Drag and Drop

As of Release 2.0, Motif uses the Uniform Transfer Method (UTM) to implement all drag and drop operations. Any drag and drop applications you write for Release 2.0 should also use UTM. Drag and drop applications written prior to Release 2.0 will work without modification at Release 2.0.

You should read Chapter 16 before reading this chapter. Chapter 16 explains UTM. Drag and drop is merely one kind of UTM transfer. That is, drag and drop uses the same convert and destination callbacks as any other UTM transfer. However, drag and drop supports richer visuals and protocols than other kinds of data transfer, and it is these topics that are the focus of this chapter.

This chapter first provides an overview of the drag and drop process and concepts from both the user's and the application developer's perspectives, then explains the actions of both initiator and receiver clients during the drag and at the drop, giving code samples.

> **Note:** Throughout this chapter, we refer to `Btn2` as the button that activates drag and drop operations. Although `Btn2` is the traditional Motif drag and drop button, some users may prefer to use `Btn1` instead of `Btn2` when doing drag and drop operations on Text, TextField, List, and Container widgets. Users choose their preferred drag and drop button with the **XmNenableBtn1Transfer** resource of **XmDisplay**.

User Overview of Drag and Drop

Drag and drop allows the user to "pick up" objects on the screen, "drag" them around the display, and "drop" them at a new location, possibly in another application.

With drag and drop the user can

- Move text or other information between windows.
- Cause application–specific actions to occur.
- Obtain help information about drop sites.

This section describes what the user does and sees during a drag and drop transaction.

Overview of User Interaction

A drag and drop transaction consists of the following actions:

1. To start a drag transaction, a user typically presses and holds `Btn2`, over a source object. The application owning that object is the initiator of the drag. The current pointer is replaced by a drag icon––a picture representing the item being dragged.
2. The user moves the pointer. From now until a drop occurs, the drag icon replaces the mouse pointer. The user may move the pointer by moving the mouse, or with the arrow keys.
3. The user drops the object, usually by releasing the mouse button.
   Locations on the screen that can accept drops are drop sites, and the application owning that drop site is the destination or receiver.

   The drag icon can be dropped anywhere on the screen. However, only certain widgets have registered themselves as drop sites and are able to process the drop.

   The receiver application usually performs some action on the information represented by the dragged icon. The initiator application may also perform some action based on the results of a drag transaction.

A drop can be between applications or within the same application. An application can be both source and destination of a drop, source only, destination only, or not participate in drag and drop at all.

The user can request help about a drop site, if available, by dragging to the drop site and pressing `osfHelp` (usually *F1*).

The user can cancel the drag at any time by pressing `osfCancel`, usually *Escape*.

Overview of Drag–Over Effects

The drag icon consists of three parts:

- The source icon is a picture representing the type of the source object, such as text.
- The state icon can be used to show whether or not the object being dragged can be dropped at its current location on the screen.
- The operation icon can be used to show what action should happen when the drop takes place.

In the following illustration, the running figure is the source icon, the arrow in the upper left is the state icon, and the rectangles with the corner folded over indicate a Copy is desired.

Figure 1 A Drag Icon

These parts can be combined (blended) and attached to each other in different ways. The default blending and attachment are shown in the previous illustration.

Parts of the drag icon may change shape or color as it is being dragged through potential drop sites, providing visual feedback about possible drop sites to the user. These changes are drag–over effects.

Applications can use the default drag icon effects, or provide more sophisticated or custom drag icons. The application or user can customize these drag–over effects in resource files.

Note: This chapter contains several illustrations of default Motif drag and drop icons. In reality, there are two different sets of default Motif icons. The **XmNenableDragIcon** resource of **XmDisplay** determines which set is used. If **XmNenableDragIcon** is False (the default), the traditional Motif drag and drop icons are used. If **XmNenableDragIcon** is True, alternate Motif drag and drop icons are used. We have used only the traditional Motif drag and drop icons for the illustrations in this chapter.

Drag States

During a drag, there are three states that describe the relationship of a drag icon to what is under it at the time:

Valid drop site

The drag icon is over a drop site on which it can potentially be dropped (this is only a hint; when the drop is actually attempted, further processing may show that the drop cannot actually be done).

Invalid drop site

The drag icon is over a drop site, but it cannot be dropped there.

No drop site

The drag icon is not over a registered drop site.

The default state icon for all three states is the same: an arrow in the upper left corner of the drag icon. Because the icon is the same for all three states, it appears not to change during the drag. The application or the user can provide custom state icons or colors in a resource file.

Drag Operations

The user specifies what action is to take place when the drop occurs by pressing certain keys when the drag starts or while the drag is in process:

Shift only

Force a move from the initiator to the receiver client (Move)

Ctrl only

Force a copy from the initiator to the receiver client (Copy)

Shift and Ctrl

Force a link between the initiator and receiver clients (Link)

The operation chosen by the user must be valid for both the drag source and the drop site, or the drop site will be considered invalid.

If the user does not specify an operation, one is chosen by the toolkit. It choses an operation that is valid for both the drag source and drop site. Move is the first choice, Copy is the second, and Link is the third. If the system cannot find a valid operation, the drop site is considered invalid.

The operation icon reflects the operation chosen by the user or by the system. If the operation is changed by the user during the drag, the operation icon changes also.

The operation icon may change as the drag icon moves to different drop sites if the drop sites accept different operations.

Overview of Drag–Under Effects

A widget registered as a drop site may change visually as a drag icon passes over it. These visual cues are drag–under effects. The sensitive area of the widget is the part that responds to drag and drop. By default it is the whole widget, but applications can specify that only parts of the widget respond to drag and drop.

In addition, a widget may support *pause drag*. This means that if a drag icon pauses over a specifed part of a widget, some action will occur, such as scrolling. See Section 17.4.2.2 for more information about this feature.

Various highlighting styles are possible:
- border around the sensitive area of the drop site widget. This is the default value.

- The sensitive area of the drop site widget looks pushed out.
- The sensitive area of the drop site widget looks pushed in.
- A special pixmap is displayed within the sensitive area of the drop site widget, overwriting what is normally there.
- No drag–under effects are used for the drop site widget.

Applications can use the default drag–under visual effects, or create more sophisticated or custom effects, such as special animation or sound effects.

Overview of Drop Effects

Visual effects also take place during the drop:

- The drag icon appears to sit over the drop site while the processing for the drop is finishing, but the standard cursor is restored and can be used normally.
- The source icon appears to melt into the drop site if the drop is successful.
- The source icon appears to snap back to the source if the drop is unsuccessful.
- The source icon appears to snap back to the source and the previous X cursor returns if Cancel is requested. All drag–under and drag–over effects are removed.

These drop effects cannot be changed by the application or the user. However, if the receiver wants to cancel these effects completely, the receiver should ask the source to convert the special **_MOTIF_CANCEL_DROP_EFFECT** target. For example, the receiver can make a call like the following to cancel the effects:

```
XmTransferValue(cs->transfer_id,
     XInternAtom(XtDisplay(wid), XmS_MOTIF_CANCEL_DROP_EFFECT, False),
     NULL, NULL, XtLastTimeStampProcessed(XtDisplay(wid)));
```

The Motif toolkit will intercept this request and fulfill it by cancelling all visual effects. The drag source should not provide any code to handle this target.

The application or widget should supply a DialogBox containing information about a drop site if the user has requested help and the receiver client provides help.

Application Programmer's Overview of Drag and Drop

This section explains some drag and drop concepts, and provides a general view of the initiator and receiver duties during the drag and at the drop.

The Motif toolkit for drag and drop consists of

- Widgets and widget classes that provide resources containing details about the source and destination of the transfer
- Functions that applications use to manage the widgets and widget classes
- Protocols that specify how interactions between source and destination clients are to take place
- Functions that manage messages, call callbacks, decide on the valid operations for a potential drop, and keep the drop site status updated

If the initiator and receiver are in the same client, they share the same toolkit. If the initiator and receiver are different clients, each client has a version of the toolkit.

An application can allow any widget to be a drag source or initiator by specifying a translation for `Btn2Down` in that widget. The corresponding action creates a DragContext, which starts the drag and drop transaction. The toolkit on the initiator side is in charge during the drag and manages all drag messages and callbacks.

An application can register any widget as a drop site. The drop site widget may change visually as a drag icon moves in and out of it, providing drag–under visual clues to the status of the drag. The application controlling the current drop site is known as the receiver. The toolkit on the receiver side is in charge of the drop operation, and manages all drop messages and callbacks.

Each drag source and drop site specifies the types of data it is prepared to handle and what operations it can perform on that data.

The state of the drag indicates whether the drag icon is over a valid drop site, an invalid drop site, or no drop site. For a drop site to be valid, there must be at least one target type and one operation in common between the drag source and drop site.

Ways an Application Can Provide Drag and Drop

Applications can use drag and drop functionality on any of several levels:

- Appendix B lists the widgets that are already defined as drag sources and as drop sites. Therefore, at the simplest, an application can compile with the Motif libraries, and have those widgets participate in drag and drop. For example, text from a Text widget could be selected from one application and moved into the Text widget in another application.
- On a slightly more advanced level, applications can let the toolkit do most of the work, but provide some customization. For example, an application could register an **XmPushButton** as a drop site, but still use default visual effects. In this case, the application would register a widget as a DropSite and provide code to handle drop and transfer duties. The example programs *simple_drag* and *simple_drop* in the following subsections are at that level.
- A complex application can take much of the control of the drag and drop itself. It can provide custom visuals for both drag icon and drop site. It can manage overlapping drop sites and can include complex transfers of information. The online example program `DNDDemo.c` contains extensive customization.

Extending a Widget To Support Drop

Any UTM widget that provides an **XmNdestinationCallback** resource has the potential to be a drop site. Conversely, if a widget does not provide an **XmNdestinationCallback** resource, it cannot be a UTM drop site.

Of the widgets that provide an **XmNdestinationCallback** resource, many already contain all the required code to become a drop site. That is, many of the standard widgets can serve as drop sites without you, the application programmer, doing anything. However, some of the widgets that do provide an **XmNdestinationCallback** resource do not provide the drop site code. You can extend these widgets to become drop sites. For example, the **XmDrawingArea** widget provides an **XmNdestinationCallback** resource but does not provide any drop site code. You can write an application that makes **XmDrawingArea** into a drop site.

Consider the *simple_drop* sample program stored online in `simple_drop.c`. This program displays both a Label widget and a DrawingArea widget. The Label widget displays whatever pixmap you pass on the invocation command line. For example, if you invoke this application as follows:

```
$ simple_drop mypixmap.xpm
```

then the Label widget will display the contents of the *mypixmap.xpm* file. To experiment with this application, simply drag the pixmap from the Label and drop it into the framed DrawingArea.

The standard Label widget knows how to be a drag source without any help from the application. However, the standard DrawingArea does not know how to be a drop site. Therefore, *simple_drop* provides both of the following procedures in order to add pixmap drop support to the DrawingArea:

- An **XmNdestinationCallback** procedure
- A transfer procedure

The only drop target that our DrawingArea supports is **PIXMAP**. The Label widget in the application provides a convenient **PIXMAP** source. In addition, you can drop a pixmap from any widget on the screen that knows how to convert a **PIXMAP** target.

The *main* routine instantiates the Label and DrawingArea widgets. *main* also calls **XmeDropSink** to establish the DrawingArea as a drop site.

```
int
main(int argc, char **argv)
{
 static Widget  MainWindow;
 XtAppContext   app_context;
 Widget         Frame1, RC1, Label1, DrawingArea1;
 Pixmap         pixmap;
 GC             gc;

   toplevel = XtAppInitialize(&app_context, "Test", NULL, 0,
                              &argc, argv, NULL, NULL, 0);

   MainWindow = XtVaCreateManagedWidget("MainWindow1",
                                  xmMainWindowWidgetClass, toplevel,
                                      NULL);

   CreateMenus(MainWindow);

   RC1 = XtVaCreateManagedWidget("RC1", xmRowColumnWidgetClass,
                                 MainWindow, NULL);

   if (!argv[1]) {
        printf("usage: %s bitmap-file\n", *argv);
        exit(1);
   }

   /* Load bitmap given in argv[1] */
    pixmap = XmGetPixmap(XtScreen(toplevel), argv[1],
        BlackPixelOfScreen(XtScreen(toplevel)),
        WhitePixelOfScreen(XtScreen(toplevel)));

    if (pixmap == XmUNSPECIFIED_PIXMAP) {
        printf("can't create pixmap from %s\n", argv[1]);
        exit(1);
    }

  /* Now instantiate an XmLabel widget that displays pixmap. */
   Label1 = XtVaCreateManagedWidget("Label1",
        xmLabelWidgetClass, RC1,
        XmNlabelType,   XmPIXMAP,
        XmNlabelPixmap, pixmap,
        NULL);

   Frame1 = XtVaCreateManagedWidget("Frame1",
        xmFrameWidgetClass, RC1,
        XmNshadowThickness, 3,
        NULL);

   DrawingArea1 = XtVaCreateManagedWidget("DrawingArea1",
        xmDrawingAreaWidgetClass, Frame1,
        XmNwidth, 150,
        XmNheight, 150,
        XmNresizePolicy, XmRESIZE_NONE,
```

```
        NULL);
    XmeDropSink(DrawingArea1, NULL, 0);
    XtAddCallback(DrawingArea1, XmNdestinationCallback,
                  DestinationCallback, NULL);

    XtRealizeWidget(toplevel);
    XtAppMainLoop(app_context);
}
```

