here are applicable to any selection transfer, such as cut-and-paste as well as drag-and-drop. This discussion assumes that you understand OLIT or XView selections, and have read the Inter-Client Communications Conventions Manual (ICCCM) X Version 11 Release. Note that this chapter only describes the data transfer part of selections. Specifying owner and requestor application, and telling the receiver at what selection rank to address the source is incorporated in each of the OPEN LOOK toolkits and is not of concern to the application programmer.

The DeskSet drag and drop communications protocol consists of a set of predefined atoms used by owner and requestor applications to exchange information about a drag and drop selection. These predefined atoms are also called *targets*. DeskSet targets, describe the nature of a drag and drop selection as well as the interaction that can occur between the owner application and the requestor application.

Targets in a DeskSet drag and drop conversation represent a series of questions or commands sent by the receiving application to the owner application. When a requesting application asks an owner application to convert a target, it is equivalent to asking the owner a predefined question or giving it a predefined command. The selection owner may or may not support answering that particular question/command. If it doesn't, the requestor must terminate the drag and drop procedure or try to ask other questions that will allow the procedure to continue. If the selection owner does support that target, it will respond with the answer to the requestor.

4

DeskSet selection conversation is similar to a game of “go fish” or “twenty questions.” One program (the selection requestor) does all the asking, and the other program (the selection owner) does all the answering.

The Deskset conventions establish a set of targets (questions) that cooperating selection owners understand. Of course, a well behaved requestor program will try to deal with selection owners that cannot answer the standard questions, but there will be some loss of functionality when dealing with a selection holder of this type.

4.1 DeskSet Drag and Drop Handshaking

Drag and drop handshaking begins with the requestor application asking the owner application to convert the *TARGETS* atom. The owner responds with a list of atoms it understands. If the requestor recognizes enough atoms to continue the procedure, it continues. If the requestor does not recognize enough atoms to complete the process, then the transfer is aborted.

Two cases of handshaking with a DeskSet application are described below. Note that these handshaking descriptions are a simplistic view of dragging and dropping between DeskSet applications. It may be necessary to examine specific DeskSet application source code to pick up procedural nuances.

4.1.1 Handshaking—Simplest Case

The simplest case of drag and drop is when a receiver application asks the owner to send it selection data without regard to data format, alternative transport method, or any other considerations. This is described below:

1. The requestor asks the owner to convert the *_SUN_ENUMERATION_COUNT* atom. The owner responds by sending the number of source objects in the current selection.

   If the response to *_SUN_ENUMERATION_COUNT* is more than 1, then the programs must agree upon the selection object to be dragged and dropped. The owner converts the *_SUN_ENUMERATION_ITEM* target which specifies a selection item to process.

2. For each selection item, the requestor asks the owner to convert the *TEXT* target which instructs the owner to send the selection data in any format.

The requestor assumes that it can convert the data into a usable format. If it cannot convert the data, it will not accept the data and the transfer is terminated. Note that the requesting application bears the responsibility of converting the data to a usable format.

4.1.2 Handshaking with _SUN_AVAILABLE_TYPES

This example describes the handshaking protocol of a requesting application that wishes to specify the format of the requested data. This is an optional feature.

1. After obtaining the list of supported targets the requestor asks the owner to convert the *_SUN_ENUMERATION_COUNT* atom. The owner responds by sending the number of source objects in the current selection.

   If the response to *_SUN_ENUMERATION_COUNT* is more than 1, then the programs must agree upon the selection object to be dragged and dropped. The owner converts the *_SUN_ENUMERATION_ITEM* target which specifies a selection item to process. For each item in the selection, the program converts the targets described in steps 2 and 3.

2. The requestor application then asks the owner to convert *_SUN_AVAILABLE_TYPES*. The owner sends a list of data type atoms which specify the format in which the selection data may be supplied.

   The formats may be GIF, PostScript, audio, or some other format. If the applications do not wish to use an alternate transport method, then the requestor application asks the owner to convert the desired data type target. The owner responds by sending the stream of data representing the object selected in the type requested.

3. The requestor application writes the data into its address space and closes the selection transfer with the *_SUN_SELECTION_END* atom.

4.1.3 Specifying _SUN_ENUMERATION_ITEM

When multiple items are selected, the *_SUN_ENUMERATION_ITEM* target specifies which object is the subject of discussion in the selection negotiation. All targets that require the specification of *_SUN_ENUMERATION_ITEM*, and the setting of the object number as its side effect, should request these conversions as part of a `MULTIPLE` conversion.

The use of MULTIPLE adds a transaction-like nature to the set of target conversions, assuring that other target conversions delivered to the selection holder will not change the state of the selection.

Specifically, a sample conversion of a target should look like this:

```
begin multiple
_SUN_ENUMERATION_ITEM = object number
FILE_NAME
_SUN_FILE_HOST_NAME
_SUN_ENUMERATION_ITEM = -1
end multiple
```

The use of MULTIPLE ensures that another client cannot convert *_SUN_ENUMERATION_ITEM* on the same selection halfway through the conversions for your targets. If this were to happen, the results could prove unpredictable with conversion results being relative to different objects within the selection, and your client not knowing this.

Within the OpenWindows Version 3.x implementation of drag and drop, each drag and drop transfer acquires a unique selection rank. So, converting *_SUN_ENUMERATION_ITEM* on the same selection halfway through target conversion, may not be an immediate problem. But if your client is ever to use the same code to support cut/copy/paste, it is likely that your client could get conversion requests on a selection from more than one client at a time.

For a more complete discussion of MULTIPLE, and how it relates to selection target conversions, see the ICCCM.

4.2 DeskSet Drag and Drop Target Atoms

This section describes the atoms used in DeskSet applications.

4.2.1 ICCCM Target Atoms

These atoms are specified by the ICCCM and should be supported by all applications.

DELETE

Converting this atom tells the selection owner to delete the selection. This is typically done when the drag and drop operation is a move command to something that can actually store the data (like a File Manager). An application like a Print Tool should always reject the *DELETE* atom. Owner application should return a zero-length property of type *NULL* if the deletion was successful.

TARGETS

Returns a list of targets supported by the owner.

TIMESTAMP

Timestamp is an integer timestamp used to acquire the selection.

FILE_NAME

When converted, *FILE_NAME* requests the file name for the current object in the selection. The response is a name of a file that contains the selection data. This file name is not useful unless the answer to the *_SUN_FILE_HOST_NAME* is a host that the recipient can access. The *FILE_NAME* response will be relative to the machine named in *_SUN_FILE_HOST_NAME*. *FILE_NAME* is not needed if *_SUN_ATM_FILENAME* is not supported.

NAME

The name of the application as represented in the title bar. Refer to the ICCCM for details.

INSERT_SELECTION

Replace the object with the contents of the named selection.

4.2.2 DeskSet Target Atoms

_SUN_ALTERNATE_TRANSPORT_METHODS

This target is like the *TARGETS* target—it returns a list of atoms which represent the alternate transports that the owner supports. Note that use of this and other ATM targets is very rare and almost always necessary.

_SUN_ATM_FILE_NAME

This target indicates that the requestor is going to get the data via a file name. A file name consists of both a pathname (as returned by *FILE_NAME*) and a host name (as returned by *_SUN_FILE_HOST_NAME*).

_SUN_ATM_TOOL_TALK

This atom specifies the ToolTalk services to pass data.

_SUN_AVAILABLE_TYPES

This atom is like the *TARGETS* atom, but instead the owner responds with the data types in which it can provide data. This may be only one, or it may be several. Note that the responsibility for type translation lies with the requestor.

The table below lists the data types supported by DeskSet.

Table 4-1 DeskSet Data Type Atoms

Data Type	Data Type Atom	Type Description
Owners Choice	`TEXT`	Can be any data type of the owners choice.
Text	`STRING`	8-bit ISO 8859-1

Graphics	`_SUN_TYPE_gif-file`
	`_SUN_TYPE_tiff-file`
	`_SUN_TYPE_postscript-file`
	`_SUN_TYPE_sun-raster`
	`_SUN_TYPE_xpm-file`
		GIF
		TIFF
		PostScript
		Sun Raster
		XPM

Audio	`_SUN_TYPE_audio-file`	SunSoft Audiotool

_SUN_DRAGDROP_DONE

This target signifies the end of a drag and drop process. It is hidden in the toolkits so the requestor application never deals with them. The owner application does, however, see them.

_SUN_SELECTION_END

Ends the selection process by indicating that the requestor is done with the current selection. Use *SUN_SELECTION_END* to notify the owner that you are done with the selection used in response to *INSERT_SELECTION* or *_SUN_LOAD*. See the section on *_SUN_LOAD* for more information.

_SUN_DATA_LABEL

A string identifier that represents the current object which is not a filename. This could be something like the last component of the pathname.

_SUN_ENUMERATION_COUNT

Specifies how many objects are in the current selection. This allows several icons to be selected and separately negotiated.

_SUN_ENUMERATION_ITEM

Specifies which object is the subject of discussion in the selection negotiation. Objects are numbered from zero to *_SUN_ENUMERATION_COUNT* -1. References to objects outside this range should be refused. References to the item -1 should be treated as resetting the current item to unspecified.

This target only has meaning within the context of a selection that contains multiple disjointed objects. It should be: 1) converted with the number of the object in the selection you wish to refer to; 2) convert the desired targets; 3) converted again to set the current item back to 0.

_SUN_FILE_HOST_NAME

The name of the host to which *FILE_NAME* is relative.

_SUN_LOAD

When converted the owner returns a selection atom to use as a reference to the current object.

_SUN_SELECTION_ERROR

If the requestor has decided that it cannot convert a selection, it should convert this target. This target has no side effects and returns no data, but can be used to inform the user that the drag and drop operation did not succeed. The

convention for informing the user of a drag and drop failure is to present an error message in the footer of the owner application, and send a pop-up message to the requestor application.

4.3 Drag and Drop and Editors

One of the primary uses of Drag and Drop with the File Manager is to drop files on cooperating applications as a shorthand for loading them in for editing. In this situation there are some difficult semantic differences that creep into the user model depending on the transport method chosen / negotiated by the applications involved in the drag and drop interaction.

Consider these two cases:

Case 1: Client A, a drag source, is running on machine X. Client B, an editor, is also running on machine X. The user drags a file from A, and drops it on B. B (the editor) realizes that the two clients share a file system, and decides to transport the data using file names and the file system as a short cut. B loads the file and runs successfully.

Case 2: Client A, a drag source, is running on machine X. Client B, an editor is also running on machine X. The user drags a file from A, and drops it on B. B (the editor) realizes that the two processes do not share a file system, and cannot rendezvous in this way. B decides to convert *TEXT*, or some more specific data type, and receives the data from A through the selection service. B loads the data and runs successfully.

Up to a point, the transfers look identical to the user, who is oblivious to the selection of transport method. The user edits the data, and then tries to execute the Save function in the editor.

In the first case, the data goes back to the place from which it came, providing an action semantically identical to loading in the file from some sort of dialog box.

In the second case, the data is either silently saved to some place within their local file system, or the user is presented with a question like “where in *this* file system do you want the data to go?” The users only option for getting the data back to where it came from would be to either drag it from the editor back to the source (assuming the editor supports sourcing drags), or to save the data into a local file system, and then use some sort of sourcing application to drag it back to the original file system and rename it to the original name.

None of the behaviors in the second case will make sense to the user, because they know nothing of the selection of transport method and how it will impact the usability of their application.

Note – There may be cases where the clients share a file system and still cannot rendezvous successfully through it. This might include selection holders whose selection resides in a special partition (databases) or whose contests are represented by more than one file (compound documents). It is important not to conclude that this is an X specific problem.

With the intent of fixing this behavior, we have defined some special mechanisms to get the data back to its original location, transparent to the user.

When a requestor application finds that it needs to obtain the source data through the selection service, and intends to allow the user to edit the data and then put it back, the recipient tries to convert the *_SUN_LOAD* target. If the selection holder supports the *_SUN_LOAD* target, it will respond with the name of a selection (called “H” for this discussion) that the holder guarantees will be unique and persistent for the life of the selection holder. This selection is associated with the original data in the drag and drop transfer.

The editor may manipulate the data as it sees fit. When it is time to save the data back to the original location, the editor creates a new selection rank and associates it with the data that it is currently holding (call the selection rank “S”). The editor then converts the *INSERT_SELECTION* target against the selection H. A selection can be guaranteed as unique by converting *SUN_SELECTION_END* against H when done. In the property associated with the target conversion, the editor places the name of the selection S.

The conversion of *INSERT_SELECTION* tells the selection holder to replace the contents of the current selection (selection H, representing the original data in storage) with the contents of the new selection (selection S, whose contents represent the edited version of the data). Refer to section 2.6.3.2 of the ICCCM.

The original selection may then make target conversions against selection S to get the data back from the editor.

4.4 Drag and Drop Example: XView Toolkit

The program `xview_dnd2.c`, located online at `$OPENWINHOME/share/src/dig_samples/dnd_xview2`, demonstrates how drag and drop can be implemented with the Xview toolkit. Once compiled, `xview_dnd2.c` allows you to drop a selection from a DeskSet application onto its rectangular drop target. `xview_dnd2` then displays the following information:

- The operation (Drag `MOVE`)
- The number of atoms contained in the *TARGETS* atom
- The atoms listed in *TARGETS*, and used in the drop operation
- The length and format of the data represented in each atom, as well as the data itself.
- Atoms listed in *TARGETS*, but not used in the operation

Note that this program does not work with multiple selections.

Section 3.4, “Drag and Drop Programming Example: OLIT Toolkit,” on page 3-7 shows a drag and drop example implemented with OLIT.

4.5 Further DeskSet Integration Information

For further information on integrating your application with DeskSet, call the SunSoft Catalyst Information Center (Appendix C, “Vendor Data Type Registration” has this number). Note, however, that the ICCCM currently does not specify the protocol supported by DeskSet. The current DeskSet protocol may change to comply with future ICCCM specifications. Refer to Appendix L of the *X Protocol Reference Manual* to see the current ICCCM.

Classing Engine 5

5.1 Overview

The Classing Engine (CE) is an OpenWindows database used to identify the characteristics, or *attributes*, of files. The CE specifies attributes such as print method, icons, and opening commands for specific *file types*. File type is defined by a file's format, its parent application, or the application executable itself. Examples of file format are:

- ASCII
- PostScript
- Sun raster files

Examples of data files created by a parent application are:

- FrameMaker® and
- Lotus 1-2-3® data files

Examples of application executables are:

- File Manager
- Mail Tool, or
- Wingz® executable files

The CE consists of two parts: a database that stores file type names and attributes, and a collection of routines that query the database. Some of the more common file attributes are:

- A filename pattern or content string to identify the file type
- Directory location of a file type icon

- Foreground and background colors of a file type icon
- Print command of a file type, if applicable
- Load and launch command for the application associated with a data file
- Edit or display command of a file

Other attributes, such as data exchange filters and text compression procedures, can be associated with a file type as well; the CE is completely extensible. In addition, it is possible to add custom databases for other data objects to the CE.

The CE acts as a central repository for all file types and their attributes. The CE also provides applications with a set of routines for determining a file's type and retrieving its attributes.

This chapter describes the CE technology and how one DeskSet program, File Manager, uses it. (File Manager is a graphical file and directory tool shipped with OpenWindows.) The CE can be used in the same manner by any Desktop application.

Note – The CE can be used by the File Manager. This chapter only discusses the CE as used by File Manager.

5.2 File Type Registration

For an application to obtain the operating attributes of a file, the file's identifier and attributes must be stored in the CE database. This requires that the file's originators, typically the vendor whose application created the file, incorporate the file's type and its attributes into the CE database. File types can be incorporated into the CE database in the following ways:

1. Software vendors may register file types and their attributes with SunSoft through the Developer Integration Format Registration program (DIFR). The new file types will be incorporated into the CE database and distributed in subsequent CE releases. Refer to Appendix C, "Vendor Data Type Registration" for detailed registration instructions.

2. Software vendors may use the CE utilities in their software installation process to update their user's CE databases with new file type information. Thus, a vendor's application can, as part of the installation process, enter its file types and attributes into the CE database.

3. Users can use CE utilities or Binder, a DeskSet application, to enter new file type information into the CE database.

5.3 Classing Engine Usage

File Manager, displayed in Figure 5-1, provides an example of how the CE can be used. File Manager is a DeskSet application that graphically displays a UNIX file system. Users may move, copy and delete files by dragging and dropping file icons onto directory icons, or onto a wastebasket icon. In addition, File Manager allows users to double-click on a data file icon to open the file in its parent application (file opening commands are stored in the CE database). For example, double-clicking on a spreadsheet data file could start the spreadsheet application program and open the data file. Double-clicking on an ASCII file will open the file with the Text Editor. File Manager also lets users print a data file by simply dropping the file's icon on the Print Tool.

Another feature of the File Manager is that different file types are represented by different icons. Thus, one application's files will have one icon, and the files of another application will have a different icon. Unique icons allow users to identify a file without opening it. File Manager retrieves the icon location from the CE. Refer to the *Solaris User's Guide* for details on how to use File Manager.

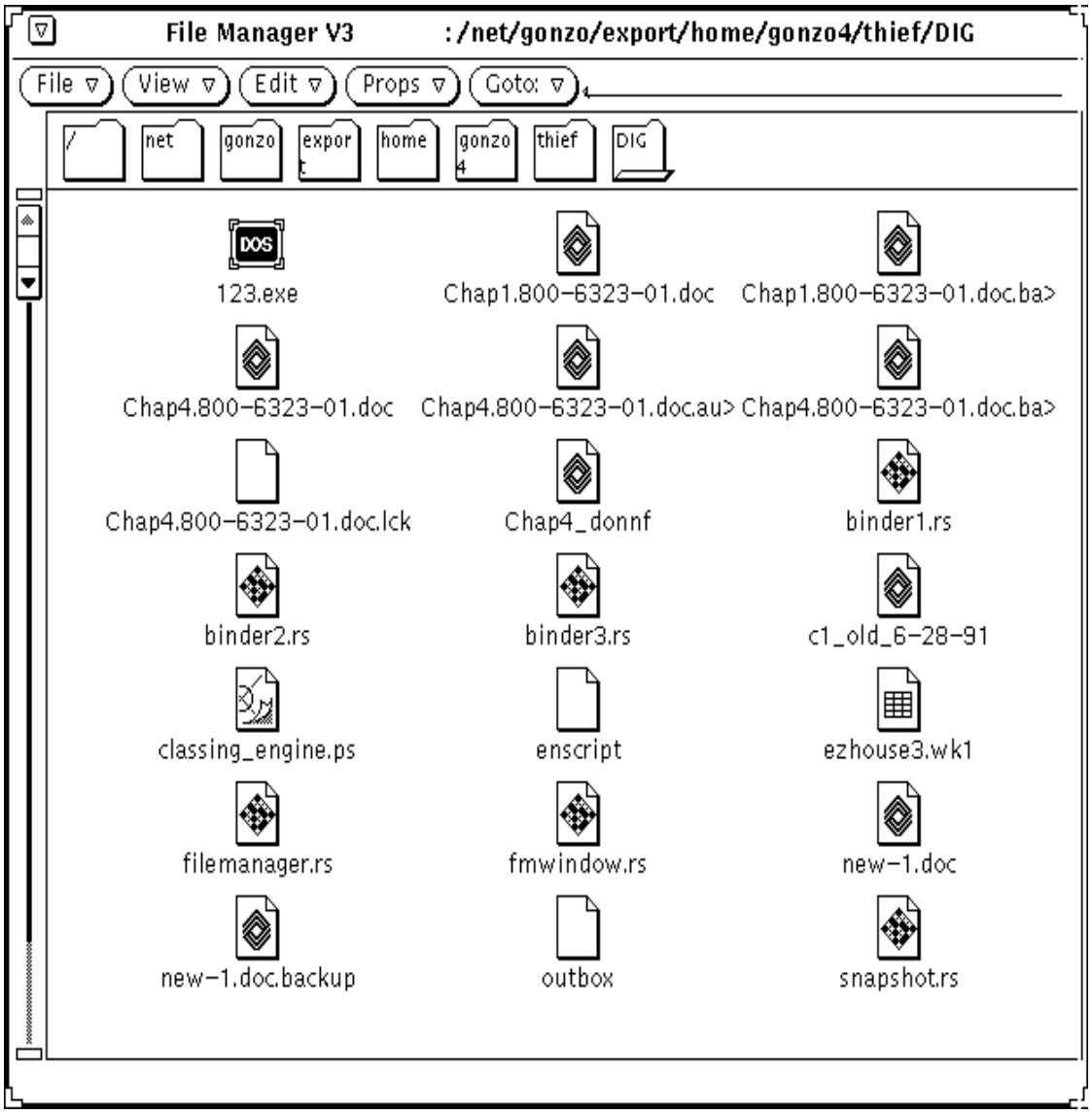


Figure 5-1 File Manager

5.4 Adding and Changing Classing Engine File Types and Attributes

Adding or changing file types and attributes in the CE consists of changing the CE database to reflect these new filetypes and attributes. Before discussing how to do this, it is necessary to discuss the structure of the CE database.

5.4.1 Classing Engine Database

The CE database contains file type names, identification patterns, and attributes. The CE database is one logical database that is the composite of three physical databases called the *user*, *system*, and *network* databases. Multiple databases allow users to personalize their environment while still having access to global data.

The user database is unique to each user and resides in the user's directory structure, the system portion is common to all users on that specific machine, and the network portion is available to everyone on the network. The CE treats these three portions as overlays. When an application queries the CE database for information, the CE will first read the entry in the user database. If an entry is not found in the user database, the CE tries the system database, and finally the network database. This assures that any CE database information customized by the user (or the system) will be used if it exists; otherwise, network information is used. The following discussion treats the three databases as a single aggregate database.

Default Location of Classing Engine Databases

Each of the three Classing Engine databases has a default location, as shown in the table below. These files are in a non-readable format. To convert these files into an ASCII-readable format, use the `ce_db_build` utility as follows:

```
ce_db_build <user | system | network> -to_ascii <file name>
```

Table 5-1 Default Classing Engine Database Locations

database	default location
user	`~/.cetables/cetables`
system	`/etc/cetables/cetables`
network	`$OPENWINHOME/lib/cetables/cetables`

5.4.2 Namespace Tables

Each CE database file consists of two *namespace tables*, which are data bases of file entries:

- A *files namespace table*, contains file type names and identifiers
- A *types namespace table*, which stores file type attributes

Both of these namespace tables are resident in the same file. To view the namespace tables use the `ce_db_build` command described in the previous section.

Each namespace table has an accompanying *namespace manager*, a collection of routines used to query that namespace table.

5.4.3 File Type Identification

Before an application can use a file's attributes, the application must identify, or *derive*, the file type. In other words, it must determine whether a file is an ASCII file, Mail Tool executable file, PostScript file, an so forth. Two methods are used to determine file types: type-by-*pattern* or type-by-*content*.

Typing by pattern involves matching the filename with a filename pattern. For example, all files whose names end in `.c` are C source files, all files that end in `.exe` are DOS executable files, and all files that end in `.ps` are PostScript files.

Typing by *content* involves matching the contents of a file to a pre-defined string or number. For example, files that have the string `WNGZWZSS` as their first characters are Wingz worksheet files. Files that contain `<Framemaker` as its first characters are FrameMaker files. This is similar to the procedure that is used by the standard UNIX `file` command that uses the `/etc/magic` file.

5.4.3.1 Files Namespace Table

The files namespace table contains entries that are used to derive file types. An excerpt of a files namespace table is shown below.

```
NS_NAME=Files      # Beginning of Files namespace table
NS_ATTR=((NS_MANAGER,junk,<$CEPATH/fns_mgr.so>))# The Files namespace manager
NS_ENTRIES=(
  ( . . .
    (FNS_TYPE,refto-Types,<filemgr-prog>)# File type = File Manager
    (FNS_FILENAME,str,<filemgr>)# File pattern = filemgr
  )( . . .
    (FNS_TYPE,refto-Types,<mailtool-prog>)# File type = Mailtool program
    (FNS_FILENAME,str,<mailtool>)# File pattern = mailtool
  )( . . .
    (FNS_TYPE,refto-Types,<lotus-spreadsheet>) # File type = lotus spreadsheet
    (FNS_FILENAME,str,<*.wk?>)# File pattern = *.wk?
  )( . . .
    (FNS_TYPE,refto-Types,<msdos-executable>)# File type = MS DOS Application
    (FNS_FILENAME,str,<*.exe>)# File pattern = *.exe
   )( . . .
    (FNS_TYPE,refto-Types,<c-file>)# File type = C source file
    (FNS_FILENAME,str,<*.c>)# File pattern = *.c
   )( . . .
    (FNS_TYPE,refto-Types,<sun-raster)# File type = Sun Raster
    (FNS_MAGIC_OFFSET,str,<0>)# Offset = 0 bytes
    (FNS_MAGIC_MATCH,str,<0x4d4d002a>)# Content Pattern = 0x4d4d002a
    (FNS_MAGIC_TYPE,str,<long>)# Content Type = long int
  )( . . .
    (FNS_TYPE,refto-Types,<framemaker-document>)#File type = Framemaker Document
    (FNS_MAGIC_OFFSET,str,<0>)# Offset = 0 bytes
    (FNS_MAGIC_MATCH,str,<<MakerFile>)# Content Pattern = <Makefile
    (FNS_MAGIC_TYPE,str,<string>)# Content Type = string
  )( . . .
    (FNS_TYPE,refto-Types,<sunwrite-document>)# File Type = SunWrite Document
    (FNS_MAGIC_OFFSET,str,<3>)# Offset = 3 bytes
    (FNS_MAGIC_MATCH,str,<pgscriptver>)# Content pattern = pgscriptver
    (FNS_MAGIC_TYPE,str,<string>)# Content Type = String
  )( . . .
    (FNS_TYPE,refto-Types,<postscript-file>)# File type = Postscript file
    (FNS_FILENAME,str,<*.ps>)# File pattern = *.ps
  )(
    (FNS_TYPE,refto-Types,<postscript-file>)# File type = Postscript file
    (FNS_MAGIC_OFFSET,str,<0>)# Offset = 0 bytes
    (FNS_MAGIC_MATCH,str,<%!>)# Content Pattern = %!
    (FNS_MAGIC_TYPE,str,<string>)# Content Type = String
  )( . . .
```

Entries in the files namespace table consist of the following arguments:

`FNS_TYPE`, or *file type name*, is the name (identifier) assigned to a file type. In the following example, the file type name for the File Manager program is `filemgr-prog`. The file type name for Lotus 1-2-3 spreadsheet files is `lotus-spreadsheet`.

`FNS_FILENAME` is the file name pattern that identifies a file's type. The file name pattern is used to match a file name to its type. For example, a file ending with `.c` is a C source file. A file ending with `.exe` is a DOS executable file.

If a file type is derived with the type-by-content method, the file type entry requires these arguments:

`FNS_MAGIC_MATCH` or *magic match*, is a string contained in all files of the type specified by `FNS_TYPE`. Thus, all FrameMaker document files contain the string `<MakerFile`. All PostScript files contain the string `%!`.

`FNS_MAGIC_TYPE` specifies the data type of the magic march. In the example, all type-by-content entries match with strings, except for sun-raster files, which use a long integer.

`FNS_MAGIC_OFFSET` specifies the number of bytes preceding the magic match. As shown in the following example, `<MakerFile` starts at the first byte in a FrameMaker document file. pgscriptver starts after the third byte in a SunWrite document file.

If both a file name pattern and a magic match are are defined like as shown in the PostScript example, a file must pass both tests before it is typed.

5.4.4 Types Namespace Table

The types namespace table contains the attribute values of the file types. Once a file type is derived, the CE can retrieve the files attributes from the types namespace table. An excerpt of a types namespace table is shown below.

```
NS_NAME=Types                              # The namespace named "Types"
NS_ATTR= ((NS_MANAGER,string, <$CEPATH/tns_mgr.so>))# The Types namespace
manager
NS_ENTRIES= (...
    (TYPE_NAME,type-id,<filemgr-prog>)
    (TYPE_ICON,icon-file,<$OPENWINHOME/include/images/filemgr.icon>)
    (TYPE_BGCOLOR,color,<79 241 255>)
    (TYPE_PRINT,string,<lpr -Plp>)
   )( . . .
    (TYPE_NAME,type-id,<lotus-spreadsheet>)
    (TYPE_OPEN,call,<dos -c 123>)
    (TYPE_ICON,icon-file,<$OPENWINHOME/include/images/spreadsheet.icon>)
    (TYPE_ICON_MASK,icon-file,<$OPENWINHOME/include/images/doc.mask.icon>)
    (TYPE_BGCOLOR,color,<255 225 255>)
    (TYPE_FILE_TEMPLATE,string,<lotus%t.wks>)
   )( . . .
    (TYPE_NAME,type-id,<compress>)
    (TYPE_OPEN,call,<uncompress>)
    (TYPE_ENCODE_PROG,call,<compress>)
    (TYPE_ENCODE_ARGS,string,<-c>)
    (TYPE_DECODE_PROG,call,<uncompress>)
    (TYPE_DECODE_ARGS,string,<-c>)
    (TYPE_ICON,icon-file,<$OPENWINHOME/include/images/compress.icon>)
    (TYPE_ICON_MASK,icon-file,<$OPENWINHOME/include/images/doc.mask.icon>)
    (TYPE_BGCOLOR,color,<255 0 0>)
    (TYPE_FILE_TEMPLATE,string,<data%t.Z>)
   )( . . .
    (TYPE_NAME,type-id,<default-app>)
    (TYPE_ICON,icon-file,<$OPENWINHOME/include/images/application.icon>)
    (TYPE_FGCOLOR,color,<0 0 0>)
    (TYPE_BGCOLOR,color,<183 229 193>)
   )(
    (TYPE_NAME,type-id,<default-doc>)
    (TYPE_OPEN,call,<textedit>)
    (TYPE_OPEN_TT,tt,<textedit>)
    (TYPE_PRINT,string,<cat $FILE | mp -lo | lpr -h>)
    (TYPE_ICON,icon-file,<$OPENWINHOME/include/images/document.icon>)
    (TYPE_ICON_MASK,icon-file,<$OPENWINHOME/include/images/doc.mask.icon>)
    (TYPE_FGCOLOR,color,<0 0 0>)
    (TYPE_BGCOLOR,color,<183 193 229>)
   )( . . .
```

Entries in the type namespace table consist of the following arguments:

`TYPE_NAME` is the name of the file type. `TYPE_NAME` matches `FNS_TYPE` in the files namespace table.

`TYPE_ICON` is the file containing the icon representation of the file type.

`TYPE_ICON_MASK` is the file containing the icon representation of file when it is selected.

