Toolkit Functions 5

This chapter describes functions that are not related to any particular widget.

Initialization and Activation Functions	*page 92*
Buffer Functions	*page 95*
Cursor and Pixmap Functions	*page 99*
Display Functions	*page 108*
Drag and Drop Functions	*page 109*
Dynamic Resource Functions	*page 140*
Error Functions	*page 142*
Help Function	*page 146*
Input Focus Functions	*page 150*
Multiple Visual Functions	*page 154*
Packed Widget Function	*page 156*
Pixel Conversion Functions	*page 158*
Protocol Function	*page 160*
Regular Expression Functions	*page 161*
Text Buffer Functions	*page 163*
Text Buffer Functions for Internationalization	*page 176*
Text Selection Operations	*page 204*
Toolkit Resource Functions	*page 206*
Virtual Event Functions	*page 207*

5

Initialization and Activation Functions

Initialization and Activation Functions

OlToolkitInitialize

```
#include <Xol/OpenLook.h>

void OlToolkitInitialize(
    XtPointer NULL);
```

`OlToolkitInitialize()` must be called by each application before any OPEN LOOK widgets are created or other OPEN LOOK routines are used.

The suggested method of initializing an OLIT application is to use `OlToolkitInitialize()` followed by some combination of:

- `XtAppInitialize()`
- `XtToolkitInitialize()`
- `XtCreateApplicationContext()`
- `XtOpenDisplay()` or `XtInitializeDisplay()`
- `XtAppCreateShell()`

or the corresponding variable argument functions.

OlInitialize

Note – The following initialization routine is now obsolete:

```
#include <Xol/OpenLook.h>

void OlInitialize(
    String                  shell_name,
    String                  application_class,
    XrmOptionDescRec        *options,
    Cardinal                num_options,
    Cardinal                *argc,
    String                  argv[]);
```

The arguments to the obsolete `OlInitialize()` routine were similar to the arguments to the X Window `XtInitialize()` routine. Use the `OlToolkitInitialize()` routine instead, but note the differences in arguments in that new version.

OlActivateWidget

```
#include <Xol/OpenLook.h>

Boolean OlActivateWidget(
    Widget          widget,
    OlVirtualName   activation_type,
    XtPointer       data);
```

`OlActivateWidget()` programmatically activates a widget in accordance with the supplied activation type (see Chapter 3, "Activation Types"). The precise semantics of activation and activation type are widget-dependent and are described in each widget section. `OlActivateWidget()` returns TRUE if the activation type was accepted by the initially supplied widget, or one of its associated *follower* widgets; see `OlAssociateWidget()`. Otherwise, the function returns FALSE. If the initially supplied widget does not accept the activation type, `OlActivateWidget()` recursively attempts to activate associated *follower* widgets until one of them accepts the supplied activation type.

`OlActivateWidget()` also accepts gadget arguments.

OlAssociateWidget

```
#include <Xol/OpenLook.h>

Boolean OlAssociateWidget(
    Widget      leader,
    Widget      follower,
    Boolean     disable_traversal)
```

`OlAssociateWidget()` associates a widget (the *follower*) with another widget (the *leader*). Associating a widget with a leader widget effectively expands the number of ways the leader widget can be activated since `OlActivateWidget()` automatically activates any follower widgets if the lead widget does not accept the supplied activation type. This routine returns TRUE if the association was successful; otherwise, it returns FALSE. Attempts to create an association-cycle are invalid and produce a warning.

It is typically desirable to prevent keyboard traversal among widgets associated with one another. The *disable_traversal* parameter is a convenient interface for setting the *follower* widget's `XtNtraversalOn` resource to FALSE.

`OlAssociateWidget()` also accepts gadget arguments.

OlUnassociateWidget

```
#include <Xol/OpenLook.h>

void OlUnassociateWidget(
    Widget follower);
```

`OlUnassociateWidget()` removes a *follower* widget from a previous association with another lead widget. No warning is generated if the supplied widget was not previously associated with another widget.

`OlUnassociateWidget()` also accepts gadget arguments.