When the user attempts a drop on the DrawingArea, UTM will call the *DestinationCallback* routine. This routine will call **XmTransferValue**, asking the source widget (the drag initiator) to convert its list of **_MOTIF_EXPORT_TARGETS**.

```
void
DestinationCallback(Widget  w,
                    XtPointer ignore,
                    XtPointer call_data)
{
 XmDestinationCallbackStruct *dcs =
          (XmDestinationCallbackStruct *)call_data;
 Atom TARGETS = XInternAtom(XtDisplay(w), "TARGETS", False);
 Atom _MOTIF_EXPORT_TARGETS = XInternAtom(XtDisplay(w),
                              "_MOTIF_EXPORT_TARGETS", False);

 /* Ask the source to return a list of all the export targets that
    it knows how to convert. */
 XmTransferValue(dcs->transfer_id, _MOTIF_EXPORT_TARGETS,
                 (XtCallbackProc)TransferProc, NULL, NULL);
}
```

UTM calls the *TransferProc* routine when the source widget has finished converting **_MOTIF_EXPORT_TARGETS**. If the source widget knows how to convert the **PIXMAP** target, the *TransferProc* routine will request it. When the source widget finishes converting **PIXMAP**, the *TransferProc* routine will paste a copy of the returned pixmap into the DrawingArea.

```
void
TransferProc(Widget  w,
             XtPointer ignore,
             XtPointer call_data)
{
 XmSelectionCallbackStruct *scs =
       (XmSelectionCallbackStruct *) call_data;
 Atom TARGETS = XInternAtom(XtDisplay(w), "TARGETS", False);
 Atom _MOTIF_EXPORT_TARGETS = XInternAtom(XtDisplay(w),
                              "_MOTIF_EXPORT_TARGETS", False);
 Atom PIXMAP = XInternAtom(XtDisplay(w), "PIXMAP", False);
 Atom  *targets = (Atom *)scs->value;
 int    PIXMAP_is_supported = 0;
 unsigned long    n;
 Widget Label2;

   if ((scs->target == _MOTIF_EXPORT_TARGETS) && (scs->type == XA_ATOM))  {
     printf("Number of supported targets is %ld\n", scs->length);
     ListAllTheTargets(w, targets, scs->length);
     for (n=0; n<=scs->length; n++)  {
  /* Look through list of returned TARGETS to see if PIXMAP is there. */
        if (targets[n] == PIXMAP)  {
          PIXMAP_is_supported = 1;
        }
     }

     if (PIXMAP_is_supported)  {
       XmTransferValue(scs->transfer_id, PIXMAP,
                   (XtCallbackProc)TransferProc, NULL, NULL);
     }
   }
```

```
  if ((scs->target == PIXMAP)) {
    Pixmap        transferred_pixmap = *(Pixmap*) scs->value;
    Pixmap        copy_of_transferred_pixmap;
    Window        root_return;
    int           x, y;
    unsigned int  width, height;
    unsigned int  border_width, depth;
    GC            gc;
    XtGCMask      valueMask;
    XGCValues     values;

    printf("TransferProc: source has transferred the PIXMAP.\n");
    printf("We need to paste it into the DrawingArea.\n");

  /* It is better to display a copy of the returned pixmap than
     to display the returned pixmap itself. The following code
     creates the copy. */
    XGetGeometry(XtDisplay(w), (Drawable)transferred_pixmap,
                 &root_return, &x, &y, &width, &height,
                 &border_width, &depth);
    copy_of_transferred_pixmap =
                 XCreatePixmap(XtDisplay(w), XtWindow(w), width,
                               height, depth);
    valueMask = GCFunction;
    values.function = GXcopy;

    gc = XtGetGC(w, valueMask, &values);
    XCopyArea(XtDisplay(w), transferred_pixmap,
              copy_of_transferred_pixmap, gc, x, y,
              width, height, x, y);

    Label2 = XtVaCreateManagedWidget("Label2", xmLabelWidgetClass, w,
        XmNlabelType,   XmPIXMAP,
        XmNlabelPixmap, copy_of_transferred_pixmap,
        NULL);
    XtReleaseGC(w, gc);
    XmTransferDone(scs->transfer_id, XmTRANSFER_DONE_SUCCEED);
  }
}
```

Extending a Widget to Support Drag

Any widget that provides an **XmNconvertCallback** resource has the potential to be a drag source. However, not all the potential drag sources are actual drag sources. For example, **XmScrollBar** supports an **XmNconvertCallback** resource (through its superclass, **XmPrimitive**) but does not provide any code to become a drag source. However, your application can turn an **XmScrollBar** into a drag source. In fact, that is just what the sample program, *simple_drag* does. This program is stored in `simple_drag.c`.

The normal action for Button 2 Press has been overridden to cause it to call the *StartDrag* routine, which causes the drag to begin. The program allows only the Copy operation, and will reply to requests for compound text.

When a drag is started on the ScrollBar, the default drag icons are used.

The *main* routine of the *simple_drag* program instantiates a ScrollBar and a Text widget. The *main* routine also alters the translations of the ScrollBar widget, which is detailed later on in this chapter.

```
int
main(int argc, char **argv)
{
 static Widget  MainWindow;
 XtAppContext   app_context;
 Widget         Frame1, RC1;
 Widget         Text1, ScrollBar1;
 char    dragTranslations[] = "#override <Btn2Down>: StartDrag()";
```

```
static  XtActionsRec  dragActions[] =
  {
   {"StartDrag", (XtActionProc)StartDrag}
  };
XtTranslations            parsed_xlations;


  toplevel = XtAppInitialize(&app_context, "Test", NULL, 0,
                             &argc, argv, NULL, NULL, 0);

  MainWindow = XtVaCreateManagedWidget("MainWindow1",
                                  xmMainWindowWidgetClass, toplevel,
                                      NULL);

  CreateMenus(MainWindow);


/* Create a RowColumn to contain the ScrollBar and Text widgets. */
   RC1 = XtVaCreateManagedWidget("RC1",
                                  xmRowColumnWidgetClass, MainWindow,
                                  NULL);

  /* Create a ScrollBar. */
   parsed_xlations = XtParseTranslationTable(dragTranslations);
   XtAppAddActions(app_context, dragActions, XtNumber(dragActions));
   ScrollBar1 = XtVaCreateManagedWidget("SB1",
                                  xmScrollBarWidgetClass, RC1,
                                  XmNorientation, XmHORIZONTAL,
                                  XmNtranslations, parsed_xlations,
                                  NULL);
 /* Associate a convert callback with the ScrollBar. */
  XtAddCallback(ScrollBar1, XmNconvertCallback, ConvertCallback, NULL);


 /* Create a text widget; it will be a potential drop site. */
  Text1 = XtVaCreateManagedWidget("Text",
                                  xmTextWidgetClass, RC1,
                                  XmNeditMode, XmMULTI_LINE_EDIT,
                                  XmNrows, 25,
                                  XmNcolumns, 25,
                                  NULL);

  XtRealizeWidget(toplevel);
  XtAppMainLoop(app_context);
}
```

When the user presses <Btn2Down>, *StartDrag* will be called. The *StartDrag* routine calls **XmeDragSource** as follows:

```
void
StartDrag(Widget w,
          XEvent *event)
{
 Arg       args[2];
 Cardinal  n=0;

  XtSetArg(args[n], XmNdragOperations, XmDROP_COPY); n++;
  XmeDragSource(w, NULL, event, args, n);
}
```

The *simple_drag* program provides an **XmNconvertCallback** procedure named *ConvertCallback*. This routine handles requests to convert targets. It knows how to convert the following targets:

- **MOTIF_EXPORT_TARGETS**
- **TARGETS**
- **COMPOUND_TEXT**

```
void
ConvertCallback(Widget  w,
                XtPointer ignore,
                XtPointer call_data)
{
 XmConvertCallbackStruct  *ccs = (XmConvertCallbackStruct *)call_data;
 int  *value;
 Atom COMPOUND_TEXT = XInternAtom(XtDisplay(w), XmSCOMPOUND_TEXT, False);
 Atom TARGETS = XInternAtom(XtDisplay(w), "TARGETS", False);
 Atom MOTIF_EXPORT_TARGETS =
   XInternAtom(XtDisplay(w), XmS_MOTIF_EXPORT_TARGETS, False);
 int n;

 /* XmeDragSource is going to call ConvertCallback and ask
    it to convert MOTIF_EXPORT_TARGETS. */
  if ( (ccs->target == MOTIF_EXPORT_TARGETS) ||
       (ccs->target == TARGETS))
    {
 /* We have to create a list of targets that ConvertCallback
    can convert. For simplicity, we are going to restrict
    that list to one target: COMPOUND_TEXT. */
     Atom *targs = (Atom *) XtMalloc(sizeof(Atom) * 1);
     if (targs == NULL) {
      ccs->status = XmCONVERT_REFUSE;
      return;
     }
     n = 0;
     targs[n] = COMPOUND_TEXT; n++;

     ccs->value = (XtPointer) targs;
     ccs->type = XA_ATOM;
     ccs->length = n;
     ccs->format = 32;
     ccs->status = XmCONVERT_DONE;  /* Yes, we converted the target. */
   }


 /* If the drop site supports COMPOUND_TEXT as an import target, then
    the drop site will ask ConvertCallback to convert the
    value to COMPOUND_TEXT format. */
   else if (ccs->target == COMPOUND_TEXT) {
     char     *passtext;
     char     *ctext;
     XmString  cstring;
     char      ValueAsAString[10];

  /* The part of the ScrollBar that we are transferring is its
     XmNvalue resource. */
     XtVaGetValues(w, XmNvalue, &value, NULL);

  /* Convert XmNvalue to COMPOUND_TEXT. */
       sprintf(ValueAsAString, "%d", value);
       cstring = XmStringCreateLocalized(ValueAsAString);
       ctext = XmCvtXmStringToCT(cstring);
       passtext = XtMalloc(strlen(ctext)+1);
       memcpy(passtext, ctext, strlen(ctext)+1);
  /* Assign converted string to XmConvertCallbackStruct. */
     ccs->value  = (XtPointer)passtext;
     ccs->type   = XA_STRING;
     ccs->length = strlen(passtext);
     ccs->format = 8;
     ccs->status = XmCONVERT_DONE;
   }
```

```
    else  {
      /* Unexpected target. */
      ccs->status = XmCONVERT_REFUSE;
    }
  }
```

The DropSite Registry

The DropSite registry contains information about widgets that have been registered as drop sites. Although the drag icon can be dropped anywhere on the screen, only widgets that have been registered as drop sites can accept information from the initiator. The receiver is the application controlling the current drop site.

The "sensitive area" is the part of the widget that responds to drag and drop. By default, the sensitive area is the whole widget. However, the application can specify that only part of the widget is sensitive.

Widgets that are drop sites can be stacked on each other, with one widget partially or completely within the boundary of another. The sensitive areas of lower drop sites are clipped if they are covered by a higher widget.

The stacking order of the widgets with drop sites can be changed by the application.

Protocols

The protocol describes how the initiator and receiver clients interact through the toolkit with each other.

Drag Protocols

There are two types of drag protocol:

Preregister

: Does not require messaging

Dynamic

: Requires messaging; this is the default and is the recommended drag protocol.

Applications can support either, both, or neither. At Motif Release 2.0, dynamic is the default; you should try to use this. The Motif toolkit automatically supports both unless a user or client sets resources to force the use of one or the other.

The user can specify which drag protocol to use when the client is the initiator or receiver. The application can specify drag protocol in an application−class defaults file. If neither the application nor the user specifies a protocol, the dynamic drag protocol is used.

The toolkit uses the requested protocols and the protocols allowed by the initiator and receiver clients to arrive at the protocol actually being used. Therefore, the protocol can change as the drag icon moves from window to window, depending on which protocols each window supports. If the initiator and receiver cannot agree on a protocol, static default drag−over effects are generated.

Even if no drag−over or drag−under visual effects are shown, a drop can still occur with the drop protocol, unless a client has specified that that window does not participate in drag and drop.

Drop Protocol

Motif Release 2.0 uses UTM as the drop protocol. See Chapter 16 for details.

Drag and Drop Widget Classes

Motif provides a number of Xt objects and widgets to encapsulate the underlying protocol; however, these are not mapped onto the screen:

XmDisplay

: An object that contains display−specific information, such as the initiator and receiver protocol styles.

XmScreen

An object that describes screen−specific information, such as font and default drag−over icons.

XmDragIcon

A widget that describes the pixmap, mask, and attachment of an icon. The source icon, state icon, operation icon, and the resulting blended drag icon are all Drag Icons.

XmDragContext

A widget that describes the resources specified by each drag initiator, such as target type, custom icons, custom colors, blending model, permitted operations, and callback routines for various situations encountered during the drag and drop transaction.

XmDropSite

A drop site database that maintains a registry of the resources unique to each drop site, such as animation for drag−under effects, valid target types and operations, and callback routines for situations encountered during a drag and drop transaction. It is not an Xt object, although it acts like one with respect to resource fetching.

XmDropTransfer

A widget that describes the information desired from the initiator client and the procedure used to process the results.

Drag and Drop Functions

Motif provides many functions to support drag and drop processing. However, many drag and drop functions were developed prior to UTM and, although not obsolete, no longer get much direct use. Application programmers will find the following drag and drop functions to be most useful:

XmeDragStart

Initiates a drag.