`TYPE_BGCOLOR` specifies the background color of the file icon. Values are in red-green-blue (RGB) values ranging from 0 (lighter) to 255 (darker).

`TYPE_FGCOLOR` specifies the foreground color of the file icon in RGB values.

`TYPE_OPEN` specifies the command to open the file. (For File Manager this is triggered by a double mouse-click.)

`TYPE_PRINT` gives the print command for the file.

`TYPE_FILE_TEMPLATE` specifies a unique filename generated and used by the application as a filename identifier.

`TYPE_OPEN_TT` is the ToolTalk identifier used when starting applications requiring the obsolete Tooltalk protocol (as used in releases prior to Solaris 2.2).

`TYPE_MEDIA` is the ToolTalk identifier used when starting applications requiring the new Message Alliance / Media Exchange protocol (as used in releases starting with Solaris 2.2).

The attribute entry for `compress` demonstrates CE extensibility. In addition to the standard attributes, `compress` file types have four additional attributes: `TYPE_ENCODE_PROG`, `TYPE_ENCODE_ARGS`, `TYPE_DECODE_PROG`, and `TYPE_DECODE_ARGS`. A program designer has added these attributes to `compress` file types in order to provide automatic file compression / decompression. For example, these attributes can be used to link large files to a mail message. Instead of pasting the file into the message, the file could be automatically compressed (using the UNIX compress command) when the file glyph is selected and dropped on the mail tool. The compressed file appears as an file glyph. After the message is sent, the file is automatically decompressed when the file glyph is selected.

The last two entries of the code segment, `default-app` and `default-doc`, demonstrate two other interesting features. Each represents the attributes of undefined data and application files. If a file does not have a definition in the files namespace, it is given a set of generic attributes depending on whether it is an application or document.

5.4.5 Adding a New File Type

The basic steps for adding a new file type to the CE database are as follows:

1. Create an ASCII description file for the new file entry. Either extract the ASCII description file for the entire CE database using `ce_db_build` (the `man` page is in the back of this chapter), or create a new ASCII description file for the new file type entry. The process for creating a single entry ASCII description file is described in the section that follows.
2. Add a file type name and file type pattern to the files namespace table in the ASCII description file. You only need to add the file type pattern if the file type is derived using the type-by-pattern method. If the file is derived using the type-by-content method, add a magic match, magic match data type, and an offset.
3. After a new file type has been added to the files namespace table, add its attributes to the types namespace table.
4. Once the attributes are added, you can overwrite the old CE database file with the one you just created using the `ce_db_build` command. Use this command only if you are replacing the entire CE database file. If you created an ASCII description file for a subset of the entire CE database (this procedure is described in the next section), merge the file into the current CE database with the `ce_db_merge` command (the `man` page for this command is in the back of this chapter).

5.4.5.1 Adding a New File Type to the Classing Engine—Example

This section shows a step-by-step example of adding a new file type to the CE.

1. Define the file type name, its unique file name pattern or content string, and its attributes. For this example we'll use a hypothetical program called `Peakstool` that works on files of a type called `twin-peaks-type`:

```
Object Name = twin-peaks-type
```

```
    Content Pattern = Good Coffee!
    Offset = 0
    Content Type = string
    Open Command (program name) = peakstool
    Icon Location = $OPENWINHOME/include/images/laura.icon
    Icon Mask Location =
$OPENWINHOME/include/images/laura.mask.icon
    Foreground Color = r=91, g= 229, b= 229
    File Pattern = *.pks
```

Only file type name and either a file content pattern (with offset and type) or file pattern are necessary to add a valid entry in the CE database. All other parameters are optional.

2. Create a CE database definition file in ASCII and give it a name. The file illustrated below, `newtype.ascii`, corresponds to our twin-peaks file. Note that the attributes go in the types namespace table, and the file/content patterns go in the files namespace table.

```
# newtype.ascii: A sample ASCII CE database description file
{
NS_NAME=Types
NS_ATTR= ((NS_MANAGER,string, <$CEPATH/tns_mgr.so>))
NS_ENTRIES= (
     (
    (TYPE_NAME,type-id,<twin-peaks-type>)
    (TYPE_OPEN,call,<peakstool>)
    (TYPE_ICON,icon-file,<$OPENWINHOME/include/images/laura.icon>)
    (TYPE_ICON_MASK,icon-file,<$OPENWINHOME/include/images/laura.mask.icon>)
    (TYPE_FGCOLOR,color,<91 229 229>)
    (TYPE_BGCOLOR,color,<91 126 229>)
    (TYPE_FILE_TEMPLATE,string,<peaks.%t>)
     )
    )
}
# Tell CE how to match files of your type. If the file begins with Good_coffee!,
# it's of type twin-peaks-type. The string begins at offset 0 in the file.
{
NS_NAME=Files
NS_ATTR=((NS_MANAGER,junk,<$CEPATH/fns_mgr.so>))
NS_ENTRIES=(
     (
     (FNS_TYPE,refto-Types,<twin-peaks-type>)
     (FNS_MAGIC_OFFSET,str,<0>)
     (FNS_MAGIC_MATCH,str,<Good_coffee!>)
     (FNS_MAGIC_TYPE,str,<string>)
     )
    )
}
```

3. After creating the ASCII description file, execute the `ce_db_merge` command to add the new file type to one of the three CE databases. The *network* database is used in this example, with an ASCII description file called `newtype.ascii`.

```
% ce_db_merge network -from_ascii newtype.ascii
```

4. Note that you can also use `cd_db_build` to add a new file type. Refer to the `man` page for details.

5.4.6 Syntax of ASCII Database Description File

The grammar that describes the Database Description File is given here in Backus-Naur Form (BNF):

```
database::=name_space
      | database name_space
name_space::={ name ns_attrs entries }
name::= NS_NAME = variable
ns_attrs::=NS_ATTR = (av_list)
av_list::=av
        |av_list av
av  ::= (av_name, av_type, av_val)
entries::=NS_ENTRIES = (entry_info_list)
entry_info_list::=entry_ent
      | entry_info_list entry_ent
entry_ent::=( av_list )
av_name::=variable
av_type::=variable
variable::=Id
av_val::=av_token
```

The terminals are:

`Id = a-z, A-Z, 0-9, _, -.`
`NS_NAME, NS_ATTR, NS_ENTRIES,`
`“{”, “}”, “(”, “)”, “,” “=”, Id, and av_token.`

`av_token` can come in two forms:

- It can begin with a “<” and end with a “>” and can have any ASCII character (except a “>”) within it.
- It can begin with one or more digits (which represent a number *n*) followed by zero or more spaces followed by a “<” followed by any *n* characters closed off by a “>.” This is the escape mechanism to allow for arbitrary byte string attributes that could have “>” characters within them.

5.4.7 Binder

Attributes can be added or changed by editing the types namespace file, or by using Binder shown in Figure 5-2. Binder is a DeskSet tool that provides an interactive display of the CE database (refer to the *"OpenWindows Reference Manual"* for operating instructions). With Binder, an advanced user can bind together a file type, its application, a print method, and an icon by setting the desired attributes.

The Binder is also helpful in understanding the Classing Engine, since it interacts directly with the CE database. When you open the Binder, you are given a selection of file types shown in icon form. These correspond to the file types contained in the files namespace table. Once you select a file type, you can view and the attributes in either the *icon properties sheet* or *files property sheet*. Binder allows you to change attributes or create new file types interactively.

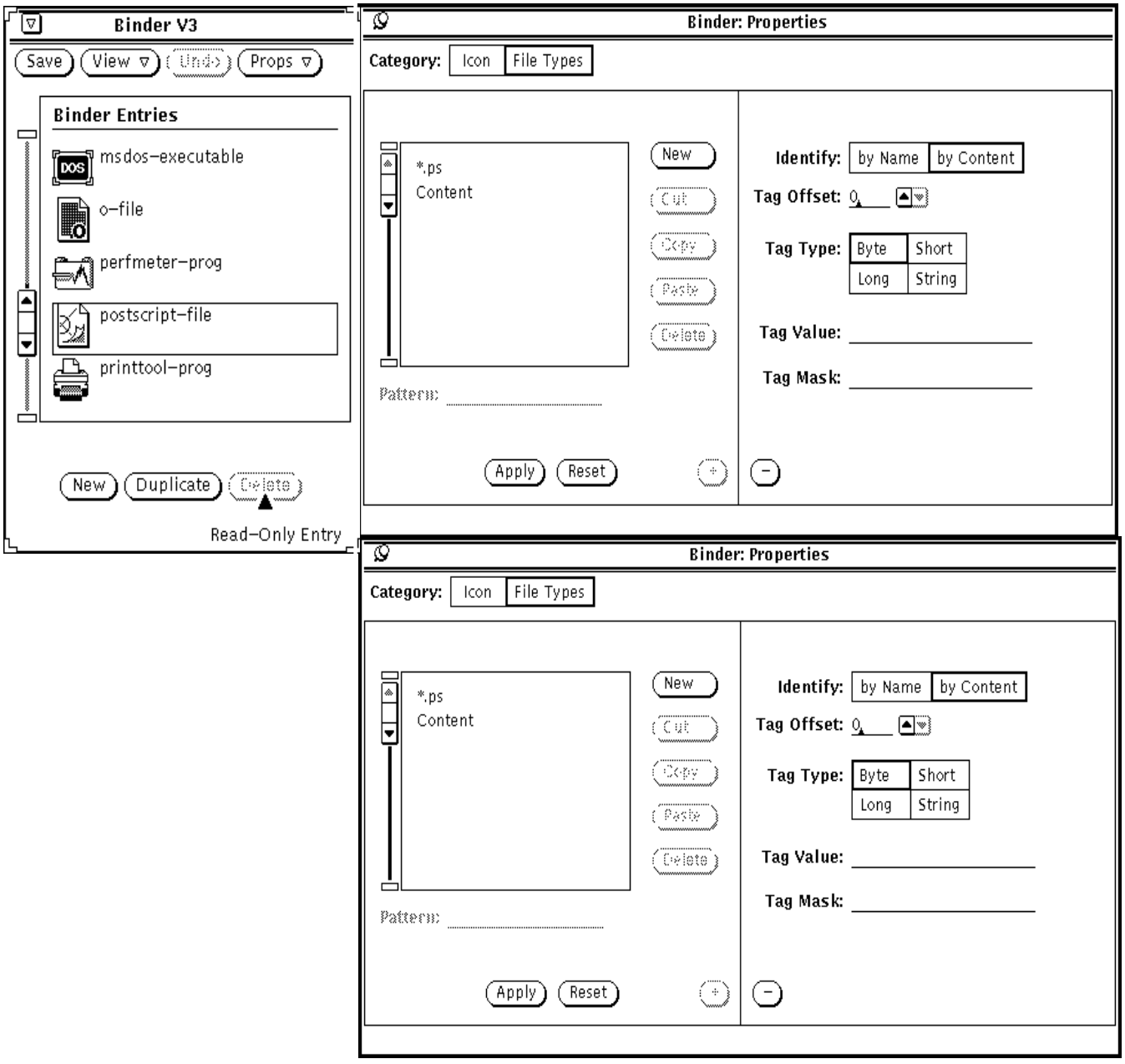


Figure 5-2 Binder—Icon and File Types Property Sheet

5.5 Accessing the Classing Engine Database

The namespace manager, a collection of routines used to query the CE database and perform other database chores provides access to the CE database. These routines are described in Section 5.6, “The Classing Engine API.” It may, however, be helpful to first see two simple programs that use the CE. In addition, there are two Classing Engine examples in `$OPENWINHOME/share/src/dig_samples/ce1` and `ce2`.

5.5.1 Example Program—Querying the Classing Engine Database

The program illustrated below shows how the CE database is queried (`$OPENWINHOME/share/src/dig_samples/ce2/ce_simple.c`). When compiled and executed, the program prompts the user to enter the name of an object (a filename). The program accesses the CE database then displays the file type and the location of its icon file on the screen. The user types “quit” to exit the program. This program must be executed on OpenWindows Version 3.1 or later.

The program is divided into code segments with a detailed explanation of how each code segment works. Table 5-2 shows the variable definitions for the sample program.

Table 5-2 Variable Definitions for ce_simple.c

Type	Variable	Comment
`CE_NAMESPAC`	`f_name_space` `t_name_space`	file namespace table handle types namespace table handle
`CE_ENTRY`	`ftype_ent` `ttype_ent`	file namespace table entry handle types namespace table entry handle

`CE_ATTRIBUTE`	`fns_type`
	`tns_icon`
	`fns_attr`
	`tns_attr`
		file type
		icon filename
		file namespace tbl: file type attr. handle
		types namespace tbl: icon attr. handle

`int`	`argcount` `fd;`	`ce_get_entry` arg counter file descriptor for file to be typed

Table 5-2 Variable Definitions for ce_simple.c

Type	Variable	Comment
char	filename[81] buf[256]	buffer for file name buffer for contents of file
int	bufsize status	return values

5.5.1.1 Preliminary Setup

This first segment includes a short program description and the compile statement. Loading the program requires the Classing Engine and dynamic linking libraries. The include files and variable definitions are:

<stdio.h>for standard io to get input and output
<desktop/ce.h>Needed for the Classing Engine variables
<desktop/ce_err.h>Error return codes from Classing Engine (not used in this program, except for ce_begin)

```
/* ce_simple.c - Simple Classing Engine Example that types a
 * file and determines its icon.
 *
 * cc -g -o ce_sample -I$OPENWINHOME/include -L$OPENWINHOME/lib
 *  ce_simple.c -lce -ldl */

#include <stdio.h>
#include <desktop/ce.h>
#include <desktop/ce_err.h>

/* variable definitions */
CE_NAMESPACEf_name_space, t_name_space;
CE_ENTRY ftype_ent, ttype_ent;
CE_ATTRIBUTEfns_type, tns_icon, fns_attr, tns_attr;
int    argcount;
```

5.5.1.2 Open the CE Database

After declaring the global variables, the program declares the variable definitions for the file to be typed: the file descriptor (fd), the file name (sufficiently long to include any likely path name), and a 256-byte buffer to hold the first characters of the file.

The CE is initialized by `ce_begin()` using the mandatory NULL argument (see the API section for details). The call returns zero if successful; otherwise, it returns a positive integer representing an error code, which is printed to standard error, after which the program exits.

```
main(argc, argv)
int argc;
char *argv[];
{
    int  fd;
    char filename[81];
    char buf[256];
    int  bufsize, status;

    /*  Initialize the Classing Engine.  */

    status = ce_begin( NULL );
    if ( status ) {
           fprintf(stderr, "Error Initializing Classing Engine
                 Database - Error no: %d.\n", status );
           exit( 0 );
    }
```

5.5.1.3 *Setting the Namespace Pointers*

The code segment below sets up the pointers in anticipation of reading the namespace entries for both the files and the types namespaces.

`ce_get_namespace_id("Files")` returns a handle to the files namespace table in `f_name_space`. If the either the file namespace table is not found, or the file namespace manager is not found, the call returns `NULL` and the program exits.

A similar `ce_get_namespace_id()` call and error routine is used for the types namespace table. These calls only need to be done once.

```
/* Read in Namespace Entries. */
f_name_space = ce_get_namespace_id( "Files" );
if ( !f_name_space ) {
        fprintf( stderr, "Cannot find File Namespace\n" );
        ce_end();
        exit(0);
}

t_name_space = ce_get_namespace_id( "Types" );
if ( !t_name_space ) {
        fprintf( stderr, "Cannot find Types namespace\n" );
        ce_end();
        exit(0);
}
```

5.5.1.4 *Retrieve Desired Attribute IDs*

`ce_get_attribute_id(f_name_space, "FNS_TYPE")` returns a handle to the file type attribute in the files namespace table and assigns it to the object ID `fns_attr`. Similarly, the second `ce_get_attribute_id()` returns a handle to the icon filename attribute in the types namespace table and assigns it to the object ID `tns_attr`.

```
/*  Get the attribute ID's that we're interested in  */

    fns_attr = ce_get_attribute_id ( f_name_space, "FNS_TYPE" );

    if (!fns_attr){
            fprintf (stderr, "Cannot find FNS_ATTR in Files\n");
            ce_end();
            exit(0);
    }

    tns_attr = ce_get_attribute_id (t_name_space, "TYPE_ICON");

    if (!tns_attr){
            fprintf (stderr, "Cannot find TYPE_ICON in Types\n");
            ce_end();
            exit(0);
    }
```

5.5.1.5 *Loop to Read File Names*

The next segment starts the loop to read in file names and derive their types. A `while` loop prompts the user for the name of the file that will be tested in the CE. If the user types "quit" the loop is exited (break) and CE database is closed (shown in next segment).

The second `if` statement attempts to open the file. If the file is found, but cannot be opened, an error message is printed and the loop starts again.

If the open is successful, an attempt is made to read the beginning of the file into the 256-byte buffer (to be used later by the CE). If the file is empty or a directory, an error message is printed and the loop starts again.

```
/*  Start loop to read in filenames  */

    while(1) {
        fprintf(stdout, "Filename: ");
        gets(filename);
        if ((strcmp(filename, "quit")) == 0)
            break;

        if ((fd = open (filename, 0)) == -1) {
            fprintf(stderr, "Cannot open: %s\n", filename);
            continue;
        }

        bufsize = read (fd, buf, sizeof (buf));
        if (bufsize <= 0) {
            fprintf(stderr, "Empty file or Directory: %s\n",
                    filename);
            close (fd);
            continue;
        }
```

5.5.1.6 *Get Entry in the Files Namespace*

This next code segment searches through the files namespace table for the file name and/or file content obtained in the previous segment. If a match is found, the file type is returned.

The program calls `ce_get_entry()` to search the files namespace table and return the handle for the matching files namespace table entry. `ce_get_entry()` requires the files namespace ID (`f_name_space`), the number of arguments used to match entries in the files namespace table (3), and the three arguments themselves (the file name entered by the user, the buffer that contains the contents of the previous read, and the length of the buffer).

`ce_get_entry()` returns a handle for the files namespace table entry that matches the filename pattern, contents of the file, or both if both are present. The handle to the entry is assigned to `ftype_ent`. If no entry is found in the files namespace table, a `NULL` is returned, and the `while` loop resumes.

The program then gets the requested attribute value (file type) by calling `ce_get_attribute()`. `ce_get_attribute()` requires the files namespace handle (`f_name_space`), the handle to the entry (`ftype_ent`), and the file type attribute handle (`fns_attr`). After the attribute value is obtained the value is printed.

```
    /*  Get a matching entry in the files namespace    */
        argcount = 3;
        ftype_ent = ce_get_entry (f_name_space, argcount,
                                  filename, buf, bufsize);
        if ( !ftype_ent ) {
            fprintf(stderr, "No match in Files Namespace\n" );
            continue;
        }


fns_type=ce_get_attribute(f_name_space,ftype_ent,fns_attr);
        if (!fns_type)  {
            fprintf(stderr,"No FNS_TYPE for entry in Files
                                  Namespace\n");
          continue;
}
else{o
        fprintf(stdout, "FNS_TYPE = %s\n", fns_type);
```

5.5.1.7 *Get Entry in the Types Namespace*

The final segment of this program retrieves the icon information from the types namespace table. Use `ce_get_entry()` to retrieve a handle for the desired entry. `ce_get_entry()` is passed the types namespace handle (`t_name_space`), the number of arguments used to match entries in the types namespace table (1), and the argument itself (`fns_type`). If a matching entry is not found, an error message is printed and the `while` loop is resumed. If a correct entry is found, the program calls `ce_get_attribute()` with `t_name_space` (types namespace), the handle to the entry (`ttype_ent`), and the icon handle (`tns_attr`) to return the icon filename.

Finally, the icon name (path and name) is printed, the Classing Engine is closed, and the program exits normally.

```
        /* Get a matching entry in the types namespace found from
         * getting type from the files namespace and find icon
         */
        argcount = 1;
        ttype_ent = ce_get_entry ( t_name_space, argcount,
                              fns_type );

        if ( !ttype_ent ) {
            fprintf( stderr, "No match in Types namespace\n" );
            continue;
        }
            tns_icon = ce_get_attribute (t_name_space, ttype_ent,
                                    tns_attr);

        if (!fns_icon)  {
           fprintf(stderr,"No TYPE_ICON in Types Namespace\n");
           continue;
        }
        else
            fprintf(stdout, "TYPE_ICON = %s\n", tns_icon);
    }
}
    cd_end ( );
    exit (0);
}
```

5.5.2 Example Program—CE Mapping Functions

This program, `ce_map1.c`, is located online at `$OPENWINHOME/share/src/dig_samples/ce1`. It demonstrates the use of the CE mapping functions. Refer to the API section that follows for further details.

```
/* ce_map1.c - Classing Engine example that print all the types
 * in the Files and Types namespaces.
 *
 * cc -g -o ce_map1 -I$OPENWINHOME/include -L$OPENWINHOME/lib
 * ce_map1.c -lce -ldl  */

#include <stdio.h>
#include <desktop/ce.h>
#include <desktop/ce_err.h>

/* variable definitions */
CE_NAMESPACE f_name_space, t_name_space;
CE_ENTRYttype_ent;
CE_ATTRIBUTE fns_attr, fns_type;

main(argc, argv)
int argc;
char *argv[];
{
    int  status;
    void *map_func(), *type_map_func();

/* Initialize the Classing Engine. */

    status = ce_begin( NULL );
    if ( status ) {
        fprintf( stderr, "Error Initializing Classing Engine
                    Database - Error no: %d.\n", status );
        exit( 0 );
    }

/* Get Files and Types Entries. */

    f_name_space = ce_get_namespace_id( "Files" );
    if ( !f_name_space ) {
        fprintf( stderr, "Cannot find File Namespace\n" );
        exit( 0 );
```

```
    }
    t_name_space = ce_get_namespace_id( "Types" );
    if ( !t_name_space ) {
        fprintf( stderr, "Cannot find Type Namespace\n" );
        exit( 0 );
    }

    /* Get the FNS_TYPE attribute ID */
    fns_attr = ce_get_attribute_id (f_name_space, "FNS_TYPE");

    if (!fns_attr){
        fprintf (stderr, "No FNS_TYPE in Files Namespace\n");
        ce_end();
        exit (0);
    }

    /* ce_map_through_entries() passes each entry handle and
     * namespace handle to the map_func()
     */
    ce_map_through_entries (f_name_space, map_func, NULL);
    ce_end ();
    exit(0);

}

/* Function to handle each entry as it is passed from the mapping
 * function  */
void
*map_func (fns_handle, ent_handle)
CE_NAMESPACE fns_handle;
CE_ENTRY ent_handle;
{
    int argcount = 1;

    /* Get File type value (FNS_TYPE) and print out */
    fns_type = ce_get_attribute (f_name_space, ent_handle,
                                    fns_attr);

    if (!fns_type)
        return (NULL);
    else
        fprintf (stdout, "FNS_TYPE = %s\n", fns_type);

    /* Get matching entry in the Type namespace */
```

```
    ttype_ent = ce_get_entry (t_name_space, argcount, fns_type);
    if (!ttype_ent){
        fprintf (stderr, "No match in Type namespace\n");
        return (NULL);
    }

    /* Map through all the attributes of the entry and send to
     * type_map_func()
     */
    ce_map_through_attrs (t_name_space, ttype_ent, type_map_func,
                NULL);
    fprintf (stdout, "\n");
    return (NULL);
}
/*
 * Function to print all the Type attributes associated with the File
 * type
 */
void
*type_map_func (tattr_handle, tattr_value, args)
CE_ATTRIBUTE tattr_handle;
char    *tattr_value;
void    *args;
{
    char    *attr_name;

    attr_name = ce_get_attribute_name (tattr_handle);

    if (attr_name)
        fprintf (stdout, "%s = %s\n", attr_name, tattr_value);

    return (NULL);
}
```

5.6 *The Classing Engine API*

The CE API can be called from C, C++, or ANSI C programs. All CE calls have names that begin with `ce_`, with each session begun with a `ce_begin()` and ending with `ce_end()`.

The arguments manipulated by the API are either Classing Engine object handles or client-decipherable argument values and return values. Classing Engine object handles are of type CE_NAMESPACE, CE_ENTRY and CE_ATTRIBUTE and are returned when a client successfully accesses a namespace, an entry, or an attribute. Client-decipherable argument values and return values are expected to be of type void *, if they are pointers, or of type int.

5.6.1 Mapping Functions

The ce_map_through_* functions loop through namespace, entry, and attribute lists, applying a client-supplied function to each member of a list. The previous example shows how the mapping functions work in detail.

5.6.2 Error Reporting

ce_begin returns 0 if it succeeds, otherwise it returns an error number. All Classing Engine ce_get_* calls return NULL if they fail, otherwise they return a valid handle or return value.

The ce_map_through_* calls map through namespaces, entries, or attributes and terminate if they encounter a non-null return value from the map function, and return the non-null value. If the map function returns NULL in every instance, the ce_map_through_* function returns NULL.

5.6.3 Location of Namespace Managers

Every namespace manager library file should be named as the NS_MANAGER namespace attribute. This should be a full pathname with both environment variables and the ‘arch’ command allowed.

If a namespace manager library name is preceded by a $CEPATH, the search rules implied by $CEPATH will be used to search for the namespace manager library.

5.7 Reading from the Classing Engine Database

5.7.1 Initializing the Classing Engine

```
int
ce_begin(void * args);
```

Reads in the CE database and makes CE internal structures suitable for subsequent CE API calls (except for another `ce_begin()`). Subsequent calls to `ce_begin()` will re-read the CE databases. `args`, which is reserved for future use, must be `NULL`.

This call returns 0 if successful. Otherwise, the return codes from this call have the following meanings:

CE_ERROR_READING_DB

This message indicates that an unrecoverable error occurred while reading a CE database. Note that the non-existence of a particular CE database file is not considered an error.

5.7.2 Determining if the Classing Engine Databases Changed

```
int
ce_db_changed();
```

Returns 0 if CE databases have not been changed since the last call to `ce_begin()`. It will return 1 if the databases have been changed.

5.7.3 Closing the Classing Engine

```
int
ce_end();
```

Frees all resources being used by the CE. All CE returned handles and values are invalid after this call. `ce_end()` returns 0 in all cases.

5.7.4 Determining Which Databases are Available

```
int
ce_get_dbs(
    int *num_db,
    char ***db_names
    char ***db_pathnames);
```

Returns a count of the databases in `*num_db`. The names of the databases read in is returned in `db_names`. The pathnames of the databases is returned in `db_pathnames`. There are three possible database names:

```
user the user-level database
systemthe system-level database
networkthe network level database
```

Returns database names and pathnames even if there was no database at a particular pathname. That is, it provides the caller information about where the CE databases would be even if one or more CE databases do not exist.

5.7.5 Accessing a Namespace

```
CE_NAMESPACE
    ce_get_namespace_id(
    char *namespace_name);
```

Returns a handle to a namespace. The namespace handle can be used in all subsequent calls to the CE in this process. This call returns `NULL` if the namespace was not found. This call also returns `NULL` if the namespace manager for the given namespace was not found.

5.7.6 Accessing an Entry in a Namespace Table

```
CE_ENTRY
    ce_get_entry(
    CE_NAMESPACE namespace,
    int argcount,
    void *arg1,
    void *arg2,...,
    void *argN);
```

Searches through a specified namespace table and returns an entry that contains a matching argument. This call requires a handle to a namespace, the number of arguments used to match entries, and the arguments themselves.

5.7.7 Getting an Attribute Handle

```
CE_ATTRIBUTE
    ce_get_attribute_id(
    CE_NAMESPACE namespace,
    char *attr_name);
```

Retrieves a handle to an attribute type within a namespace table. All attributes with the same name, within a namespace, can be retrieved using the same attribute handle. This handle is retrieved with this call.

For example, all attributes named ICON will have the same attribute handle within a single namespace. This call returns `NULL` if the named attribute was not found in this namespace.

5.7.8 Getting an Attribute

```
char
*ce_get_attribute(
    CE_NAMESPACE namespace,
    CE_ENTRY entry,
    CE_ATTRIBUTE attribute);
```

Retrieves the value of an individual attribute. This call returns `NULL` if the attribute could not be found in this entry. It requires a handle for the namespace table (`ce_get_namespace_id()`), entry (`ce_get_entry()`), and attribute (`ce_get_attribute_id()`).

5.7.9 Getting the Size of an Attribute

```
int
ce_get_attribute_size(
    CE_NAMESPACE namespace,
    CE_ENTRY entry,
    CE_ATTRIBUTE attribute);
```

Returns the size (in bytes) of an attribute value. Returns 0 if the attribute was not found in this entry.

5.7.10 Getting an Attribute's Type String

```
char
*ce_get_attribute_type(
    CE_NAMESPACE namespace,
    CE_ENTRY, entry
    CE_ATTRIBUTE attribute);
```

Returns the character string denoting the type of an attribute. Attribute types are not enforced nor understood by the CE. Returns `NULL` if the attribute was not found in this entry.

5.7.11 Getting a Namespace Entry

```
CE_ENTRY
    ce_get_ns_entry(
    CE_NAMESPACE namespace);
```

Returns the namespace entry handle for the specified namespace. Namespaces can have attributes of their own; for example, a range of bytes to read for magic number information in the case of files. Namespace attributes are stored in a namespace entry. This call returns a handle to a namespace's entry. All calls that apply to entries can be made using the returned entry handle. Returns `NULL` if the namespace entry was not found.

5.7.12 Mapping Through Namespaces

```
void
*ce_map_through_namespaces(
    void *(*map_func)(),
    void *args);
```

Maps through all installed namespaces, calls `map_func()` for each namespace, and passes each namespace handle as the first argument to `map_func()` and any other `args` as subsequent arguments. `map_func()` is a user defined function. `args` are optional additional arguments for `map_func()`. If no arguments are to be passed, use `NULL`.

The map will be stopped either when there are no more namespaces or when `map_func` returns a non-`null` value, which will be returned to the caller.

5.7.13 Mapping Through Entries

```
void
*ce_map_through_entries(
    CE_NAMESPACE namespace,
    void *(*map_func)(),
    void *args);
```

Maps through all the entries in a namespace, calls `map_func()` for each entry, and passes the namespace handle as the first argument to `map_func()`, entry handle as the second argument, and any other `map_func()` `args` as subsequent arguments. `map_func()` is a user defined function. `args` are optional additional arguments for `map_func`. If no arguments are to be passed, use `NULL`.

The map will be stopped either when there are no more entries or when `map_func` returns a non-`NULL` value, which will be returned to the caller.

5.7.14 Mapping Through Attributes

```
void
*ce_map_through_attrs(
    CE_NAMESPACE namespace,
    CE_ENTRY entry,
    void *(*map_func)(),
    void *args);
```

Maps through all the attributes in an entry, calls `map_func()` for each attribute, and passes the attribute handle as the first argument to `map_func()`, each attribute value as the second argument, and `args` as the subsequent arguments to `map_func`. `map_func()` is a user defined function. `args` are optional additional arguments for `map_func`. If no arguments are to be passed, use `NULL`.