Buffer Functions

These functions manipulate a generic “Buffer” object.

AllocateBuffer

```
#include <Xol/buffutil.h>

Buffer *AllocateBuffer(
    int     element_size,
    int     initial_size);
```

`AllocateBuffer()` allocates a Buffer for elements of the given *element_size*. The *used* member of the Buffer is set to zero and the size member is set to the value of *initial_size*. If *initial_size* is zero, the pointer *p* is set to NULL; otherwise, the amount of space required (*initial_size* × *element_size*) is allocated and the pointer *p* is set to point to this space. The function returns the pointer to the allocated Buffer. It is the responsibility of the caller to free this storage (using `FreeBuffer()`) when it is no longer needed.

The Buffer structure is defined as follows:

```
typedef struct _Buffer {
    int                 size;
    int                 used;
    int                 esize;
    BufferElement       *p;
} Buffer;
```

Buffer Macros

The following macros are provided for use with the Buffer functions.

Table 5-1 Buffer Utilities Macros

Macro	Returns
`BufferFilled(`*buffer*`)`	Indicates whether *buffer* is filled
`BufferLeft(`*buffer*`)`	Evaluates to the number of unused elements in *buffer*
`BufferEmpty(`*buffer*`)`	Indicates whether *buffer* is empty

CopyBuffer

```
#include <Xol/buffutil.h>

Buffer *CopyBuffer(
    Buffer *buffer);
```

`CopyBuffer()` allocates a new Buffer with the same attributes as the given *buffer* and copies the data associated with the given *buffer* into the new Buffer. A pointer to the newly allocated and initialized Buffer is returned. It is the responsibility of the caller to free this storage (using `FreeBuffer()`) when it is no longer needed.

FreeBuffer

```
#include <Xol/buffutil.h>

void FreeBuffer(
    Buffer *buffer);
```

`FreeBuffer()` deallocates (frees) storage associated with the given *buffer* pointer.

GrowBuffer

```
#include <Xol/buffutil.h>

void GrowBuffer(
    Buffer      *buffer,
    int         increment);
```

`GrowBuffer()` expands (or compresses) a given *buffer* size by *increment* elements. If the increment is negative, the operation results in a reduction in the size of the Buffer.

InsertIntoBuffer

```
#include <Xol/buffutil.h>

int InsertIntoBuffer(
    Buffer      *target,
    Buffer      *source,
    int         offset);
```

`InsertIntoBuffer()` inserts the elements stored in the *source* Buffer into the *target* Buffer before the element stored at *offset*. If the *offset* is invalid or if the *source* Buffer is empty, the function returns zero; otherwise, it returns 1 after

completing the insertion. The `GrowBuffer()` function will be used as needed to ensure that the *target* Buffer is large enough to hold the contents of the *source* Buffer.

ReadFileIntoBuffer

```
#include <Xol/buffutil.h>

int ReadFileIntoBuffer(
    FILE        *fp,
    Buffer      *buffer);
```

`ReadFileIntoBuffer()` reads a previously opened file associated with *fp* and adds the characters read to the end of the *buffer*. The `GrowBuffer()` function will be used as needed to ensure that the Buffer is large enough to hold the contents of the file. The function returns when either an end-of-file is detected or a newline character is encountered while reading the file. The function returns EOF if end-of-file is detected and '\n' if a newline character is encountered.

ReadStringIntoBuffer

```
#include <Xol/buffutil.h>

int ReadStringIntoBuffer(
    Buffer      *sp,
    Buffer      *buffer);
```

`ReadStringIntoBuffer()` reads a previously opened Buffer (see "stropen" on page 98) associated with *sp* and adds the characters read to the end of *buffer*. The `GrowBuffer()` function will be used as needed to ensure that the Buffer is large enough to hold the data to be copied. The function returns when either a newline character is encountered or an end-of-buffer condition is detected while reading the Buffer associated with *sp*. The function returns a '\n' if it encounters a newline character and EOF if it detects an end-of-buffer condition.