XmeDropSink

Establishes a widget as a drop site.

XmCreateDragIcon

Creates any of the parts of a drag icon (status icon, operation icon, or source icon) from a cursor or pixmap. This allows custom icons for all or part of the drag icon, rather than the default icons.

XmDragCancel

Cancels a drag that is in progress. This function is called when the user presses `osfCancel`.

XmDragStart

This function is typically called by **XmeDragSource** when the user asks to start a drag. This function creates a DragContext object, which is referenced by other functions whenever information about the drag initiator is needed. The **XmDragStart** function also calls the DragStart callback procedure, if one has been nominated with the **XmNdragStartCallback** resource of the **XmDisplay** widget.

XmDropSiteConfigureStackingOrder

Sets the order of overlapping drop sites. The default order is with the first−registered drop site on the bottom and the last−declared drop site on top.

XmDropSiteEndUpdate

Causes the **XmDropSiteUpdate** requests made after **XmDropSiteStartUpdate** to take place.

XmDropSiteQueryStackingOrder

Provides information about the stacking order of overlapping drop sites. The order can be changed with

XmDropSiteConfigureStackingOrder.

XmDropSiteRegister

Registers a drop site. Resources describing the drop site are defined. **XmeDropSink** calls **XmDropSiteRegister**.

XmDropSiteRetrieve

Retrieves the values of drop site resources.

XmDropSiteStartUpdate

Signals the toolkit to wait until **XmDropSiteEndUpdate** is called to process drop site changes requested by **XmDropSiteUpdate**. This provides a more efficient way to update several drop sites than changing them one at a time.

XmDropSiteUpdate

Updates drop site resources for a single drop site. If a series of **XmDropSiteUpdate** requests are surrounded by **XmDropSiteStartUpdate** and **XmDropSiteEndUpdate**, then the changes will be made all at once after the end update request.

XmDropSiteUnregister

Removes a drop site. After a drop site has been unregistered, it is unavailable as a destination for a drag.

XmDropTransferAdd

Adds additional transfer requests once a transfer has started.

XmDropTransferStart

Specifies what information should be requested from the drag initiator, and starts the process to get the information.

XmGetDragContext

Returns the DragContext ID associated with a particular time stamp.

XmGetXmDisplay

Returns the ID for the specified display.

XmGetXmScreen

Returns the ID for a specified screen. Some resources, such as the drag icons, are screen−specific.

XmTargetsAreCompatible

Checks if there are any matching targets between the initiator and destination to help determine the correct drag state.

Targets

Each drag source and drop site specifies what kinds of data types it can process, called targets. These targets are atoms, such as **XA_STRING**.

The DragContext resources **XmNexportTargets** and **XmNnumExportTargets** provide a list and number of the data types provided by the drag source. These are export targets.

The DropSite resources **XmNimportTargets** and **XmNnumImportTargets** provide a list and number of the data types accepted by the drop site. These are known as import targets. The primary purpose of the targets in **XmNimportTargets** is to provide visual feedback about the validity of the drop site. The **XmNdestinationCallback** procedure need not use the same targets as those specified by **XmNimportTargets**.

Any number of targets may be listed for each source and site. A drop site is considered valid for a particular drag if at least one of its targets matches any of the source's targets and if the source and drop site operations are compatible.

An application can define anything it wants as a target. Be aware, however, that other applications might not recognize that target.

Operations

There are three ways that the initiator and receiver can interact with each other:

- Data can be moved from the initiator to the receiver (Move).
- Data can be copied from the initiator to the receiver (Copy).
- Data can be linked from the receiver to the initiator (Link).

When a drag is started, the initiator provides a list of valid operations in the DragContext **XmNdragOperations** resource. When a drop site is registered, the receiver provides a list of operations it supports in the DropSite **XmNdropSiteOperations** resource. These lists are the values **XmDROP_MOVE**, **XmDROP_COPY**, or **XmDROP_LINK**, connected by the bitwise OR operator (|). For example, the following value means that Move and Copy are valid operations, but Link is not:

```
XmDROP_MOVE | XmDROP_COPY
```

The value **XmDROP_NOOP** indicates that there are no operations possible for a drop at the current site.

The user can specify an operation by using key combinations discussed earlier in this chapter. The user can also change the operation at any time until the drop starts.

The initiator and the receiver need to be able to handle all the operations their application supports. If the operation is Move, the receiver first gets a copy of the data, then tells the initiator that it can delete the data. If the operation is Copy, both applications have the data, making two copies of it. If the operation is Link, there is only one copy of the data, and the receiver establishes a link to that copy.

Drop Site Status

The drag and drop callbacks for both receiver and initiator contain a *dropSiteStatus* field. This field is initialized and maintained by the receiver through the toolkit, although the receiver's drag and drop procedures can update it if they wish. This field is used by the toolkit to determine what drag–over and drag–under visual effects to use.

The *dropSiteStatus* field indicates the relationship of the drag source to the drop site over which the drag icon is located:

XmDROP_SITE_VALID

A drop can take place. There is at least one matching target and operation between the drag source and the drop site.

XmDROP_SITE_INVALID

A drop cannot take place. Either there were no matching targets, no matching operations, or the receiver's **XmNdragProc** or **XmNdropProc** discovered some other problem that would make a drop impossible.

XmNO_DROP_SITE

The drag icon is not over a drop site.

If the toolkit on the receiver's side has set either the *operation* or the *operations* field to **XmDROP_NOOP**, it also sets the *dropSiteStatus* field to **XmDROP_SITE_INVALID**.

Drag and Drop Protocols

The protocols refer to how the initiator and receiver clients use the toolkit to communicate with each other. There are two drag protocol styles that are available. The drop protocol is based on the Xt selection protocol.

Drag Protocols

The toolkit on the initiator side is in charge during the drag. The protocol in effect determines how it will find the information about drop sites that it needs to manage visuals, and how extensively the initiator and receiver clients are involved during the drag.

There are two kinds of drag protocol styles:

Dynamic

Uses messages from the toolkit to the receiver to find out drop site information. The toolkit on the receiver side can reply to the messages, or the application can take action based on these messages. The receiver manages the drag−under effects.

Preregister

Stores drop site information in a database when the drop site is registered. The receiver is not involved in the drag until a drop occurs. All drag −over and drag−under visual effects are managed by the toolkit on the initiator side.

The code for the initiator is the same regardless of the protocol. The code for the receiver applications is the same except that in the dynamic mode, or in the preregister mode when the initiator and receiver are the same client, the receiver's **XmNdragProc** is called.

The drag protocol in use can change during the course of a drag. When the drag icon enters or leaves a top−level window, the source and potential drop receiver negotiate a mutually acceptable drag protocol, as described in Section 17.3.2.

The toolkit uses pixmap source icons if the client provides them. If not, the toolkit uses bitmap source icons if the client provides them. If the client provides neither, the toolkit uses **XmScreen** icons. The **XmScreen** icons can be either the default icons or ones provided by the client or user.

The Dynamic Drag Protocol

With the dynamic drag protocol, the initiator and receiver communicate with messages through the toolkit.

As the drag icon moves within the receiver's window, messages are sent from the toolkit on the initiator side to the toolkit on the receiver side. Based on these messages, the receiver determines whether the drag icon is entering, within, or leaving a drop site. Although the toolkit on the receiver side initializes state and operation information, the receiver can check and update this information further if it registers an **XmNdragProc** for the drop site. The initiator receives the updated message in one of its drag−related callbacks (described later in this chapter), and can take action accordingly.

The dynamic drag protocol allows the receiver to provide more sophisticated visual effects using the **XmNdragProc** than the toolkit can provide alone.

With the dynamic drag procotol

- The receiver can provide custom drag processing and drag−under visual effects.
- The drag icon can be any size supported by the system on which the application is running.

The dynamic protocol is the default for drag and drop operations.

The Preregister Drag Protocol

When a receiver supports the preregister protocol, the toolkit on the receiver side stores drop site information in a database. The toolkit on the initiator side manages all drag−under effects based on the information in the drop site database. By setting some DropSite resources appropriately, the receiver can have the toolkit use different highlighting or pixmaps, but the receiver only participates directly in the drag−under effects when it is the same as the initiator and an **XmNdragProc** procedure has been registered with the drop site.

With the preregister protocol

- The server is grabbed.
- The only customization a receiver can perform is providing custom values for the DropSite visual resources.
- The drag icon can be any size supported by the system on which the application is running.

Choosing the Protocol and Visual Style

The user can specify which drag protocol to use or the application can specify the drag protocol in resource file.

The preregister drag protocol can be used with a minimum of additional coding in an application, because the toolkit manages the drag–over visual effects using the default drag icons specified in the **XmScreen** object. Or the application can override the default **XmScreen** icons with custom icons, but still allow the toolkit to manage the effects.

The dynamic drag protocol requires more work for the application program, but allows a receiver application to provide visual effects beyond the capabilities of the toolkit.

The drag protocol in use has an effect on the system performance as described later in this section.

Specifying Drag Protocols

Two Display resources specify which protocol the toolkit should try to use when a client is an initiator or receiver. These resources can be set by the client in a resource file or by the user.

- **XmNdragInitiatorProtocolStyle**
- **XmNdragReceiverProtocolStyle**

These resources can take the following values (the letter in brackets following the value is used in Table 17–1):

XmDRAG_NONE [N]

Does not participate in drag and drop. There are no drag–under effects. The drag–over effects depend on the value of **XmNdragInitiatorProtocolStyle**.

XmDRAG_DROP_ONLY [X]

Does not support either the preregister mode or the dynamic mode, but does data transfer after the drop occurs. There are no drag–over or drag –under visual effects.

XmDRAG_PREREGISTER [P]

Supports the preregister mode only. The visual effects are managed by the toolkit.

XmDRAG_PREFER_PREREGISTER [PP]

Supports both protocols, but prefers the preregister protocol. This is the default for receivers. The visual effects are determined by the protocol actually used.

XmDRAG_PREFER_RECEIVER [R]

Used by initiators only. Uses the protocol that the receiver specifies. This is the default for initiators. The visual effects are determined by the protocol actually used.

XmDRAG_PREFER_DYNAMIC [PD]

Supports both protocols, but prefers the dynamic mode. The visual effects are determined by the protocol actually used.

XmDRAG_DYNAMIC [D]

Supports the dynamic protocol only. The drag–over and drag–under visual effects are managed by the clients.

For example:

```
myclient*dragInitiatorProtocolStyle: DRAG_PREFER_DYNAMIC
myclient*dragReceiverProtocolStyle: DRAG_PREFER_DYNAMIC
```

If the initiator and receiver have specified the same protocol, that protocol is used. If they specify different protocols, the protocol that is used is shown in the following table.

Table 1 Initiator and Receiver Protocols

Initiator Protocol	*Receiver Protocol*				
	P	**PP**	**PD**	*D*	*X*
P	P	P	P	X	X
PP	P	P	P	D	X
R	P	P	D	D	X
PD	P	D	D	D	X
D	X	D	D	D	X
X	X	X	X	X	X
N	N	N	N	N	N

The **XmGetXmDisplay** function returns the Display object ID associated with a specific display. **XtGetValues** can be used to check the protocol style resources.

If an **XmNdragProc** is specified for a drop site, it will be performed only if the protocol is dynamic, or, if the protocol is preregister, only if the initiator and receiver are the same client. In this case, the application should set the **XmNdragReceiverProtocolStyle** resource to the value **XmDRAG_PREFER_DYNAMIC** in the application−class defaults file rather than use the default value.

Protocols and Visuals

When the resulting protocol is preregister, a preregister visual style is used, and the server is grabbed by the initiator. With the dynamic protocol, the initiator and the receiver exchange messages. The visual look of the two protocols is very similar. The most significant visual difference between the two protocols is associated with sophisticated drag−under effects, such as autoscroll drag, or any drag−under operation that depends on widget data not displayed at the initiation of the drag. These effects are only available with the dynamic protocol, or with the preregister protocol, under the condition that the initiator and the receiver are the same client.

When the resulting protocol is dynamic, a dynamic visual style is always used. The drag−over visual can be a pixmap with an arbitrary size whose depth and colormap are the same as those of the widget associated with the drag source. When a pixmap is used in dynamic mode, a shaped window is used to contain it. The pixmap, if desired, is specified in the DragContext **XmNsourcePixmapIcon** resource. Otherwise, the drag−over visual is implemented with the X cursor, which must be a bitmap, and often has limited size (use **XQueryBestSize** to find out the largest size available per screen). This cursor is specified using **XmNsourceCursorIcon**.

Consequently, when the resulting protocol is **XmDRAG_NONE** or **XmDRAG_DROP_ONLY**, the visual style depends upon the value of **XmNdragInitiatorProtocolStyle**. When it is **XmDRAG_DYNAMIC** or **XmDRAG_PREFER_DYNAMIC**, the dynamic visual style is used; otherwise, the preregister visual style is used.

Drop Protocol

When a drop is made, the receiver checks what action should happen:

- If the user requested help, the receiver should display a dialog explaining the consequences of a drop on the site and determine if the user wants to continue or cancel the drop.
- If the user requests a cancel from the help dialog or presses `osfCancel`, or if the receiver determines that the drop cannot continue, the receiver must call **XmTransferDone** to terminate the transfer. The *status* argument to **XmTransferDone** must be **XmTRANSFER_DONE_FAIL**.
- If the drop can continue normally, the receiver starts a transfer.