The function will be stopped either when there are no more attributes or when `map_func` returns a non-null value, which will be returned to the caller.

5.7.15 Mapping Through the Attributes of a Namespace

```
void
*ce_map_through_ns_attrs(
    CE_NAMESPACE namespace,
    void *(*map_func)(),
    void *args);
```

Maps through all the attributes of a namespace, calls `map_func()` for each attribute, and passes each attribute handle as the first argument to `map_func()`, each attribute value as the second argument, and `args` as subsequent arguments. `map_func()` is a user defined function. `args` are optional additional arguments for `map_func`. If no arguments are to be passed, use `NULL`.

The map will be stopped either when there are no more attributes or when `map_func` returns a non-null value, which will be returned to the caller.

5.7.16 Getting the Name of a Namespace

```
char
*ce_get_namespace_name(CE_NAMESPACE namespace);
```

We envision some namespace mapping functions requiring to know the name of a namespace, given a namespace handle. This function will return a namespace name, when passed a namespace handle.

5.7.17 Getting the Name of an Attribute

```
char
*ce_get_attribute_name(CE_ATTRIBUTE attribute);
```

There may be some attribute mapping functions that need to know the name of an attribute when passed a handle to it. This function will return an attribute name, when passed an attribute handle.

5.7.18 Determining Which Database Contains an Entry

```
int
ce_get_entry_db_info(
    CE_NAMESPACE namespace,
```

```
CE_ENTRY entry,
char **name_ptr,
char **path_ptr);
```

Returns the name of the database (either user, system, or network) in which an entry is stored. The name is returned in `*name_ptr` and the pathname of the database in `*path_ptr`. This call returns 0 if it is successful, otherwise it returns `CE_ERR_WRONG_ARGUMENTS`.

5.8 Classing Engine Utility Programs

Two utilities that enable reading and writing the Classing Engine database files to and from an ASCII form are available to allow developers to view the database. The `man` pages for these utilities follows on the next pages.

5.8.1 ce_db_build

The build utility, `ce_db_build`, will generate a readable ASCII file from the CE database, if given the `-from_ascii` argument. The user must also indicate the desired database (user, system, or network) and the filename where the file should be written. This allows a developer to print and peruse a hard copy of the database for familiarization or troubleshooting.

Caution – The `ce_db_build` utility will overwrite an existing CE database if given the `-from_ascii` argument. This will overwrite the existing CE database and replace it with the information from an ASCII file.

An optional argument, `-db_file` *filename*, can be given to generate a CE database file without disturbing the existing CE database files.

5.8.2 cd_db_merge

The merge utility, `ce_db_merge`, permits the merging of an ASCII database description file with an existing CE database file. This utility permits the merging of custom CE entries to the database.

NAME

ce_db_build - build an entire CE database

SYNOPSIS

ce_db_build user|system|network -from_ascii|-to_ascii *filename* \

[**-db_file** *db-filename*]

DESCRIPTION

cd_db_build reads from / writes to the Classing Engine databases and an ASCII description file.

user|system|network indicates which CE database is to be used, ether the user, the system, or the network database.

-from_ascii *filename* indicates that the user wishes to write to the stated CE database from the ASCII file *filename*. The entire CE database will be re-written. This is an all or nothing update of the CE database; that is, effectively the old database is erased and a new one is created based solely on the contents of the ASCII file.

-to_ascii *filename* indicates that the file named *filename* should be written with the ASCII description of the stated CE database. This ASCII description may then be modified and supplied as input to an invocation of **ce_db_build** with the **-from_ascii** argument.

OPTIONS

-db_file should be used in the case that a particular database is to be read from / written to using *db-filename* as the pathname of the CE database, instead of the default database files noted below.

FILES

The Classing Engine uses the following default database files:

```
user~/.cetables/cetables
system/etc/cetables/cetables
network$OPENWINHOME/lib/cetables/cetables
```

EXAMPLES

Create an ascii definition file `newdef` from the existing **user** CE database.

```
ce_db_build user -to_ascii newdef
```

Create the **user** CE database from file `new_db`.

```
ce_db_build user -from_ascii new_db
```

NAME

ce_db_merge - merge a Classing Engine ASCII database description file into the CE database

SYNOPSIS

ce_db_merge user|system|network -from_ascii *filename* \
[**-db_file** *db-filename*]

DESCRIPTION

ce_db_merge will attempt to merge namespace and entry definitions from an ASCII description file into an existing CE database. It will overwrite namespace attributes; that is, namespace attributes from the ASCII file will replace existing namespace attributes.

user|system|network indicates whether the user wants to update the user, the system, or the network CE database.

-from_ascii *filename* indicates that the user wishes to write the stated CE database from the ASCII file *filename*. The named CE database will be updated based on the ASCII description file. Any existing entries that also exist in the ASCII description file will be updated. Any new ASCII descriptors will be entered in the database.

OPTIONS

-db_file should be used in the case that a particular CE database is to be written to, using *db-filename* as the pathname of the CE database, instead of the default database files noted below.

FILES

The Classing Engine uses the following default database files:

database:default location:

```
user~/.cetables/cetables
system/etc/cetables/cetables
network$OPENWINHOME/lib/cetables/cetables
```

EXAMPLES

Merge an ascii definition file `newdef` into the existing user CE database.

```
ce_db_merge user -from_ascii newdef
```

Merge the ascii file `newdef` into the Classing Engine system database at /foo/bar/sysfile.

```
cd_db_merge system -from_ascii newdef -db_file /foo/bar/sysfile
```

5

The ToolTalk Service 6

This chapter describes how the *ToolTalk* 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 chapter.

The ToolTalk service is a network-spanning, interapplication communication service. It provides *multicast* messaging; that is, an application can send 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 chapter 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 *ToolTalk User's Guide*.

6

6.1 The ToolTalk Service Overview

The ToolTalk™ service enables independent applications to communicate with each other without having direct knowledge of each other. Applications create and send ToolTalk messages to communicate with each other. The ToolTalk service receives these messages, determines the recipients, and then delivers the messages to the appropriate applications. See Figure 6-1.

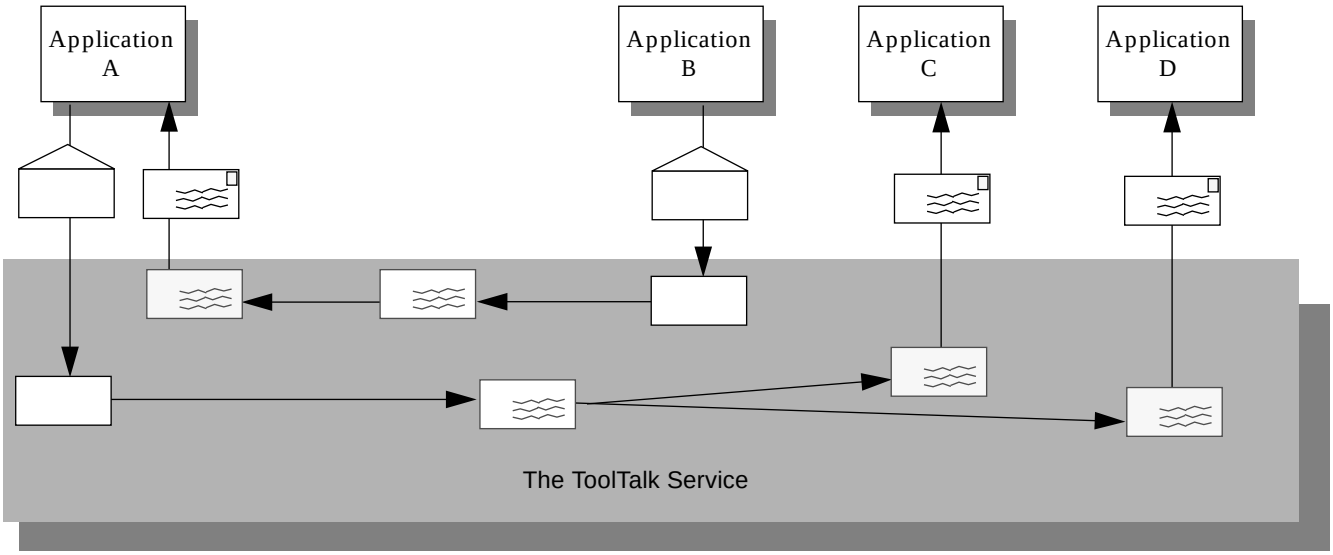


Figure 6-1 Applications Using The ToolTalk Service To Communicate

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

6.2 ToolTalk Scenarios

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

Using the ToolTalk Desktop Services Message Set

The ToolTalk Desktop Services Message Set allows an application to integrate and control other applications without user intervention. A complete and detailed description of the messages in this set is given in Appendix F, "The ToolTalk Desktop Services Message Set."

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

The Smart Desktop

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

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

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

1. **Diane double clicks on the File Manager icon.**
   - The File Manager opens and displays the files in Diane's current directory.
2. **Diane double clicks on an icon for a data file.**
   a. The File Manager requests that the file represented by the icon be displayed. The File Manager encodes the file type in the *display* message.
   b. The ToolTalk session manager matches the pattern in the *display* message to a registered application (in this case, the Icon Editor), and finds an instance of the application running on Diane's desktop.

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

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

3. **Diane manually edits the file.**

Integrated Toolsets

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

1. **George is working on a compound document using his favorite editor.** He decides to change the some of the source code text.
2. **George double clicks on the source code text.**
   a. The Document Editor first determines the text represents source code and then determines what file contains the source code.
   b. The Document Editor sends an *edit* message request, using the file name as a parameter for the message.
   c. The ToolTalk session manager matches the pattern in the *edit* message to a registered application (in this case, the Source Code Editor), and finds an instance of the application running on George's desktop.

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

d. The Source Code Editor accepts the *edit* message request.

e. The Source Code editor determines that the source code file is under configuration control, and sends a message to check out the file.

f. The Source Code Control application accepts the message and creates a read / write copy of the requested file. It then passes the name of the file back to the Source Code Editor.

g. The Source Code Editor opens a window that contains the source file.

3. George edits the source code text.

Using the ToolTalk Document and Media Exchange Message Set

The ToolTalk Document and Media Exchange Message Set is very flexible and robust. A complete and detailed description of the messages in this set is given in Appendix G, "The ToolTalk Document and Media Exchange Message Set."

This section illustrates three applications of the ToolTalk Document and Media Exchange Message Set:

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

Integrating Multimedia Functionality

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

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

1. Daniel opens a document that contains multimedia objects.

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

Adding Multimedia Extensions to Existing Applications

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

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

When Karin’s appointment reminder is executed, the calendar manager will start the audiotool and play Karin’s recorded reminder.

Extending the X Cut and Paste Facility

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

1. Maria opens two documents that are different media types.
2. Maria selects a portion of *Document A* and cuts the portion using the standard *X*-windowing *cut* facility.

3. Maria then pastes the cut portion into *Document B*.
   a. *Document B* negotiates the transfer of the cut data with *Document A*.
   b. If *Document B* does not understand any of the types offered by *Document B*, it requests a *tagged media type*. *Document B* uses the tagged media type to broadcast a ToolTalk message requesting a translation of the media type to a media type it understands.
   c. A registered translation utility accepts the request and returns the translated version of the media type to *Document B*.
   d. The paste of the translated data into *Document B* is performed.

6.3 How Applications Use ToolTalk Messages

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

Sending ToolTalk Messages

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

- Is the message a notice or a request? (that is, should the recipient respond to the message?)
- What interest does the recipient share with the sender? (for example, is the recipient running in a specific user session or interested in a specific file?)

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

Message Patterns

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

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

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

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

Receiving ToolTalk Messages

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

The process retrieves the message and examines its contents.

- If the message contains a notice that an operation has been performed, the process reads the information and then discards the message.

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

6.4 ToolTalk Message Distribution

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

Process-Oriented Messages

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

Object-Oriented Messages

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

Determining Message Delivery

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

Sessions

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

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

Files

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

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

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

Note – The file scoping feature is restricted to NFS and UFS file systems; it does not work, for example, across tmpfs filesystems.

6.5 Modifying Your Application to Use the ToolTalk Service

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

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

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

6

The ToolTalk Service and DeskSet Integration

7

The ToolTalk services allow your application to exchange messages with DeskSet applications. These messages can be commands to start an application, to load a specified file, or to execute a process on a file or object.

This chapter describes:

- DeskSet's ToolTalk message protocol
- Instructions for integrating DeskSet services into your application using ToolTalk
- An example scenario of ToolTalk services used to communicate between two desktop applications

For information about ToolTalk beyond the scope of this manual, see the *ToolTalk User's Guide* and the *ToolTalk Reference Guide*.

7.1 The ToolTalk Messaging Protocol

In the ToolTalk messaging protocol, one process (the *requestor*) requests Desktop services from another process, (the *handler*). For example, the requestor may request that another running tool (the handler) prepare to receive some data from the requestor.

When implemented correctly in tools that run on the Solaris Desktop, the ToolTalk protocol guarantees that any two autonomous processes cooperate effectively in observing and responding to such requests. (Refer to "The ToolTalk Message Sets" section below.)

7.1.1 How the Tooltalk Protocol Works

Under the Tooltalk / Message Alliance protocol, a Tooltalk request is sent by a tool or process that needs a service provided by another tool. If a process is not available to handle the request, an appropriate tool is started and the message is delivered to that tool. The receiving tool (handler) then decides either to service it, or to *reject* or *fail* it.

The message is *failed* if it seems to be improperly formed, or is otherwise not legitimate. The recipient or intended handler sets the error status code and fails the message. The message is *rejected* if it is legitimate and properly formed, but the recipient or intended handler cannot handle it right now. Tooltalk will then look for another handler and will fail the message if one can not be found.

If the message is properly formed and otherwise legitimate, its processing is normally started immediately by the handler (the process that seems to be the intended recipient of the message). While processing, the handler keeps track of two ID codes: the process ID of the requestor, and the message ID. The responder keeps track of these IDs until it is no longer responsible for the message; that is, until it has done something with it, either rejected or performed the requested action. Immediately upon receiving the message and deciding not to reject it, the handler sends a status message, letting the requestor know that the original message is being handled.

Then the handler immediately begins to process the request, which generally requires that it begin to handle data.

7.1.2 New Duties of the Handler

The ToolTalk / Message Alliance protocol gives handlers the responsibility of acquainting themselves with their surroundings when servicing a request. Simply put, the handler must issue requests for any information the need for which is specific to its situation, and thus not included in the initial message. These inquiries are performed with the *Get_** messages.

This permits tools with different implementations to interchangeably implement the same message interface. For a given operation, one brand of handler might need to know what host the requestor is on, while another brand might want to know what the value of *$PATH* is in the requestor's

environment. Instead of enclosing in the initial request the answers to all possible questions a handler might have, handlers are made responsible for making just the inquiries they care about.

7.2 The ToolTalk Message Sets

SunSoft has specified a set of ToolTalk messages commonly used by most Solaris DeskSet applications. (Refer to Appendix F, "The ToolTalk Desktop Services Message Set"). Each application running on the Solaris Desktop, whether a Solaris DeskSet application or a third party application, should be capable of handling these messages.

The following Solaris DeskSet applications support this same core set of ToolTalk messages:

- Mail Tool
- Audiotool
- Text Editor
- Binder
- Color Changer
- Icon Editor

Applications may require more specialized messaging operations for their interaction, and developers can add to the current list of messages. For example, the Solaris Calendar Tool and Mail Tool require three special message types for their interaction. (These messages are discussed below in "Editing with the Open Request.")

There are other standard ToolTalk message sets defined for this release. Developers of multimedia applications will find the Media Exchange message set of interest. (Refer to Appendix G, "The ToolTalk Document and Media Exchange Message Set.")

7.3 Example ToolTalk Messaging Scenarios

Following are two typical scenarios of tools cooperating on the Solaris Desktop, with ToolTalk messages.

7.3.1 Display Request

One scenario arises when a tool (for example, Mailtool) requests that another tool (a text editor) display some of its data. The scenario begins when the Mailtool process (the requestor) sends a message requesting a display of data. The text editor recognizes the message is intended for it, and becomes the handler. It notes the process ID of the requestor, and the message ID of the display request, and immediately sends a message to the requestor indicating that it will service the original message. At this time, it also can send a message to the original requestor asking for additional data regarding the intended display.

When data transmission is complete and the handler has completed the display, it replies to the Display message so the requestor can then clean up any scratch data or other outstanding activities related to the display request.

7.3.2 Edit Request

A different scenario arises when one tool (the requestor) requests that an editor tool (the handler) perform edits on a data file. The sequence might be as follows:

1. **The requestor sends an Edit message to the handler.**
2. **The handler takes note of the process ID of the requestor, and the message ID of the Edit message.**
   The handler then sends back a Status message, taking responsibility for servicing this request.
3. **The handler may send a series of incremental updates with the Deposit messages to the requestor.**
4. **At any time while this edit session is being handled, the requestor can send additional service requests to the same message ID.**
   Such as Iconify, Lower or Raise messages to change the visual appearance of the target on the Desktop.
5. **When all desired data has been transferred, requestor replies to the Edit message with the final, modified, data.**
   The handler saves the file with all changes.

7.3.3 Editing with the Open Request

A different scenario arises when one tool requests that another tool perform edits on some data controlled by the second tool. For example, a Calendar tool might wish to send data to a Mail Tool; namely, a request to mail a reminder to the other party of a planned meeting. Scenarios such as this caused SunSoft to develop three additional messages: *Open*, *Paste*, and *Close*. The sequence might be as follows:

1. **The requestor sends an Open message to the handler.**
   This requests that a data file be made available for edits by the requestor.
2. **The handler takes note of the process ID of the requestor, and replies to the Open request with a bufferID.**
3. **The requestor sends a series of Paste messages to the handler.**
   These correspond to small editing transactions on the target data file.
4. **The requestor sends a Close message to the handler.**
   This indicates that all intended edits are complete.
5. **The handler (Mailtool) can then proceed with its job by displaying the compose window and allowing the user to actually send the email.**

7.4 Example Tooltalk Program with Deskset

The source files for this program can be found online in the Solaris 2.2 distribution at `$OPENWINHOME/share/src/dig_samples/`Tooltalk. This sample code creates a simple yet working editor that can be launched from Mailtool when a user double clicks on a text attachment. To test it out, perform the following steps:

1. **Save your original setup with the command**
   `tt_type_comp -p > save_file`
2. **Run** `make`
3. **Run** `make tooltalk`
4. **Exit mailtool *and* ttsession**
5. **In *one* shelltool:**

a. **Set the environment variable** *XENVIRONMENT* **to be the full path to the** `Resources` **file:**
```
setenv XENVIRONMENT
/usr/openwin/demo/tooltalk/Resources
```

b. **Set your** *path* **environment variable to be this directory.**

c. **Restart ttsession so it has *both* these environment variables.**

6. **Restart mailtool and select a text attachment.**

When your testing is done and you want to go back to your original setup, you can give the command:

```
tt_type_comp save_file
```

or if you had no local changes to begin with, you can give the command:

```
rm -rf ~/.tt
```

7.4.1 Files for this Example

Here is a summary of the files that comprise the example source.

Table 7-1 Overview of the Modules

`olit_tt.c`	These are the routines that support the GUI.
`tt_code.c`	These are the routines that initialize tooltalk and set up the other callbacks.
`tt_callbacks.c`	These are the routines that get called in response to tooltalk and/or GUI events.
`types.file`	Contains tooltalk static ptypes used to identify to Tooltalk that this application should be started for text applications.
`Resources`	Contains X resources for the GUI.
`Makefile`	This builds the application and installs the ptypes.

7.4.2 olit_tt.c

```
/* olit_tt.c  (continued)  */
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <Xol/OpenLook.h>
#include <Xol/TextEdit.h>
#include <Xol/ScrolledWi.h>
#include <Xol/RubberTile.h>
#include <Xol/MenuButton.h>
#include <Xol/OblongButt.h>
#include <Xol/StaticText.h>
#include <Xol/DropTarget.h>
#include <Xol/Exclusives.h>
#include <Xol/RectButton.h>

#defineTEXT_UNMODIFIEDFALSE
#define TEXT_MODIFIEDTRUE

staticWidgettoplevel, base, textedit;
staticchar*saved_text = NULL;

/* callback to hide the window */
hide_frame()
{
    XtPopdown(base);
}

/* callback to expose the window */
show_frame()
{
    XtPopup(base, XtGrabNone);
}

/* callback to place the data in the text window */
display_data(char *text)
{
    XtVaSetValues(textedit,
    XtNsourceType,(XtArgVal)OL_STRING_SOURCE,
    XtNsource,(XtArgVal)text,
    XtNuserData,(XtArgVal)TEXT_UNMODIFIED,
    XtNdisplayPosition,(XtArgVal)0,
    XtNcursorPosition,(XtArgVal)0,
```

```
/* olit_tt.c  (continued)  */
    XtNselectStart,(XtArgVal)0,
    XtNselectEnd,(XtArgVal)0,
    NULL);

    if (saved_text != NULL)
    {
    XtFree(saved_text);
    }

    OlTextEditCopyBuffer((TextEditWidget)textedit, &saved_text);
}

/* callback to get the data from the text window */
Boolean
get_data(char **text, int *len)
{
    if (!OlTextEditCopyBuffer((TextEditWidget)textedit, text))
    {
    OlWarning("getData: error trying to copy textedit buffer\n");
    return(FALSE);
    }

    *len = strlen(*text);
    return(TRUE);

}

/* callback to check if the data in the text window is modified */
Boolean
data_is_modified(void)
{
    inttext_state;

    XtVaGetValues(textedit, XtNuserData, &text_state, NULL);

    return(text_state);
}

/* callback to restore the data of the text window to its last
unmodified
 * state
 */
restore_data()
{
```

```
/* olit_tt.c  (continued)  */
    if(data_is_modified())
    {
    display_data(saved_text);
    }
}

/* callback to clear the data in the text window */
clear_data()
{
    OlTextEditClearBuffer((TextEditWidget)textedit);
}

/* callback to quit this application */
void
quit(void)
{
    quit_tt();
    exit(0);
}

/* Save button callback to save back this data  */
static void
saveTextCB(Widget w, XtPointer client_data, XtPointer callData)
{
    save_tt();
}

/* Quit button callback to quit  */
static void
quitTextCB(Widget w, XtPointer client_data, XtPointer callData)
{
    quit_tt();
    exit(0);
}

/* callback to be called when we get a tooltalk message  */
static void
handleMessageCB(Widget w, XtPointer client_data, XtPointer
callData)
{
    handle_tt_message();
}

/* main initialization routine */
```

```
/* olit_tt.c  (continued)  */
main(int argc, char **argv)
{
    XtAppContext    appContext;
    Widgetcontrol, scrolledwin;
    Widgetsave_btn, quit_btn;
    Widgetblank;
    intfd;

    /* check if we were started by tooltalk */
    check_tt_startup(&argc, &argv);

    /* initialize and build the widgets for the application */
    OlToolkitInitialize((XtPointer) NULL);
    toplevel = XtAppInitialize(&appContext, "AsciiEdit",
           (XrmOptionDescList) NULL,
           0, &argc, argv, (String *) NULL,
           (ArgList) NULL, 0);

    base = XtVaCreateManagedWidget("base",
        rubberTileWidgetClass,
        toplevel,
        NULL);

    control = XtVaCreateManagedWidget("control",
        rubberTileWidgetClass,
        base,
        NULL);

    save_btn = XtVaCreateManagedWidget("save",
        oblongButtonWidgetClass,
        control,
        NULL);

    blank = XtVaCreateManagedWidget("blank",
        staticTextWidgetClass,
        control,
        NULL);

    quit_btn = XtVaCreateManagedWidget("quit",
        oblongButtonWidgetClass,
        control,
        NULL);

    scrolledwin = XtVaCreateManagedWidget("scrolledwin",
```

```
/* olit_tt.c  (continued)  */
        scrolledWindowWidgetClass,
        base,
        NULL);

    textedit = XtVaCreateManagedWidget("textedit",
        textEditWidgetClass,
        scrolledwin,
        NULL);

    /* add the callbacks for the save and quit buttons */
    XtAddCallback(save_btn, XtNselect, saveTextCB, NULL);
    XtAddCallback(quit_btn, XtNselect, quitTextCB, NULL);

    /* start to handle the tooltalk messages and set the callback
       when we get messages in
     */
    fd = start_handling_messages();
    XtAppAddInput(appContext, fd, XtInputReadMask,
handleMessageCB, NULL);

    /* realize the widgets and start the notification process */
    XtRealizeWidget(toplevel);
    XtAppMainLoop(appContext);

}
```

7.4.3 tt_code.c

```
/* tt_code.c  (continued)  */
#include <desktop/tt_c.h>
#include <poll.h>

externTt_callback_actionxinfo_cb(Tt_message m, Tt_pattern p);
externTt_callback_actionlocale_cb(Tt_message m, Tt_pattern p);
externTt_callback_actionhandle_desktop(Tt_message m, Tt_pattern
p);
externTt_callback_actionhandle_display(Tt_message m, Tt_pattern
p);
externTt_callback_actionhandle_edit(Tt_message m, Tt_pattern p);
```

```
/* tt_code.c  (continued)  */

#ifndefDEBUG
#defineDPif(0)
#else
#defineDPif(1)
#endif

Tt_messagesave_msg;/* startup message */

structstartup
{
    char*new[40];/* pointers for new argv */
    intargcount;/* count for new argv pointers */
    intddvl;/* or'ed flag to test for messages */
}newargs;

Tt_status
check_tt_startup(int *argc, char ***argv)
{
    structpollfdmyfds;/* structure for poll command */
    Tt_messagem;/* temporary message */
    Tt_statusstatus;/* message status holder */
    char*toolid;/* procid of tool sending msg */
    int i=0, n, j = 0;/* counters */
    char**tmp;/* tmp location for argv array */
    /* open Tooltalk session and check for errors */
    status = tt_ptr_error(tt_open());
    if(status != TT_OK)
    {
    return(status);
    }

    /* get and save the incoming message */
    save_msg = tt_message_receive();

    /* if Tooltalk started us up */
    if(TT_WRN_START_MESSAGE == tt_message_status(save_msg))
    {
    /* argv initalization */
    newargs.argcount = 0;
    newargs.ddvl = 0;

    /* continue to deliver messages, I am just working on
     * this one for a while
```

```
/* tt_code.c  (continued)  */
            * If you are using a TT library prior to Solaris 2.2
this line
            * is not available and you will not be able to handle
multiple
           * messages until you reply to the message that started
you up.
     */
    /* tt_message_accept(save_msg); */

    /* get the procid of the requesting application */
    toolid = tt_message_sender(save_msg),

    /* request the display, visual & depth */
    m = tt_message_create();
    tt_message_address_set(m, TT_HANDLER);
    tt_message_class_set(m, TT_REQUEST);
    tt_message_scope_set(m, TT_SESSION);
    tt_message_session_set(m,  tt_default_session());
    tt_message_op_set(m, "Get_XInfo");
    tt_message_handler_set(m, toolid);
    tt_message_disposition_set(m, TT_DISCARD);
    tt_message_arg_add(m, TT_OUT, "string", NULL);
    tt_message_arg_add(m, TT_OUT, "string", NULL);
    tt_message_arg_add(m, TT_OUT, "integer", "");
    tt_message_scope_set(m, TT_SESSION);
    tt_message_callback_add(m, xinfo_cb);
    tt_message_user_set(m, 0, &newargs);
    tt_message_send(m);

    /* request the locale info */
    m = tt_message_create();
    tt_message_address_set(m, TT_HANDLER);
    tt_message_class_set(m, TT_REQUEST);
    tt_message_scope_set(m, TT_SESSION);
    tt_message_session_set(m,  tt_default_session());
    tt_message_op_set(m, "Get_Locale");
    tt_message_handler_set(m, toolid);
    tt_message_disposition_set(m, TT_DISCARD);
    tt_message_arg_add(m, TT_OUT, "string", "LC_CTYPE");
    tt_message_arg_add(m, TT_OUT, "string", NULL);
    tt_message_arg_add(m, TT_OUT, "string", "LC_TIME");
    tt_message_arg_add(m, TT_OUT, "string", NULL);
    tt_message_arg_add(m, TT_OUT, "string", "LC_NUMERIC");
    tt_message_arg_add(m, TT_OUT, "string", NULL);
```

```
/* tt_code.c  (continued)  */
    tt_message_arg_add(m, TT_OUT, “string”, “LC_MESSAGES”);
    tt_message_arg_add(m, TT_OUT, “string”, NULL);
    tt_message_scope_set(m, TT_SESSION);
    tt_message_callback_add(m, locale_cb);
    tt_message_user_set(m, 0, &newargs);
    tt_message_send(m);

    /* poll for messages */
    myfds.fd = tt_fd();
    myfds.events = POLLIN;
    while(newargs.ddvl != 3)
    {
        poll(&myfds, 1, -1);
        /* got one */
        m = tt_message_receive();
        if(m)
        {
            /* it’s not one we are looking for */
            tt_message_reject(m);
        }
    }

    /* take all the new args that we got and create a
     * new argv/argc
     */
    n = *argc+newargs.argcount;
    tmp = (char **)malloc((n+1)*sizeof(char *));
    for(i = 0 ; i < *argc; i++)
    {
        tmp[i] = *argv[i];
    }
    for(j = 0; i < n; i++,j++)
    {
        tmp[i] = newargs.new[j];
    }
    tmp[i] = 0;
    *argc = n;
    *argv = tmp;
    }
    return(TT_OK);
}

int
start_handling_messages()
```

```
/* tt_code.c  (continued)  */
{
    /* patterns for desktop messages and display/edit messages */
    Tt_patterndesktop_pat;
    Tt_patterndisplay_pat;
    Tt_patternedit_pat;
    char*op;/* op of saved message */

    /* prepare to handle Desktop messages */
    desktop_pat = tt_pattern_create();
    tt_pattern_op_add(desktop_pat, “Set_Mapped”);
    tt_pattern_op_add(desktop_pat, “Quit”);
    tt_pattern_scope_add(desktop_pat, TT_SESSION);
    tt_pattern_session_add(desktop_pat, tt_default_session());
    tt_pattern_category_set(desktop_pat, TT_HANDLE);
    tt_pattern_callback_add(desktop_pat, handle_desktop);
    tt_pattern_register(desktop_pat);

    /* prepare to handle Display messages */
    display_pat = tt_pattern_create();
    tt_pattern_op_add(display_pat, “Display”);
    tt_pattern_arg_add(display_pat, TT_IN, “ISO_Latin_1”, NULL);
    tt_pattern_scope_add(display_pat, TT_SESSION);
    tt_pattern_session_add(display_pat, tt_default_session());
    tt_pattern_category_set(display_pat, TT_HANDLE);
    tt_pattern_class_add(display_pat, TT_REQUEST);
    tt_pattern_callback_add(display_pat, handle_display);
    tt_pattern_register(display_pat);

    /* prepare to handle Edit messages */
    edit_pat = tt_pattern_create();
    tt_pattern_op_add(edit_pat, “Edit”);
    tt_pattern_arg_add(edit_pat, TT_OUT, “ISO_Latin_1”, NULL);
    tt_pattern_arg_add(edit_pat, TT_INOUT, “ISO_LATIN_1”, NULL);
    tt_pattern_scope_add(edit_pat, TT_SESSION);
    tt_pattern_session_add(edit_pat, tt_default_session());
    tt_pattern_category_set(edit_pat, TT_HANDLE);
    tt_pattern_class_add(edit_pat, TT_REQUEST);
    tt_pattern_callback_add(edit_pat, handle_edit);
    tt_pattern_register(edit_pat);

    /* if we have one we haven’t handled because we were doing
     * the window/application setup, handle it now
     */
    if(save_msg != NULL)
```

```
/* tt_code.c  (continued)  */
    {
    /* find out its type */
    op = tt_message_op(save_msg);

    if(strcmp(op, “Display”) == 0)/* if Display message */
    {
        handle_display(save_msg, display_pat);
    }
    else if(strcmp(op, “Edit”) == 0)/* if Edit message */
    {
        handle_edit(save_msg, edit_pat);;
    }
    else /* all others we don’t want yet */
    {
        tt_message_reject(save_msg);
        tt_message_destroy(save_msg);
    }
    }

    /* return the file descriptor to watch for tooltalk activity
on */
    return(tt_fd());
}
```