strclose

```
#include <Xol/buffutil.h>

void strclose(
    Buffer  *sp);
```

The `strclose()` function closes a string Buffer that was opened using the `stropen()` function. It frees the Buffer allocated by `stropen()`.

strgetc

```
#include <Xol/buffutil.h>

int strgetc(
    Buffer *sp);
```

The `strgetc()` function reads the next character stored in the string *sp*. The function returns the next character in the Buffer. When no characters remain, the function returns EOF.

stropen

```
#include <Xol/buffutil.h>

Buffer *stropen(
    char   *string);
```

The `stropen()` function allocates a Buffer large enough for *string* and copies *string* into this Buffer. A pointer to the newly allocated Buffer is returned. It is the responsibility of the caller to close this Buffer (using `strclose()`) when it is no longer needed.

See Also

"Text Buffer Functions" on page 163,
"Text Buffer Functions for Internationalization" on page 176.

Cursor and Pixmap Functions

These functions fall into several categories:

- `OlGet`*xyz*`Cursor` – These are Version 3 functions for obtaining the cursors defined for Drag and Drop functionality. They take a `Widget` argument, unlike the `GetOl`-style cursor functions. The middle part of the name indicates the OPEN LOOK-specified function of the cursor. These `OlGet*` functions are the preferred functions to use in every case, especially if you are not using the standard colormap for the root window of your display.
- `GetOl`*xyz*`Cursor` – These are not the preferred interface, but are included to provide compatibility with the Version 2 style of functions, taking a `Screen *`*`screen`* argument. They are listed separately, without figures. The cursors for the `GetOl*` functions are identical to their `OlGet*` counterparts.
- Other Version 2 functions, also with names of the form `GetOl`*xyz*`Cursor`, for cursors unrelated to Drag and Drop. Figures for these cursors are in the *OPEN LOOK GUI Functional Specification*.
- Two functions returning gray `Pixmap` IDs.

Version 3 Cursors

OlGetDataDupeDragCursor

```
#include <Xol/OlCursors.h>
Cursor OlGetDataDupeDragCursor(
    Widget widget);
```

OlGetDataDupeDropCursor

```
#include <Xol/OlCursors.h>
Cursor OlGetDataDupeDropCursor(
    Widget widget);
```

OlGetDataDupeInsertCursor

```
#include <Xol/OlCursors.h>
Cursor OlGetDataDupeInsertCursor(
    Widget widget);
```

5

Cursor and Pixmap Functions

OlGetDataDupeNoDropCursor

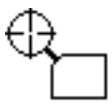
```
#include <Xol/OlCursors.h>
Cursor OlGetDataDupeNoDropCursor(
    Widget widget);
```

OlGetDataMoveDragCursor

```
#include <Xol/OlCursors.h>
Cursor OlGetDataMoveDragCursor(
    Widget widget);
```

OlGetDataMoveDropCursor

```
#include <Xol/OlCursors.h>
Cursor OlGetDataMoveDropCursor(
    Widget widget);
```



OlGetDataMoveInsertCursor

```
#include <Xol/OlCursors.h>
Cursor OlGetDataMoveInsertCursor(
    Widget widget);
```

OlGetDataMoveNoDropCursor

```
#include <Xol/OlCursors.h>
Cursor OlGetDataMoveNoDropCursor(
    Widget widget);
```

OlGetDocCursor

```
#include <Xol/OlCursors.h>
Cursor OlGetDocCursor(
    Widget widget);
```

OlGetDocStackCursor

```
#include <Xol/OlCursors.h>
Cursor OlGetDocStackCursor(
    Widget widget);
```

OlGetDropCursor

```
#include <Xol/OlCursors.h>

Cursor OlGetDropCursor(
    Widget widget);
```

OlGetDupeDocCursor

```
#include <Xol/OlCursors.h>

Cursor OlGetDupeDocCursor(
    Widget widget);
```

OlGetDupeDocDragCursor

```
#include <Xol/OlCursors.h>

Cursor OlGetDupeDocDragCursor(
    Widget widget);
```