See the Chapter 16 for details on how UTM implements a drop.

Drop Receiver Responsibilities for Dragging

In the dynamic protocol, motion messages go first to the receiver client. The receiver evaluates the state of the drag and sends an updated message to the initiator, which then manages its drag−over visuals based on the results.

If the drag protocol in effect is preregister, the drop site information is put in the database as the drop sites are registered and the receiver client does nothing until a drop is made. All visual effects are handled by the toolkit.

If the drag protocol is dynamic, messaging begins when the pointer enters the window containing the drop site. The receiver is given the opportunity to provide additional processing in its **XmNdragProc**. If the protocol is preregister, the opportunity is still available, as long as the initiator and receiver are the same client. The **XmNdragProc**

- Receives messages when the drag icon enters or leaves the drop site, the operation changes, the drag icon is in motion, or the drag is cancelled.
- Provides information back to the toolkit about the state of the drag (valid drop site, invalid drop site, no drop site) and the operation to be performed when a drop is made.
- Manages any custom drag−under visual effects.

Establishing a Drop Site

The widgets shown in Appendix B are already registered as drop sites. An application must register any other widget it wants to use for a drop site, but only if the widget provides an **XmNdestinationCallback** procedure. A widget may be registered as only one drop site.

XmeDropSink registers a widget as a drop site, establishes callbacks to be used when a drag is made through the drop site or a drop is made in the drop site, and provides target and operation information. If the protocol is preregister, the information is stored in a database, which is read by the toolkit during the drag. If the drag protocol is dynamic, messaging is used to check for possible drop sites within a widget.

The optional **XmNdragProc** routine is executed only if the drag protocol is dynamic, or, if the protocol is preregister, only if the initiator and receiver are the same client. It is called in response to events during the drag, and allows the receiver to provide additional drag−under effects or additional drag processing.

The **XmNdropSiteOperations** resource lists all operations that the drop site will support, combined by the bitwise OR operations (|). For example, the default value

```
XmDROP_COPY | XmDROP_MOVE
```

means that Copy and Move are valid operations, but Link is not. During a drag, the toolkit on the receiver side compares this list with the DragContext's **XmNdragOperations** list and the user−selected operation to arrive at the operation that will be performed if a drop occurs on this site, along with a list of all operations possible between the initiator and the current drop site.

If an application wants to use only one operation, such as Copy, it should set the **XmNdropSiteOperations** field to just that operation to ensure that the toolkit chooses the correct operation and drag icon during the drag and drop transaction.

Drop sites that represent copying devices, such as printers, or transformation devices, such as compilers, should perform a Copy rather than a Move if both are possible.

The **XmNdropSiteActivity** resource indicates whether the drop site is available for use:

XmDROP_SITE_ACTIVE

The drop site is available for use. This is the default value.

XmDROP_SITE_INACTIVE

The drop site is not available for use. If the drag icon is moved over the drop site, both the icon and drop site act as if the icon were not over a drop site.

XmDROP_SITE_IGNORE

Ignore this drop site for all purposes.

Use the **XmDropSiteRegistered** function to determine whether a widget is already a registered drop site. The function accepts a widget as an

argument, and returns a Boolean value, which is TRUE if the widget is a registered drop site.

The **XmDropSiteUnregister** function removes a widget from the DropSite registry. Once a widget is unregistered, it displays no drag–under visual effects and cannot accept a drop.

The difference between an unregistered drop site and an inactive drop site is that the inactive drop site is still registered; it still uses memory, but does not engage in any drag and drop transactions. One use for inactive drop sites is to provide the correct clipping on overlapping drop sites. An unregistered drop site is no longer involved in the drag and drop system. It is the same as a widget that was never registered.

When the drag and drop is operating in preregister mode, and the receiving client is not the same as the initiating client, drop effects will be generated even if a drop site's widget is unmanaged. To avoid this, you can deactivate the drop site by calling **XmDropSiteUpdate** and setting **XmNdropSiteActivity** to **XmDROP_SITE_INACTIVE**. Specifying the resource as **XmDROP_SITE_IGNORE** at creation time will disable the drop site completely, producing some savings of computational overhead, since inactive drop sites are still used for clipping drag–under effects.

The following code registers a DrawingArea widget as a drop site. The only operation it will accept is Copy. The only import target it will accept is **COMPOUND_TEXT**. The other resources, including drag–under effects, are left at their default values.

```
DrawingArea = XmCreateDrawingArea(Frame, "Draw1", NULL, 0);
XtManageChild(DrawingArea);
Atom targets[2];
Arg  args[4];
int  n, nt;
Atom COMPOUND_TEXT = XInternAtom(XtDisplay(w), "COMPOUND_TEXT", False);

  /* Register the DrawingArea as a drop site */
  nt=0
  targets[nt++] = COMPOUND_TEXT;
  n = 0;
  XtSetArg(args[n], XmNdropSiteOperations, XmDROP_COPY); n++;
  XtSetArg(args[n], XmNimportTargets, targets); n++;
  XtSetArg(args[n], XmNnumImportTargets, nt); n++;
  XmeDropSink(DrawingArea, args, n);
  XtAddCallback(DrawingArea1, XmNdestinationCallback,
                DestinationCallback1, NULL);
```

Changing a Drop Site

The **XmDropSiteUpdate** function is used to change drop site resources for a single drop site. For multiple requests, **XmDropSiteStartUpdate** signals that a series of **XmDropSiteUpdate** requests will follow, and **XmDropSiteEndUpdate** ends the series and processes the requests at one time.

XmDropSiteUpdate can also be used to change the resource values of the widgets that register themselves as drop sites.

Specially Shaped Drop Sites

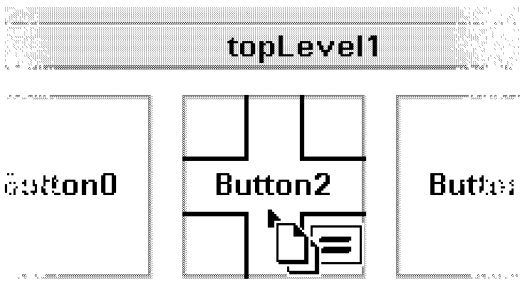
The shape of a simple drop site can be specified as the union of a set of specified rectangles clipped by the associated widget.

If only part of the widget is to be sensitive to a drop, it is defined by a list of rectangles in the **XmNdropRectangles** resource. If the resource is NULL, the drop site is the smallest enclosing widget and the shape of the drop site is the shape of the widget.

The rectangles that make up the drop site do not need to be contiguous. All the noncontiguous segments of the drop site act as one; they are all highlighted the same way at the same time. A drop on one segment is the same as a drop on any of the other segments. This might look to the user as if there were several drop sites on a single widget, but the application handles nested drop sites differently from drop sites made of noncontiguous segments. Nested drop sites, whether simulated or real, may have different drag–under effects, targets, operations, or callback procedures.

The following example establishes a sensitive area shaped like a plus sign on a DrawnButton widget named Button2. Even if the drag icon is within the Button2 widget, no drag–under effects are shown until the drag icon is within the sensitive area. The area is visible only when a drag icon enters it and highlighting occurs. The sensitive area is the only part of the widget that accepts a drop.

Figure 1 Specially Shaped Drop Site




```
XRectangle plus[] = {
    {30, 0, 30, 30},
    {0, 30, 90, 30},
    {30, 60, 30, 30},
    };
...
n = 0;
XtSetArg(args[n], XmNimportTargets, importList); n++;
XtSetArg(args[n], XmNnumImportTargets, 1); n++;
XtSetArg(args[n], XmNdropRectangles, plus); n++;
XtSetArg(args[n], XmNnumDropRectangles, 3); n++;
XmeDropSink(Button2, args, n);
```

Nested Drop Sites

A widget can be registered as only one drop site. However, widgets that are registered as drop sites can be nested within each other, providing nested drop sites.

The **XmNdropSiteType** indicates the complexity of the drop site:

XmDROP_SITE_SIMPLE

The drop site contains no other drop sites.

XmDROP_SITE_COMPOSITE

The drop site contains other drop sites. This value is generally associated with a Manager.

A composite drop site must be registered before any of its children are registered. If a composite drop site is inactive, so are all of its children.

The composite and children drop sites do not need to have the same operations or targets.

A manager that contains a number of widgets with their associated drop sites does not need to be a composite drop site unless it is possible to drop in the background of the manager.

It is possible for an application to simulate nested drop sites on a single widget, for example, a DrawingArea. The process is described as part of the discussion of the duties of the optional **XmNdragProc** routine in Section 15.4.2.

Overlapping Drop Sites

Drop sites can overlap. Their stacking order is assumed to correspond to the order in which they are registered, with the first−registered one on top. **XmDropSiteQueryStackingOrder** checks the stacking order, whereas **XmDropSiteConfigureStackingOrder** changes it.

When a drop site is overlapped by another drop site, the drag−under effects of the drop site underneath are clipped as appropriate by the obscuring drop site.

A widget or gadget that is not a drop site can overlap and partially obscure a drop site. To ensure that the drop−site's drag−under visuals are appropriately clipped by the obscuring widget, such sibling widgets should be registered as inactive drop sites. Parent widgets, whether drop sites or not, will clip their children's drop site visuals. If a parent has some active and some inactive drop site children, it should be registered as an active drop site.

Drag−Under Visual Effects

Drag−under visual effects are displayed only when the pointer is within the sensitive area of the drop site widget. An application can either handle all or none of the animation effects for a particular drop site. That is, an application should never do a partial job of animation on a particular drop site. Various drag−under styles can be chosen in the **XmNanimationStyle** DropSite resource:

XmDRAG_UNDER_HIGHLIGHT

A solid border around the sensitive area of the drop site is used to show the drop site is valid. This is the default value.

XmDRAG_UNDER_SHADOW_OUT

The sensitive area of the drop site looks pushed out when it is valid.

XmDRAG_UNDER_SHADOW_IN

The sensitive are of the drop site looks pushed in when it is valid.

XmDRAG_UNDER_PIXMAP

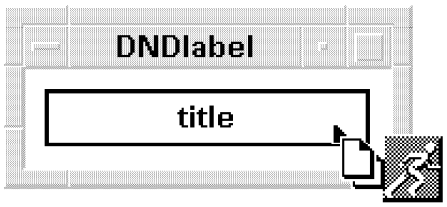
A custom pixmap is used to indicate the drop site is valid. The pixmap is specified in **XmNanimationPixmap**.

XmDRAG_UNDER_NONE

No indication is given that the drop site is valid.

The following illustration shows the default drag−under animation around the Label widget drop site.

Figure 2 Default Drag−Under Animation



If the value of **XmNanimationStyle** is **XmDRAG_UNDER_PIXMAP**, the resources **XmNanimationPixmap**, **XmNanimationMask**, and **XmNanimationPixmapDepth** are used to provide more information about the pixmap. If the depth does not match the depth of the window controlling the drop site widget, no animation occurs. Except for **XmDRAG_UNDER_PIXMAP**, the colors used for the visual effects are based on the colors of the widget associated with the drop site.

The dynamic protocol provides the most control over the drop site animation. It is the only way to get visual effects that do not remain the same for the duration of the drag icon's stay in the drop site; for example, a background that flashes.

XmNdragProc

The procedure registered in the DropSite's **XmNdragProc** resource is called only when the dynamic protocol is in effect, or, in the case of the preregister protocol, if the initiator and receiver are the same client. This procedure is optional. Applications that need to provide special drag −under effects or other special processing during a drag can do so with this procedure.

The **XmNdragProc** procedure is called in response to messages from the toolkit, before the initiator's equivalent drag callback. The pointer from the drop site's **XmNclientData** resource is passed as the client data to this procedure. Applications may use this resource to pass any manner of information desired to the **XmNdragProc** procedure. Fields in the callback structure provide information to the receiver about the drag.

The *reason* field in the callback structure indicates why the procedure was called.

XmCR_DROP_SITE_ENTER_MESSAGE

The drag icon hotspot has entered the drop site.

XmCR_DROP_SITE_LEAVE_MESSAGE

The drag icon hotspot has left the drop site.

XmCR_DRAG_MOTION_MESSAGE

The drag icon hotspot has moved.

XmCR_OPERATION_CHANGED_MESSAGE

The operation has changed.

The *operations* field lists all the operations that are valid for the drop site with the current drag source. The *operations* field is initialized by the toolkit as follows:

- If the user has selected an operation, the value of *operations* is initialized to that operation if it is in the DragContext's **XmNdragOperations** list.
- Otherwise, the *operations* field is initialized to the list in the DragContext's **XmNdragOperations** list.

The *operation* field indicates the type of action a successful drop will perform. The toolkit initializes the *operation* field by taking the following steps, in order of precedence from highest to lowest:

1. If Move is a valid operation (in both the *operations* field and the DropSite's **XmNdropSiteOperations** list), *operation* is initialized to **XmDROP_MOVE**.
2. If Copy is a valid operation, *operation* is initialized to **XmDROP_COPY**.
3. If Link is a valid operation, *operation* is initialized to **XmDROP_LINK**.
4. Otherwise, *operation* is initialized to **XmDROP_NOOP**.