7.4.4 tt_callbacks.c

```
/* tt_callbacks.c  (continued)  */
#include <desktop/tt_c.h>
#include <locale.h>

#ifndefDEBUG
#defineDPif(0)
#else
#defineDPif(1)
#endif

#defineTT_DESKTOP_ENOENT1538
#defineTT_DESKTOP_EINVAL1558
#defineTT_DESKTOP_EXITING1697
#defineTT_DESKTOP_CANCELED1698
#defineTT_DESKTOP_UNMODIFIED1699
```

```
/* tt_callbacks.c  (continued)  */

#defineTT_MEDIA_ERR_SIZE1700
#defineTT_MEDIA_ERR_FORMAT1701
#defineTT_MEDIA_NO_CONTENTS1702

structstartup
{
    char*new[40];
    intargcount;
    intddvl;
};

staticchar*argname[] = {“-display”, “-visual”, “-depth”, “-
lc_basiclocale”};

externvoidshow_frame();
externvoidhide_frame();
externvoidquit();

#defineMSG_DISPLAY1
#defineMSG_EDIT2

structcur_msg_state
{
    Tt_messagett_msg;
    int type;
    char*msgid;
} cur_msg_state = { NULL, NULL};

void close_out_old_msg(struct cur_msg_state *);

/* Tooltalk call back to handle the Get_XInfo request */
Tt_callback_action
xinfo_cb(Tt_message m, Tt_pattern p)
{
    char*display;/* Xdisplay */
    char*visual;/* Screen visual */
    intdepth;/* Screen depth */
    intcount;/* Number of args in this msg */
    charbuff[10];/* tmp format buffer */
    structstartup*newargs;/* new argv data structure */

    /* check the state of the returned message */
    switch(tt_message_state(m))
```

```
/* tt_callbacks.c  (continued)  */
    {
    caseTT_HANDLED:/* everything came back ok */

    /* get the structure to save the argv data to */
    newargs = tt_message_user(m, 0);

    /* create the display argument */
    display = tt_message_arg_val(m, 0);
    if(display)
    {
        newargs->new[newargs->argcount++] = “-display”;
        newargs->new[newargs->argcount++] = (char
*)strdup(display);
    }

    /* create the visual argument */
    visual = tt_message_arg_val(m, 1);
    if(visual)
    {
        newargs->new[newargs->argcount++] = “-visual”;
        newargs->new[newargs->argcount++] = (char
*)strdup(visual);
    }

    /* create the depth argument */
    tt_message_arg_ival(m, 2, &depth);
    if(depth > 0)
    {
        newargs->new[newargs->argcount++] = “-depth”;
        sprintf(buff, “%d”, depth);
        newargs->new[newargs->argcount++] = (char *)strdup(buff);
    }
    break;
    default:
    /* just in case something goes wrong */
    DP printf(“xinfo_cb: tt_message_state = %d\n”,
tt_message_state(m));
    }

    /* set the lage so we know we got this one */
    newargs->ddvl |= 1;

    /* clean up the message and return */
    tt_message_destroy(m);
```

```
/* tt_callbacks.c  (continued)  */
    return(TT_CALLBACK_PROCESSED);
}


/* Tooltalk call back to handle the Get_Locale request */
Tt_callback_action
locale_cb(Tt_message m, Tt_pattern p)
{
    inti;   /* counter */
    intcount = 0;/* number of locale values */
    char*cat;/* category */
    char*locale;/* locale */
    structstartup*newargs;/* new argv data structure */

    /* check the state of the returned message */
    switch(tt_message_state(m))
    {
    caseTT_HANDLED:/* everything came back ok */

    /* get the structure to save the argv data to */
    newargs = tt_message_user(m, 0);

    /* get the number of values */
    count = tt_message_args_count(m);

    /* for each set of category/locale */
    for(i = 0; i < count/2; i++)
    {
        /* get the category and locale info */
        cat = tt_message_arg_val(m, i);
        locale = tt_message_arg_val(m, i+1);

        /* if the locale has been set */
        if(locale)
        {
            /* and if the category is one we are
             * interested in add the arguments
             */
            if(strcmp(cat, “LC_CTYPE”) == 0)
            {
            newargs->new[newargs->argcount++] = “-
lc_basiclocale”;
            newargs->new[newargs->argcount++] = (char
*)strdup(locale);
```

```
/* tt_callbacks.c  (continued)  */
           newargs->new[newargs->argcount++] = “-lc_inputlang”;
           newargs->new[newargs->argcount++] = (char
*)strdup(locale);
           }
           else if(strcmp(cat, “LC_TIME”) == 0)
           {
           newargs->new[newargs->argcount++] = “-lc_timeformat”;
           newargs->new[newargs->argcount++] = (char
*)strdup(locale);
           }
           else if(strcmp(cat, “LC_NUMERIC”) == 0)
           {
           newargs->new[newargs->argcount++] = “-lc_numeric”;
           newargs->new[newargs->argcount++] = (char
*)strdup(locale);
           }
           else if(strcmp(cat, “LC_MESSAGES”) == 0)
           {
           newargs->new[newargs->argcount++] = “-
lc_displaylang”;
           newargs->new[newargs->argcount++] = (char
*)strdup(locale);
           }
       }
   }
   /* set the flag so we know we got this one */
   newargs->ddvl |= 2;
   break;
   default:
   DP printf(“locale_cb: tt_message_state = %d\n”,
tt_message_state(m));
   }
   /* clean up the message and return */
   tt_message_destroy(m);
   return(TT_CALLBACK_PROCESSED);
}

/* when the tooltalk file descriptor becomes active */
handle_tt_message()
{
   Tt_messagem;

   /* receive the message */
   m = tt_message_receive();
```

```
/* tt_callbacks.c  (continued)  */
    if(m)
    {
    /* this means none of our callbacks got called so it
     * is not one of ours so throw it back
     */
    tt_message_reject(m);
    }
}

/* when the message is one from the desktop pattern */
Tt_callback_action
handle_desktop(Tt_message m, Tt_pattern p)
{
    intmapped;/* the map operator */
    char*op;/* the type of message */

    /* get the message name */
    op = tt_message_op(m);

    if(strcmp(op, “Set_Mapped”) == 0)/* if it is a mapped message
*/
    {
    /* get the map operator */
    tt_message_arg_ival(m, 0, &mapped);

    /* set it to mapped or not based on the operator */
    if(mapped == 0)
    {
        hide_frame();
    }
    else
    {
        show_frame();
    }
    }
    else if(strcmp(op, “Quit”) == 0)/* if it is the quit message
*/
    {
    /* since this is a simple demo just quit */
    quit();
    }
    return(TT_CALLBACK_PROCESSED);
}
```

```
/* tt_callbacks.c  (continued)  */
/* if we get a Display message */
Tt_callback_action
handle_display(Tt_message m, Tt_pattern p)
{
    char*file;/* not used for this simple app */
    char*media;/* media (for this it SHOULD be ISO_Latin_1 */
    char*type;/* vtype of message arg */
    char*data;/* contents of message */
    int size;/* size of the data */
    char*msgid;/* request's ID */
    char*title;/* request's title */
    int count;/* argument count */
    int i;  /* tmp counter */
    Tt_messagett_msg;/* out going messages */

    /* get the media type (paranoids should check it */
    media = tt_message_arg_type(m, 0);

    /* get the count to see if we have the optional msgID/title */
    count = tt_message_args_count(m);
    DP printf("count = %d\n", count);
    for(i = 1; i < count; i++)
    {
    /* get the msg type */
    type = tt_message_arg_type(m, i);
    DP printf("type = '%s'\n", type);
    if(strcmp(type, "messageID") == 0) /* its optional msgID */
    {
        /* save it */
        msgid = tt_message_arg_val(m, i);
        DP printf("msgid = '%s'\n", msgid);
    }
    else if(strcmp(type, "title") == 0) /* its optional title */
    {
        /* save it */
        title = tt_message_arg_val(m, i);
    }
    }
    if(file = tt_message_file(m)) /* its a file type */
    {
    DP printf("Displaying a file\n");
    /* this type of message is not handled for simplicity sake */
    tt_message_reject(m);
    tt_message_destroy(m);
```

```
/* tt_callbacks.c  (continued)  */
    return(TT_CALLBACK_PROCESSED);
    }
    else/* its a contents type */
    {
    /* get the data */
    tt_message_arg_bval(m, 0, (unsigned char **)&data, &size);
    if(data == NULL || *data == '\0')
    {
        DP printf("data is NULL so fail this message\n");
        /* its not good so fail it */
        tt_message_status_set(m, TT_MEDIA_NO_CONTENTS);
        tt_message_fail(m);
        return(TT_CALLBACK_PROCESSED);
    }
    else
    {
        /* if we have an outstanding msg handle it */
        close_out_old_msg(&cur_msg_state);
        cur_msg_state.tt_msg = m;
        DP printf("Displaying data\n");

        /* display the new data */
        display_data(data);
    }
    }

    /* save the current message state */
    cur_msg_state.type = MSG_DISPLAY;
    cur_msg_state.msgid = msgid;

    /* send back pt-pt a status msg to say all is well */
    tt_msg = tt_message_create();
    tt_message_address_set(tt_msg, TT_HANDLER);
    tt_message_handler_set(tt_msg, tt_message_sender(m));
    tt_message_op_set(tt_msg, "Status");
    tt_message_class_set(tt_msg, TT_NOTICE);
    tt_message_scope_set(tt_msg, TT_SESSION);
    tt_message_session_set(tt_msg,  tt_default_session());
    tt_message_disposition_set(tt_msg, TT_DISCARD);
    tt_message_arg_add(tt_msg, TT_IN, "string", "Request
Received");
    tt_message_arg_add(tt_msg, TT_IN, "string", "ACME Vendor");
    tt_message_arg_add(tt_msg, TT_IN, "string", "Sample Editor");
    tt_message_arg_add(tt_msg, TT_IN, "string", "0.1");
```

```
/* tt_callbacks.c  (continued)  */
    tt_message_arg_add(tt_msg, TT_IN, “messageID”,
cur_msg_state.msgid);
    tt_message_arg_add(tt_msg, TT_IN, “domain”,
setlocale(LC_CTYPE, NULL));
    tt_message_scope_set(tt_msg, TT_SESSION);
    tt_message_send(tt_msg);

    return(TT_CALLBACK_PROCESSED);
}

/* if we get a Edit message */
Tt_callback_action
handle_edit(Tt_message m, Tt_pattern p)
{

    char*file;/* not used for this simple app */
    char*media;/* media (for this it SHOULD be ISO_Latin_1 */
    char*type;/* vtype of message arg */
    char*data;/* contents of message */
    int size;/* size of the data */
    char*msgid;/* request’s ID */
    char*title;/* request’s title */
    int count;/* argument count */
    int i;  /* tmp counter */
    Tt_messagett_msg;/* out going messages */

/*
 *  REJECT MESSAGES
 */
    /* get the media type (paranoids should check it */
    media = tt_message_arg_type(m, 0);

    /* get the count to see if we have the optional msgID/title */
    count = tt_message_args_count(m);
    DP printf(“count = %d\n”, count);
    for(i = 1; i < count; i++)
    {
    /* get the msg type */
    type = tt_message_arg_type(m, i);
    DP printf(“type = ‘%s’\n”, type);
    if(strcmp(type, “messageID”) == 0)
    {
        /* save it */
        msgid = tt_message_arg_val(m, i);
```

```
/* tt_callbacks.c  (continued)  */
        DP printf("msgid = '%s'\n", msgid);
    }
    else if(strcmp(type, "title") == 0)
    {
        /* save it */
        title = tt_message_arg_val(m, i);
    }
    }
    if(file = tt_message_file(m)) /* its a file type */
    {
    /* this type of message is not handled for simplicity sake */
    tt_message_reject(m);
    tt_message_destroy(m);
    return(TT_CALLBACK_PROCESSED);
    }
    else/* its a contents type */
    {
    /* get the data */
    tt_message_arg_bval(m, 0, (unsigned char **)&data, &size);
    close_out_old_msg(&cur_msg_state);
    if(data == NULL || *data == '\0')
    {
        /* its compose time */
        clear_data();
    }
    else
    {
        cur_msg_state.tt_msg = m;
        DP printf("Displaying data\n");
        /* display the new data */
        display_data(data);
    }
    }
    /* save the current message state */
    cur_msg_state.type = MSG_EDIT;
    cur_msg_state.msgid = msgid;

    /* send back pt-pt a status msg to say all is well */
    tt_msg = tt_message_create();
    tt_message_address_set(tt_msg, TT_HANDLER);
    tt_message_handler_set(tt_msg, tt_message_sender(m));
    tt_message_op_set(tt_msg, "Status");
    tt_message_class_set(tt_msg, TT_NOTICE);
    tt_message_scope_set(tt_msg, TT_SESSION);
```

```
/* tt_callbacks.c  (continued)  */
    tt_message_session_set(tt_msg,  tt_default_session());
    tt_message_disposition_set(tt_msg, TT_DISCARD);
    tt_message_arg_add(tt_msg, TT_IN, “string”, “Request
Received”);
    tt_message_arg_add(tt_msg, TT_IN, “string”, “ACME Vendor”);
    tt_message_arg_add(tt_msg, TT_IN, “string”, “Sample Editor”);
    tt_message_arg_add(tt_msg, TT_IN, “string”, “0.1”);
    tt_message_arg_add(tt_msg, TT_IN, “messageID”,
cur_msg_state.msgid);
    tt_message_arg_add(tt_msg, TT_IN, “domain”,
setlocale(LC_CTYPE, NULL));
    tt_message_scope_set(tt_msg, TT_SESSION);
    tt_message_send(tt_msg);

    return(TT_CALLBACK_PROCESSED);
}

void
close_out_old_msg(struct cur_msg_state *old_msg)
{
    if(old_msg->tt_msg == NULL)
    {
    return;
    }
    if(strcmp(tt_message_op(old_msg->tt_msg), “Display”) ==
NULL)
    {
    tt_message_reply(old_msg->tt_msg);
    tt_message_destroy(old_msg->tt_msg);
    }
    else
    {
    /* more work here for save/old data etc. */
    tt_message_reply(old_msg->tt_msg);
    tt_message_destroy(old_msg->tt_msg);
    }
    old_msg->tt_msg = NULL;
}

/* when we want to quit we need to make sure the
 * current message gets handled
 */
void
quit_tt()
```

```
/* tt_callbacks.c  (continued)  */
{
    char*data;/* current data */
    intsize;/* current size */

    /* if no current message we are done */
    if(cur_msg_state.tt_msg == 0)
    {
    return;
    }
    if(cur_msg_state.type == MSG_DISPLAY) /* we're handling a
Display msg */
    {
    /* just reply to it */
    tt_message_reply(cur_msg_state.tt_msg);
    tt_message_destroy(cur_msg_state.tt_msg);
    }
    else if(cur_msg_state.type == MSG_EDIT)/* handling an Edit
msg */
    {
    /* get the correct data */
    if(data_is_modified())
    {
        restore_data();
    }
    get_data(&data, &size);

    /* use that data to reply to the message */
    tt_message_arg_val_set(cur_msg_state.tt_msg, 0, data);
    tt_message_reply(cur_msg_state.tt_msg);
    tt_message_destroy(cur_msg_state.tt_msg);
    }
    cur_msg_state.tt_msg = 0;
}

/* called when a deposit has completed */
Tt_callback_action
save_cb(Tt_message m, Tt_pattern p)
{
    switch(tt_message_state(m))
    {
    caseTT_HANDLED:
    /* show a successfull save was done */
    break;
    caseTT_FAILED:
```

```
/* tt_callbacks.c  (continued)  */
    /* error state */
    break;
    default:
    DP printf("some thing else\n");
    }
}

/* called when you need to save the data back to the
 * calling process. (This only needs to be done if you're
 * handling a Display msg or you want to save an intermediate
 * step in Edit)
 */
save_tt()
{
    Tt_message      tt_msg = 0;
    int             null = 0;
    Tt_status       rc;
    char*toolid;
    char*data;
    int size;

    /* if we really have a tooltalk msg to save */
    if(cur_msg_state.tt_msg != NULL)
    {
    /* create and send a Deposit message */
    tt_msg = tt_message_create();
    tt_message_address_set(tt_msg, TT_PROCEDURE);
    tt_message_class_set(tt_msg, TT_REQUEST);
    tt_message_scope_set(tt_msg, TT_SESSION);
    tt_message_session_set(tt_msg,  tt_default_session());
    tt_message_disposition_set(tt_msg, TT_DISCARD);

    tt_message_address_set(tt_msg, TT_HANDLER);

    toolid = tt_message_sender(cur_msg_state.tt_msg);
    tt_message_handler_set(tt_msg, toolid);
    tt_message_op_set(tt_msg, "Deposit");
    get_data(&data, &size);
    tt_message_arg_add(tt_msg, TT_IN, "ISO_Latin_1", data);

    tt_message_arg_add(tt_msg, TT_IN,
           "messageID", cur_msg_state.msgid);
    tt_message_callback_add(tt_msg, save_cb);
```

```
/* tt_callbacks.c  (continued)  */
    tt_message_send(tt_msg);
    }
}
```

7.4.5 *types.file*

```
ptype Sun_MA_textedit
{
start “olit_tt”;
per_session 5;
handle:
/*
 *
 * Optional extra arguments for these requests:
 *            in    string      title
 *            in    messageID   text
 */
/* content display */
session Display (in ISO_Latin_1 text) => start;
session Display (in ISO_Latin_1 text, in title text) => start;
session Display (in ISO_Latin_1 text, in messageID text) =>
start;
session Display (in ISO_Latin_1 text, in messageID text, in title
text) => start;

/* content compose */
session Edit (out ISO_Latin_1 text) => start;
session Edit (out ISO_Latin_1 text, in title text) => start;
session Edit (out ISO_Latin_1 text, in messageID text) => start;
session Edit (out ISO_Latin_1 text, in messageID text, in title
text) => start;

/* content edits */
session Edit (inout ISO_Latin_1 text) => start;
session Edit (inout ISO_Latin_1 text, in title text) => start;
session Edit (inout ISO_Latin_1 text, in messageID text) =>
start;
session Edit (inout ISO_Latin_1 text, in messageID text, in title
text) => start;

/*
```

```
 * Optional extra arguments for these requests:
 *            in    string    title
 *            in    messageID   text
 */

/* file display */
file Display (in ISO_Latin_1 text) => start;
file Display (in ISO_Latin_1 text, in title text) => start;
file Display (in ISO_Latin_1 text, in messageID text) => start;
file Display (in ISO_Latin_1 text, in messageID text, in title
text) => start;

/* file compose */
file Edit (out ISO_Latin_1 text) => start;
file Edit (out ISO_Latin_1 text, in title text) => start;
file Edit (out ISO_Latin_1 text, in messageID text) => start;
file Edit (out ISO_Latin_1 text, in messageID text, in title text)
=> start;

/* file edits */
file Edit (inout ISO_Latin_1 text) => start;
file Edit (inout ISO_Latin_1 text, in title text) => start;
file Edit (inout ISO_Latin_1 text, in messageID text) => start;
file Edit (inout ISO_Latin_1 text, in messageID text, in title
text) => start;
};
```

7.4.6 Resources

```
AsciiEdit.base.orientation: vertical
AsciiEdit.base.control.orientation: horizontal
AsciiEdit.base.control.weight: 0

AsciiEdit.base.control.save.weight: 0
AsciiEdit.base.control.blank.weight: 1
AsciiEdit.base.control.quit.weight: 0

AsciiEdit.base.scrolledwin.weight: 1

AsciiEdit.base.scrolledwin.forceHorizontalSB: False
AsciiEdit.base.scrolledwin.forceVerticalSB: True
AsciiEdit.base.scrolledwin.textedit.charsVisible: 80
```

```
AsciiEdit.base.orientation: vertical
AsciiEdit.base.scrolledwin.textedit.linesVisible: 60
AsciiEdit.title: Simple Text Editor
AsciiEdit*font: lucidasans
!-----------------------------------------------
AsciiEdit.base.control.save.label: Save
AsciiEdit.base.control.quit.label: Quit
```

7.4.7 *Makefile*

```
#
########################################################################
#
#

SRC += olit_tt.c tt_code.c tt_callbacks.c
HDR += Resources types.file
OBJ += $(SRC:%.c=%.o)

INCLUDE+= -I${OPENWINHOME}/include

#CFLAGS+= -g -DDEBUG
CFLAGS+= ${INCLUDE}

LDFLAGS+= -L${OPENWINHOME}/lib -R${OPENWINHOME}/lib

LIBS+= -lXol -lXt -lX11 -ltt

PROGRAM+= tt_demo

.KEEP_STATE:

$(PROGRAM):$(OBJ)
    $(CC) -o $(PROGRAM) $(OBJ) $(CFLAGS) $(LDFLAGS) $(LIBS)

tooltalk:
    tt_type_comp types.file

clean:
    rm -f core $(PROGRAM) $(OBJ) types.file.deps

.INIT: $(SRC) $(HDR)
```

```
#

# End makefile
########################################################################
########
```

Drag and Drop User Interface Specification

A

A.1 Executive Summary

Drag and drop is a convenient, powerful, general purpose accelerator for transferring data within and between applications. This specification establishes conventions for the user interface of the drag and drop mechanism. It is intended to guide the implementation of drag and drop for OpenWindows Version 3.0.1 or greater, and to guide application developers toward consistent uses of the technique. It does not describe implementation details of the drag and drop mechanism, nor does it describe the API.

This document includes descriptions of:

- the kinds of objects that can be dragged
- the meanings of dropping objects on specific locations (such as on a window header, on a pane in a window, or on a drag and drop target)
- the differences between dragging with and without the Duplicate modifier key held down
- the visual feedback associated with the stages of a drag and drop operation
- how the process of data translation appears to users
- how users can cancel drag operations in progress, and undo completed drag operations[1]

1. In this document, drag and drop operations are sometimes referred to as *drag operations* and *drags*.

- how error messages are presented to users

A.2 Introduction

A.2.1 Classic Examples

Drag and drop is a technique for manipulating data and applications by directly manipulating graphical objects on the display screen. It has become a standard accelerator on the SunSoft desktop for transferring data between applications and for moving data around within an application. A classic example of the use of drag and drop is to move documents around in the directory hierarchy. For example, in File Manager you can move a document into a folder by dragging a document glyph and dropping it on a folder glyph. Technically speaking, the document is the *source object*, and the folder is the *destination object*. First you press and hold the Select mouse button while the pointer is on the document you want to move (the source) and then you drag it onto the folder glyph (the destination) and release the mouse button.

In addition to dragging documents between folders in File Manager, you can also drag documents from a File Manager folder into the wastebasket to delete them, or onto Print Tool to print them. See Figure A-1. Whereas moving documents among folders in File Manager or from a folder to the wastebasket involves only one application (File Manager), dragging documents to the Print Tool involves the transfer of data between two applications, File Manager and

Print Tool. In other words, in the latter case the *source application* and the *destination application* are different, whereas in the former cases they are the same.

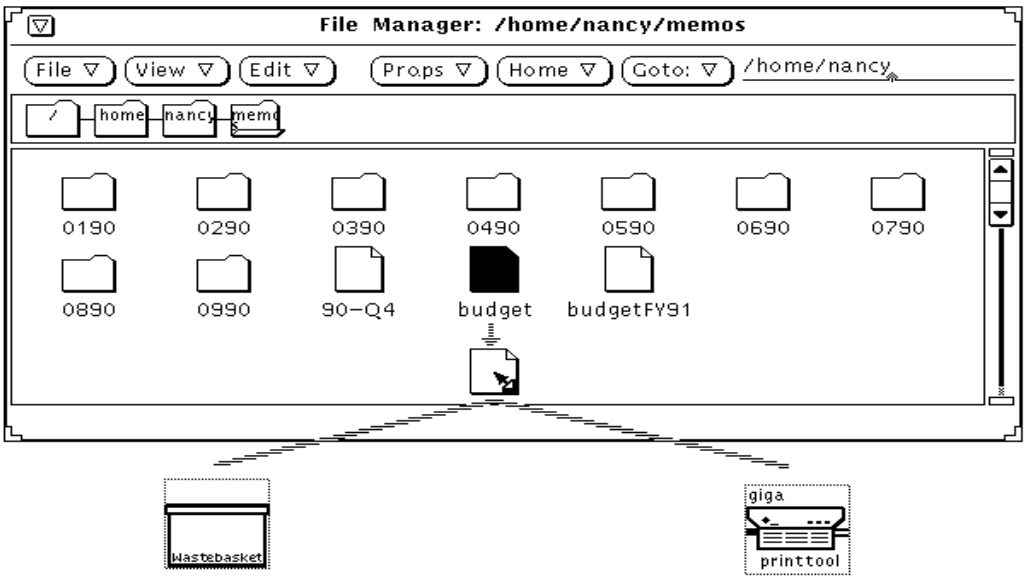


Figure A-1 Dragging File Manager Documents

A.2.2 Drag and Drop as Cut and Paste

Another classic use of drag and drop is as an alternative to the Cut and Paste commands. For example, Text Editor allows you to move selected text from one document to another either by using the Cut and Paste commands, or by using drag and drop. To use drag and drop, you follow these steps. Before you begin, you need to have the two documents loaded into Text Editor, and visible in two windows. Then you select the part of the first document that you want

to move. Next you press the Select mouse button on the selection, and drag it to the location where you want to insert it in the other document. Releasing the mouse button completes the drag and drop operation. See Figure A-2.

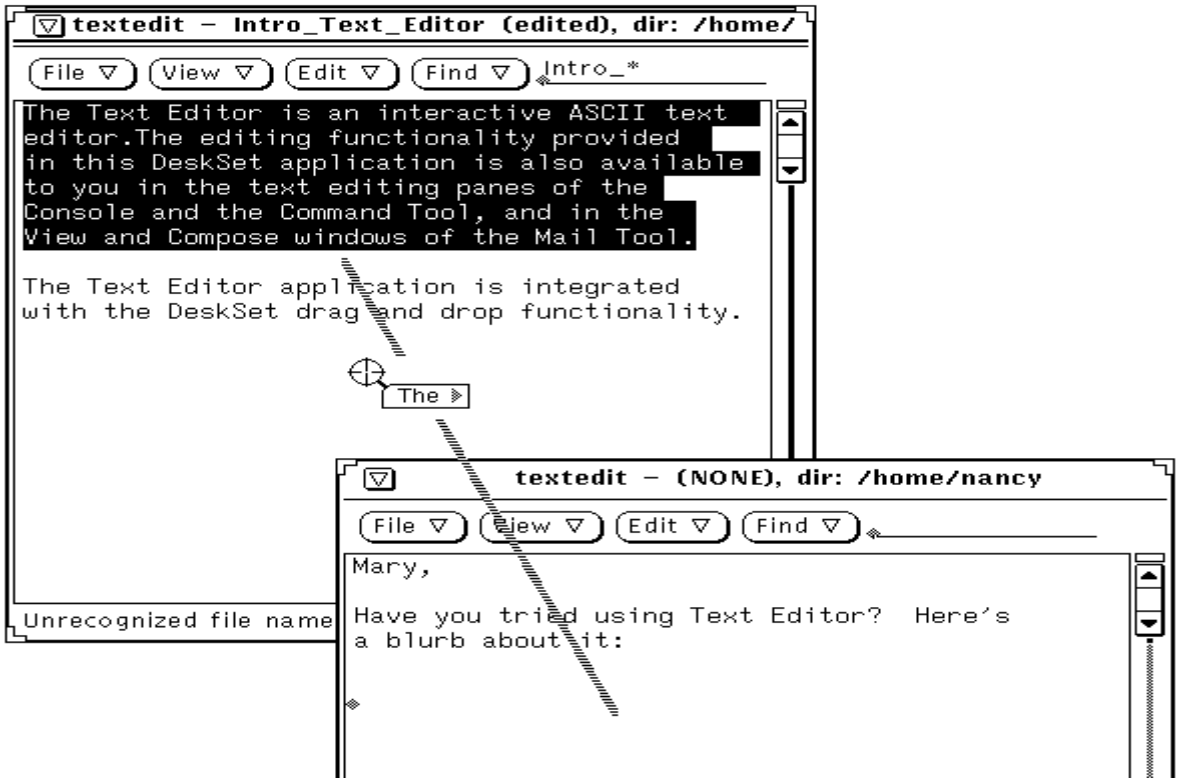


Figure A-2 Dragging Text Between Text Edit Documents

Although drag and drop is often used as an alternative to Cut and Paste, as described above, the two techniques have subtly different effects. First, whereas the Paste command inserts the source object at the caret in the destination document (replacing the selection if there is one), drag and drop inserts it at the hot spot of the pointer. Second, drag and drop does not involve the clipboard, whereas Cut and Paste do. Third, after a drag and drop operation the newly inserted text is selected, whereas after a Paste it is not selected.

A.2.3 To Cut or To Copy?