OlGetDupeDocDropCursor

```
#include <Xol/OlCursors.h>

Cursor OlGetDupeDocDropCursor(
    Widget widget);
```

OlGetDupeDocNoDropCursor

```
#include <Xol/OlCursors.h>

Cursor OlGetDupeDocNoDropCursor(
    Widget widget);
```

OlGetDupeStackCursor

```
#include <Xol/OlCursors.h>

Cursor OlGetDupeStackCursor(
    Widget widget);
```

OlGetDupeStackDragCursor

```
#include <Xol/OlCursors.h>

Cursor OlGetDupeStackDragCursor(
    Widget widget);
```

OlGetDupeStackDropCursor

```
#include <Xol/OlCursors.h>
Cursor OlGetDupeStackDropCursor(
    Widget widget);
```

OlGetDupeStackNoDropCursor

```
#include <Xol/OlCursors.h>
Cursor OlGetDupeStackNoDropCursor(
    Widget widget);
```

OlGetFolderCursor

```
#include <Xol/OlCursors.h>
Cursor OlGetFolderCursor(
    Widget widget);
```

OlGetFolderStackCursor

```
#include <Xol/OlCursors.h>
Cursor OlGetFolderStackCursor(
    Widget widget);
```

OlGetMoveDocCursor

```
#include <Xol/OlCursors.h>
Cursor OlGetMoveDocCursor(
    Widget widget);
```

OlGetMoveDocDragCursor

```
#include <Xol/OlCursors.h>
Cursor OlGetMoveDocDragCursor(
    Widget widget);
```

OlGetMoveDocDropCursor

```
#include <Xol/OlCursors.h>
Cursor OlGetMoveDocDropCursor(
    Widget widget);
```

OlGetMoveDocNoDropCursor

```
#include <Xol/OlCursors.h>
Cursor OlGetMoveDocNoDropCursor(
    Widget widget);
```

OlGetMoveStackCursor

```
#include <Xol/OlCursors.h>
Cursor OlGetMoveStackCursor(
    Widget widget);
```

OlGetMoveStackDragCursor

```
#include <Xol/OlCursors.h>
Cursor OlGetMoveStackDragCursor(
    Widget widget);
```

OlGetMoveStackDropCursor

```
#include <Xol/OlCursors.h>
Cursor OlGetMoveStackDropCursor(
    Widget widget);
```

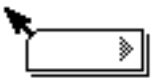
OlGetMoveStackNoDropCursor

```
#include <Xol/OlCursors.h>
Cursor OlGetMoveStackNoDropCursor(
    Widget widget);
```

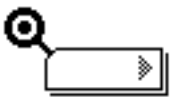
OlGetNoDropCursor

```
#include <Xol/OlCursors.h>
Cursor OlGetNoDropCursor(
    Widget widget);
```

OlGetTextDupeDragCursor

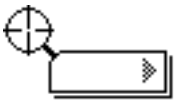


```
#include <Xol/OlCursors.h>
Cursor OlGetTextDupeDragCursor(
    Widget widget);
```



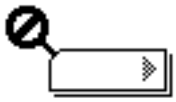
OlGetTextDupeDropCursor

```
#include <Xol/OlCursors.h>

Cursor OlGetTextDupeDropCursor(
    Widget widget);
```



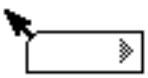
OlGetTextDupeInsertCursor

```
#include <Xol/OlCursors.h>

Cursor OlGetTextDupeInsertCursor(
    Widget widget);
```



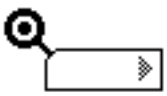
OlGetTextDupeNoDropCursor

```
#include <Xol/OlCursors.h>

Cursor OlGetTextDupeNoDropCursor(
    Widget widget);
```



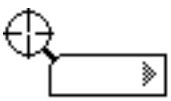
OlGetTextMoveDragCursor

```
#include <Xol/OlCursors.h>

Cursor OlGetTextMoveDragCursor(
    Widget widget);
```