The *dropSiteStatus* field provides an indication of whether a transfer between the initiator and this drop site could occur. The value that the toolkit selects for the *dropSiteStatus* field depends on the reason the **XmNdragProc** procedure was entered:

- If the reason is motion or drop site leave, and the drop site is the same as in the last call to **XmNdragProc**, the *dropSiteStatus* field is the same as at the end of the previous call.
- Otherwise, if there is at least one target in common and at least one operation in common, the value is initialized to **XmDROP_SITE_VALID**. If not, the value is initialized to **XmDROP_SITE_INVALID**.
- If the *operation* field is **XmDROP_NOOP**, the *dropSiteStatus* field is initialized to **XmDROP_SITE_INVALID**.

The **XmNdragProc** procedure can update *operation*, *operations*, or *dropSiteStatus* further during its execution. The final values for these fields are available to the initiator in its drag callback structures. If the receiver's **XmNdragProc** procedure is called more than once while the drag icon is within the drop site (for example, because of motion events), the values used by the toolkit when it initializes the drag callback *operations*, *operation*, and *dropSiteStatus* fields are the ones at the end of the previous call to **XmNdragProc**.

The *animate* field tells the toolkit who should provide the drag–under visual effects. It is initially set to True, but the **XmNdragProc** routine can set it to False.

True

The toolkit provides the drag–under visuals as if the protocol were preregister.

False

The receiver provides the drag–under visuals. The application can provide special visual effects, such as a blinking background, that are not possible with the toolkit.

The *DragProcCallback* routine in the `DNDDemo.c` program is an example of a *DragProc* routine. It can process every drag message, changes the *operations*, *operation*, and *dropSiteStatus* as necessary, and sets the *animate* field to **True**, allowing the toolkit to manage the drag–under effects. The *DragProcCallback* routine is shown in the next section of this chapter.

Simulating Nested Drop Sites

A widget can be registered as only a single drop site. However, if the application needs one or more drop sites entirely enclosed within another drop site, there are two ways to accomplish this:

- Widgets that contain other widgets that are drop sites should be registered as composite drop sites as described earlier in this chapter. This method allows the toolkit to manage drop site messages and drag–under effects for each nested drop site.

- An application can simulate multiple drop sites on a single widget in the **XmNdragProc** and **XmNdropProc** routines. Because the **XmNdragProc** routine is executed only in the dynamic drag protocol mode, this method would only work if the drag procotol chosen is preregister in the case where the initiator and receiver are the same client.
  This method requires that the application manage all drag–under effects, because the toolkit is not aware of the simulated nesting.

To simulate nested drop sites on a single widget:

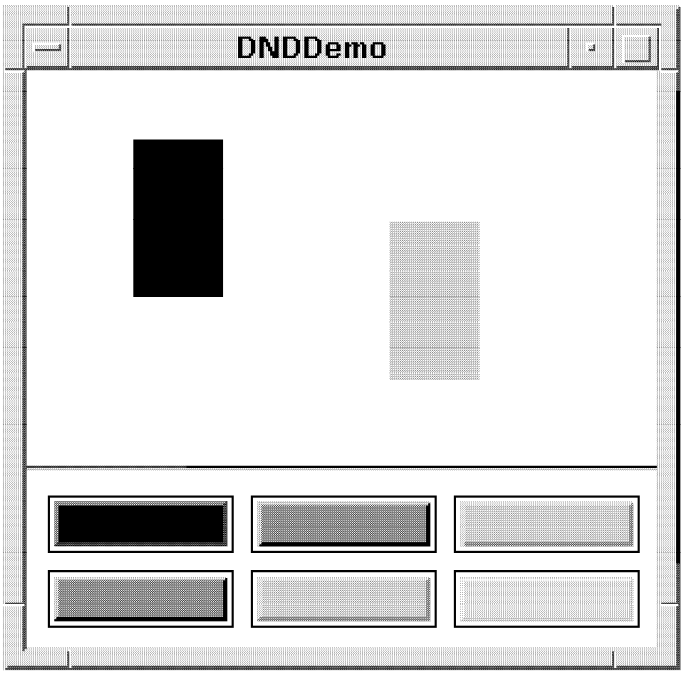
1. Register the widget as a single active drop site. Set **XmNdropSiteOperations** to all the operations possible for any of the nested drop sites. Set **XmNimportTargets** to all the targets possible for any of the nested drop sites. Register an **XmNdragProc** routine to provide any special drag –under effects for the simulated drop sites.
   The *operations*, *operation*, and *dropSiteStatus* fields are initialized by the toolkit only when this outer drop site is entered or left. The simulated drop sites must be managed by the application.
2. When either **XmNdragProc** or **XmNdropProc** is called, check the *x* and *y* fields in the callback structure to determine which of the nested drop sites contains the pointer.
3. If the pointer is within a simulated nested drop site, update the callback fields as follows:
   - When the pointer enters the simulated nested drop site, save the value of the *operations* and *operation* fields.
   - Remove any operations from the *operations* field that do not apply to the simulated drop site.
   - Set *operation* to the valid operation preferred by the simulated drop site, or to **XmDROP_NOOP** if the *operations* list does not contain the preferred operation.
   - The *dropSiteStatus* field must reflect the status of the simulated drop site so that the initiator can manage drag–over effects correctly:
     - Set the *dropSiteStatus* to **XmDROP_SITE_VALID** if the *operation* is allowed in the simulated drop site and if there is at least one target in common between the simulated drop site and the initiator. (Use the **XmTargetsAreCompatible** routine to check the targets.)
     - Set the *dropSiteStatus* to **XmDROP_SITE_INVALID** if the *operation* is not allowed in the simulated drop site, if there are no targets in common, or if the *operation* is **XmDROP_NOOP**.
   - Display appropriate drag–under visual effects.
   - When the pointer leaves the simulated drop site, restore the original values of *operations* and *operation* that apply to the outer drop site.
4. If the pointer is not within a simulated drop site, but drops are allowed in the outer drop site, update the fields as described in the previous step.
5. If the pointer is not within a simulated drop site, and drops are not allowed in the outer drop site, set the *dropSiteStatus* field to **XmDROP_SITE_INVALID**.

If the preregister protocol is in effect, the simulated drop sites cannot be managed during the move, because **XmNdragProc** is not performed; but they can be managed at the drop with **XmNdropProc**.

In the following example, only the top–level window, *DNDDemo*, is registered as a drop site. The user can create rectangles within the window that then act like drop sites themselves. The user can drag and drop colors from one of the six buttons in the lower part of the window onto the rectangles to change the color of the rectangle. However, these rectangles are not registered drop sites; they are simulated.

The user can also drag these rectangles to new locations.

Figure 3 Simulated Drop Sites



The *RegisterDropSite* routine registers the DrawingArea widget as a drop site. The list of operations and targets may not be valid for each simulated drop site, but are valid for other simulated drop sites.

```
static void
RegisterDropSite(Widget w)
{
    Display *display = XtDisplay(w);
    Atom    targets[3];
    Arg     args[5];
    int     n = 0;

    /* Only accept moves or copies */
    XtSetArg(args[n], XmNdragOperations, XmDROP_COPY | XmDROP_MOVE); n++;

    /* set all possible targets for any of the nested drop sites */
    targets[0] = XmInternAtom(display, "_MY_RECTANGLE", False);
    targets[1] = XmInternAtom(display, "BACKGROUND", False);
    targets[2] = XmInternAtom(display, "PIXMAP", False);
    XtSetArg(args[n], XmNimportTargets, targets); n++;
    XtSetArg(args[n], XmNnumImportTargets, 3); n++;

    /* register a dragProc - necessary for simulating nested drop sites */
    XtSetArg(args[n], XmNdragProc, DragProcCallback); n++;
    XmeDropSink(w, args, n);
}
```

The **XmNdragProc** routine, *DragProcCallback*, is called whenever a drag icon enters the registered drop site (the top–level window). The *RectFind* routine from `DNDDraw.c` determines if the pointer is in a simulated drop site. The *CheckTargets* routine determines if the object being dragged is one of the six colors (*bgFound*) or one of the created rectangles (*rectFound*). (The value *pixFound* to represent a pixmap being dragged is coded in this routine, but not in the rest of the program.)

The only drag–under visual is displayed when a color is dragged to a rectangle. The outline of the rectangle is highlighted.

The entire *DragProcCallback* routine is too long to be listed in its entirety here. The section dealing with the drop site enter message is used as an example.

```
static void
DragProcCallback(Widget w, XtPointer client, XtPointer call)
{

    XmDragProcCallbackStruct    *cb = (XmDragProcCallbackStruct *) call;
    Display                     *display = XtDisplay(w);
    Boolean                     rectFound, bgFound, pixFound;
    static unsigned char        initial_operations;
    static unsigned char        initial_operation;
    RectPtr                     rect;

    CheckTargets(cb->dragContext, display, &rectFound, &bgFound, &pixFound);

    switch(cb->reason) {

        case XmCR_DROP_SITE_ENTER_MESSAGE:

            /* save the value of the operations and operation fields */
            initial_operations = cb->operations;
            initial_operation = cb->operation;

            rect = RectFind(cb->x, cb->y);

            /* Remove any operations for the operations field which do not
             * apply to the simulated drop site.
             */
            if (rect) {

                if (bgFound || pixFound) {
                    cb->operations = XmDROP_COPY;
                    RectHighlight(w, rect);
                }
                else if (rectFound) {
                    cb->operations = cb->operations &
                                     (XmDROP_COPY | XmDROP_MOVE);
                    RectUnhighlight(w);
                }

            }
            else {
                cb->operations = initial_operations &
                                 (XmDROP_COPY | XmDROP_MOVE);
                RectUnhighlight(w);
            }

            /* Set operation to the valid operation preferred by the simulated
             * drop site or to XmDROP_NOOP if the operations list does not
             * contain the preferred operation.
             */
            if (rect) {

                if (bgFound || pixFound) {

                    if (cb->operations & XmDROP_COPY)
                        cb->operation = XmDROP_COPY;
                    else
                        cb->operation = XmDROP_NOOP;

                }
                else if (rectFound) {
```

```
            if (cb->operations & XmDROP_MOVE)
                cb->operation = XmDROP_MOVE;
            else if (cb->operations & XmDROP_COPY)
                cb->operation = XmDROP_COPY;
            else
                cb->operation = XmDROP_NOOP;

        }

    }
    else {

        if (rectFound) {

            if (cb->operations & XmDROP_MOVE)
                cb->operation = XmDROP_MOVE;
            else if (cb->operations & XmDROP_COPY)
                cb->operation = XmDROP_COPY;
            else
                cb->operation = XmDROP_NOOP;

        }
        else
            cb->operation = initial_operation;

    }

    /*
     * Set dropSiteStatus to XmDROP_SITE_INVALID if the operation
     * field is XmDROP_NOOP, or if there are no common targets
     * between the source and the nested drop site. Otherwise, set
     * dropSiteStatus to XmDROP_SITE_VALID.
     */
    if (cb->operation == XmDROP_NOOP ||
        (rect && (!rectFound && !bgFound && !pixFound)) ||
        (!rect && !rectFound))
        cb->dropSiteStatus = XmINVALID_DROP_SITE;
    else
        cb->dropSiteStatus = XmVALID_DROP_SITE;

    /*
     * Display appropriate drag under visuals. Only highlight
     * the rectangle if we are changing rectangle attributes.
     */
    if (rect && bgFound || pixFound &&
        cb->dropSiteStatus == XmVALID_DROP_SITE)
        RectHighlight(w, rect);
    break;

case XmCR_DROP_SITE_LEAVE_MESSAGE:
    ...
```

Autoscrolling (or Pause) Drag

Using the **XmNdragProc** routine, drag and drop can support *pause drag*. This is when, during a drag, a dragged object is held over a drop site (without releasing), with some drag−under effect resulting. The most common form of this is when a scrolled window will scroll in some direction while a dragged object is held over the border of the window, or in the scrollbar.

Since this feature depends on the **XmNdragProc** routine, it is only available with the dynamic protocol, or, with the preregister protocol, when the initiator and receiver share the same top−level shell.

This feature is fully implemented in the Motif ScrolledWindow widget. That widget has a resource called **XmNautoDragModel**, which, when set to **XmAUTO_DRAG_ENABLED**, will allow autoscroll drag. This is the default setting. The scrolling characteristics are set with the

XmNinitialDelay, **XmNrepeatDelay**, and the **XmNincrementFactor** resources of that widget. Please refer to the documentation for **XmScrolledWindow** in the *Motif Programmer's Reference* for more information about these resources.

Drag Initiator Responsibilities for Dragging

This section explains what an application can do on behalf of the widget that initiates the drag. The drag initiator can do any of the following:

- If a widget does not already contain drag code, an application must recognize the start of a drag (`Btn2Down`) within a widget controlled by the application.
- If an application wants a widget to act as a drag source, the application must attach an **XmNconvertCallback** to the widget.
- If a widget does not already contain drag code, an application can establish a DragContext for the widget, providing information about operations, targets, and drag–over visuals, using the **XmeDragSource** function. If a widget does already contain drag code, an application can customize the DragContext.
- Optionally, an application can provide special drag–over effects.

These steps are described in the following sections.

If the user tries to drag objects from a widget that is not recognized as a drag source by either the toolkit or the source application, nothing happens.

Recognizing a Drag Has Started

The initiator client must be able to recognize the `Btn2Down` event within a widget it allows to be a drag source. It may have to override an already–assigned translation for the widget.