In the example above, drag and drop was used as an alternative to Cut and Paste. However, if the user had held down the Duplicate modifier key[1] during the drag operation, the source would have been copied. As a result, the drag and drop operation would be analogous to Copy and Paste rather than Cut and Paste.

A.2.4 Where We Are Headed

As these examples indicate, the drag and drop technique is used in a variety of different ways by OPEN LOOK applications. It has proven to be a convenient, powerful, general purpose accelerator for transferring data within and across applications. To exploit the paradigm to its fullest, we need conventions for its use so that applications will use it in similar ways, and consequently, users will know what to expect of it. Conventions are necessary for everything from the meaning of dropping onto an iconified application base window (a *mini-window*), to the feedback that appears when the user attempts a drop in an inappropriate place.

The following sections of this document describe the details of the user interface for drag and drop. They include a formal definition of drag and drop, and a description of the kinds of operations that applications may use drag and drop for. They also specify the meaning of dragging with and without the Duplicate modifier key held down, and the meanings of dropping on specific types of destination objects. Finally, they specify the visual feedback associated with the stages of the drag and drop operation, and describe how a variety of special conditions should be handled.

A.3 Formal Definition

Technically speaking, drag and drop is a gestural technique for manipulating objects,[2] with the following characteristics:

1. The Duplicate modifier key is the Ctrl key by default.

2. The term *object* is used in the loose, generic sense in this document.

- The source is indicated by initiation of the drag operation on an object that typically has been selected, or that will become selected as the drag operation begins.
- The drag operation is initiated by pressing and holding down a mouse button while dragging the mouse. *Dragging* the mouse involves moving it by five pixels or more.[1]
- Following initiation of the drag operation, a drag mode persists in which the user indicates a continuous path from the source to the destination.
- The drag operation terminates when the user releases the mouse button.
- The destination is indicated by the pointer position at the end of the drag operation. More specifically, the destination is indicated by the location of the pointer's hot spot when the user releases the mouse button.

On the SunSoft desktop, drag and drop is defined as an *accelerator*—anything that you can do using drag and drop you should also be able to do in another way, often by selecting commands from menus.

A.4 The Source

Any object that is selectable can potentially be dragged, excluding, of course, selections in most controls (such as exclusive and non-exclusive settings and menus). Typically, the source is a data object, such as a document or a text selection, or a container of data objects, such as a folder.

When the source object is a text selection, a data object, or a container of data objects, the source is the primary selection. After the drag operation has completed, the new object at the destination location is the primary selection. For example, if you drag a text selection from one window to another, after the drag operation the text that has been inserted at the destination location is selected.

1. Users should be able to adjust the *drag threshold* through a workspace property.

A.4.1 Multiple Source Objects

You can drag many different source objects in a single drag operation, provided that you can create a selection that includes all the objects. When the primary selection includes objects in a window, this naturally restricts you to dragging objects only from a single window, since the primary selection cannot span windows.

If the source objects have a natural logical ordering in the source application, the drag operation should preserve the ordering. For example, if the source objects are document glyphs that are displayed in the source application organized by filename, the drag operation should order them alphabetically by filename. However, the destination application should not presume that the source objects it receives are ordered in any way.

A.4.2 Windows as Source Objects

Open windows and iconified windows (that is, mini-windows) also may be source objects in drag and drop operations, however, these drag and drop operations are atypical in several regards.

First, when you drag a window it does not become selected. Because the window is not selected, you can drag it without losing the current primary selection. So, for example, you can make a selection in a window; then drag the window to reposition it; and your selection in the window will still be there.[1]

Second, you can't duplicate a window by holding down the Duplicate key when starting a drag operation. Whenever you drag an open window or a mini-window, the effect of the drag action is to move the window, not to clone it.

Third, when you are dragging an open window or mini-window, the only place you can drop it is onto the workspace. In other words, when a mini-window or an open window is the source, the only *legal* destination is the workspace. Of course, you can drop one window onto another, because our workspace supports overlapping window placement.[2] However, even in this case the destination is the workspace. That is, the overlaid window is not the destination, the workspace is.

1. In the future we may identify other cases where it is useful to be able to drag an object without selecting it. However, presently only open windows and mini-windows can be dragged without being selected.

Fourth, when you drag an open window or a mini-window, the mouse pointer does not change into one of the pointers that are typically used for drag and drop operations (see Figure A-6 on page A-22). The normal pointer was chosen because users are unlikely to view dragging open windows and mini-windows as drag and drop operations. And due to all of the restrictions on dragging open windows and mini-windows, users should *not* view dragging a window as a drag and drop operation.

A.5 The Destination

The destination of a drag operation is determined by the location of the pointer's hot spot at the time the user releases the mouse button. If the source object is a data object or a collection of data objects, the destination may be a data object; a container of data objects such as a directory (i.e., folder); the workspace; a mini-window; or a location in an open window, such as a data pane, a text field, or a *drag and drop target*. Drag and drop targets are a new type of graphical element, whose purpose is to support drag and drop operations. They are described in a following section of this document. If the source object is a mini-window or an open window, the only allowed destination is the workspace.

The legal source and destination combinations are shown below.

Table A-1 Legal combinations of sources and destinations

Source	Data Object/Container	Destination Mini-Window	Open Window	Workspace
Data Object / Container	Yes	Yes	Yes	Yes
Mini-Window	No	No	No	Yes
Open Window	No	No	No	Yes

2. There is one exception to this. You cannot drop a mini-window onto an open window. More specifically, you cannot terminate a drop when the source is a mini-window and the hot spot of the pointer is within the border of an open window. If you attempt such a drop, a Notice will be presented which will tell you the drop operation is not allowed, and the drop will be terminated.

A.5.1 The Drop Method

The primary purpose of the *drop method* is to specify the processing that the source object undergoes at the destination. That is, the drop method determines the *effect* of the drag and drop operation on the destination. The application that owns the graphical element underneath the pointer at the time of a drop (the *destination application*) identifies the *drop method*. The destination may use different drop methods depending on what type of object the source is and depending on where the user dropped the source object.

To ensure conformity among applications and to make it easy for users to guess what the results of a drag operation will be, we have established guidelines for the drop methods applications may use with different parts of the workspace. These guidelines specify which standard elements of the workspace can be used as destinations, and they describe appropriate types of drop methods.

A.5.2 Dropping onto Specific Locations

Text Fields and Text Panes

When you drop a source object onto a single-line text field, multi-line text field, or text pane, the source object should be inserted into the destination text at the position of the pointer's hot spot. If the source object is a text selection, then the text selection is inserted, whereas if the source object is a named object (such as a document), the name of the source object is inserted.

Naturally, source objects that are neither named objects nor text selections cannot be dropped onto text fields.

Non-Text Panes

As is the case with text panes, when the user drops a source object onto a non-text pane, the source is inserted into the destination object. However, whereas in a text pane the source is always inserted at the pointer's hot spot, in a non-text pane the destination application has several options to choose from. The destination application may choose either to insert the source object at the pointer's hot spot, or to:

- insert the source object at a location that depends solely on characteristics of the source object

Calendar Manager processes mail messages dropped on it in this fashion. If you drop a mail message onto an open Calendar Manager window, and the mail message contains a correctly formatted appointment, Calendar Manager will insert the message into the calendar at the appropriate date and time.

- place all source objects at a single location in the destination pane

  For example, imagine a graphical cartridge tape manager that has a data pane that displays glyphs for the files on the tape. Imagine that you can drag a document from File Manager onto the Tape Manager pane to add the document to the tape. Because tapes are sequential media, regardless of where you drop the document in the pane, the document file is added to the end of the tape.

- apply processing specific to the glyph the source object was dropped onto

  For example, File Manager's Path Pane and Folder Pane behave this way. If you drop a document onto a folder in either pane, the document moves into the folder you dropped it on. In contrast, if you drop a document onto the background of the Folder Pane, the document is moved into the directory displayed in the pane.

Scrolling Lists

A scrolling list may accept a source object and insert it as a new entry in the list. The destination application may insert the source into the list either:

- at a location that depends on the pointer's hot spot[1]
- at a location that depends on characteristics of the source object

  For example, in an alphabetical list the source object could be inserted alphabetically by name (or by content if the source object is a text selection).

- at a single fixed location

  For example, when you drop a document onto the Print Tool scrolling list, the document is inserted at the end of the queue.

1. By default, when an object is dropped on a list item, the source object is inserted above the item it was dropped on.

Scrolling lists that allow users to drop items into the list, and/or to drag items already in the list, should have a small icon to the left of each item in the list.

Mini-Windows

When you drop a source object on an iconified application base window (a mini-window), the result of the drop should match the results of a drop method that the open base window supports. If the base window supports more than one drop method, the mini-window should use the drop method that is most closely associated with the base window as a whole. For example, if the application supports a *load* drop method, that drop method should be supported by the mini-window.

Naturally, a mini-window cannot use a drop method that inserts the source object into a data pane at the pointer's hot spot (since the pointer's hot spot is over the mini-window, not over a data pane). For example, a drop onto the Text Editor mini-window *cannot* correspond to a drop onto the Text Editor base window's text pane, because the results of a drop onto the text pane depend on the precise location of the pointer's hot spot in the text pane.[1]

Applications should follow these guidelines in choosing a drop method for a mini-window:

- If the associated open window uses only one drop method, and the drop method does not insert the source object at the pointer's hot spot, then that drop method should also be used for the mini-window.
- If the associated open window supports more than one drop method that does not involve an insertion at the pointer's hot spot, then the mini-window should use the drop method that is most closely associated with the base window as a whole.
- If the associated open window has a drop method which loads the source object into the application (replacing the data there) that drop method should be used for the mini-window.
- If the associated open window allows drops onto its header, dropping onto the header should have the same effect as dropping onto the mini-window.

1. Do not use the caret as a substitute for the pointer hot spot.

Window Backgrounds

Applications may not allow objects to be dropped onto the backgrounds of open windows, except, in some cases, onto the window header.[1] In addition to the header, the background of a window includes:

- the footer
- areas to the left and right of data panes, excluding areas immediately adjacent to scrollbar drag boxes and cables
- the backgrounds of control areas

When an application wants to provide a drop method that there is no obvious receptacle (i.e., destination object) for, the application should use a drag and drop target in a control area. For example, when an application supports a *load* drop method, a drag and drop target should be provided for it. Applications that don't have control areas may use their window headers instead of drag and drop targets.

The Workspace

Dropping an object onto the workspace should not cause the object to transform, such as becoming a mini-window for a running application (which is what File Manager does).

In the future, it may be possible to drop data objects onto the workspace and have them appear to rest on the workspace. However, because there is presently no mechanism in place for displaying data objects on the workspace, this guideline represents a long-term objective. It is included here as a hint to applications about how we intend to use the workspace in the future. Also, it is intended to preclude applications from using drops onto the workspace for other purposes.

1. Drops onto the background are not allowed for two reasons. First, a background should be a neutral zone, which means that it should not have magical properties, such as the ability to accept dropped objects. Second, if a background had a drop method and elements on it had other drop methods, it could be difficult for users to predict the effects of a drop. In cases where a destination doesn't have a clear boundary, as a text field doesn't, it would be hard to know where one destination object ends and the other begins.

Drag and Drop Targets

If an application wants to support a drop method and there is no obvious destination receptacle for the drag and drop operation, it should use a drag and drop target. Such obvious receptacles include text panes, single-line text fields, glyphs displayed in non-text panes, and scrolling lists, among others.

What Drag and Drop Targets Are. A drag and drop target is a rectangular graphical element, typically located in a control area, whose primary purpose is to serve as a destination for drag and drop operations. See Figure A-3.

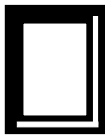


Figure A-3 A Drag and Drop Target

A typical use of a drag and drop target is as a receptacle for dropping an object to be loaded into the destination application. Imagine an editor window that has a drag and drop target in its control area. Imagine further that the data pane is displaying `The_Simpsons`, the file currently loaded in the editor. Imagine that this editor window supports two types of drag and drop operations, one which uses the text pane as a destination, and one which uses the drag and drop target. If you drag a document—call it `Bart`—from File Manager and drop it onto the text pane, `Bart` will be inserted into `The_Simpsons` at the location where you dropped it. If instead of dropping `Bart` onto the text pane, you dropped it on the drag and drop target, `Bart` would replace `The_Simpsons` as the document presently loaded. If you had unsaved edits in `The_Simpsons`, the editor would present a Notice window asking whether you want to save them before closing `The_Simpsons`.

As a secondary feature, some drag and drop targets contain images which can themselves be dragged. That is, the images can be source objects in drag operations. Consider again the Text Editor example above. Imagine that the drag and drop target contains a glyph which can serve as a source object that represents the document presently loaded. For example, if Bart is currently loaded, you can drag the Bart image out of the drag and drop target and onto

the Print Tool to print Bart. This action prints the version of Bart which currently appears in the window (which may contain unsaved edits), and does not unload Bart from the Text Editor. See Figure A-4.

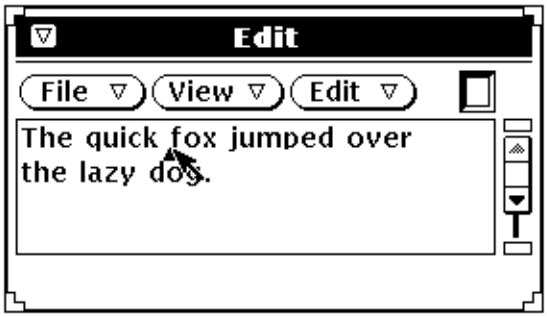


Figure A-4 An Editor Window with a Drag and Drop Target

Windows are not required to include a drag and drop target. When an explicit drag and drop target is used, there should typically be only one per window or, at most, one per control area. Multiple drag and drop targets should be used only when the control areas in which they appear have explicit borders separating one panel from another. The drag and drop target always applies to the entire window or control area in which it appears. In particular, drag and drop targets should not be used to load data into single-line text fields or other individual controls, since these objects can accept drops directly when appropriate and do not require explicit targets of their own.

An explicit drag and drop target may, however, be included as an alternative to the *primary* drop site in a window or control area - provided there is a clear primary drop site that applies to the window or control area as a whole. In such cases, the explicit target will indicate to the user that drops are permitted when the presence of a drop site might not be sufficiently obvious based on the appearance of the drop site itself. An application whose primary drop site is a scrolling list, for example, might choose to provide a drop target to indicate that drops are permitted. In such cases, dropping on the drag and drop target should have the same effect as dropping on the primary drop site. Because it will typically be smaller and thus more difficult for the user to hit, the alternative drop site should only be added if the primary drop site will not be apparent to the user.

Introducing a drag and drop target to an existing application should not cause larger, more accessible drop sites to ignore drop requests. For example, many read-only data viewing applications permit users to drop files onto their data panes for immediate display. This method should continue to be supported for backward compatibility with established conventions even after a drag and drop target is added, because it is easier for the user to point at the data pane than at the drag and drop target and because drops over read-only data panes do not create any ambiguity over whether the data being dropped should replace, or be inserted into, the current data.

Visual Appearance of a Drag and Drop Target. As Figure A-3 on page A-13 shows, a drag and drop target appears to be a box whose open top is flush with the screen. The *sunken* appearance signifies that the object is a receptacle. Drag and drop targets have two standard sizes (see "Drag and Drop Target Engineering Specification" on page A-31). The smaller standard size allows the drag and drop target to be added to the control area that typically appears at the top of an OPEN LOOK base window without increasing the normal height of the control area. Drag and drop targets should use the smaller standard size whenever the control area contains only one row of buttons. The larger standard size provides a target that is somewhat easier to drop on and that is also large enough to permit the display of an application-specified image inside the target's frame. The larger standard size should be used whenever there is sufficient room in the control area containing the drag and drop target. Drag and drop targets can be created in arbitrary sizes if necessary, but the two standard sizes should be used whenever possible, since the size and proportions of the target are important means of identification.

Like other standard OPEN LOOK controls, drag and drop targets should appear only in control areas; they should never appear in data panes. The drag and drop target is typically located in the upper right-hand corner of the control area. When it is located in a control area above a data pane, the drag and drop target should be right-aligned with the right edge of the data pane. If the drag and drop target has a textual label, the label should appear to the left of the drag and drop target in the standard bold font and be followed by a colon. The bottom of the drag and drop target should be positioned slightly below the baseline of the text.

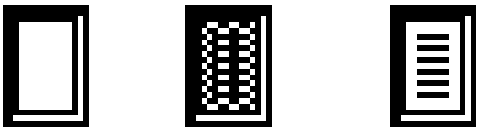
When a window containing a drag and drop target is resizable, the target should be positioned relative to the top and right-hand edges of the window or control area. The drag and drop target should remain in the same relative position whenever the window is resized to ensure its continuous visibility

when the size of the window is reduced. If the application permits its window to be resized such that the drag and drop target would extend into the space occupied by another control, the drag and drop target should appear to overlap the other control.

Drag and Drop Target Content Images - In their normal states, some drag and drop targets are empty, whereas others contain object images. A drag and drop target is ordinarily empty if it doesn't allow objects to be dragged out of it. These *empty* drag and drop targets contain an image only while they are processing dropped objects. This image has a grayed-out, or *busy* appearance. Refer to the center figure in Figure A-5. Once the drop has finished being processed, the object image and the busy feedback vanish and the drag and drop target is empty again.

In contrast, if a drag and drop target allows an object to be dragged out of it, there is an object image inside the drag and drop target at all times that dragging-out is possible. For example, in the editor example described above, the drag and drop target always contains an object image, except when there is no document presently loaded in the window. The default content image is a series of horizontal lines spaced evenly across the receptacle. Applications may choose to provide other, customized images. The object image is overlaid with the standard OPEN LOOK *busy* feedback while a drop is being processed. After the drop completes, the object image resumes its normal appearance. Refer to the right figure in Figure A-5. When a drag and drop target is inactive, the borders of the box as well as its content and label should be dimmed.

Figure A-5 Drop Targets: Empty, Busy, and Containing an Image



Applications may occasionally need to display an object that can serve as the source for a drag operation, but which nevertheless cannot serve as a legal drop site. The standard, "sunken" drag and drop target should not be used in

these cases. The recommended solution is to display a glyph that represents the data and serves as a source for drag operations. This *drag source image* should appear in one of the standard sizes defined for use with the drag and drop target and should be positioned according to the same set of rules. The drag source image should be surrounded by a one-pixel border line that matches the interior dimensions (i.e., the "sunken" rectangle inside the bevel) of an appropriately sized drag and drop target. In color implementations, the border should be a standard "chiseled" line comparable to the border of a control area.

Drag and drop targets (and drag sources) appearing in the smaller standard size should normally use the default content image (see Figure A-5) because the available imaging area is not large enough to make distinctions between images representing different data types practical. If an application-specified content image is required, or if space for a larger target is already available, the drag and drop target should use the larger standard size, which is designed to accommodate a standard (32 x 32) File Manager document glyph for the data in the window. If the content image is used to represent a specific type of data object, it should use the same image that appears in the File Manager for data objects of that type. (The application should query the Classing Engine for the appropriate glyph rather than using a hard-coded image, since users can change the glyph assigned to a particular type of data object at any time.)

A.6 To Copy or Not to Copy?

Drag and drop operations *transfer* an object. Transferring an object may mean relocating a document in the file system; loading a document into an editor; printing a document; inserting a text selection into a document; or any number of other actions determined by the characteristics of the source object, the nature of the destination application, and where in the destination application the source object is dropped.

You can use drag operations simply to transfer a source object, or to duplicate the source object and transfer the duplicate. To support these two forms of drag and drop, there are two types of drag operations which differ in whether the user holds down a modifier key while initiating the drag. The standard form of drag and drop is the unmodified form, where the user does not hold down a modifier key. In this section this form is referred to as *unmodified-drag*. The second form involves holding down the Duplicate modifier while

initiating the drag operation, and is referred to as Duplicate-*drag*. Whereas Duplicate-drag always copies the source object, an unmodified-drag may or may not, depending on what is most intuitive in the current context.[1]

A.6.1 Unmodified Drag

Because users are most likely to learn the unmodified form of drag and drop first, and to use it when they are exploring new drag and drop actions, it has been designed to do the most obvious thing in a given situation. That is, it either does or does not duplicate the source object depending on what the source object is and what the destination is doing with it.

Typically, when a drag operation is relocating data, the source object is not duplicated. For example, when you drag a document from one folder to another in File Manager, it is clear that you meant to reorganize your directories, and the document is not duplicated. Similarly, when you drag a document from a folder onto the wastebasket, it is clear that you meant to relocate the document to the wastebasket, and in this case as well the document is not duplicated. Similarly, when a drag operation loads data into an application, it does not duplicate the data.[2]

In contrast, in many cases when a drag operation carries data from one application to another, the data are transformed, and the user would typically prefer that the operation not affect the original source object. For example, when you drag a document from the File Manager onto the Print Tool, the data are transformed into a hardcopy document, and you are not likely to want to lose the original document. As another example, consider dragging a message from Mail Tool onto Calendar Manager. This action transforms the mail message into a scheduled appointment, assuming the mail message is formatted correctly. In this case as well, it is not clear that a user would be happy to lose the original mail message.

Note that both the source and destination applications play a role in determining whether or not an unmodified drag operates on a duplicate of the source object. The impact of the drag operation on the original source object in the source application depends on where the user drops it. For example,

---

1. Another type of drag operation may be added in the future to support link creation.

2. In fact a copy of the source object is loaded. However, from the user's perspective he or she is operating on the original object, since the original source object's name appears in the destination application header, and by default changes will ultimately be committed to the original object.

imagine that you drag a document from a File Manager folder. The source may or may not eventually be removed from the folder, depending on whether you drop the document on Print Tool, or on the wastebasket, or onto a *load* drag and drop target in Text Editor. When a drop has been completed, the destination application advises the source application as to whether the source object should be removed from its original location.[1]

Naturally, the successful completion of the drop is a necessary condition for removing the source. That is, any time that a drag and drop operation does not complete successfully, the source will not be removed.

A.7 Loading Data

In many cases, using drag and drop to load a file into a destination application is identical, in effect, to loading the file via more conventional means (such as by choosing "Open" from the application's File menu). Specifically:

- If there are any unsaved modifications to the currently-loaded file, a Notice window is presented that gives the user the opportunity to save the changes.
- The currently-loaded file is closed and the new file is loaded.
- The newly-loaded file's name and path are displayed in the window header following the application name.
- After the user modifies the newly-loaded file, he or she can save the changes back to the original file, typically using "Save" in the File menu.

In other cases, a load resulting from a drag and drop operation may differ in one or more regards from loading a file via more conventional means. First, occasionally, such as when the file is dragged from a File Manager running on a remote machine with an inaccessible file system, only the filename (not the path) is accessible. In such cases the window header should display the filename and the name of the application the file came from. Specifically, the window header should display:

Current Application -- *Filename* From *Source Application*

1. Generally, the destination application should recommend that the source be removed only when it is clear that the user intended to relocate the source object. The original source object should be left behind whenever it is not intuitively obvious that the user would expect the operation to remove the source.

For example, if you were to drag a file called Lisa from a File Manager running on a remote machine to a Text Edit application window running on the local machine, the window header should display:

Text Edit -- Lisa From File Manager

Second, occasionally it may not be possible to save the modified document back to the original file. For example, if you had dragged the file from a File Manager running on a remote system, and the remote File Manager application then died, you could not save the file back. In cases such as this the “Save” item in the File menu should be inactive (i.e., grayed out). Users presumably will still be able to use the “Save As” command to save the file to the local file system. They may also be able to restart the remote File Manager and drag the file into it.

Third, unlike the more conventional methods of loading files, when you are loading a file via drag and drop you have the option to duplicate the original source file, and then load the duplicate. If you press the Duplicate key and then perform a drag operation whose drop method is a load, the source object is duplicated in the source application and then the copy is loaded into the destination application. Ordinarily, if the original source object was named “Bart”, the duplicate is called “copy_of_Bart”. However, if the original source object name begins with “copy_of_”, or if there is already a file named “copy_of_Bart” in the current directory, then the duplicated name begins with the string “copy2_of_”, and so forth.

A.8 Data Format Conversion

Frequently the source object is in a data format that differs from the destination’s data format. For example, imagine that you drag some text from Text Editor into a painting application’s window and drop it onto the painting canvas. Whereas Text Editor stores data in ASCII format, the painting application might store it in Postscript format. In order for the painting application to insert the source object into its document, the source must be converted from ASCII to Postscript.

Ideally, when a drop entails data format conversion, the conversion should occur transparently. That is, the user shouldn’t even need to know it happened. However, in some cases the destination application may not be able to decide

how to handle the source data format. In those cases, the destination application should let the user choose among alternative formats listed in a Notice window.

A.9 Handling Multiple Source Objects

Typically, when the destination receives multiple source objects during a single drag and drop operation, it should treat them as independent drag and drop events. However, they may be treated as a single, atomic event in cases where:

- undesirable results would be obtained if all the source objects were not successfully processed by the destination; and
- the destination can reverse the effects of any processing already completed at the time that a failure occurs.

When the destination application treats multiple source objects as independent drag and drop events, it should present a Notice window for each source object that is not successfully processed. The user may terminate processing of all the source objects by pressing the STOP key (once).

A.10 Visual Feedback

A.10.1 While Dragging

When you begin a drag operation, the pointer changes shape and an image of the source object is attached to the pointer to provide feedback that a drag and drop operation has begun. As you drag the pointer over different graphical objects, it changes shape to indicate whether a drop is allowed. In addition, the prospective destination object may animate to provide visual feedback about whether it can accept the source object. For example, a folder might open to show that it can accept the source object.

The visual appearance of the pointer, and the visual image of the source object that the pointer drags along, differ depending on whether the source object is a text selection or not. The two sets of visuals are described in the following sections.

While Dragging Data Objects and Containers

When you begin dragging a data object or a container of data objects, the pointer changes to either the *move* pointer or the *copy* pointer (see Figure A-6). It changes to the *move* pointer if you initiated an unmodified-drag, and to the *copy* pointer if you initiated a Duplicate-drag.

Figure A-6 Normal Pointer, Move Pointer, and Copy Pointer

In addition to changing the shape of the pointer, the source application should attach to the pointer a graphic image to represent the source object. See Figure A-7. The source object image should be a relatively compact representation of the source that fits around the pointer. If the source object itself is a small graphical object, the shape of the image that is dragged should be the same as the shape of the original source object. If the source object has no obvious visual representation or is too large to be previewed in its entirety during the drag operation, an image that is roughly the size of a File Manager glyph should be designed to represent the source object.

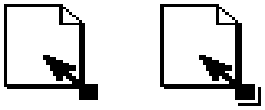


Figure A-7 Move and Copy Pointers with Source Images

The source image should be transparent, and should not have much internal detail, so that users can see through the source image to the object underneath the pointer's hot spot. The *move* or *copy* pointer should be placed on the source

image in a way that: (a) the hot spot of the pointer is as near as possible to the middle of the source image; and (b) the “tail” of the pointer is not obscured by the outline of the source image. When a user drags multiple source objects at once, a representation of the collection of source objects should surround the pointer

Feedback About Prospective Destinations. Whenever possible, when you drag the pointer over a graphical object on the screen during the drag operation, the *drop allowed* or the *drop not allowed* symbol should be added to the pointer. See Figure A-8. To ensure that these symbols will be legible when overlaid onto the image being dragged, an area equal to the size of the symbol should be cleared in the center of the source image before the *drop allowed* or *drop not allowed* symbol is added. The object under the pointer may also change its appearance to indicate that it can accept the source object.

Figure A-8 Drop Allowed and Drop Not Allowed Pointers

In some cases applications may not be able to predict with certainty whether a drop on the destination object will succeed or not. However, applications should try to be as accurate as possible. So long as the feedback is typically accurate, and errors seem like reasonable errors, users will forgive occasional misinformation.

With respect to the *drop allowed* and *drop not allowed* pointers, three areas of the screen are considered *neutral*: the workspace itself, window and control area backgrounds in general, and the background of the data pane (if any) from which the drag operation was initiated (all areas of the data pane except those explicit graphical objects that are either legal or illegal destinations for a drop are considered part of its background). With one exception, the pointer image always changes to the *move* or *copy* pointer while it is over these areas. The exception to the rule is: If an application supports drag and drop actions

within a single window, but not between windows, then the pointer should change to the *drop not allowed* shape as soon as the pointer leaves the source window.

While Dragging a Text Selection

When you begin dragging a text selection, the pointer image changes immediately to the *text move* or *text copy* pointer, depending on whether you are holding down the Duplicate key. These pointers include a rectangular area containing at least the first three characters of the text selection as a "preview" of the data being dragged. If the selection contains more characters than will fit within the rectangle, a dimmed More arrow follows the characters in the rectangle.Text Move and Text Copy Pointers.

Figure A-9 Text Move and Text Copy Pointers

The *text move* and *text copy* pointers in are *neutral* pointers. In other words, they are pointers that appear whenever the pointer's hot spot is not over graphical objects that are either legal or illegal destinations for the drop.

These pointer shapes appear while the pointer is over the workspace, over the backgrounds of windows or control areas, or over objects that don't subscribe to the drag and drop protocol.[1]

When the pointer's hot spot is over a text data pane or a text field, its image changes to one of the *text insert drop allowed* pointers shown in Figure A-10. Specifically, the arrow changes to look like a cross-hair. To facilitate the accurate insertion of the data being dragged into the existing text, the interior

1. These pointers are also used over graphical objects that *do* subscribe to the protocol, but for some reason cannot provide feedback about whether a drop is allowed.

of the cross-hair itself must be transparent. Ideally, the cross-hair pointer should be used only when the pointer is over a drop site whose semantics call for insertion of the data being dragged into the data at the drop site. Note that the change to the *text insert drop allowed* pointer should take place immediately when dragging a text selection in a data pane (unless it is read only), since the text can be dropped anywhere within the same pane.

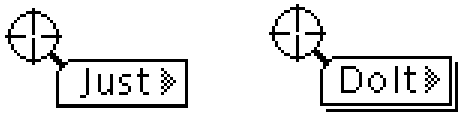


Figure A-10 Text Inset Drop Allowed Pointers

When the pointer is over a drag and drop target (or any other drop site where the drop semantics indicate a replacement of the current data), the pointer should change to one of the *text replace drop allowed* images shown in Figure A-11. Specifically, the arrow in the pointer should change to a bull's-eye that is the same as the *drop allowed* feedback used elsewhere. If an implementation is unable to support different pointer images over explicit drag and drop targets and implicit drop sites (data panes or individual controls), then the *text insert drop allowed* (cross-hair) pointer should be used to provide drop allowed feedback over all legal drop sites (including drag and drop targets) while the text is being dragged.