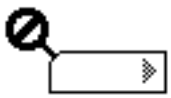
OlGetTextMoveDropCursor

```
#include <Xol/OlCursors.h>

Cursor OlGetTextMoveDropCursor(
    Widget widget);
```



OlGetTextMoveInsertCursor

```
#include <Xol/OlCursors.h>

Cursor OlGetTextMoveInsertCursor(
    Widget widget);
```



OlGetTextMoveNoDropCursor

```
#include <Xol/OlCursors.h>

Cursor OlGetTextMoveNoDropCursor(
    Widget widget);
```

Version 2 Drag and Drop Cursors

All of these version 2 drag and drop cursor functions have similar synopses:

```
#include <Xol/OlCursors.h>

Cursor GetOl<cursor-name>Cursor(
    Screen      *screen);
```

The functions are:

```
GetOlDataDupeDragCursor()
GetOlDataDupeDropCursor()
GetOlDataDupeInsertCursor()
GetOlDataDupeNoDropCursor()
GetOlDataMoveDragCursor()
GetOlDataMoveDropCursor()
GetOlDataMoveInsertCursor()
GetOlDataMoveNoDropCursor()
GetOlDocCursor()
GetOlDocStackCursor()
GetOlDropCursor()
GetOlDupeDocCursor()
GetOlDupeDocDragCursor()
GetOlDupeDocDropCursor()
GetOlDupeDocNoDropCursor()
GetOlDupeStackCursor()
GetOlDupeStackDragCursor()
GetOlDupeStackDropCursor()
GetOlDupeStackNoDropCursor()
GetOlFolderCursor()
GetOlFolderStackCursor()
GetOlMoveDocCursor()
GetOlMoveDocDragCursor()
GetOlMoveDocDropCursor()
GetOlMoveDocNoDropCursor()
GetOlMoveStackCursor()
GetOlMoveStackDragCursor()
GetOlMoveStackDropCursor()
GetOlMoveStackNoDropCursor()
GetOlNoDropCursor()
GetOlTextDupeDragCursor()
GetOlTextDupeDropCursor()
GetOlTextDupeInsertCursor()
GetOlTextDupeNoDropCursor()
GetOlTextMoveDragCursor()
GetOlTextMoveDropCursor()
GetOlTextMoveInsertCursor()
GetOlTextMoveNoDropCursor()
```

Other Version 2 Cursors

GetOlBusyCursor

```
#include <Xol/OlCursors.h>

Cursor GetOlBusyCursor(
    Screen      *screen);
```

`GetOlBusyCursor()` obtains the cursor ID for *screen* that complies with the *OPEN LOOK GUI Functional Specification* description of the **Busy** cursor.

GetOlDuplicateCursor

```
#include <Xol/OlCursors.h>

Cursor GetOlDuplicateCursor(
    Screen      *screen);
```

`GetOlDuplicateCursor()` obtains the cursor ID for *screen* that complies with the *OPEN LOOK GUI Functional Specification* description of the **Duplicate** cursor.

GetOlMoveCursor

```
#include <Xol/OlCursors.h>

Cursor GetOlMoveCursor(
    Screen      *screen);
```

`GetOlMoveCursor()` obtains the cursor ID for *screen* that complies with the *OPEN LOOK GUI Functional Specification* description of the **Move** cursor.

GetOlPanCursor

```
#include <Xol/OlCursors.h>

Cursor GetOlPanCursor(
    Screen      *screen);
```

`GetOlPanCursor()` obtains the cursor ID for *screen* that complies with the *OPEN LOOK GUI Functional Specification* description of the **Pan** cursor.

GetOlQuestionCursor

```
#include <Xol/OlCursors.h>

Cursor GetOlQuestionCursor(
    Screen      *screen);
```

`GetOlQuestionCursor()` obtains the cursor ID for *screen* that complies with the *OPEN LOOK GUI Functional Specification* description of the **Question** cursor.

GetOlStandardCursor

```
#include <Xol/OlCursors.h>

Cursor GetOlStandardCursor(
    Screen      *screen);
```

`GetOlStandardCursor()` obtains the cursor ID for *screen* that complies with the *OPEN LOOK GUI Functional Specification* description of the **Standard** cursor.