The following example from the main routine of *simple_drag* in Section 17.2.1.2 overrides the existing mouse button 2 translation for the ScrollBar widget, and maps it to the *StartDrag* routine, which will start the drag transaction.

```
char    dragTranslations[] = "#override <Btn2Down>: StartDrag()";
static  XtActionsRec  dragActions[] =
   {
    {"StartDrag", (XtActionProc)StartDrag}
   };
XtTranslations            parsed_xlations;
...
/* override button two press to start a drag */
parsed_xlations = XtParseTranslationTable(dragTranslations);
XtAppAddActions(app_context, dragActions, XtNumber(dragActions));

/* create a scroll bar widget */
    ScrollBar1 = XtVaCreateManagedWidget("SB1",
        xmScrollBarWidgetClass, RC1,
        XmNorientation, XmHORIZONTAL,   /* for cosmetic reasons */
        XmNtranslations, parsed_xlations,
        NULL);
```

Translation may be more complicated in some editable widgets, in which `Btn2DownBtn2Up` is used for primary transfer, and `Btn2Motion` is used for drag and drop.

Establishing an XmNconvertCallback Procedure

If your widget is to act as a drag initator, it must provide an **XmNconvertCallback** procedure that converts target requests. Your application must attach an **XmNconvertCallback** to the drag initiator widget; for example:

```
/* Associate a convert callback with the ScrollBar. */
  XtAddCallback(ScrollBar1, XmNconvertCallback, ConvertCallback, NULL);
```

The **XmNconvertCallback** procedure must be able to convert two kinds of requests:

- A request to convert **_MOTIF_EXPORT_TARGETS**. The callback procedure must respond to this request by providing a list of all the targets that the callback procedure knows how to convert. For example, in the in the *simple_drag* demo, the only target that the *ConvertCallback* procedure can convert the value to is **COMPOUND_TEXT**. Therefore, the list of targets that *ConvertCallback* returns consists solely of the **COMPOUND_TEXT** target.
- A request to convert all of the targets in the list returned by **_MOTIF_EXPORT_TARGETS**. For example, the *ConvertCallback* of *simple_drag* must be able to return the current ScrollBar **XmNvalue** in **COMPOUND_TEXT** format.

Starting a Drag With XmeDragSource

XmeDragSource initiates a drag and creates an **XmDragContext** widget. In the *simple_drag* demo, **XmeDragSource** is called as follows from the *StartDrag* routine:

```
void
StartDrag(Widget w,
          XEvent *event)
{
 Arg       args[2];
 Cardinal  n=0;

  XtSetArg(args[n], XmNdragOperations, XmDROP_COPY); n++;
  XmeDragSource(w, NULL, event, args, n);
```

StartDrag is called when the user preses `Btn2Down` on the **XmScrollBar**.

See the *Motif Widget Writer's Guide* for syntactic details about **XmeDragSource**. In brief, **XmeDragSource** is the UTM replacement for the **XmDragStart** call. The *args* passed to **XmeDragSource** will eventually become resources of the DragContext.

If **XmDisplay** has the **XmNdragStartCallback** resource, the nominated procedure is called at the beginning of **XmeDragSource**. This allows the client to cancel the drag initiation, as well as perform any other necessary operation.

The **XmNdragOperations** resource lists all the operations that the initiator will support for this drag source, combined by the bitwise OR operation (|). During a drag, the toolkit compares this list with the receiver's **XmNdropSiteOperations** list and the user−selected operation to arrive at the operation that will be performed if a drop occurs on this site.

If an application wants to use only one operation, it should set the **XmNdragOperations** resource to just that operation to ensure that the correct operation and drag icon are chosen by the toolkit during the drag and drop transaction.

This drag source does not have any custom drag icons or any drag callbacks.

Overriding Existing Drag Sources

XtGetValues is used to check the values of widgets resources established as drag sources earlier in the application, and **XtSetValues** is used to update these values. The widget ID used is the DragContext, not the source widget ID, so that the change applies only to the widget during the drag.

If the widget is a predefined drag source (for example, **XmText**), overriding the default behavior becomes more complex. The widget calls **XmeDragSource** when the drag starts, and the application cannot call **XmeDragSource** again for the widget. Instead, it must update the existing DragContext. First it must find the DragContext for the widget, then establish the new behavior. One possible means to accomplish this is as follows:

- Override the existing Btn2Down translation with a new translation that calls the widget's action and also an action supplied by the application. For the Text widget, this new translation might look as follows:
  ```
  <Btn2Down>: process-bdrag() my-drag-start()
  ```
- Register the new action, using **XtAppAddActions**.
- In the new action procedure, call **XmGetDragContext** to get the DragContext, and then call **XtSetValues** to change resource values. The timestamp argument to **XmGetDragContext** can be the timestamp from the event passed to the action routine.

For instance, Text allows the Copy and Move operations. If an application can support only Copy, it must update the DragContext's **XmNdragOperations** resource.

Drag−Over Visual Effects

When the user moves the mouse, a drag icon representing the object being dragged moves around the screen instead of the usual pointer. As the icon is dragged over portions of the screen, the icon may change to show the status of a possible drop. These drag−over visual effects help the user know how to proceed with the drag.

There are four ways to provide drag−over visual effects:

- Use the default drag−over visuals, specified in the **XmScreen** object. The toolkit manages all the drag−over effects.

- Put custom icons and pixmaps in the **XmScreen** visual resources to be used as default icons for all drag and drop transactions running on that **XmScreen**. The toolkit manages all the drag–over effects using these new icons. These resources can be modified by the application or the user in a resource file.
- Put custom icons and pixmaps in the DragContext visual resources for source, state, or operation icons. The application must monitor the state of the drag using the drag callbacks and update the DragContext icon values as necessary. The default icons specified in the **XmScreen** object are used only if the value for the equivalent DragContext visual resource is NULL.
- Manage the drag–over effects entirely in the application by drawing directly to the screen. The toolkit is not used, nor are the **XmScreen** and DragContext visual resources.

If the application provides custom icons and they are unsuitable for some reason, the toolkit defaults to the **XmScreen** drag–over visuals.

The drag icon consists of a source icon, optionally combined with a state icon and an operation icon.

Each drag icon has a hotspot. Since a drag icon could be quite large, the hotspot provides a single pixel that is used in providing drag–over and drag–under effects. For example, if the drag icon moves into the area of a valid drop site, neither the drag icon or the drop site will provide visual clues until the hotspot has moved into the area. By default, the hotspot is the upper left corner of the state icon.

In the following illustration, the running figure is the source icon, the state icon is the arrow in the corner, and the operation icon shows a Copy will happen if a drop is made. The default blending and attachment values are used.

Figure 1 A Drag Icon

Source Icon

The source icon is a picture representing the object being dragged. It can be either a pixmap or cursor. The client can create a custom pixmap through the **XmNsourcePixmapIcon** resource or a custom cursor through the **XmNsourceCursorIcon** resource. If these resources are NULL or not usable (too large, not a bitmap, or created on a different screen, for example), the default cursor given in the **XmScreen** resource **XmNdefaultSourceCursorIcon** is used.

For example, the following code establishes the *Pixmap* stored in the *pixmap* variable as the custom pixmap:

```
    /* Create a drag icon using "pixmap" as the DragIcon pixmap. */
      XtSetArg(wargs[n],XmNpixmap,pixmap); n++;
      drag_icon = XmCreateDragIcon((Widget)cw,"dragIcon",wargs,n);

    /* Set the XmNsourcePixmapIcon resource to the drag icon widget */
      n = 0;
      XtSetArg(wargs[n],XmNsourcePixmapIcon,drag_icon); n++;
```

The pixmap icon is used with the preregister visual style. The colormap is based on the source widget. The cursor icon is used for the dynamic visual style.

The following illustration shows the default source icons for general purpose, List, Label, and Text widgets.

Figure 2 Source Icons

State Icon

The state icon is a cursor that indicates if the drag is over a valid drop site, invalid drop site, or no drop site. The default state icons are in the **XmScreen** resources **XmNdefaultValidCursorIcon**, **XmNdefaultInvalidCursorIcon**, and **XmNdefaultNoneCursorIcon**.

A custom state icon can be specified in the DragContext resource **XmNstateCursorIcon**. This icon must be changed appropriately as the state of the drag changes, using the drag callbacks to change the CursorForeground resources outlined below. If **XmNstateCursorIcon** is NULL, not a bitmap, or not defined on the same screen as **XmScreen**, the default **XmScreen** icons are used.

The default state icon for all three states is an arrow, usually shown at the upper left corner of the operation icon.

Three DragContext resources can be used to change the color of the drag icon based on the state of the drag: **XmNvalidCursorForeground**, **XmNinvalidCursorForeground**, **XmNnoneCursorForeground**. This allows visual feedback about the drag to the user, without changing the icon shape. For example, the following lines in a resource file would make the drag icon green when it was over a valid drop site, red when it was over an invalid drop site, and yellow when it was not over any drop site:

```
*.validCursorForeground:          green
*.invalidCursorForeground:        red
*.noneCursorForeground:           yellow
```

Operation Icon

The operation icon is a cursor that indicates what operation is to happen when the drop is made. The default operation icons are the values of the **XmScreen** resources **XmNdefaultMoveCursorIcon**, **XmNdefaultCopyCursorIcon**, and **XmNdefaultLinkCursorIcon**.

A custom operation icon can be specified in the DragContext resource **XmNoperationCursorIcon**. The icon should be changed as the operation changes, using the drag callbacks. If **XmNoperationCursorIcon** is NULL, not a bitmap, or not defined on the same screen as **XmScreen**, the default **XmScreen** icons are used.

The following illustration shows the default Copy, Link, and Move operation icons.

Figure 3 Operation Icons

If the operation in effect is **XmDROP_NOOP**, meaning that no operation is possible, then the operation icon is left blank, as shown in the following illustration. This condition also sets the *dropSiteStatus* to **XmDROP_SITE_INVALID**.

Figure 4 Copy and Noop Drag Icons

Drag Icon Blending and Attachment

The client can specify which of the three icons to mix together to form the drag icon with the **XmNblendModel** DragContext resource:

XmBLEND_ALL

Use the source icon, state icon, and operation icon. The hotspot comes from the state icon. This is the default value. The order listed is also the order of the blend.

XmBLEND_STATE_SOURCE

Use only the source icon and state icon. The hotspot comes from the state icon.

XmBLEND_JUST_SOURCE

Use only the source icon. The hotspot comes from the source icon.

XmBLEND_NONE

Do not display any drag icon. The client handles all drag–over effects.

The **XmNattachment** DragIcon resource specifies where the state and operation icons will be placed on the source icon. The default placement is both the state and operation icons at the attachment point of the source icon, with the operation icon on top. The default value is **XmATTACH_NORTH_WEST**.

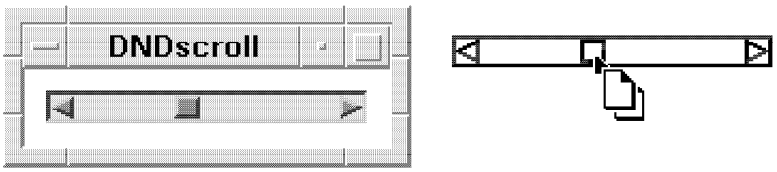
XmNoffsetX and **XmNoffsetY** are used to place the icon relative to the attachment point.

If the attachment point is **XmATTACH_HOT**, the state and operation icons are attached to the source icon at a point the same x and y distance from the upper left corner of the source icon as the pointer is from the upper left corner of the widget containing the source. This attachment style

is particularly useful when the application makes a custom source icon that exactly reflects the source widget at the time the drag starts.

In the following illustration, the custom source icon is an outline of the scrollbar. When the drag was started, the pointer was on the slider. The operation and state icons are placed at the same location on the source icon.

Figure 5 An Attach_Hot Icon



When the state or operation icon is blended with a source icon, a specified point of the icon's **XmNpixmap** is aligned with the upper left corner of the source icon. The resulting **XmNpixmap** is large enough to include both, and the resulting **XmNmask** has 1 bits wherever either the source icon or source mask did.

If a dynamic cursor style is being used, and the resulting blended cursor is too large for the screen, the blending is done with the **XmScreen XmNdefaultSourceCursorIcon** instead of the DragContext's **XmNsourceCursorIcon**. If it is still too large, it is clipped relative to the hotspot (that is, if the hotspot is at an edge, the other edge is clipped; if the hotspot is in the center, opposite edges are clipped equally).

Visual Style Notes

If **XmNsourcePixmapIcon** is used, the colormap used for rendering is that of the DragContext's reference widget.

If the DragContext **XmNblendModel** is **XmBLEND_NONE**, and the dynamic cursor style is in use, the application must use **XChangeActivePointerGrab** to change the cursor. If **XmBLEND_NONE** is specified, and the preregister cursor style is in use, the application can render the cursor directly onto the screen, saving and restoring the image underneath.

The cursor style can change as the pointer moves from window to window. An application can tell which style is in use by looking at the *dragProtocolStyle* field in the **XmNtopLevelEnterCallback** structure, or looking at the **XmNdragInitiatorProtocolStyle** Display resource in the case of **XmDRAG_NONE** or **XmDRAG_DROP_ONLY**.

The resolution and best cursor size can vary from screen to screen. This is why the default cursor icons are **XmScreen** resources. An application that wants its source cursor or pixmap to be screen dependent can look for changes in the *screen* field in the **XmNtopLevelEnterCallback** struct, and update the various icon DragContext resources appropriately.