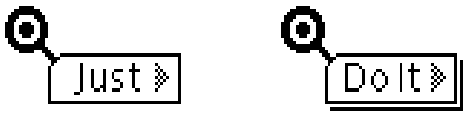


Figure A-11 Text Replace Drop Allowed Pointers

When the pointer is over a graphical object that cannot accept the text selection as a drop, the pointer changes to one of the *text drop not allowed* pointers. See Figure A-12. Specifically, the arrow changes to look like the *drop not allowed* symbol shown in Figure A-8 on page A-23.

Figure A-12 Text Drop Not Allowed Pointers

While Dragging Selected Data other than Text

When dragging a selection containing non-text data that does not itself represent an object or a container, the pointer changes to the *selection move* or *selection copy* pointer, depending on whether you are holding down the Duplicate key. See Figure A-13. These pointers are analogous to the *text move* and *text copy* pointers shown in Figure A-9 on page A-24, but they do not include any “preview” of the data being dragged (that is, there is no indication of the actual contents of the selection). The source application may choose to include an optional glyph within the rectangular area of the pointer to indicate the type of data being dragged (see Figure A-13 on page A-27) but, by default, the rectangle is empty.

As in the case of text selections, the implementation should allow for the use of both *selection insert drop allowed* and *selection replace drop allowed* pointers (see Figure A-13) when it can make the appropriate distinctions between drop sites with insert semantics and those with replace semantics. If the implementation cannot support different pointer images over drag and drop targets and implicit drop sites (data panes or individual controls), then the *selection insert drop allowed* (cross-hair) pointer should be used to provide drop allowed feedback over all legal drop sites (including drag and drop targets).

When dragging selections in data panes containing *sequential* data types (that is, types such as audio that are characterized by a one-dimensional array in which new data displaces existing data at a specific insert point), the pointer image should change immediately to the *selection insert drop allowed* pointer, since an insert point must be specified even in the source data pane.

When dragging selections within data panes containing *non-sequential* data types (that is, types such as structured graphics, in which data can be moved to arbitrary spatial locations and can overlap any data that is already displayed in those locations), the pointer image should change immediately to the *move* or *copy* pointer, but should not display the *drop allowed* or *drop not allowed* symbol while over the original data pane, since any point in the source data pane constitutes a legal drop site. In addition to changing the pointer's shape, the source application should attach a graphical image - as similar as possible to the size and shape of the actual selected data - that provides a WYSIWYG preview of the effect of a drop. If the pointer and image being dragged are moved out of the source data pane, the appropriate *selection drop allowed* pointer should be displayed whenever the hot spot is over any legal drop site, including other compatible data panes or the original source data pane.

When the pointer is over a graphical object that cannot accept the data being dragged, the pointer image changes to one of the *selection drop not allowed* pointers. See Figure A-13. As in the case of text drags, the arrow changes to look like the *drop not allowed* symbol shown in Figure A-8 on page A-23.

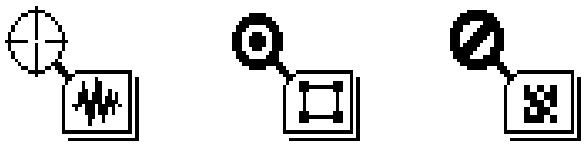


Figure A-13 Drop Feedback Pointers for Non-Text Selections

A.10.2 During the Drop

The destination assumes a *busy* appearance while processing a drop. Once the operation is complete, the destination resumes its normal appearance. If the application can process the drop in the time it would take to post and clear the busy appearance change, then the application may choose not to post the busy appearance.

A.11 Input Focus Management

When you drag an object between windows, the input focus moves to the destination window. Within the destination window, the input focus moves to the element the object was dropped on, assuming it is an element that ordinarily receives the input focus. For example, if you drop a text selection into a text pane, the text pane receives the input focus. If the destination element cannot receive the input focus (as, for example, drag and drop targets can't), the input focus goes to the element in the window that ordinarily receives it when the window receives the input focus.

When you drag an object from one location to another within the same window, the input focus moves to the destination element, assuming it is capable of receiving the input focus. If the destination element cannot receive the input focus, the input focus remains at the source location.

A.12 Error Handling

The best user interfaces are designed for error, and drag and drop is no exception. Errors inevitably occur as a result of user mistakes and as a result of system errors. Drag operations may fail for any of the following reasons:

- The user dropped the source object over a destination that does not subscribe to the drag and drop protocol.
- The source object is of a type the destination cannot accept.

  Although the visual feedback on the pointer is designed to minimize this sort of problem, the feedback is not infallible. And, of course, we can't count on users' actions conforming to the recommendations of the feedback, in any case.
- For some reason the drop operation was aborted.

A drop might be aborted either because of a failure of the transport mechanism used by drag and drop; or because of complications the destination application encounters while processing the drop (such as running out of space in the file system); or for other reasons.

When a drag operation fails, either, but not both, the source application or the destination application presents a Notice window telling the user what has happened. During the drag before the source application has established communication with the destination, the source is responsible for all Notice windows. After communication with the destination has been established, the destination assumes responsibility for Notice windows. The application that does not present the Notice window may choose to display an error message in its base window footer.

In cases of intra-application drags, the application may present a message in a window footer rather than in a Notice window. In either case, the message should explain why the drop failed, and provide constructive guidance to the user about how to avoid failure in the future (if possible).

A.13 Undoing the Effects of Drag and Drop

You can undo the effects of a drag operation by using an Undo menu item or command button, or the Undo function key, in both the source and destination applications (assuming the operation is undo-able).[1] An Undo action in the source application undoes the effect of the drag operation on the source; whereas an Undo action in the destination application undoes the effect there.

A.14 Canceling a Drag Operation in Progress

If you decide to cancel a drag operation while you still have the mouse button held down, you can press the STOP key and then release the mouse button.

1. The Undo function key operates on the window with the keyboard input focus.

A.15 Deviations from the OPEN LOOK Style Guidelines[1]

For the most part, the guidelines in this document extend the guidelines in the *OPEN LOOK Application Style Guidelines*. However, a few of the guidelines described in this document differ from those of the style guide. Applications designed to run on the OpenWindows Environment should follow the guidelines described here rather than those in the style guide.

A summary of the discrepancies follows:

- Differences between unmodified-drag and Duplicate-drag

  According to the style guide, whenever a user initiates a drag operation without holding down the Duplicate key (i.e., Ctrl), the drag operation should be interpreted as a request to relocate the source object. In other words, unmodified-drags should always be interpreted as requests to move the source object from its original location to the destination. In cases where such actions would result in unexpected loss of data, the destination application may refuse to receive data transferred by unmodified-drag operations. The destination application should present a Notice window to allow users either to cancel the drag operation or to change it to a duplicate operation.

  The guidelines described in this document allow applications to interpret unmodified-drag operations as identical to Duplicate-drag operations to prevent unanticipated loss of data.

  Refer to *OPEN LOOK Application Style Guidelines* page 165 and to “To Copy or Not to Copy?” on page A-17 in this document.

- Dropping one mini-window onto another

  The style guide recommends that applications allow users to drop mini-windows onto one another, which should transfer or copy data from the source application to the destination application.

1. Sun Microsystems, Inc. (1990) *OPEN LOOK Graphical User Interface Application Style Guidelines*. Reading, MA: Addison-Wesley Publishing Company, Inc.

This document states that when one mini-window is dropped onto another it is as if they are resting on top of one another on the workspace. That is, when you drop one mini-window onto another, the destination application is not the overlaid mini-window, it is the application that owns the workspace (i.e., the window manager).

Refer to the *OPEN LOOK Application Style Guidelines* and to "The Source" on page A-6 of this document.

- Dropping objects onto window backgrounds

  The style guide says that a user may drop a source object onto the background of a base window, resulting in loading the source into the window (replacing the previous content).

  This document specifies that applications should use drag and drop targets in their control areas for this purpose. If an application doesn't have a control area, and, consequently, doesn't have a place to put a drag and drop target, it may allow drops onto its window header.[1]

  Refer to page 164 in the style guide and to the section called "Drag and Drop Targets" on page A-13 of this document.

A.16 Drag and Drop Target Engineering Specification

Two standard sizes are defined for the drag and drop target. The smaller size (see Figure A-14 on page A-32) is used in the control area above an OPEN LOOK base window when that control area contains only one row of buttons. The larger standard size (see Figure A-15 on page A-33) is designed to display a standard File Manager document glyph within its borders. Its dimensions are the same for all scaling factors because the same set of File Manager glyphs is used in all cases.

Applications can specify the position of the drag and drop target as well as its width and height. The standard "3D" border must always be used, since this is the only aspect of the target itself that directly identifies the drag and drop target as an explicit drop site.

1. Applications that convert from the old policy to the new one should provide constructive guidance in error messages to help users with the transition. Specifically, if a user drops onto the window background, the application should present an explanatory error message that describes that the drag and drop target (or window header) should be used in place of the window background.

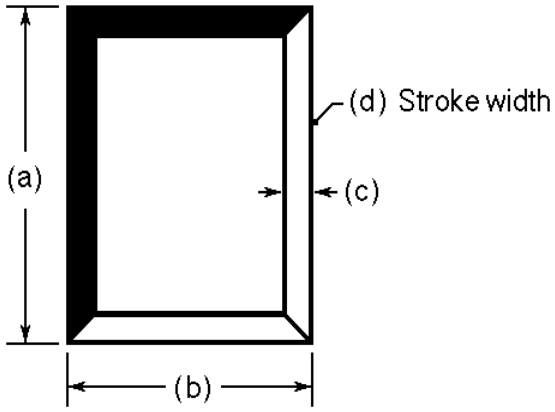


Figure A-14 Small Drag and Drop Target

Table A-2 Dimensions for Small Drag and Drop Target (in points)

	10 pt	**12 pt**	**14 pt**	**19 pt**
(a)	19.0	21.0	23.0	30.0
(b)	14.0	15.5	17.0	22.0
(c)	2.6	3.0	3.4	4.4
(d	0.8	1.0	1.2	1.6

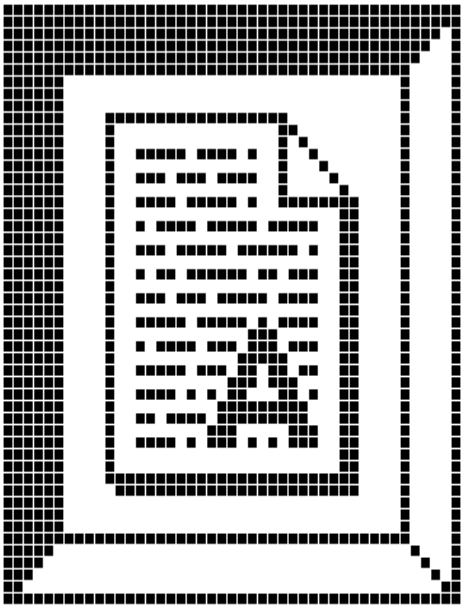


Figure A-15 Large Drag and Drop Target

Table A-3 Dimensions for Large Drag and Drop Target (in pixels)

All
(a)50
(b)45
(c)6
(d)1

Examining a Classing Engine Database

B

You may use the `ce_db_build` utility program to create an ASCII Classing Engine database file that you may print out, or examine using any ASCII editor. Perform the following steps to create the readable CE database file:

1. Determine which CE database you wish to transcribe. You may select either the *user*, the *system*, or the *network* CE database. For the purposes of this sample, let's create an ASCII file of the *system* CE database in a file called `ce.system.txt`.

2. Issue the following command:

   ```
   ce_db_build system -to_ascii ce.database.txt
   ```

3. If you would like to create ASCII readable files for the *user* or *network* CE databases, substitute `user` or `network` for `system` in the above command example

B

Vendor Data Type Registration

If you want your application to be able to exchange data with other applications on the OpenWindows Desktop, you need to make your application's file format, process types, object types, and file attributes public. SunSoft facilitates the dissemination of this information through its Developer Integration Format Registration (DIFR) program.

Registration is required for the three technologies discussed in this guide: drag and drop, Classing Engine, and the ToolTalk service. As data type information is gathered, SunSoft will make it available to other developers.

Call the number below for instructions on receiving your DIFR packet. The packet contains detailed information on the program, as well as the forms you need to register your data types.

Phone number: 1-510-460-3267.

The remainder of this appendix discusses the technical issues of why data types must be registered.

C.1 Drag and Drop Data Types

If an application is to receive a drop from a source application, the source application must send the data in a format readable by the receiving application. For example, if Text Editor wishes to drop data into Mail Tool, Text Editor must be able to provide the data in a format readable by Mail Tool. Conversely, if Mail Tool wishes to drop data into Text Editor, Mail Tool must be able to provide the data in a format Text Editor can read. Although the source

application is responsible for providing data in a format readable by the receiving application, it is important that the receiving application be able to receive data in some of the more common data formats such as ASCII, Sun raster imaging format, or PostScript page description language.

Programmatically, drag and drop handshaking works as follows: (1) data is selected from the source application; (2) the receiving application requests a list of the data formats in which the source application can send the drop; (3) the source application replies with a list of data formats; (4) the receiving application tells the source application which format it would like the data sent; (5) data is transferred.

The SunSoft data type registration program helps standardize the data format names by which applications request data formats from one another. All companies that wish to share their data with other applications are encouraged to register data format names for their application's data files. This name will be used by other applications to reference desired data formats.

Data format names, as well as additional format information, will be made available from SunSoft.

C.2 Classing Engine File Types and Attributes

As described in Chapter 5, "Classing Engine," File Manager and other applications identify a file's type with a unique identifier. Once the file is typed, the file's attributes can be determined.

C.2.1 File Type Identifiers

The file type identifier is used to derive a file's type. File type identifiers can be associated with a filename pattern (such as *.ps or *.wk), a unique string value within the file, or both. If the type-by-pattern method is used, you will need to register a file pattern. If the type-by-content method is used, you will need to register a content pattern, byte offset, and content data type (short, long, string). Two file type registration examples are shown below.

1. Content Value = SSQLReport; Offset = 0; Type = string

   This file type can be identified by the string "SSQLReport" starting at byte zero.

2. Content Value = 0x4d4d002a; Offset = 10; Type = long

This file can be identified by the longword value 0x4d4d002a starting at offset 10 (decimal) in the file.

It is important that your file identifier be unique. The best identifier is a string that identifies your company, the application, and the file type.

C.2.2 File Type Attributes

File type attributes are used to specify the correct method to open or read the file, print the file, and the display icon. The current Classing Engine database attributes are shown below. Refer to Chapter 5, “Classing Engine,” for more information on these attributes.

Table C-1 Classing Engine Database Attributes

Attribute	Description
TYPE_NAME	File type name
TYPE_OPEN	String used to open the file
TYPE_PRINT	String used to print the file
TYPE_ICON	icon file `$OPENWINHOME/include/images/compress.icon`
TYPE_ICON_MASK	icon-file, `<$OPENWINHOME/include/images/doc.mask.icon`
TYPE_FGCOLOR	Icon foreground color
TYPE_BGCOLOR	Icon background color
TYPE_OPEN_TT	ToolTalk identifier used when starting applications
TYPE_FILE_TEMPLATE	Unique filename generated a nd used by the application as a filename.

C.2.3 File Type and Attribute Reference

If you want to peruse a file of previously registered file types, you may view the Classing Engine database by using the program `map1.c` in the Classing Engine chapter, or use the `ce_db_build` utility to create an ASCII description file.

C.3 ToolTalk Type Information

ToolTalk messages can be addressed to a specific application, a type of application, a specific object, or a type of object. To send messages addressed to types of applications or objects, you must know the application's process type or object type. It is the name of an application's process types and object types that need to be registered. For more information on process and object types, see the *ToolTalk User's Guide*.

To provide process type and/or object type information to the ToolTalk service you must supply static type information at installation time by compiling your type file (which puts your type information into the Classing Engine database) and register your process type with the ToolTalk service. When you register your `ptype` with the ToolTalk service, it will read the type information from the Classing Engine database. If you use `otypes`, you need to also create a `ptype` for your application.

C.3.1 Process Type

To send messages to a particular type of application, an application needs to know the process type (ptype) of the receiving application(s). The `ptype` is identified by the process type identifier (ptid). A ptid must be unique for every installation. This identifier cannot be changed at installation time, so it is important that a unique name be chosen. Ideally you will use a name that includes the trademarked name of your product or company, such as `Sun_EditDemo`. Also use upper-case letters to help make your ptid unique. The ptid cannot exceed 32 characters, and should not be one of the reserved identifiers (start, queue, file, session, observe, handle, ptype, otype, per_file, per_session, and opnum.

C.3.2 Object Type

To send messages to a particular type of object, an application needs to know the object type (otype). The otype is identified by the object type identifier (otid). An otid must be unique for every installation. This identifier cannot be changed at installation time, so it is important that a unique name be chosen. It is recommended that the name begin with the ptid of the tool that implements the `otype`; e.g., `Sun_EditDemo_object`. The otid is limited to 64 characters, and should not be one of the reserved identifiers (start, queue, file, session, observe, handle, ptype, otype, per_file, per_session, and opnum).

C.3.3 Ptype and Otype Reference

If you want to peruse a file of previously registered ptypes and otypes, you may view the Classing Engine database by using `tt_type_comp`. Refer to the *ToolTalk User's Guide* for details.

ToolTalk Example Program for XView Toolkit

D

This appendix presents two code examples (`ttreceive.c` and `ttsend.c`) and a header file (`ttdig.h`) which illustrate the use of ToolTalk service with the XView toolkit.

The source and header files, and the Makefile for this example can be found online at `$OPENWINHOME/share/src/dig_samples/tooltalk_simple`.

D

D.1 ttreceive.c

Code Example D-1 ttreceive.c *(1 of 3)*

```
 /* ttreceive - show receiving tooltalk message based on pattern.
 * This simple example program is the counterpart to ttsend.  It registers
 * a pattern which describes the message it is interested in, and then
 * waits for them. */

#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/tt_c.h>

#include “ttdig.h”

Frame base_frame;
Panel_item controls;
Panel_item gauge;

char *my_procid;

voidreceive_tt_message();
voidcreate_ui_components();

void
main(argc, argv)
int argc;
char **argv;
{
  int ttfd;
  Tt_pattern pat;

  /* Initialize XView. */
  xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, 0);
  create_ui_components();

 /* Initialize ToolTalk and obtain file descriptor for incoming messages. */
  my_procid = tt_open();
  ttfd = tt_fd();

  /* Tell XView to call my receive procedure when there are messages. */
  notify_set_input_func(base_frame,
           (Notify_func)receive_tt_message, ttfd);
  /*
   * Create and register the pattern we are interested in.  We are
```

Code Example D-1 ttreceive.c *(2 of 3)*

```
   * registering as an observer; all observers will receive a message
   * destined for them (try a few ttreceives).  If we had registered
   * as a TT_HANDLE, we would be the one to handle the message.
   */
  pat = tt_pattern_create();
  tt_pattern_category_set(pat, TT_OBSERVE);
  tt_pattern_scope_add(pat, TT_SESSION);
  tt_pattern_op_add(pat, RECEIVE_PATTERN);
  tt_pattern_register(pat);

  /* Join the default session to get messages. */
  tt_session_join(tt_default_session());
  xv_main_loop(base_frame);

  /* Clean up ToolTalk on exit. */
  tt_close();
  exit(0);
}

/*
 * receive_tt_message is the procedure that gets called by the XView
 * notifier when my tooltalk file descriptor becomes active with a message.
 */
void
receive_tt_message()
{
  Tt_message msg_in;
  int mark;
  int val_in;

  /*
   * Pull in my message handle.  If it is null, we became active even
   * though there wasn't a real message for us.
   */
  msg_in = tt_message_receive();
  if (msg_in == NULL) return;

  /*
   * Get a storage mark so we can free storage that tt obtains for
   * our message contents.
   */
  mark = tt_mark();
```

Code Example D-1 ttreceive.c *(3 of 3)*

```
  /* If the message pattern matches our interest, fetch the value.  */
  if (0==strcmp(RECEIVE_PATTERN, tt_message_op(msg_in))) {
    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;
}

/*
 * create_ui_components is the procedure called to set up the panel.
 */
void
create_ui_components()
{
  base_frame = xv_create(NULL, FRAME,
            XV_LABEL, “TT Receiver Example”,
            FRAME_SHOW_RESIZE_CORNER, FALSE,
            NULL);
  controls = xv_create(base_frame, PANEL,
              WIN_BORDER, FALSE,
              NULL);
  gauge = xv_create(controls, PANEL_GAUGE,
            PANEL_LABEL_STRING, “Received:”,
            PANEL_MIN_VALUE, RECEIVE_MIN,
            PANEL_MAX_VALUE, RECEIVE_MAX,
            PANEL_SHOW_RANGE, FALSE,
            NULL);
  window_fit(controls);
  window_fit(base_frame);
}
```

D.2 ttsend.c

Code Example D-2 ttsend.c *(1 of 3)*

```
/* ttsend - Demonstrate sending a message with a particular pattern.
 *
 * This simple program is the counterpart to ttreceive.  It sends
```

Code Example D-2 ttsend.c *(2 of 3)*

```
 * a message with a particular pattern that all receivers that are
 * listening will receive.
 */

#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/tt_c.h>

#include “ttdig.h”

Frame base_frame;
Panel_item controls;
Panel_item slider;

char *my_procid;

voidbroadcast_value();
voidcreate_ui_components();

void
main(argc, argv)
int argc;
char **argv;
{

  /* Initialize XView and Tooltalk; enter XView main loop.  */
  xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, 0);
  create_ui_components();
  my_procid = tt_open();
  xv_main_loop(base_frame);

  /* Clean up ToolTalk on exit.  */
  tt_close();
  exit(0);
}


/*
 * broadcast_value is the procedure that gets called when you
 * release the slider.  It gets the current slider
 * value and broadcasts it with ToolTalk.
 */
void
```

Code Example D-2 ttsend.c *(3 of 3)*

```
broadcast_value(item, value, event)
Panel_item item;
int value;
Event *event;
{
  Tt_message msg_out;

  /* Create and send ToolTalk msg.  */
  msg_out = tt_pnotice_create(TT_SESSION, RECEIVE_PATTERN);
  tt_message_arg_add(msg_out, TT_IN, “integer”, NULL);
  tt_message_arg_ival_set(msg_out, 0, value);
  tt_message_send(msg_out);

  /* Destroy the handle since we don’t expect a reply.  */
  tt_message_destroy(msg_out);
}

/*
 * create_ui_components is the procedure called to set up the panel.
 */
void
create_ui_components()
{
  base_frame = xv_create(NULL, FRAME,
            XV_LABEL, “TT Send Example”,
            FRAME_SHOW_RESIZE_CORNER, FALSE,
            NULL);
  controls = xv_create(base_frame, PANEL,
              WIN_BORDER, FALSE,
              NULL);
  slider = xv_create(controls, PANEL_SLIDER,
             PANEL_LABEL_STRING, “Send:”,
             PANEL_SLIDER_END_BOXES, FALSE,
             PANEL_SHOW_RANGE, FALSE,
             PANEL_SHOW_VALUE, FALSE,
             PANEL_MIN_VALUE, RECEIVE_MIN,
             PANEL_MAX_VALUE, RECEIVE_MAX,
             PANEL_TICKS, 0,
             PANEL_NOTIFY_PROC, broadcast_value,
             NULL);
  window_fit(controls);
  window_fit(base_frame);
}
```

D.3 ttdig.h

Code Example D-3 ttdig.h

```
/*
 * RECEIVE_PATTERN is the message identifier for our tooltalk messages.
 * It is prefixed with Sun_ as a simple mechanism to avoid namespace
 * conflicts with other apps in the default session.
 */
#define RECEIVE_PATTERN “Sun_ttexample_pattern”

/*
 * RECEIVE_MIN and _MAX is our slider/gauge range.
 */
#define RECEIVE_MIN 0
#define RECEIVE_MAX 100
```

Drag and Drop Programming Example for X View Toolkit

This example program illustrates an implementation of drag and drop using the XView toolkit. Its source file, icon resources, and Makefile can be found online at `$OPENWINHOME/share/src/dig_samples/`dnd_xview1. When the program is executed, it opens a text window with a drag and drop target. Users may drag any text file from the file manager and drop it on the window's drop site. The text will be displayed in the text pane, and the filename path will appear in the window header. The file can also be imported by entering the filename in the window header.

The document can be exported by dragging the drag and drop target to another window. A portion of the text can be moved by selecting the desired text and dropping it at a specific insert point. Section 3.4, "Drag and Drop Programming Example: OLIT Toolkit" shows a drag and drop example implemented with the OLIT toolkit.

Table E-1 Overview of the Modules

`xview_dnd.c`	Calls `DnD_init()` and `create_user_interface()`

Table E-1 Overview of the Modules

`busy_site.icon`	Contains the data to display the icon on the desktop indicating a busy drop site
`drop_site.icon`	Contains the data to display the icon on the desktop indicating a normal drop site
`Makefile`	Contains the commands to compile and link the example executable

Table E-2 Overview of the Functions

`main()`	Calls `DnD_init()` and `create_user_interface()`
`create_user_interface()`	Creates the frame and text window
`DnD_init()`	Creates drop site & drag object
`drop_proc()`	Event callback procedure; the event procedure for the drop
`get_primary_selection()`	Called from `drop_proc()`; gets the data from the source
`load_file_proc()`	Event callback procedure; callback that displays the file name on the panel

The following sections describe the contents of `xview_dnd.c` in more detail.

E.1 Opening Declarations

The program begins with the compiler include directives and the global object definitions. Note that the header file `dragdrop.h` is only distributed with OpenWindows Version 3.0.1 or later.

Four global data types are defined:

Table E-3 Global Data Type Declarations

`Frame`	Pointer to opaque structure defining the frame
`Panel`	Pointer to opaque structure defining the panel
`Textsw`	Pointer to opaque structure defining the text subwindow
`Panel_item`	Pointer to opaque structure defining a panel item (the `load_file` prompt)

A structure with two members (*atom* and **name*) is declared to store three server atoms. It is initialized with zeros at this time. Actual server atom values will be loaded during the initialization (in the `DnD_init()` function called later). Note that the structure does not have a formal name declared. A formal structure name is not required when a structure is declared if the storage is allocated at the same time.

Here are the contents of the top of `xview_dnd.c`, before the definition of the main function:

```
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/textsw.h>
#include <xview/dragdrop.h>
#include <xview/xv_xrect.h>

/* Global Object definitions
 *
 */

Frame        frame;
Panel        panel;
Textsw       textsw;
Panel_item   load_file;


#define FILE_NAME_ATOM        0
#define _SUN_AVAILABLE_TYPES_ATOM        1
#define XA_STRING_ATOM        2
#define TOTAL_ATOMS                3
```

```
struct
{
    Atomatom;
    char*name;
} atom_list[TOTAL_ATOMS] =
{
    {0,“FILE_NAME”},
    {0,“_SUN_AVAILABLE_TYPES”},
    {0,“XA_STRING”},
};

Drag_dropdrag_object; /* The drag object */
```

E.2 Function: Main()

The program’s main function is straightforward. Two functions without return values, `create_user_interface()` and `DnD_init()`, are declared. The `xv_init()` procedure establishes connections with the X server, initializes the Notifier, reads the `~/.Xdefaults` database and reads any passed arguments.

The program then calls the two functions: `create_user_interface()` creates the frame, the panel, and the text sub window; and `DnD_init()` creates the drop site and the drag object.

Finally, `xv_main_loop()` is executed, telling the Notifier to start dispatching events.

```
main(int argc, char **argv)
{
    Xv_Server   server;

        void        DnD_init(), create_user_interface();

    server = xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

        create_user_interface();


        DnD_init(server);
```

```
    xv_main_loop(frame);
}
```

E.3 Function: *create_user_interface()*

The `create_user_interface()` function, called from `main()`, uses the `xv_create()` procedure call to create the frame, the panel, the panel text (the file name prompt), and the text subwindow where the file is displayed. Notice that the `xv_create()` procedure with the `load_file` handle that creates the Filename: prompt also registers the `load_file_proc()` function with the Notifier.

```
/*
 *  create_user_interface: Create the user interface components.
 */

void
create_user_interface()
{
        Panel_setting    load_file_proc();
    frame = xv_create(NULL,       FRAME,
                        XV_LABEL, "Drag-n-Drop Demo",
                            XV_WIDTH,  600,
                            XV_HEIGHT, 300,
                            FRAME_SHOW_FOOTER, TRUE,
                            NULL);

        panel = xv_create(frame,      PANEL,
                            XV_X,     0,
                            XV_Y,     0,
                            XV_WIDTH, WIN_EXTEND_TO_EDGE,
                            XV_HEIGHT,50,
                            NULL);

        load_file = xv_create(panel,  PANEL_TEXT,
                                PANEL_VALUE_DISPLAY_LENGTH, 45,
                                PANEL_VALUE_STORED_LENGTH, 80,
                                PANEL_LABEL_STRING, "Filename:",
                                PANEL_LAYOUT,PANEL_HORIZONTAL,
                                PANEL_READ_ONLY, FALSE,
                              PANEL_NOTIFY_PROC, load_file_proc,
```

```
                                    NULL);


        textsw = xv_create(frame,         TEXTSW,
                                WIN_BELOW,  panel,
                                XV_WIDTH,   WIN_EXTEND_TO_EDGE,
                                XV_HEIGHT,  WIN_EXTEND_TO_EDGE,
                                NULL);
}
```

E.4 Function: DnD_init()

The `DnD_init()` function creates the drag and drop target as well as the drag and drop target *busy* glyph. The `for` loop gets the three server atoms and loads them into the structure (which was declared in the global object definitions at the beginning of the program). Note that the last `xv_create()` procedure registers the `drop_proc()` function with the Notifier.

```
/*
 *  DnD_init: Create a drop site, and a drag object.
 */
void
DnD_init(Xv_Server server)
{
    Xv_drop_sitedrop_site;
    Xv_opaquedrop_glyph;
    Xv_opaquebusy_glyph;

        static unsigned short   drop_icon[] = {
#include “drop_site.icon”
        };
        static unsigned short   busy_icon[] = {
#include “busy_site.icon”
        };

    inti;

    for(i = 0; i < TOTAL_ATOMS; i++)
    {
    atom_list[i].atom = xv_get(server,
            SERVER_ATOM,
            atom_list[i].name);
```

```
    }

    atom_list[XA_STRING_ATOM].atom = XA_STRING;

    drag_object = xv_create(panel, DRAGDROP, NULL);

        drop_glyph = xv_create(XV_NULL, SERVER_IMAGE,
                SERVER_IMAGE_BITS, drop_icon,
                SERVER_IMAGE_DEPTH, 1,
                XV_WIDTH, 32,
                XV_HEIGHT, 32,
                NULL);

        busy_glyph = xv_create(XV_NULL, SERVER_IMAGE,
                SERVER_IMAGE_BITS, busy_icon,
                SERVER_IMAGE_DEPTH, 1,
                XV_WIDTH, 32,
                XV_HEIGHT, 32,
                NULL);

    xv_create(panel,                          PANEL_DROP_TARGET,
        PANEL_DROP_DND,drag_object,
        PANEL_DROP_GLYPH,drop_glyph,
        PANEL_DROP_BUSY_GLYPH,busy_glyph,
        PANEL_NOTIFY_PROC,drop_proc,
        PANEL_DROP_FULL,TRUE,
        NULL);
}
```

E.5 Function: drop_proc()

The `drop_proc()` routine is the event callback procedure that initiates the drag and drop operation. If the operation is a drag, the case statement handles it either as a move or a copy. If the operation is a drag from the drag and drop target, the third case statement (`LOC_DRAG`) is used. This code determines whether the filename or the data string is passed.