GetOlTargetCursor

```
#include <Xol/OlCursors.h>

Cursor GetOlTargetCursor(
    Screen      *screen);
```

`GetOlTargetCursor()` obtains the cursor ID for *screen* that complies with the *OPEN LOOK GUI Functional Specification* description of the **Target** cursor.

Pixmap Functions

OlGet50PercentGrey

```
#include <Xol/OlCursors.h>

Pixmap OlGet50PercentGrey(
    Screen      *screen);
```

`OlGet50PercentGrey()` obtains the ID of a 50% grey Pixmap for *screen.*

OlGet75PercentGrey

```
#include <Xol/OlCursors.h>

Pixmap OlGet75PercentGrey(
    Screen      *screen);
```

`OlGet75PercentGrey()` obtains the ID of a 75% grey Pixmap for *screen.*

Return Values

Each cursor function returns a cursor ID. Each Pixmap function returns a `Pixmap`.

See Also

"Drag and Drop Functions" on page 109.

Display Functions

OlUpdateDisplay

```
#include <Xol/OpenLook.h>

void OlUpdateDisplay(
    Widget      w);
```

`OlUpdateDisplay()` processes all pending exposure events so that the appearance of a given widget can be updated immediately. Normally, an operation is accomplished by a set of callback functions. If one of the callback functions performs a time-consuming action, it is possible that some portion of an application window will not be redrawn right away after an `XtSetValues()` call. This is because normal exposure processing does not occur until all callback functions have been invoked. This situation can be resolved by calling this function before starting a time-consuming action.

Example

```
extern Widget  status_area;     /* a staticText widget */
void fooCB(
    Widget      w,
    XtPointer   client_data,
    XtPointer   call_data);
{
    ...
        Arg     args[5];
            /* display the status in the footer area */
            /* before the actual operation */
        XtSetArg(args[0], XtNstring,
            "Start the operation, please wait ...");
        XtSetValues(status_area, args, 1);
            /* show the status in the footer area right away */
        OlUpdateDisplay(status_area);
            /* now we can start the actual operation */
    ...
}
```

See Also

`XtSetValues()` in *Xt Intrinsics Reference Manual.*

Drag and Drop Functions

Drag and Drop is a direct-manipulation data transfer operation with the following steps:

1. The user "picks up" an object (a graphical representation of data inside a client application) by pressing the SELECT mouse button on the object.
2. The user "drags" the object across the display with the SELECT mouse button pressed.
3. The user "drops" the object over an eligible drop site by releasing the SELECT mouse button over the drop site.

An example of a drag and drop operation is picking up a file from a file manager and dropping it on a trash can icon to delete the file. The terminology associated with the drag and drop operation is described below in detail.

Drop Rectangle

A drop rectangle is a rectangular area of the screen selected by some application to be eligible for drops. It is the building block of a drop site. There may or may not be any graphical feedback associated with a drop rectangle.

Drop Site

A drop site is a list of possibly overlapping drop rectangles. Drop sites need to be registered with the toolkit before they can participate in the Drag and Drop "protocol." A drop site may be registered with the toolkit by calling either of the functions `OlDnDRegisterWidgetDropSite()` or `OlDnDRegisterWindowDropSite()`.

5

Drag and Drop Functions

Drop Target

A drop target is a visible receptacle for a drop. It is marked by one of the glyphs (empty or full versions) in the following figure:

Figure 5-1 Drop Target Glyphs

See "DropTarget Widget" on page 266 for details on creating a drop target.

Owner of Drop Site

The owner of a drop site is the window or widget that registered the drop site with the toolkit. The owner of a drop site is notified (see Preview Message Notify Procedure) when the user moves the pointer over one of its drop rectangles during a drag operation. Drop sites die when the owner dies.

Do not confuse the widget owner of a drop site with the widget owner of an X11 selection (see "Drop and Data Transfer" on page 112).

