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.

3

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.