This first `xv_create()` associates the selection targets with a corresponding selection atom. The second `xv_create()` will determine if a filename is being passed, and the third, if the text string is to be passed. In addition, the message

"Start dragging" is printed in the lower left of the frame. Notice that the argument lists of `xv_create()` and `xv_set()` are variable length and must be terminated with NULL statements.

The `create_user_interface()` function, described earlier, registers `drop_proc()` with the Notifier.

```
/* drop_proc: Setup the drag operation and handle the drop.
 *
 */
void
drop_proc(Xv_opaque item, unsigned int value, Event *event)
{
    long    length;
    int     format;
    char    *sel_string;
    char    *string;
    Selection_requestorsel_req;
    char    *buff;
    int     txt_len;
    Atom    list[4];
       void           get_primary_selection(Selection_requestor
sel_req);

    sel_req = xv_get(item, PANEL_DROP_SEL_REQ);


    printf("sel_req = %X\n", sel_req);
    switch(event_action(event))
    {
    caseACTION_DRAG_MOVE:/* they are moving the object */
    printf("drag move\n");
    get_primary_selection(sel_req);
    break;

    caseACTION_DRAG_COPY:/* they are copying the object */
    printf("drag copy\n");
    get_primary_selection(sel_req);
    break;

    caseLOC_DRAG:
    list[0] = atom_list[_SUN_AVAILABLE_TYPES_ATOM].atom;
    list[1] = atom_list[FILE_NAME_ATOM].atom;
    list[2] = atom_list[XA_STRING_ATOM].atom;
    list[3] = NULL;
```

```
    xv_create(drag_object, SELECTION_ITEM,
        SEL_DATA, list,
        SEL_FORMAT,32,
        SEL_LENGTH,4,
        SEL_TYPE,    atom_list[_SUN_AVAILABLE_TYPES_ATOM].atom,
        SEL_OWN,TRUE,
        NULL);

    string = (char *)xv_get(load_file, PANEL_VALUE);

    xv_create(drag_object, SELECTION_ITEM,
        SEL_DATA, string,
        SEL_FORMAT,8,
        SEL_LENGTH,strlen(string),
        SEL_TYPE,atom_list[FILE_NAME_ATOM].atom,
        SEL_OWN,TRUE,
        NULL);

    txt_len = xv_get(textsw, TEXTSW_LENGTH) + 1;
    string = (char *)calloc(txt_len,1);
    xv_get(textsw,
        TEXTSW_CONTENTS, 0, string, txt_len);

    xv_create(drag_object, SELECTION_ITEM,
        SEL_DATA, string,
        SEL_FORMAT,8,
        SEL_LENGTH,strlen(string),
        SEL_TYPE,atom_list[XA_STRING_ATOM].atom,
        SEL_OWN,TRUE,
        NULL);

    xv_set(frame,
        FRAME_LEFT_FOOTER,“Start draging”,
        NULL);
    printf(“Start draging\n”);
    break;
    default:
    printf(“unknown event %d\n”, event_action(event));
    }

}
```

E.6 Function: get_primary_selection()

The `get_primary_selection()` function is called from either the `move` or `copy` switch statements of the `drop_proc()` callback function. This function will get data from the source in the format mutually agreed upon. The first `xv_get()` function determines from the passed atom the selection datatype. If the selection is a filename, the text string is retrieved from the file and placed in the text subwindow. If the selection is a text string, the last `xv_get()` function retrieves the string and places it in the text subwindow.

```
void
get_primary_selection(Selection_requestor sel_req)
{
    long            length;
    int             format;
    char*sel_string;
    char*string;
    Atom*list;
    int i;

    list = NULL;
    xv_set(sel_req, SEL_TYPE,
atom_list[_SUN_AVAILABLE_TYPES_ATOM].atom, 0);
    list = (Atom *) xv_get(sel_req, SEL_DATA, &length, &format);
    if (length == SEL_ERROR)
    {
    printf("*** Unable to get target list.\n");
    }
    else
    {
    printf("length = %d format = %d\n", length, format);
    while(*list)
    {
        printf("list = %X\n", list);
        for(i = 0; i < TOTAL_ATOMS; i++)
        {
            if(*list == atom_list[i].atom)
            {
            printf("supports %d %s\n", i,
                 atom_list[i].name);
            break;
            }
        }
        list++;
```

```
    }
    }
    xv_set(sel_req, SEL_TYPE, atom_list[FILE_NAME_ATOM].atom,
0);
    string = (char *) xv_get(sel_req, SEL_DATA, &length,
&format);
    if (length != SEL_ERROR)
    {
    printf("length = %d format = %d\n", length, format);
    /* Create a NULL-terminated version of 'string' */
    sel_string = (char *) calloc(1, length + 1);
    strncpy(sel_string, string, length);

    xv_set(load_file, PANEL_VALUE, string, NULL);
    xv_set(textsw,
        TEXTSW_FILE,string,
        NULL);
    return;
    }
    else
    {
    printf("*** Unable to get FILE_NAME_ATOM selection.\n");
    }

    xv_set(sel_req, SEL_TYPE, atom_list[XA_STRING_ATOM].atom,
0);
    string = (char *) xv_get(sel_req, SEL_DATA, &length,
&format);
    if (length != SEL_ERROR)
    {
    printf("length = %d format = %d\n", length, format);
    /* Create a NULL-terminated version of 'string' */
    sel_string = (char *) calloc(1, length + 1);
    strncpy(sel_string, string, length);

    textsw_reset(textsw, 0, 0);
    textsw_insert(textsw, string, length);
    }
    else
    {
    printf("*** Unable to get XA_STRING_ATOM selection.\n");
    }
}
```

E.7 Function: load_file_proc()

The function `load_file_proc()` is the event callback procedure that loads the selected file into the text subwindow when the user enters a valid file name followed by a Return.

```
/*
 * Notify callback function for `filename'. This routine loads the
 * named file into the textpane.
 */

Panel_setting
load_file_proc(Panel_item item, Event *event)
{
    char *value = (char *) xv_get(item, PANEL_VALUE);

    fprintf(stderr, "DnD_demo: load_file: value: %s\n", value);


    xv_set(textsw,
    TEXTSW_FILE,value,
    NULL);

    return panel_text_notify(item, event);
}
```

The ToolTalk Desktop Services Message Set

F

This appendix contains a description of each of the generic messages that comprise the ToolTalk Desktop Services Message Set. The ToolTalk Desktop Services Message Set is an *open* specification with no royalty or license fees. It will be continuously revised and modified to meet the expanding needs of desktop services developers and users. Send questions, comments, and requests for information to the Desktop Services Messaging Alliance at *ToolTalk_desktop_services@sun.com*.

F.1 General Description of the ToolTalk Desktop Services Message Set

The ToolTalk Desktop Services Message Set conventions apply to any tools in a POSIX or X11 environment. In addition to standard messages for these environments, the Desktop conventions define data types and error codes that apply to all of the ToolTalk inter-client conventions.

F.2 Desktop Definitions and Conventions

This section defines terms and error messages unique to the Desktop Services message set. Specific to the desktop services messages are values associated with fields:

boolean
A vtype for logical values. The underlying data type of boolean is integer; manipulate arguments of this vtype with `tt_*_arg_ival[_set]()` and `tt_*_iarg_add()`. A zero value means *false*; a non-zero value means *true*.

buffer
A volatile, non-shared (for example, in-memory) representation of persistent data.

bufferID
A vtype that uniquely identifies buffers. The underlying data type of bufferID is *string*. To guarantee bufferID uniqueness, use the form

```
<internal_counter> <procID>.
```

messageID
A vtype that uniquely identifies messages. The underlying data type of messageID is *string*; manipulate arguments of this vtype with `tt_*_arg_val[_set]()` and `tt_*_arg_add()`. To guarantee messageID uniqueness, use the form

```
<internal_counter> <procID>
```

`tt_message_id()` returns an opaque string of similar uniqueness. Use `tt_message_id()` to generate a message's messageID; however, the inter-client conventions explicitly include the messageID as a message argument to support inter-operation with other versions of the ToolTalk service.

type
Any vtype that is the name of the kind of objects in a particular persistent-object system. For example, the vtype for the kind of objects in filesystems is `File`; the vtype for ToolTalk objects is `ToolTalk_Object`.

vendor
toolName
toolVersion
Names of arguments. These strings appear in several of the Desktop Service messages. These strings are not defined rigorously; they are intended to present to the user descriptions of these three attributes of the relevant procID.

view
A screen display, such as a (portion of a) window, that presents to the user part or all of a document.

viewID
A vtype that uniquely identifies views. The underlying data type of viewID is *string*. To guarantee viewID uniqueness, use the form

```
<internal_counter> <procID>
```

Errors

Table F-1 describes the Desktop Services error messages; the error messages are listed in order of their message id.

Table F-1 Desktop Services Error Messages

Message ID	Error Message	Error Message String	Description
1538	TT_DESKTOP_ENOENT	No such file or directory	
1549	TT_DESKTOP_EACCES	Permission Denied	
1558	TT_DESKTOP_EINVAL	Invalid argument	An argument's value was not valid; for example, a locale in `Set_Locale` that is not valid on the handler's host. Use this error status only when a more-specific error status does not apply.
1571	TT_DESKTOP_ENOMSG	No message of desired type	A messageID does not refer to any message currently known by the handler.
1610	TT_DESKTOP_EPROTO	Protocol error	A message was not understood because: a. A required argument was omitted. b. An argument had the wrong vtype, or the vtype is not allowed in this message; for example, the vtype `boolean` in the `Get_Geometry` message. c. An argument's value was not legal for its vtype; for example, negative values for width in the `Set_Geometry` message. d. An argument's value was not legal for this message; for example, the `PATH=/foo` variable in `Get_Environment` message. In general, this error status indicates that the message is malformed.

Table F-1 Desktop Services Error Messages (Continued)

Message ID	Error Message	Error Message String	Description
1688	TT_DESKTOP_CANCELED	Operation was canceled	The operation was canceled because of direct or indirect user intervention. An example of indirect intervention is termination of the handling process caused by the user, or receipt of a Quit() request. (All messages should be taken as authentically representing the wishes of the user whose uid is indicated by tt_message_uid().)
1689	TT_DESKTOP_ENOTSUP	Operation not supported	The requested operation is not supported by this handler. This error indicates that a handler assumes that, if it rejects a request, no other handler will be able to perform the operation. For example, a request such as Set_Iconified() or a request that refers to a state (such as a bufferID) that is managed by this handler alone. A request failed with this error distinguishes the case of an incompletely-implemented handler from the case of the absence of a handler. **Note:** Do not use TT_ERR_UNIMP in place of TT_DESKTOP_ENOTSUP as TT_ERR_UNIMP means that a particular feature of ToolTalk itself is not implemented.
1699	TT_DESKTOP_UNMODIFIED	Operation does not apply to unmodified entities	

Warnings

The vtype namespace for persistent objects currently only contains `File` and `ToolTalk_Object`. Vendors who want to define a type should either give it a vendor-specific name or register it through SunSoft's Developer Integration Format Registration program. SunSoft can be reached at 1-800-227-9227.

F.3 The ToolTalk Desktop Services Message Set

This section contains a description of each of the generic messages which constitute the ToolTalk Desktop Services Message Set.

Created, Deleted (Notice)

Notification that entities (for example, files) have been created or deleted.

Synopsis

```
[file] Created(in type ID[...]);
[file] Deleted(in type ID[...]);
```

Description

The Created notice is sent whenever a tool creates or deletes one or more entities that may be of interest to other tools.

Required Arguments

type ID
: The identity of the created entity. If more than one entity are created in the same logical event, extra ID arguments may be present.

  When *type* is File, each non-empty ID argument is the name of an entry which has been created in the directory named in the message's file attribute. (Each argument is, therefore, a single, final component of a pathname.)

  When *type* is File and this argument is empty (that is, has a value of (char *)0), it refers to the file or directory named in the message's file attribute.

Optional Arguments

type ID
: Extra instances of this argument may be included.

Do_Command (Request)

Requests in a tool's native command language that a command be performed.

Synopsis

```
Do_Command(  in      string        command,
             out     string        results
             [in     messageID     counterfoil] );
```

Description

The Do_Command message requests that the receiving tool perform a command. The request is stated in the receiving tool's native command language.

When the request includes the optional *counterfoil* argument, the handler can send an immediate point-to-point status notice back to the requesting tool if the requested operation is expected to require an extended amount of time.

Required Arguments

string command
: The command being requested to be performed.

string results
: The results of the completed command. The results are returned as if the command had been executed locally to the requesting tool.

Optional Arguments

messageID counterfoil

Unique string created by the message sender (typically by concatenating a counter and a procID) to give both sender and receiver a way to refer to this request in other correspondence. Include this argument if the sender anticipates a need to communicate with the handler about this request before it is completed; for example, to cancel it.

When this argument is included and the handler determines that an immediate reply is not possible, then the handler should immediately send at least one Status notice point-to-point back to the requestor to identify itself to the requestor.

Warnings

This request allows tools to provide a message interface to functionality that is not supported through any standard (or even tool-specific) message interface. This message, therefore, constitutes a deprecated interface when the intended function is available through an existing message interface.

Get_Modified (Request)

Asks whether an entity (for example, a file) has been modified.

Synopsis

```
[file] Get_Modified(intype          ID,
             out     boolean        modified);
```

Description

The Get_Modified message asks whether any tool has modified a volatile, non-shared for example, in-memory) representation of the persistent state of an entity (such as a file) with the intention of eventually making that representation persistent. Therefore, a tool should register a dynamic pattern for this request when it has modified an entity of possible shared interest.

Required Arguments

type ID
: The identity of the entity that may have been modified.

  When *type* is `File`, this argument is empty (that is, it has a value of (char *) 0) and references the file or directory named in the message's file attribute.

boolean modified
: The boolean value that indicates whether a volatile, non-shared (for example, in-memory) representation of the entity has been modified with the intention of eventually making that representation persistent.

Errors

TT_ERR_NO_MATCH
: The Get_Modified request failed because no handler was found and the named entity is assumed not to be modified.

Get_Status (Request)

Requests that a tool's current status be returned.

Synopsis

```
Get_Status(  out     string         status,
             out     string         vendor,
             out     string         toolName,
             out     string         toolVersion
             [in     messageID      operation2Query]);
```

Description

The Get_Status message retrieves either the current status of a tool or the current status of a specific operation that is being performed by a tool.

Required Arguments

string status
: The status to be retrieved.

string vendor
: The name of the vendor of the receiving tool.

string toolName
: The name of the receiving tool.

string toolVersion
: The version of the receiving tool.

Optional Arguments

messageID operation2Query
: The ID of the request that initiated the operation the status of which is being requested.

Get_Sysinfo (Request)

Retrieves information about a tool's host.

Synopsis

```
Get_Sysinfo( out     string        sysname,
             out     string        nodename,
             out     string        release,
             out     string        version,
             out     string        machine,
             out     string        architecture,
             out     string        provider,
             out     string        serial);
```

Description

The Get_SysInfo message retrieves information about the receiver's host.

Required Arguments

string sysname
The name of the host's operating system.

string nodename
The name of the host.

string release
string version
Vendor-determined information about the host's operating system.

string machine
A vendor-determined name that identifies the hardware on which the operating system is running (such as sun4, sun4c, or sun4m).

string architecture
A vendor-determined name that identifies the instruction set architecture of the host (such as sparc, mc68030, m32100, or i80486).

string provider
The name of the hardware manufacturer.

string serial
The ASCII representation of the hardware-specific serial number of the host.

See Also

sysinfo(2), umane(2)

Modified, Reverted (Notice)

Notification that an entity (for example, a file) has been either modified or reverted to its prior state.

Synopsis

```
[file] Modified(in   type          ID);

[file] Reverted(in   type          ID);
```

Description

The Modified message notifies interested tools whenever a tool first makes changes to a volatile, non-shared (for example, in-memory) representation of the persistent state of an entity (such as a file). The Reverted message notifies interested tools whenever a tool discards the modifications made to a volatile, non-shared (for example, in-memory) representation of the persistent state of an entity (such as a file).

Required Arguments

type ID
: The identity of the modified or reverted entity.

  When *type* is File, this argument is empty (that is, has a value of (char *)0) and refers to the file or directory named in the message's file attribute.

Moved (Notice)

Notification that an entity (for example, a file) has been moved.

Synopsis

```
[file] Moved(in      type          oldID,
             in      type          newID);
```

Description

The Moved message notifies interested tools whenever a tool changes the location of a persistent entity.

Required Arguments

type newID
: The new identity of the moved entity.

  When *type* is File, this argument is empty (that is, has a value of (char *)0), and refers to the file or directory named in the message's file attribute.

type oldID
: The old identity of the moved entity.

  When *type* is File, this argument is either an absolute pathname, or a pathname relative to the directory named in (or containing) the path in the message's file attribute.

Pause, Resume (Request)

Requests the specified tool, operation, or data performance to pause or resume.

Synopsis

```
Pause(      [in     messageID     operation] );

Pause(      in      bufferID      docBuf );

Resume(     [in     messageID     operation] );

Resume(     in      bufferID      docBuf
            [in     locator       whither
            |in     vector        duration] );
```

Description

The Pause or Resume messages requests that the specified tool, operation, or data performance pause or resume, respectively.

- If the optional *operation* argument is included, the handler should pause or resume the operation that was invoked by the specified request. Use a Tt_address of TT_HANDLER to send this form of the request.
- If the optional *docBuf* argument is included, performance of the data in the specified buffer should be paused or resumed. Use a Tt_address of TT_PROCEDURE to send this form of the request.
- If both of the optional arguments are omitted, the handling procid should pause or resume its operations. Use a Tt_address of TT_HANDLER to send this form of the request.

Caution – The Pause and Resume requests may also be sent as a multicast notices; however, the consequences can be severe and unexpected.

Optional Arguments

bufferID docBuf
The buffer in which data performance is to be paused or resume.

messageID operation
The request to be paused.

locator whither
The buffer location to which performance is to be resumed.

vector duration
The duration for which performance is to be resumed.

Note – If neither the *whither* nor the *duration* argument is included in this message, the performance is resumed indefinitely.

Errors

TT_ERR_NOMATCH
The bufferID may not be valid; no editor has a pattern handling this request for docBuf.

TT_DESKTOP_EINVAL
The value for the *whither* is not a legal locator for the media type of the document in docBuf.

TT_DESKTOP_EINVAL
The destination is not a legal vector for the media type of the document in docBuf.

TT_DESKTOP_EFAULT
The value for the *whither* argument is not a valid locator for the document in docBuf.

TT_DESKTOP_EFAULT
The value for the *duration* argument is not a valid vector for the document in docBuf.

TT_DESKTOP_ENOMSG
The operation does not refer to any message currently known by the handler.

Quit (Request)

Requests that an operation, or an entire tool, terminate.

Synopsis

```
Quit(       in      boolean       silent,
            in      boolean       force
            [in     messageID     operation2Quit]);
```

Description

Without the optional *operation2Quit* argument, this request asks the recipient procID to quit. If the request succeeds, one or more ToolTalk procID's should call tt_close(), and zero or more processes should exit. ("Zero or more process" are indicated because a single process can instantiate multiple independent procID's, and a single procID can conceivably be implemented by a set of cooperating processes.)

With the optional *operation2Quit* argument, this request asks the recipient to terminate the indicated request. (Whether the terminated request must be failed depends on its semantics. Often, termination can be considered to indicate that the requested operation has been carried out to the requestor's satisfaction.)

This request should be failed (and the status code set appropriately) when the termination is not performed -- for example, because the *silent* argument was false and the user canceled the quit operation.

Caution – The Quit request may also be sent as a multicast notice; however, the consequences can be severe and unexpected.

Required Arguments

boolean silent
: Boolean value that indicates whether the recipient tool is allowed to block on user input before terminating itself, or the indicated operation. If this value is *false*, the handler is not required to seek user input.

boolean force
Boolean value that indicates whether the recipient tool should terminate itself even if circumstances are such that the tool ordinarily would not terminate under them.

For example, a tool might have a policy of not quitting with unsaved changes unless the user has been asked whether the changes should be saved. When this argument is true, such a tool should terminate even when doing so would lose changes that the user has not been asked about saving.

Optional Arguments

messageID operation2Quit
The request that should be terminated. For a request to be terminable, an (optional) counterfoil messageID shall have been included in the request, and the handler shall have sent a Status notice back to the requestor (thus identifying itself to the requestor).

Errors

TT_DESKTOP_ECANCELED
The Quit request was over-ridden by the user.

TT_DESKTOP_ENOMSG
The operation2Quit argument does not refer to any message currently known by the handler.

Raise, Lower (Request)

Raises or lowers a tool's window(s) to the front or back, respectively.

Synopsis

```
Raise(      [in     messageID     commission...]
            [in     viewID        view2Raise...]);

Lower(      [in     messageID     commission...]
            [in     viewID        view2Lower...]);
```

Description

The Raise and Lower message raise or lower, respectively, the window(s) associated with the recipient's procid. If any optional arguments are present, only the indicated window(s) are raise or lowered.

Caution – The Raise and Lower requests may also be sent as a multicast notice; however, the consequences can be severe and unexpected.

Optional Arguments

messageID commission
: The identifier of the message (if any) that resulted in the creation of the raised or lowered window(s).

viewID view2Raise
viewID view2Lower
: The identifier of the view whose associated window(s) is (are) be raised or lowered.

Save, Revert (Request)

Saves or discards any modifications to an entity (for example, a file).

Synopsis

```
[file]      Save(   in          typeID);
[file]      Revert( in          typeID);
```

Description

The Save and Revert messages requests that any pending, unsaved modifications to a persistent entity (such as a file) be saved or discarded, respectively.

Required Arguments

type ID
The identity of the entity to save or revert.

When *type* is `File`, this argument is empty (that is, it has a value of (char *) 0) and references the file or directory named in the message's file attribute.

Errors

TT_DESKTOP_UNMODIFIED
The entity had no pending, unsaved modifications.

`TT_DESKTOP_ENOENT`
The file to save or revert does not exist.

Saved(Notice)

Notification that an entity (such as a file) has been saved to persistent storage.

Synopsis

```
[file] Saved(in      type          ID);
```

Description

The Saved message notifies interested tools whenever a tool saves an entity (such as a file) to persistent storage.

Required Arguments

type ID
: The identity of the saved entity.

  When *type* is File, this argument is empty (that is, has a value of (char *)0), and refers to the file or directory named in the message's file attribute.

Set_Environment, Get_Environment (Request)

Requests that a tool's environment either be set or retrieved.

Synopsis

```
Set_Environment(      in             stringvariable,
                      in             stringvalue
                      [...]);

Get_Environment(      in             stringvariable,
                      out            stringvalue
                      [...]);
```

Description

The Set_Environment and Get_Environment messages request that the value of the indicated environment variable(s) either be replaced or reported, respectively.

Caution – The Set_Environment request may also be sent as a multicast notice; however, the consequences can be severe and unexpected.

Required Arguments

string variable
The name of the environment variable to be set or retrieved.

string value
The value of the environment variable to be set or retrieved.

- If this argument does not contain a value for the Set_Environment request, the variable is removed from the environment. It is not considered an error if the specified variable does not exist.
- If this argument does not contain a value when used in the Get_Environment request, the variable was not present in the receiving tool’s environment. This condition is not considered an error.

Optional Arguments

string variable
string value
Extra pairs of these arguments may included.

Set_Geometry, Get Geometry (Request)

Requests that a tool's on-screen geometry either be set or retrieved.

Synopsis

```
Set_Geometry(inout   width          w
             inout   height         h
             inout   xOffset        x
             inout   yOffset        y
             [in     messageID      commission]
             [in     viewID         view2Set]);

Get_Geometry(out     width          w
             out     height         h
             out     xOffset        x
             out     yOffset        y
             [in     messageID      commission]
             [in     viewID         view2Get]);
```

Description

The Set_Geometry and Get_Geometry messages request that the value of the on-screen geometry of the optionally-specified window, or the value of the on-screen geometry of the window primarily associated with the receiving tool's procID if no window is specified, be either set or retrieved (respectively).

Required Arguments

width w
height h
xOffset x
yOffset y

The integer geometry values in pixels.

The return values for the Get_Geometry request are the actual new values, not the requested new values.

Note – Negative offset values are interpreted according to X11 rules.

Optional Arguments

messageID commission
: The identifier of the message (if any) that resulted in the creation of the set or retrieved window(s).

viewID view2Set
viewID view2Get
: The identifier of any view associated with the window(s) that is (are) to be set or retrieved.

Set_Iconified, Get_Iconified(Request)

Requests that a tool's iconic state be set or retrieved.

Synopsis

```
Set_Iconified(inout   boolean       conic
              [in     messageID     commission]
              [in     viewID        view2Iconify]);

Get_Iconified(out     boolean       iconic
              [in     messageID     commission]
              [in     viewID        view2Query]);
```

Description

The Set_Iconified and Get_Iconified messages request that the value of the iconic state of the optionally-specified window, or the iconic state of the window primarily associated with the receiving tool's procID if no window is specified, be either set or retrieved (respectively).

Caution – The Set_Iconified and Get_Iconified requests may also be sent as a multicast notice; however, the consequences can be severe and unexpected.

Required Arguments

boolean iconic
The boolean value that indicates whether the specified window is iconified.

Optional Arguments

messageID commission
The identifier of the message (if any) that resulted in the creation of the iconified or queried window(s).

viewID view2Iconify
viewID view2Query
The identifier of any view associated with the window(s) that is (are) to be iconified or queried.

Set_Locale, Get_Locale (Request)

Sets or retrieves a tool's locale.

Synopsis

```
Set_Locale(  in      string        category,
             in      string        locale
             [...]);

Get_Locale(  in      string        category,
             out     string        locale
             [...]);
```

Description

The Set_Locale and Get_Locale messages replace or report (respectively) the locale of the POSIX locale categories.

Caution – The Set_Locale request may also be sent as a multicast notice; however, the consequences can be severe and unexpected.

Required Arguments

string category
The locale category to set or retrieve.

A locale category is a group of data types whose formatting varies according to locale; for example, ANSI C and X/OPEN locale categories include:

- LC_CTYPE
- LC_NUMERIC
- LC_TIME
- LC_COLLATE
- LC_MONETARY
- LC_ALL
- LC_MESSAGES (Solaris-specific)

string locale
The name of the current locale of the indicated category, or the locale to which to set the indicated category; example of these locales defined in UNIX SVR4 are "C", "de", "fr", and "it".

Optional Arguments

string category
string locale
Extra pairs of these arguments may be included.

Set_Mapped,Get_Mapped(Request)

Requests that a tool's mapping to the screen be set or retrieved.

Synopsis

```
Set_Mapped(   inout    boolean        mapped
              [in      messageID      commission]
              [in      viewID         View2Map]);

Get_Mapped(   out      boolean        mapped
              [in      messageID      commission]
              [in      viewID         view2Query]);
```

Description

The Set_Mapped and Get_Mapped messages request that value of the mapped state of the optionally-specified window, or the mapped state of the window primarily associated with the receiving tool's procID if no window is specified, be either set or retrieved (respectively).

Caution – The Set_Mapped request may also be sent as a multicast notice; however, the consequences can be severe and unexpected.

Required Arguments

boolean mapped
: The boolean value that indicates whether the specified window is mapped to the screen.

Optional Arguments

messageID commission
: The identifier of the message (if any) that resulted in the creation of the set or retrieved window(s).

viewID view2Map
viewID view2Query
: The identifier of any view associated with the window(s) that is (are) to be set or retrieved.

Set_Situation, Get_Situation

Requests that a tool's current working directory be set or reported.

Synopsis

```
Set_Situation(in     string        path );
Get_Situation(out    string        path );
```

Description

The Set_Situation and Get_Situation messages request that value of the current working directory be either set or reported (respectively).

Caution – The Set_Situation request may also be sent as a multicast notice; however, the consequences can be severe and unexpected.

Required Arguments

string path
: The pathname of the working directory that the recipient is either using or is to use.

Set_XInfo, Get_XInfo (Request)

Requests that a tool's X11 attributes be set or retrieved.

Synopsis

```
Set_XInfo(    inout    string        display,
              inout    string        visual,
              inout    integer       depth
              [in      messageID     commission]
              [inout   string        resourceName,
              inout    string        resourceVal,...]);

Get_XInfo(    out      string        display,
              out      string        visual,
              out      integer       depth
              [in      messageID     commission]
              [in      string        resourceName,
              out      string        resourceVal,...]);
```

Description

The Set_XInfo and Get_XInfo messages request that the X11 attributes of the optionally-specified window, or the X11 attributes of the window primarily associated with the receiving tool's procID if no window is specified, be either set or retrieved (respectively).

Required Arguments

string display
   An X11 display.

Note – Since the handler may be running on a different host, use the value `hostname:n[.n]` rather than `:n[.n]`.

---

string visual
An X11 visual class, which determines how a pixel will be displayed as a color. Values include:

StaticGray
GrayScale
StaticColor
PseudoColor
TrueColor
DirectColor

integer depth
The number of bits in a pixel.

Optional Arguments

string resourceName
string resourceVal
An X11 resource name and resource value.

messageID commission
The ID of the message with respect to which X11 attributes are being set or reported. This is useful to the extent that the handler employs different attributes for the different operations it may be carrying out.

Signal(Request)

Requests that a (POSIX-style) signal be sent to a tool.

Synopsis

```
Signal(      in      integer        theSignal);
```

Description

The Signal message requests that the receiving tool's procID send the indicated signal to itself.

Required Arguments

integer theSignal
The signal to be sent.

Caution – The Signal request may also be sent as a multicast notice; however, the consequences can be severe and unexpected.

Started, Stopped (Notice)

Notification that a tool has started or terminated.

Synopsis

```
Started(     in      string        vendor,
             in      string        toolName,
             in      string        toolVersion);

Stopped(     in      string        vendor,
             in      string        toolName,
             in      string        toolVersion);
```

Description

The Started and Stopped messages notify interested tools whenever a tool starts or terminates, respectively.