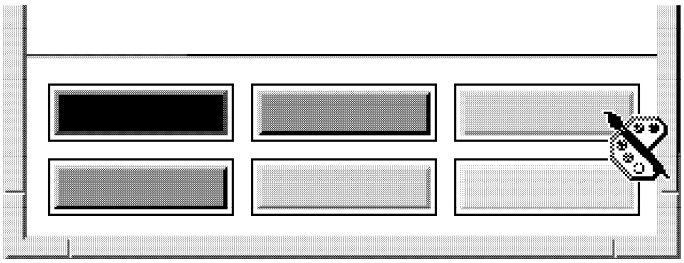
Creating a Drag Icon

Any of the three parts of a drag icon can be customized: the source icon, the state icon, and the operation icon.

Use the **XmCreateDragIcon** function to create any of these parts. The **XmNattachment** resource is not used for the source icon. The other resources specify pixmap, size, and hotspot details. The DragContext **XmNblendModel** resource indicates which hotspot is used for the entire drag icon.

The following example from `DNDDemo.c` creates a source icon from a bitmap. The source icon is the palette and the state icon is the paintbrush. (Actually, the state icon is not shown when the drag starts, because the blend style is **XmBLEND_JUST_SOURCE**. It is shown here as if the blend style were **XmBLEND_ALL**.)

Figure 6 Custom Source Icon



The *ColorRect* function is called when a drag starts from one of the color rectangles in the lower portion of the window. Among its other duties, it establishes the drag icon from source bits from the `DNDDraw.c` file.

```
void
ColorRect(Widget w, XEvent *event, String *params, Cardinal *num_params)
{
    ...
    Atom        targets[1];
    Widget      sourceIcon, stateIcon;
    Pixel       background, foreground;
    char        *source_bits, *source_mask;
    char        *state_bits, *state_mask;
    Dimension   width, height;
    Arg         args[16];
    int         n = 0;

    n = 0;
    XtSetArg(args[n], XmNbackground, &background); n++;
    XtSetArg(args[n], XmNforeground, &foreground); n++;
    XtGetValues(w, args, n);

    /* If the server will handle a large icon, create one */
    if (appInfo->maxCursorWidth >= ICON_WIDTH &&
        appInfo->maxCursorHeight >= ICON_HEIGHT) {

        source_bits = (char *)SOURCE_ICON_BITS;
        source_mask = (char *)SOURCE_ICON_MASK;
        state_bits = (char *)STATE_ICON_BITS;
        state_mask = (char *)STATE_ICON_MASK;
        width = ICON_WIDTH;
        height = ICON_HEIGHT;

    }
    else {

        /* If the server will handle a small icon, create one */
        source_bits = (char *)SMALL_SOURCE_ICON_BITS;
        source_mask = (char *)SMALL_SOURCE_ICON_MASK;
        state_bits = (char *)SMALL_STATE_ICON_BITS;
        state_mask = (char *)SMALL_STATE_ICON_MASK;
        width = SMALL_ICON_WIDTH;
        height = SMALL_ICON_HEIGHT;

    }

    /* Create the drag cursor icons */
    sourceIcon = GetDragIconFromBits(w, source_bits, source_mask, width,
                                     height, background, foreground);

    stateIcon = GetDragIconFromBits(w, state_bits, state_mask, width,
```

```
                                    height, background, foreground);

    /* Setup the arglist for the drag context that is created at drag start */
    n = 0;
    ...
    /* set args for the drag cursor icons */
    XtSetArg(args[n], XmNsourceCursorIcon, sourceIcon); n++;
    XtSetArg(args[n], XmNstateCursorIcon, stateIcon); n++;

    /* identify the necessary callbacks */
    ...
    /* start the drag. This creates a drag context. */
    myDC = XmeDragSource(w, NULL, event, args, n);

}
```

The *GetDragIconFromBits* function turns the bits into a bitmap.

```
static Widget
GetDragIconFromBits(Widget w, char *bits, char *mask, Dimension width,
                    Dimension height, Pixel background, Pixel foreground)
{

    Pixmap     icon, iconMask;
    Display    *display = XtDisplay(w);

    icon = XCreateBitmapFromData(display, DefaultRootWindow(display), bits,
                                 width, height);

    iconMask = XCreateBitmapFromData(display, DefaultRootWindow(display),
                                     mask, width, height);

    return(GetDragIcon(w, icon, iconMask, width, height,
                       background, foreground));

}
```

The *GetDragIcon* function uses the bitmap created by the *GetDragIconFromBits* function to create a drag icon.

```
static Widget
GetDragIcon(Widget w, Pixmap icon, Pixmap iconMask, Dimension width,
            Dimension height, Pixel background, Pixel foreground)
{

    Widget  dragIcon;
    Arg     args[10];
    int     n = 0;

    XtSetArg(args[n], XmNhotX, ICON_X_HOT); n++;
    XtSetArg(args[n], XmNhotY, ICON_Y_HOT); n++;
    XtSetArg(args[n], XmNwidth, width); n++;
    XtSetArg(args[n], XmNheight, height); n++;
    XtSetArg(args[n], XmNmaxWidth, appInfo->maxCursorWidth); n++;
    XtSetArg(args[n], XmNmaxHeight, appInfo->maxCursorHeight); n++;
    XtSetArg(args[n], XmNbackground, background); n++;
    XtSetArg(args[n], XmNforeground, foreground); n++;
    XtSetArg(args[n], XmNpixmap, icon); n++;
    if (iconMask != XmUNSPECIFIED_PIXMAP &&
        iconMask != None) {
      XtSetArg(args[n], XmNmask, iconMask); n++;
    }
    dragIcon = XmCreateDragIcon(w, "dragIcon", args, n);

    return(dragIcon);
}
```

Drag Callbacks

Callbacks notify the initiator of how the drag is proceeding. The receiver's **XmNdragProc** (if any) is first notified of the action and given a chance to update the *operation*, *operations*, and *dropSiteStatus* fields in its callback structure. The new values are available to the initiator's drag callback in the appropriate callback structure.

These drag callbacks are all optional. They enable the initiator to monitor the progress of the drag and manage its visual effects accordingly. Otherwise, the toolkit on the initiator side handles the drag–over effects.

XmNdragMotionCallback

Called when the drag icon is in motion

XmNoperationChangedCallback

Called when the user requests a different operation be performed on the drop than was previously in effect

XmNdropSiteEnterCallback

Called when the drag icon enters a drop site

XmNdropSiteLeaveCallback

Called when the drag icon leaves a drop site

XmNtopLevelEnterCallback

Called when the drag icon enters a top–level window or root window (when changing screens)

XmNtopLevelLeaveCallback

Called when the drag icon leaves a top–level window or root window (when changing screens)

Callback structures for these routines contain information about the drag. The callback structures for **XmNdragMotionCallback**, **XmNoperationChangedCallback**, and **XmNdropSiteEnterCallback** contain the *operations*, *operation*, and *dropSiteStatus* fields (among others), which are initialized by the toolkit before the callback is called.

The *operations* field lists all operations possible for a drop on the current site, whether the site is registered as a DropSite or not. The toolkit initializes the *operations* field as follows:

- If the receiver's **XmNdragProc** was called, the value of *operations* is the list of operations common to the value of the **XmNdragProc**'s *operations* field at the end of **XmNdragProc** and the DropSite's **XmNdropSiteOperations** list.
- If the **XmNdragProc** routine was not called but the user selected an operation, *operations* is set to that operation if it is in the **XmNdragOperations** list. If it is not in the list, *operations* is set to **XmDROP_NOOP**.
- Otherwise, the *operations* field is initialized to the list in the DragContext's **XmNdragOperations** resource.

The *operation* field shows the operation that will occur if a drop happens at the current cursor location. It is initialized as follows:

- If the receiver's **XmNdragProc** was called, *operation* is initialized to the value of *operation* at the end of the **XmNdragProc**.
- If the **XmNdragProc** routine was not called but the pointer is in or entering an active drop site, the toolkit initializes *operation* by taking the following steps, in order of precedence from highest to lowest:
  1. If Move is in both the *operations* field and the DropSite's **XmNdropSiteOperations** list, *operation* is set to **XmDROP_MOVE**.
  2. If Copy is in both the *operations* field and the DropSite's **XmNdropSiteOperations** list, *operation* is set to **XmDROP_COPY**.
  3. If Link is in both the *operations* field and the DropSite's **XmNdropSiteOperations** list, *operation* is set to **XmDROP_LINK**.
  4. Otherwise, *operation* is set to **XmDROP_NOOP**.
- Otherwise, the toolkit initializes *operation* by taking the following steps, in order of precedence from highest to lowest:
  1. If Move is in the *operations* field, *operation* is set to **XmDROP_MOVE**.
  2. If Copy is in the *operations* field, *operation* is set to **XmDROP_COPY**.
  3. If Link is in the *operations* field, *operation* is set to **XmDROP_LINK**.
  4. Otherwise, *operation* is set to **XmDROP_NOOP**.

The *dropSiteStatus* field in the callback structure indicates if the drag icon is over a valid drop site, an invalid drop site, or no drop site. The callback procedure can use this information to display the appropriate drag–over visuals. The toolkit initializes the *dropSiteStatus* field as follows:

- If the pointer is over an active drop site:
  - If the receiver's **XmNdragProc** was called, *dropSiteStatus* is initialized to the value of *dropSiteStatus* at the end of the **XmNdragProc** procedure.
  - If the **XmNdragProc** routine was not called but the initiator and receiver have at least one target and one operation in common, *dropSiteStatus* is initialized to **XmDROP_SITE_VALID**.
  - Otherwise, *dropSiteStatus* is initialized to **XmDROP_SITE_INVALID**.
- If the pointer is not over an active drop site, *dropSiteStatus* is initialized to **XmNO_DROP_SITE**.
- If the *operation* field is **XmDROP_NOOP**, *dropSiteStatus* is initialized to **XmDROP_SITE_INVALID**.

If the application has not stored the DragContext ID in a global location, these callbacks can find the DragContext ID by passing the *timeStamp* field from the callback structure to the **XmGetDragContext** function.

This example shows a callback that is called when a new drop site is entered. It checks the validity of the drop site, and uses one of three custom source icons, depending on the status.

```
static void EnterCB(w, client_data, call_data)
Widget          w;
XtPointer       client_data, call_data;
{

   XmDragContext                   dc;
   XmDropSiteEnterCallback         EnterData;
   Cardinal                        n;
   Arg                             args[MAX_ARGS];

   dc = (XmDragContext)w;
   EnterData = (XmDropSiteEnterCallback )call_data;

   n = 0;

   if (EnterData->dropSiteStatus == XmVALID_DROP_SITE) {
      XtSetArg(args[n], XmNsourceCursorIcon,
                GetValidIcon(w)); n++;
      XtSetValues(dc, args, n);
      }
   if (EnterData->dropSiteStatus == XmINVALID_DROP_SITE) {
      XtSetArg(args[n], XmNsourceCursorIcon,
                GetInvalidIcon(w)); n++;
      XtSetValues(dc, args, n);
      }
   if (EnterData->dropSiteStatus == XmNO_DROP_SITE) {
      XtSetArg(args[n], XmNsourceCursorIcon,
                GetNeutralIcon(w)); n++;
      XtSetValues(dc, args, n);
      }
}
```

If a drag callback is desired, it is added to the DragContext's callback resources. The following example adds a callback named *EnterCB* that is performed when the pointer enters an active drop site:

```
Widget       dc;

   dc = XmeDragSource(w, location_data, event, args, n);
   XtAddCallback(dc, XmNdropSiteEnterCallback, EnterCB,
                 NULL);
```

Getting Data about the Current Drop Site

The initiator can find information about the current drop site with the **XmDropSiteRetrieve** function. It must pass in the DragContext so that the toolkit knows what drop site the request is for. The initiator can find the value of any drop site resource except the callback routines.

The following example gets the number and list of import targets for a drop site. The example shows a drop site enter callback, but it could be in any of the initiator's drag callbacks.

```
XmDropSiteEnterCallback      DragData;
```

```
...
n = 0;
XtSetArg(args[n], XmNimportTargets, &importTargets); n++;
XtSetArg(args[n], XmNnumImportTargets,
          &numImportTargets); n++;
XmDropSiteRetrieve(DragData->DragContext, args, n);
```

Cancelling the Drag

The drag in progress can be cancelled in either of two ways. Both ways are treated the same by the toolkit.

- The user can press `osfCancel`.
- The initiator can call the **XmDragCancel** function if it decides the drag should not continue for some reason.

The initiator is notified of the cancel by the **XmNdropStartCallback** with a *dropAction* field value of **XmDROP_CANCEL**.

The receiver is notified by a **XmCR_DROP_SITE_LEAVE_MESSAGE** message. This message is processed by the **XmNdragProc** in the dynamic protocol mode. This allows any drag–under effects to be undone.

Drop Receiver Responsibilities for Dropping

The drop receiver's responsibilities for dropping are the same as any widget's responsibilities as the destination widget in a UTM transfer. See Chapter 16 for more details.

This chapter explains how a drop receiver can provide help to the user.

Providing Help

It might not always be obvious to the user what the result of dropping a particular source on a drop site might be. The user can request more information about the drop site by pressing `osfHelp` while the drag icon is over the drop site.