Preview Message Notify Procedure

When a drop site is registered with the toolkit, the owner of the drop site may provide Preview Hints (see "OlDnDSitePreviewHints" on page 117) and a Preview Message Notify Procedure (see "OlDnDPMNotifyProc" on page 119). During the drag operation, when the user moves the pointer over a drop site, the drop site's Preview Message Notify Procedure may be invoked in a manner consistent with the Preview Hints.

Trigger Message Notify Procedure

When a drop site is registered with the toolkit, the owner of the drop site may provide a Trigger Message Notify Procedure (see "OlDnDTMNotifyProc" on page 121). When a drop occurs on the drop site, the toolkit informs the drop site of the drop by invoking the drop site's Trigger Message Notify Procedure.

A brief description follows of the interaction between a source and destination during a Drag and Drop operation.

Setup

Source

The source may or may not be registered with the toolkit. The source must arrange to be notified of mouse SELECT events in whatever manner it deems appropriate.

Destination

The destination must register its drop site(s) with the toolkit by calling `OlDnDRegisterWidgetDropSite()` if the destination is a widget, or `OlDnDRegisterWindowDropSite()` if the destination is a window. Registering the drop site may involve providing the toolkit with a set of Preview Hints (see "OlDnDSitePreviewHints" on page 117), a Preview Message Notify Procedure (see "OlDnDPMNotifyProc" on page 119), and a Trigger Message Notify Procedure (see "OlDnDTMNotifyProc" on page 121).

Begin Drag

The user "picks" an object by pressing the mouse SELECT button over an object. The owner of the object is deemed to be the source. With the mouse SELECT button pressed, the user moves the pointer over the display, "dragging" the picked object away from the source in search of a destination.

Source

Once the source is notified of a mouse SELECT event and the source is in an internal state consistent with the beginning of a Drag and Drop operation, the source should take the following steps:

1. Allocate an X11 selection atom using `OlDnDAllocateTransientAtom()`.
2. Grab the pointer by calling `OlGrabDragPointer()`.

3. Begin the Drag and Drop operation by calling `OlDnDDragAndDrop()`. `OlDnDDragAndDrop()` does not return until a drop occurs or the Drag and Drop operation is aborted. `OlDnDDragAndDrop()` returns TRUE if a successful "drop" occurs. Do not confuse a successful drop with a successful transfer of data associated with a dropped object.

Destination

The Destination does not exist at this point.

Preview And Animate

Each time the user moves the pointer over a drop site during the drag operation, the source and the drop site under the pointer may be notified, depending on whether they have registered suitable notify procedures.

Source

When calling `OlDnDDragAndDrop()` (see "Begin Drag" on page 111), the source may provide a Preview and Animate Callback Procedure (see "OlDnDPreviewAnimateCbP" on page 118). The toolkit calls the source's Preview and Animate Callback Procedure as the cursor moves over each potential drop site, giving the source the opportunity to change the cursor appearance as needed to provide the appropriate user feedback.

Drop Site Under The Pointer

As the cursor moves over each potential drop site, the toolkit will also call the Preview Message Notify Procedure of the drop site under the pointer, as appropriate for the drop site's Preview Hints. This affords the drop site the opportunity to change its appearance to indicate its ability (or inability) to actually receive the drop.

Drop and Data Transfer

A "drop" occurs when the user releases the mouse SELECT button over a drop site. The drop site over which the drop occurs is deemed to be the destination.

Source

`OlDnDDragAndDrop()` (see page 124) returns TRUE if a successful drop occurs. It has three arguments used to return values:

```
Window      *window,
Position    *x, *y
```

These arguments contain the values of the drop window ID and the x- and y-coordinates of the drop. At this point the source may:

1. Obtain ownership of the X11 selection atom allocated during the Begin Drag step. This is done by calling `OlDnDOwnSelection()`. See page 126.
2. Inform the toolkit of the ID of the X11 selection atom by calling `OlDnDDeliverTriggerMessage()`. See page 123.
3. Release the pointer by calling `OlUngrabDragPointer()`. See page 128.