Required Arguments

string vendor
The name of the vendor of the started or terminated tool.

string toolName
The name of the started or terminated tool.

string toolVersion
The version of the started or terminated tool.

Status(Notice)

Notification that a tool has status information to announce.

Synopsis

```
Status(      in      string        status,
             in      string        vendor,
             in      string        toolName,
             in      string        toolVersion
             [in     messageID     commission]);
```

Description

The Status message notifies interested tools of a tool's general status information.

Required Arguments

string status
: The status which is being announced.

string vendor
: The name of the vendor of the tool whose status is being announced.

string toolName
: The name of the tool whose status is being announced.

string toolVersion
: The version of the tool whose status is being announced.

Optional Arguments

messageID commission
: The ID of the request, if any, that initiated the operation the status of which is being announced.

The ToolTalk Document and Media Exchange Message Set

G

Multimedia is an important emerging technology. While the base of multimedia-aware applications has expanded, no single vendor provides a completely integrated solution which meets the complex needs of today's market. The *ToolTalk Document and Media Exchange™ Message Set* is a genuine breakthrough in multimedia technologies. A powerful messaging protocol designed to benefit both developers and users of multimedia technologies, the ToolTalk Document and Media Exchange Message Set allows applications to easily share each others multimedia functionality. Using the ToolTalk Document and Media Exchange Message Set, multimedia applications can communicate with each other in a transparent manner, both locally and over networks, regardless of data formats, compression technology, and other technical issues which has previously confined the use of this technology.

The ToolTalk Document and Media Exchange Message Set is an *open* specification with no royalty or license fees. It will be continuously revised and modified to meet the expanding needs of multimedia developers and users. Send questions, comments, and requests for information to the Document and Media Exchange Messaging Alliance at *media_exchange@Sun.Com*.

This appendix contains a description of each of the messages that constitute the ToolTalk Document and Media Exchange Message Set. Each message described contains the information described in Table G-1. In addition the next sections describe common information to all messages. These sections describe ToolTalk unique definitions and error messages common to all messages.

G.1 General Tooltalk Message Definitions and Conventions

In the ToolTalk messages there are terms used with specific ToolTalk definitions. This section defines these terms and conventions used in the ToolTalk message man pages.

Table G-1 ToolTalk Document and Media Exchange Message Set Descriptions

Type of Information	Description
header	A single line that describes the message in the following format: *MsgName*(*Tt_class*) where *MsgName* is the name of the message and *Tt_class* is either Request or Notice.
name	The name of the message and a one-line description of the message.
description	An explanation of the operation (event) that the message requests (announces).

Table G-1 ToolTalk Document and Media Exchange Message Set Descriptions

Type of Information	**Description**
synopsis	A representation of the message in the ToolTalk types-file syntax (similar to the syntax understood by the ToolTalk type compiler `tt_type_comp`) in the following format: <fileAttrib> <opName> (<requiredArgs> [<optionalArgs>]); A synopsis entry is given for each interesting variant of the message. <fileAttrib> - An indication of whether the file attribute of the message can/should be set. <opName> - The name of the operation or event is called the "op name" (or "op"). It is important that different tools not use the same opName to mean different things. Therefore, unless a message is a standard one, its opName should be made unique. A good way to do this is to prefix it with: <Company><Product> e.g., "Acme_Hoarktool_My_Frammistat". <requiredArgs>, <optionalArgs> - The arguments that must always be included in the message. A particular argument is described in the following format: <mode> <vtype> <argument name> where *mode is one of "in","out", or "inout", vtype* is a programmer-defined string that describes what kind of data a message argument contains; and *argument name* is the name of the argument. The ToolTalk service uses vtypes to match sent message instances with registered message patterns. By convention, a vtype maps to a single, well-known data type.

Table G-1 ToolTalk Document and Media Exchange Message Set Descriptions

Type of Information	Description
required arguments	The arguments that must always be in the message. <vtype> <argumentName> A description of a particular argument. A 'vtype' is a programmer-defined string that describes what kind of data a message argument contains. ToolTalk uses vtypes for the sole purpose of matching sent message instances with registered message patterns. Every vtype should by convention map to a single, well-known data type. The data type of a ToolTalk argument is either integer, string, or bytes. The data type of a message or pattern argument is determined by which ToolTalk API function is used to set its value. The argument name is merely a comment hinting to human readers at the semantics of the argument, much like a parameter name in a C typedef.
optional arguments	The extra arguments that may be included in a message. Unless otherwise noted, any combination of the optional arguments, in any order, may be appended to the message after the required arguments.
description	An explanation of the operation that the request entreats, or the event that the notice announces.
errors	A list of the error codes that can be set by the handler of the request (or the sender of the notice).

Edict

An *edict* is a notice that looks like a request. If a request returns no data (or if the sender does not care about the returned data), it can sometimes be useful to broadcast that request to a set of tools. Since the message is a notice, no data is returned, no replies are received, and the sender is told if any tool gets the message.

Handler

The *handler* is the distinguished recipient procid of a request. This procid is responsible for completing the indicated operation.

Notice
A *notice* is a message that announces an event. Zero or more tools may receive a given notice. The sender does not know whether any tools receive its notice. A notice cannot be replied to.

Procid
A *procid* is a principal that can send and receive ToolTalk messages. A procid is an identity, created and handed over by the ToolTalk service on demand (via tt_open()), that a process must assume in order to send and receive messages. A single process can use multiple procids; and a single procid can be used by a group of cooperating processes.

Request
A request is a message that asks an operation to be performed. A request has a distinguished recipient, called a handler, who is responsible for completing the indicated operation. A handler may fail, reject, or reply to a request. Any number of handlers may reject a request but ultimately only one handler can fail it or reply to it. If no running handler can be found to accept a request, the ToolTalk service can automatically start a handler. If no willing handler can be found, or if a handler fails the request, then the request is returned to the sender in the 'failed' state.

G.1.1 Errors

A `Tt_status` code can be read from a reply via `tt_message_status()`. This status defaults to `TT_OK`, or can be set by the handler via `tt_message_status_set()`. In extraordinary circumstances (such as no matching handler) the ToolTalk service itself sets the message status.

In addition to the `Tt_status` values defined by the ToolTalk API, the overview reference page for each set of messages lists the error conditions defined for that set of messages. For each error condition, the overview reference page provides

- Its name
- Its integer value
- A string in the "C" locale that explains the error condition

Since the ToolTalk Inter-Client Conventions (TICC) are a binary message interface, the integer and string are part of that binary interface; the name is not.

- The string may be used as a key in the `SUNW_TOOLTALK_INTERCLIENTCONVENTIONS` domain to retrieve a localized explanation of the error condition. See `dgettext(3)`.
- The integer values of these status codes begin at 1537 (`TT_ERR_APPFIRST + 1`). The first 151 codes correspond to the system error list defined in `intro(2)`.

A standard programming interface for these conventions that binds the name to the integer value does not yet exist.

The ToolTalk service allows an arbitrary status string to be included in any reply. Since a standard localized string can be derived for each status code, this status string may be used as a free-form elucidation of the status. For example, if a request is failed with `TT_DESKTOP_EPROTO`, the status string could be set to `The vtype of argument 2 was ‘string’; expected ‘integer’`. Handling tools should try to compose the status string in the locale of the requestor. See the `Get_Locale()` request.

Generic messages can be sent and received by any tool.

G.2 Media Exchange Definitions and Conventions

Specific to the media exchange messages there are values associated with fields. The following paragraphs define those fields.

Editor messages are sent and received by tools that display or edit some kind of media. The parts of an editor message is defined as follows:

<document>
A vector of bytes with an associated mediaType.

<mediaType>
The name of a media format. The mediaType allows messages about that document to be dispatched to the right editor. Standard mediaTypes include:

• ISO_Latin_1	ISO 8859-1 (+tab+newline)	ISO
• PostScript	Postscript Lang Ref. Manual	Adobe
• RTF	MS Word Technical Ref	Microsoft
• MIF	Maker Interchange Format	FrameMaker
• WKS		

• EPS		
• GIF	Graphics Interchange Format	Compuserve
• TIFF	“TIFF Rev. 5” Technical Memo	Aldus / Microsoft
• XPM	XPM --The X PixMap Format	Groupe Bull
• Sun_Raster		
• Sun_XView_icon		
• Sun_Audio	audio_intro(3),audio_hdr(3)	Sun Microsystems
• JPEG		ISO / CCITT
• JPEG_Movie		Parallax Graphics
• RFC_822_Message	RFC 822	IETF
• Unix_Mail_Folder		
• Sun_CM_Appointment		Sun Microsystems

Note – The mediaType list will be extended as required. You can extract a list of the installed mediaTypes from the ToolTalk Types Database.

abstract mediaType
A family of similar mediaTypes, such as flat text or structured graphics.

vector
A string vtype describing a distance and a direction in a document. The syntax of vectors varies by abstract mediaType.

locator
A string describing a location in a document. The syntax of locators varies by abstract mediaType, but should usually be a superset of vector syntax.

flat text
A family of mediaTypes (such as ISO_Latin_1) which consist of a sequence of characters from some character set.

Legal vectors for flat text are:

```
lineVec ::= Line:[-][0-9]+
charVec ::= Character:[-][0-9]+
vector ::= <lineVec>
vector ::= [<lineVec>,]<charVec>
```

Legal locators for flat text are vectors.

G.2.1 Errors

These definitions are common to all messages. Any differences or additions will be noted in the man pages.

`1700 TT_MEDIA_ERR_SIZE`

The specified size was too big or too small.

`1701 TT_MEDIA_ERR_FORMAT`

The data do not conform to their alleged format.

`1702 TT_MEDIA_NO_CONTENTS`

The message neither contains nor refers to any document.

Abstract(Request)

Requests a summary representation of a document.

Synopsis

```
[file] Abstract (in     <mediaType>    contents,
                 out      <mediaType>   output
                 in       boolean       inquisitive,
                 in       boolean       covert
                [in       messageID     counterfoil]
                [inout    vector        size] );
```

Description

The Abstract message requests that a summary representation of a document (for example, an icon or a video frame raster) be returned. The abstraction is the best possible representation of the document within the size constraints of the sending tool.

Note – You can extract a list of the installed mediaType-to-mediaType mappings from the ToolTalk Types Database.

Required Arguments

<mediaType> contents
The contents of the document.

If this argument is empty (that is, it has a value of (char *) 0), the contents of the document are contained in the file named in the message's file attribute. If nulls are not legal in the given mediaType, the data type of the contents argument is `string`; otherwise, the data type is `bytes`.

<mediaType> output
The abstracted document.

boolean inquisitive
The boolean value that indicates whether the recipient is allowed to seek user input about interpretation options.

Note – However, even if this value is *true*, the recipient is not required to seek the input.

If both the *inquisitive* and `covert` values are true, the recipient should attempt to limit (for example, through iconification) its presence to the minimum required to receive any user input requested.

boolean covert
The boolean value that indicates whether the recipient is allowed to make itself apparent to the user as it performs the interpretation.

Note – However, even if the value is *false*, the recipient is not required to make itself apparent.

If both the *inquisitive* and `covert` values are true, the recipient should attempt to limit (for example, through iconification) its presence to the minimum required to receive any user input requested.

Optional Arguments

messageID counterfoil
A unique string created by the message sender, typically by concatenating a procid and a counter. The sending application includes this argument if it anticipates a need to communicate with the handler about this request before the request is completed; for example, you could include this argument to cancel the request.

Note – When this argument is included and the handler determines that an immediate reply is not possible, then the handler should immediately send at least one Status notice point-to-point back to the requestor so as to identify itself to the requestor.

vector size

- On input, the maximum size of the abstraction. The recipient returns an abstraction as close to this size as possible without exceeding this size.
- On output, the actual size of the abstraction to be returned; or, if the error `TT_MEDIA_ERR_SIZE` is returned, the smallest possible size the recipient is capable of returning.

Examples

In this scenario, a container application requires a representation of some video data. To abstract a representation frame of the video tool, you could send an Abstract request such as:

```
Abstract (in Acme_Video, out Sun_Raster output, ...);
```

to obtain a custom raster representation; or

```
Abstract (in Acme_Video, out Sun_XView_Icon output, ...);
```

to obtain a generic icon representation. In either case, the container application does not need to understand the Acme_Video format.

Errors

`1700 TT_MEDIA_ERR_SIZE`

The specified size was too big or too small

`1701 TT_MEDIA_ERR_FORMAT`

The data do not conform to their alleged format

`1702 TT_MEDIA_NO_CONTENTS`

The message neither contains nor refers to any document

Deposit(Request)

Saves the document to its backing store.

Synopsis

```
[file] Deposit(      in       <mediaType> contents
                    {in       bufferID     beingDeposited
                     in       messageID    commission} );

[file] Deposit(      in       <mediaType> contents,
                     out      bufferID     beingDeposited
                    [in       title         docName] );
```

Description

Save this document to its backing store. This request is different from the Save() request, because here the requestor (and not the handler) has the data that needs to be written. Deposit() should almost never be file-scoped, because if the sending tool knows what file the document belongs in, that tool should be able to perform the save itself.

Required Arguments

<mediaType> contents
The contents of the document.

If this argument is empty (that is, it has a value of (char *) 0), the contents of the document are contained in the file named in the message's file attribute. If nulls are not legal in the given mediaType, the data type of the contents argument is `string`; otherwise, the data type is `bytes`.

bufferID beingDeposited
messageID commission
The Identifier of the buffer to be deposited to backing store. The identifier is either a bufferID returned or the messageID of the edit request that created this buffer.

If the `beingDeposited` argument is an `out` parameter, a new document is created and the handling container application must save the document and return a new bufferID for it.

Optional Arguments

title docName
The name of the document.

Example

This request is especially useful for when the user checkpoints (e.g., via a "Save" menu item) her modifications to a document that is the subject of a purely-session-scoped Edit request in progress.

The second variant of this request can be issued by editors that allow the user to create, as an afterthought, extra documents 'near' the document that was just edited. This can be useful if the each document in the series can serve as the template or starting point for the next document. Of course, if the handling container application does not support the notion of accommodating uninvited documents, it should reject the request.

Errors

`TT_DESKTOP_ENOENT`
The file that was alleged to contain the document does not exist.

`TT_MEDIA_NO_CONTENTS`
The in-mode contents arg had no value and the file attribute of the message was not set.

`TT_MEDIA_ERR_FORMAT`
See general info for description.

Display(Request)

Displays a document.

Synopsis

```
[file] Display( in <mediaType>  contents
                [in   messageID  counterfoil]
                [in   string      docName   ] );
```

Description

The Display message requests that a document be displayed. *Display* is a generic term for the operation the player performs; for example, an audiotool displays sound. The Display request invokes the requested playback mechanism (such as a video tool, or an audio tool). The receiving tool decides:

- when the display operation is complete.
- what user gesture signals that the display is completed (that is, what determines that the user has signaled “I have completed the display.”).
- the action it takes after it has replied to the request.

Note – The display request does not allow changes to be saved back to the source data; however, a tool that supports a “save as” operation may allow edits to be saved back to the document.

Required Arguments

<mediaType> contents

The contents of the document. If this argument is empty (i.e., has a value of (char *)0), then the contents of the document are in the file named in the message’s file attribute. The data type of the contents argument shall be string, unless nulls are legal in the given mediaType, in which case the data type shall be bytes.

Optional Arguments

messageID counterfoil

The unique string created by the message sender (typically by concatenating a procID and a counter) to give both sender and receiver a reference to this request in other correspondence. Include this argument if the sender anticipates a need to communicate with the handler about this request before it is completed (for example, to cancel the request).

Note – When this argument is included and the handler determines that an immediate reply is not possible, then the handler should immediately send at least one Status notice point-to-point back to the requestor so as to identify itself to the requestor.

---

title docName

The name of the document.

Examples

To display a PostScript document, send a Display request with a first argument whose vtype is “PostScript”, and whose value is a vector of bytes such as “%!^J/inch {72 mul} def...”. (By “^J” here we mean the newline character, octal 12.)

To display a PostScript document contained in a file, send a Display request, scoped to that file, with a first argument whose vtype is “PostScript”, and whose value is not set.

Errors

`TT_DESKTOP_ENOENT`

The file that was alleged to contain the document does not exist.

`TT_MEDIA_NO_CONTENTS`

The in-mode contents arg had no value and the file attribute of the message was not set.

TT_MEDIA_ERR_FORMAT

See general info for description.

Edit(Request)

Edits or composes a document.

Synopsis

```
[file] Edit ( [in]out <mediaType>  contents
                [in   messageID   counterfoil]
                [in   string       docName   ] );
```

Description

The Edit message requests that a document be edited and a reply containing the new contents be returned when the editing is completed. The receiving tool decides:

- when the edit operation is complete.
- what user gesture signals that the edit is completed (that is, what determines that the user has signaled “I have completed the edit.”).
- the action it takes after it has replied to the request.

If a tool supports a “save” or “checkpoint” operation during editing, it can send a Deposit request back to the tool that requested the edit.

Required Arguments

<mediaType> contents
The contents of the document. If the message is file-scoped, the contents argument has no value, and the document is contained in the scoped file. The data type of the contents argument is *string* unless nulls are legal in the given mediaType; if nulls are legal, the data type is *bytes*. If the contents argument is mode `out`, a new document is to be composed and its contents to be returned in this argument.

Optional Arguments

messageID counterfoil
: The unique string created by the message sender (typically by concatenating a procID and a counter) to give both sender and receiver a reference to this request in other correspondence. Include this argument if the sender anticipates a need to communicate with the handler about this request before it is completed (for example, to cancel the request).

Note – When this argument is included and the handler determines that an immediate reply is not possible, then the handler should immediately send at least one Status notice point-to-point back to the requestor so as to identify itself to the requestor.

---

title docName
: The name of the document.

Examples

To edit an X11 “xbm” bitmap, send an Edit request with a first argument whose vtype is “XBM”, and whose value is a a string such as “#define foo_width 44^J#define foo_height 94^J...”. (By “^J” here we mean the newline character, octal 12.)

To edit an X11 “xbm” bitmap contained in a file, send an Edit request, scoped to that file, with a first argument whose vtype is “XBM”, and whose value is not set.

Errors

`TT_DESKTOP_ENOENT`
: The file that was alleged to contain the document does not exist.

`TT_MEDIA_NO_CONTENTS`
: The in-mode contents arg had no value and the file attribute of the message was not set.

`TT_MEDIA_ERR_FORMAT`

Interpret(Request)

Translates a document and displays the translation.

Synopsis

```
[file] Interpret(   in       <mediaType>   contents,
                    in       <mediaType>   targetMedium,
                    in        boolean      inquisitive,
                    in        boolean      covert
                   [in        messageID     counterfoil]
                   [in        title         docName ] );
```

Description

The Interpret message translates a document from one media type to another and displays the translation.

Note – The translation is the best possible representation of the document in the target media type; however, it is possible that the resulting representation cannot be perfectly translated back into the original document.

The Interpret request is equivalent to issuing a Translate request followed by a Display request. The Interpret message is a useful optimization when the sender has no interest in retaining the translation.

Note – It is possible to extract from the ToolTalk types database a list of the installed Translate() mediaType-to-mediaType mappings.

Required Arguments

<mediaType> contents
The contents of the document.

If this argument is empty (that is, it has a value of (char *) 0), the contents of the document are contained in the file named in the message's file attribute. If nulls are not legal in the given mediaType, the data type of the contents argument is `string`; otherwise, the data type is `bytes`.

<mediaType> targetMedium
An empty argument whose vtype indicates the mediaType into which the document is to be translated before it is displayed.

boolean inquisitive
The boolean value that indicates whether the recipient is allowed to seek user input about interpretation options.

Note – However, even if this value is *true*, the recipient is not required to seek the input.

If both the *inquisitive* and `covert` values are true, the recipient should attempt to limit (for example, through iconification) its presence to the minimum required to receive any user input requested.

boolean covert
The boolean value that indicates whether the recipient is allowed to make itself apparent to the user as it performs the interpretation.

Note – However, even if the value is *false*, the recipient is not required to make itself apparent.

If both the *inquisitive* and `covert` values are true, the recipient should attempt to limit (for example, through iconification) its presence to the minimum required to receive any user input requested.

Optional Arguments

messageID counterfoil
The unique string created by the message sender (typically by concatenating a procID and a counter) to give both sender and receiver a reference to this request in other correspondence. Include this argument if the sender anticipates a need to communicate with the handler about this request before it is completed (for example, to cancel the request).

Note – When this argument is included and the handler determines that an immediate reply is not possible, then the handler should immediately send at least one Status notice point-to-point back to the requestor so as to identify itself to the requestor.

---

title docName
The name of the document.

Examples

Text-to-Speech Translation

To request a string to be spoken, send an Interpret request such as the following:

```
Interpret( in ISO_Latin_1 contents, in Sun_Audio targetMedium )
```

ToolTalk will then pass this request to the appropriate third party server in your environment.

Errors

`TT_DESKTOP_ENOENT`
The file that was alleged to contain the document does not exist.

`TT_MEDIA_NO_CONTENTS`
The in-mode contents arg had no value and the file attribute of the message was not set.

`TT_MEDIA_ERR_FORMAT`
See general description for definition.

Print(Request)

Prints a document.

Synopsis

```
[file] Print( in       <mediaType> contents,
              in       boolean      inquisitive,
              in       boolean      covert
             [in       messageID    counterfoil]
             [in       title        docName ] );
```

Description

The Print message prints a document. In effect, the recipient assumes the user issued a “print...” command via the recipient's user interface. The recipient tool decides issues such as what it should do with itself after replying.

Required Arguments

<mediaType> contents
The contents of the document.

If this argument is empty (that is, it has a value of (char *) 0), the contents of the document are contained in the file named in the message's file attribute. If nulls are not legal in the given mediaType, the data type of the contents argument is `string`; otherwise, the data type is `bytes`.

boolean inquisitive
The boolean value that indicates whether the recipient is allowed to seek user input about interpretation options.

Note – However, even if this value is *true*, the recipient is not required to seek the input.

If both the *inquisitive* and `covert` values are true, the recipient should attempt to limit (for example, through iconification) its presence to the minimum required to receive any user input requested.

boolean covert
The boolean value that indicates whether the recipient is allowed to make itself apparent to the user as it performs the interpretation.

Note – However, even if the value is *false*, the recipient is not required to make itself apparent.

If both the *inquisitive* and `covert` values are true, the recipient should attempt to limit (for example, through iconification) its presence to the minimum required to receive any user input requested.

Optional Arguments

messageID counterfoil
The unique string created by the message sender (typically by concatenating a procID and a counter) to give both sender and receiver a reference to this request in other correspondence. Include this argument if the sender anticipates a need to communicate with the handler about this request before it is completed (for example, to cancel the request).

Note – When this argument is included and the handler determines that an immediate reply is not possible, then the handler should immediately send at least one Status notice point-to-point back to the requestor so as to identify itself to the requestor.

title docName
The name of the document.

Examples

Printing a PostScript Document

To print a PostScript document,

```
Print(     in PostScript         contents,
           in boolean            inquisitive,
           in boolean            covert)
```

where the first argument is vtype `PostScript` whose value is a a vector of bytes.

Printing a PostScript Document Contained in a File

To print a PostScript document contained in a file,

```
Print(      in PostScript          contents,
            in boolean             inquisitive,
            in boolean             covert)
```

where the file attribute is set to filename, and the first argument is vtype `PostScript` whose value is not set.

Errors

`TT_DESKTOP_ENOENT`

The file that was alleged to contain the document does not exist.

`TT_MEDIA_NO_CONTENTS`

The in-mode contents arg had no value and the file attribute of the message was not set.

`TT_MEDIA_ERR_FORMAT`

Translate(Request)

Translates a document from one media type to another media type.

Synopsis

```
[file] Translate(       in       <mediaType> contents,
                         out      <mediaType> output,
                         in       boolean      inquisitive,
                         in       boolean      covert
                        [in       messageID    counterfoil] );
```

Description

The Translate message requests that a document be translated from one media type to another media type and that a reply containing the translation be returned. The translation is the best possible representation of the document in the target media type; however, it is not guaranteed that the resulting translation can be perfectly translated back into the original document.

Note – You can extract a list of the installed mediaType-to-mediaType mappings from the ToolTalk Types Database.

Required Arguments

<mediaType> contents
The contents of the document.

If this argument is empty (that is, it has a value of (char *) 0), the contents of the document are contained in the file named in the message's file attribute. If nulls are not legal in the given mediaType, the data type of the contents argument is `string`; otherwise, the data type is `bytes`.

<mediaType> output
The translated document.

boolean inquisitive
The boolean value that indicates whether the recipient is allowed to seek user input about interpretation options.

Note – However, even if this value is *true*, the recipient is not required to seek the input.

If both the *inquisitive* and `covert` values are true, the recipient should attempt to limit (for example, through iconification) its presence to the minimum required to receive any user input requested.

boolean covert
The boolean value that indicates whether the recipient is allowed to make itself apparent to the user as it performs the interpretation.

Note – However, even if the value is *false*, the recipient is not required to make itself apparent.

If both the *inquisitive* and `covert` values are true, the recipient should attempt to limit (for example, through iconification) its presence to the minimum required to receive any user input requested.

Optional Arguments

messageID counterfoil
The unique string created by the message sender (typically by concatenating a procID and a counter) to give both sender and receiver a reference to this request in other correspondence. Include this argument if the sender anticipates a need to communicate with the handler about this request before it is completed (for example, to cancel the request).

Note – When this argument is included and the handler determines that an immediate reply is not possible, then the handler should immediately send at least one Status notice point-to-point back to the requestor so as to identify itself to the requestor.

Examples

Speech-to-Text Translation

To translate speech to text, send a Translate request such as the following:

```
Translate (in Sun_Audio contents, out ISO_Latin_1 output);
```

Optical Character Recognition (OCR)

To translate optical characters to text, send a Translate request such as the following:

```
Translate (in GIF contents, out ISO_Latin_1 output);
```

Errors

`TT_DESKTOP_ENOENT`

The file that was alleged to contain the document does not exist.

`TT_MEDIA_NO_CONTENTS`

The in-mode contents arg had no value and the file attribute of the message was not set.

`TT_MEDIA_ERR_FORMAT`

Glossary

accelerator

Any efficient, alternate method of implementing a series of commands. Specifically, drag-and-drop is an accelerator because it replaces one or more commands that could be executed through a series of command line scripts or menu selections.

additional action

Any action that takes place after the completion of a drag-and-drop data transfer. This includes actions that comprise normal termination of the conversation. Addenda (or “side effects”) are included in this category.

adjust mouse button

The mouse button (the center one, by default) that is used to adjust (add or remove) selections.

alternate transport medium

(ATM) Any communication channel other than the X wire; for instance, the ToolTalk service and sockets.

anchor

A connection point in a data file which supports one end of a link.

animation

There are two primary types of animation, tracking and previewing.

animation, previewing

A visual indication of receptivity to a drop. This may be indicated by a pointer change, for instance.

animation, tracking

The process of making an image of the object move across the desktop in synchronization with the user's mouse movement.

API

application programming interface.

Classing Engine

A mechanism that permits an application to query a database (the Classing Engine database) to determine the attributes of a desktop object.

conversation

The negotiations necessary to determine the format, transport method, and other considerations pertinent to the data to be transferred in a drag-and-drop operation.

data span

A segment of on-screen data. It can be a segment of text, digitized audio, video, and so forth.

desktop object

A discrete on-screen representation of data. This could be a data span, an application icon, file glyph, etc.

drag and drop

The overall concept of using a mouse to select a desktop object and move or copy it to another desktop object.

Drop Site Database Manager (DSDM)

A process (not the sending client) responsible for maintaining a registry or database of potential drop sites for drag-and-drop operations.

File type

Refers to a file's format (e.g., ASCII, PostScript, and Sun raster files), its parent application (FrameMaker, Lotus 1-2-3 data files) or the application executable itself (File Manager, Mail Tool, or Wingz® executable file).

gesture, drag and drop

Holding down the selection button over an existing selection, moving the mouse, then releasing the selection button. This causes a representation of the selection to move to the point where the button is released.

glyph

Any graphical element on the desktop. A glyph may be a button, a folder, or other graphical element representing a document or file.

hints

Suggestions that a source can provide to a destination concerning possible ways to deal with the data exchange.

icon

A closed representation of a window. An window displayed as an icon is still running. An icon may change in appearance to show the state of the application; for instance, the familiar Mail Tool icon shows whether new mail has arrived by displaying envelopes in a tray.

interplay

The ability of various objects on the OpenWindows desktop to exchange data without requiring user intervention.

match attributes

Each entry in a namespace table will have one or more match attribute values, which will be used by the namespace manager in matching client arguments.

menu mouse button

The mouse button (with a three-button mouse, by default the right one) that is used to display a menu associated with a desktop object.

namespace

Refers to the space from which an object name is derived and understood. Files are named within the file namespace, printers are named within the printer namespace. You cannot name a printer by using a file name.

namespace table

A namespace table is the place where all namespace information is stored, for use by the CE as well as a namespace manager. Each namespace table consists of entries (rows) and each entry consists of a set of named attributes.

namespace manager

Every namespace has a namespace manager. A namespace manager is a piece of code that performs the matching of client supplied arguments with the attributes of entries in a table.

Different namespace managers will use different matching logic. Consider two examples - (1) mapping file information to a file type and (2) mapping a type name to type attributes.

In case (1), the namespace manager for files might have to know a file name, a magic number and fstat() information in order to match with its own match attributes of file name pattern, magic number and an fstat() mask.

In case (2), all the type namespace manager has to know is the name of a type, which it will exact match with its type name match attribute.

Notice

A special pop-up window initiated by an application. A Notice warns of errors or potential loss of data. The application will not accept further input until the user dismisses the Notice by selecting a desired action.

object

See desktop object.

object type derivation

The process by which an object's name and content information is examined to determine the type of an object. This is not the same process as the one used to give an object its type.

OLIT

The OPEN LOOK Intrinsics Toolkit.

rendezvous

The set of events that are necessary to identify the sending client to the receiving client.

select

To distinguish an object (or objects) on the desktop so they may be operated on.

select mouse button

The mouse button (the left one, by default) that is used to select objects, set the insert point, drag objects, and set/reset buttons.

ToolTalk

A service for communications between applications on the desktop.

transport

The means by which an object is passed from one process to another.

type-specific attributes

Each object type has a set of attributes associated with it e.g. its methods, icons etc. These attributes are referred to as type-specific attributes.

type database

The database used to map an object type to its attributes. In the context of the CE, a type database is a namespace table, where types are named and their attributes stored.

workspace

The background area of a display screen on which windows and icons are displayed.

Index

D

E

F

G

P

Q

R

S

T

V

W