When the user presses `osfHelp`, UTM starts a drop by doing the following:

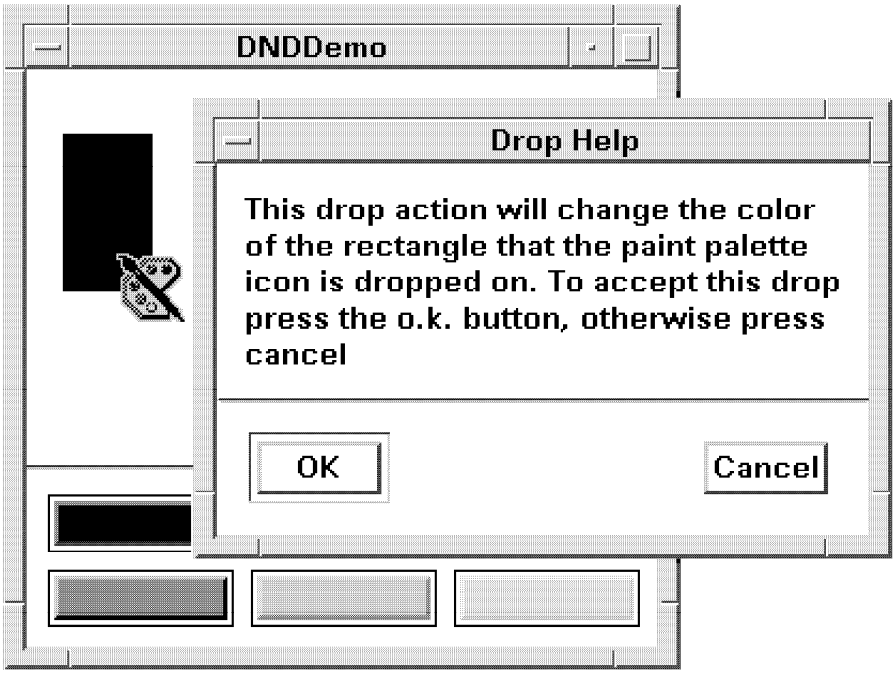
- Sets the *dropAction* member of the **XmDropProcCallbackStruct** to **XmDROP_HELP**.
- Sets the *operation* member of the **XmDestinationCallbackStruct** to **XmOTHER**.

When the user presses `osfHelp`, your **XmNdestinationCallback** procedure must realize a DialogBox. This DialogBox must provide help about the drop site, and must give users the choice of either continuing or cancelling the drop. While the DialogBox is realized, your **XmNdestinationCallback** procedure must neither perform any transfers nor call **XmTransferDone**. Your **XmNdestinationCallback** procedure must save the *transfer_id* member for any later **XmTransferValue** calls to be made.

After the user accepts or rejects the transfer, the processing continues from the callbacks on the dialog buttons. In the case of accepting the transfer, the transfer may be continued by using the *transfer_id* in calls to all the normal transfer calls in UTM. In the case of rejecting the transfer, the programmer *must* call **XmTransferDone** with a status of **XmTRANSFER_DONE_FAIL**. This will clean up the internal state of UTM and provide drag and drop zap effects to indicate the failure to the user.

The following example taken from `DNDDemo.c` shows how the help dialog shown in the illustration was created.

Figure 1 Help Dialog



The *handleDestination* routine examines the *operation* member of the **XmDestinationCallbackStruct**. If *operation* holds **XmOTHER**, then the *HandleHelp* routine is called:

```
void
handleDestination(Widget w, XtPointer client, XtPointer call)
{
  XmDestinationCallbackStruct *cs = (XmDestinationCallbackStruct *) call;

  if (appInfo->highlightRect != NULL)
    RectUnhighlight(w);

  if (cs->operation != XmOTHER)
    HandleDrop(w, call, NULL);
  else
    HandleHelp(w, call, NULL);
}
```

The *HandleHelp* routine displays the help dialog. The text presented in the dialog depends on the drop site and on the requested operation. Callback routines are registered to be performed when either of the dialog pushbuttons is pressed.

```
static void
HandleHelp(Widget w, XtPointer call, XtPointer ignore)
{
  XmDestinationCallbackStruct *cs = (XmDestinationCallbackStruct *) call;
  XmDropProcCallbackStruct *ds =
    (XmDropProcCallbackStruct *) cs -> destination_data;
  Boolean                      rectFound, bgFound, pixFound;
  XmString                     helpStr;
  RectPtr                      rect;
  Arg                          args[5];
  XmString                     tempStr, buttonArray[2];
  int                          n = 0;
```

```
savedCB = cs;

/* if we haven't created a help dialog, create one now */
if (helpDialog == NULL) {
  XtSetArg(args[n], XmNdialogStyle, XmDIALOG_FULL_APPLICATION_MODAL); n++;
  XtSetArg(args[n], XmNtitle, "Drop Help"); n++;
  helpDialog = XmCreateMessageDialog(topLevel, "Help", args, n);

  n = 0;
  buttonArray[0] = XmStringCreateSimple("Move");
  buttonArray[1] = XmStringCreateSimple("Copy");
  XtSetArg(args[n], XmNbuttons, buttonArray); n++;
  XtSetArg(args[n], XmNbuttonCount, 2); n++;
  XtSetArg(args[n], XmNbuttonSet, 0); n++;
  XtSetArg(args[n], XmNsimpleCallback, ChangeOperation); n++;
  tempStr = XmStringCreateSimple("Operations:");
  XtSetArg(args[n], XmNoptionLabel, tempStr); n++;
  helpMenu = XmCreateSimpleOptionMenu(helpDialog, "helpMenu", args, n);
  XmStringFree(tempStr);
  XmStringFree(buttonArray[0]);
  XmStringFree(buttonArray[1]);

  XtAddCallback(helpDialog, XmNokCallback,
                (XtCallbackProc) HandleDrop, NULL);
  XtAddCallback(helpDialog, XmNcancelCallback,
                (XtCallbackProc) CancelDrop, NULL);
  XtUnmanageChild(XmMessageBoxGetChild(helpDialog, XmDIALOG_HELP_BUTTON));
  XtRealizeWidget(helpDialog);
}

/* find the valid targets */
CheckTargets(ds->dragContext, XtDisplay(w), &rectFound,
             &bgFound, &pixFound);

/* determine the appropriate help message */
if (rectFound) {
  if (ds->operations == XmDROP_MOVE | XmDROP_COPY) {
    XtManageChild(helpMenu);
    helpStr = XmStringCreateLtoR(HELP_MSG4, XmFONTLIST_DEFAULT_TAG);
    XtManageChild(XmMessageBoxGetChild(helpDialog, XmDIALOG_OK_BUTTON));
  } else if (ds->operation == XmDROP_MOVE) {
    XtUnmanageChild(helpMenu);
    helpStr = XmStringCreateLtoR(HELP_MSG2, XmFONTLIST_DEFAULT_TAG);
    XtManageChild(XmMessageBoxGetChild(helpDialog, XmDIALOG_OK_BUTTON));
  } else if (ds->operation == XmDROP_COPY) {
    XtUnmanageChild(helpMenu);
    helpStr = XmStringCreateLtoR(HELP_MSG3, XmFONTLIST_DEFAULT_TAG);
    XtManageChild(XmMessageBoxGetChild(helpDialog, XmDIALOG_OK_BUTTON));
  }
} else if (bgFound || pixFound && ds->operation == XmDROP_COPY) {
  XtUnmanageChild(helpMenu);
  rect = RectFind(ds->x, ds->y);
  if (rect) {
    helpStr = XmStringCreateLtoR(HELP_MSG1, XmFONTLIST_DEFAULT_TAG);
    XtManageChild(XmMessageBoxGetChild(helpDialog, XmDIALOG_OK_BUTTON));
  } else {
    helpStr = XmStringCreateLtoR(HELP_MSG5, XmFONTLIST_DEFAULT_TAG);
    XtUnmanageChild(XmMessageBoxGetChild(helpDialog,
                                         XmDIALOG_OK_BUTTON));
  }
} else {
  XtUnmanageChild(helpMenu);
```

```
    helpStr = XmStringCreateLtoR(HELP_MSG5, XmFONTLIST_DEFAULT_TAG);
    XtUnmanageChild(XmMessageBoxGetChild(helpDialog, XmDIALOG_OK_BUTTON));
  }

  /* set the help message into the dialog */
  XtSetArg(args[0], XmNmessageString, helpStr);
  XtSetValues(helpDialog, args, 1);

  /* Free the XmString */
  XmStringFree(helpStr);

  /* map the help dialog */
  XtManageChild(helpDialog);
}
```

If the user selects the **OK** button to continue the drop, the program calls *HandleDrop* again. If the user selects the *Cancel* button, the program calls the *CancelDrop* routine. This routine calls **XmTransferDone** as follows to cancel the drop:

```
static void
CancelDrop(Widget w, XtPointer call, XtPointer ignore)
{
  XmDestinationCallbackStruct *cs;

  /* For drop help we save the XmDestinationCallbackStruct in a global
     variable named savedCB. */
  cs = savedCB;

  XmTransferDone(cs -> transfer_id, XmTRANSFER_DONE_FAIL);
}
```

Drag Initiator Responsibilities for Dropping

The drag initiator

- Registers an **XmNconvertCallback** procedure to format data and send the formatted data to the receiver.
- Optionally, registers an **XmNdropStartCallback** to be performed at the drop.
- Optionally, registers an **XmNdropFinishCallback** to be performed after the drop and transfer have finished.
- Optionally, registers an **XmNdragDropFinishCallback** to be performed after the entire drag and drop transaction has finished.

XmNdropStartCallback

The receiver's **XmNdropProc** routine receives the drop message first if the drop occurred over a widget that was registered as a drop site. It verifies that a drop is possible, and updates fields in its callback structure. These fields become available to the initiator in its **XmNdropStartCallback** callback structure. The initiator can perform any actions necessary before the information is transferred; for example, providing a new drag icon.

The toolkit initializes the *operation*, *operations*, and *dropSiteStatus* fields as described in Section 17.5.5, with one difference: the initialization for the drag callbacks uses the values at the end of the receiver's **XmNdragProc**, while the initialization for the drop callbacks uses the values at the end of the receiver's **XmNdropProc**.

The *dropAction* field indicates the action that the receiver has taken. **XmDROP** shows that a normal drop is in progress. **XmDROP_CANCEL** shows that the receiver has cancelled the drop. If the action is **XmDROP_HELP**, the initiator is not expected to do anything, although this callback provides the opportunity to do so if desired (for example, changing the drag icon to reflect the Help request).

This procedure will not know the resolution of the help dialog. However, if the user chooses to continue, the initiator's **XmNconvertProc** routine is called as part of the transfer process and, if the user chooses to cancel, the initiator's **XmNdropFinishCallback** is called with a *dropAction* of **XmDROP_CANCEL**.

Dealing with Requests for Transfer

As of Motif Release 2.0, the role of the **XmNconvertProc** can be done more easily through the UTM **XmNconvertCallback** procedures. Nevertheless, we provide the information in this subsection for those maintaining older drag and drop applications.

The drag initiator must register a callback to process transfers in the **XmNconvertProc** DragContext resource. This routine is called when the receiver client invokes **XmDropTransferStart**. Before calling **XmDropTransferStart**, the receiver makes a list of the target formats it wants.

The initiator's **XmNconvertProc** callback routine processes transfer requests from the receiver. The routine should be able to return information about each object being dragged in each possible target format for that item.

If the DropTransfer **XmNincremental** resource is True, information is transferred between the initiator and the toolkit using the Xt selection incremental protocol. If the value is False, the information is transferred between the initiator and the toolkit in one pass. The initiator and receiver need not be using the same incremental or nonincremental protocol.

The **XmNconvertProc** routine is called for each target type desired by the receiver, a single target type for each request. The **XmNconvertProc** routine should be able to perform any of the operations listed in the DragContext's **XmNdragOperations** resource on data in any of the target types listed in the **XmNexportTargets** resource:

- If the operation is Copy or Link, the **XmNconvertProc** returns a pointer to the data. The receiver will use this pointer to copy this data into its own storage, or establish a link using this pointer.
- If the operation is Move, the first transfer request has a normal *target* type. The **XmNconvertProc** routine should return a pointer to the data, as it would for a Copy.
  A second transfer request for the data has a *target* type of DELETE. The receiver does not issue this request until it has received the data and handled it appropriately (such as storing it in a file). Only then should the initiator delete the data.

XmNdropFinishCallback

The **XmNdropFinishCallback** is called when the receiver's **XmNtransferProc** routine has finished processing all the transfers desired by the receiver.

The *completionStatus* field indicates whether the entire drop was successful or not.

The *operations*, *operation*, *dropSiteStatus*, and *dropAction* fields are initialized as for the **XmNdropStartCallback** procedure.

XmNdragDropFinishCallback

The **XmdragDropFinishCallback** routine is performed when the complete drag and drop transaction has finished. This routine is called immediately after the initiator's **XmNdropFinishCallback** has finished. The initiator frees any remaining structures it has allocated during the drag.

The following sample code from `DNDDemo.c` removes the icons when the drop is complete:

```
static void
ColorDragDropFinishCB(Widget w, XtPointer client, XtPointer call)
{
    Widget  sourceIcon;
    Widget  stateIcon = (Widget) client;
    Arg     args[1];

    XtSetArg(args[0], XmNsourceCursorIcon, &sourceIcon);
    XtGetValues(w, args, 1);

    XtDestroyWidget(sourceIcon);
    XtDestroyWidget(stateIcon);
}
```