Destination

When `OlDnDDeliverTriggerMessage()` is called, the toolkit invokes the Trigger Message Notify Procedure of the destination drop site, passing along the ID of the X11 selection atom obtained from the source. If the destination is interested in the drop, it may call `XGetSelection()` or `XtGetSelectionValue()` to obtain the contents of the selection. Refer to Section 10.2 of the *Xt Intrinsics Reference Manual* for details on obtaining the contents of a selection.

Closing Handshake

Source

The source may provide an `OlDnDTransactionStateCallback()` function as an argument to `OlDnDOwnSelection()` (see "Drop and Data Transfer" on page 112). This function is invoked by the toolkit with its `OlDnDTransactionState` argument set to an appropriate type, when the destination invokes any of the following functions:

- `OlDnDBeginSelectionTransaction()`
- `OlDnDEndSelectionTransaction()`
- `OlDnDErrorDuringSelectionTransaction()`
- `OlDnDDragNDropDone()`

Destination

If the toolkit invokes the Trigger Message Notify Procedure with the *send_done* argument set to TRUE, the destination is expected to call `OlDnDBeginSelectionTransaction()` at the beginning of the selection transfer and call `OlDnDEndSelectionTransaction()` at the end of the selection transfer. The destination may invoke `OlDnDErrorDuringSelectionTransaction()` to indicate an error during selection transfer.

Cleanup

Source

The source releases the selection with `OlDnDDisownSelection()`, and frees the X11 selection atom it originally allocated (see "Setup" on page 111) with `OlDnDFreeTransientAtom()`. This is typically done from within the source's Transaction State Callback when it is invoked with an `OlDnDTransactionState` value of `OlDnDTransactionDone`.

Destination

The destination calls `OlDnDDragNDropDone()`. If the source specified a Transaction State Callback to `OlDnDOwnSelection()`, the toolkit will now call it with an `OlDnDTransactionState` value of `OlDnDTransactionDone`.

Common Arguments

The following arguments are used in most drag and drop functions. When an argument to a specific function differs in interpretation from what is listed here, it is noted for that individual function.

client_data	Application-defined data
dropsiteid	ID of the drop site over which the drop occurred
detail	An `XEvent` type: either `EnterNotify`, `MotionNotify`, or `LeaveNotify`
num_sites	The number of drop rectangles defining the drop site
operation	Either `OlDnDTriggerCopyOp` or `OlDnDTriggerMoveOp`

pmnotify	A pointer to a Preview Notify Procedure, of type `OlDnDPMNotifyProc()`. The Preview Notify Procedure is given by the destination when it registers its drop site(s). It is called when the cursor passes over it following the start of a drag, subject to the conditions given in the *preview_hints*.
preview_hints	An enumerated data type `OlDnDSitePreviewHints`; possible values are listed on page 117.
root	The root window.
rootx	The horizontal (x-) coordinate at which the drop occurred, relative to the root window.
rooty	The vertical (y-) coordinate at which the drop occurred, relative to the root window.
selection	The X11 selection atom actually used to transfer the data.
site_rects	A list of `OlDnDSiteRect` structures, which defines the drop rectangles, in the coordinate system of the widget that registers the drop site.
timestamp	The current server time. Often, this argument indicates the time of an event such as the cursor entering a drop site. In these cases it will be explicitly noted.
tmnotify	A pointer to a Trigger Message Notify Procedure, of type `OlDnDTMNotifyProc()`.
widget	The widget associated with the owner of a drop site.
window	The window associated with the owner of a drop site.

Function Groups

The Drag and Drop API contains the following function groups:

Drop Site Manipulation Functions

`OlDnDRegisterWidgetDropSite()`	Register a widget-based drop site
`OlDnDRegisterWindowDropSite()`	Register a window-based drop site
`OlDnDSetDropSiteInterest()`	Activate or inactivate a drop site
`OlDnDSetInterestInWidgetHier()`	Activate or inactivate drop sites in a widget hierarchy

5

Drag and Drop Functions

`OlDnDUpdateDropSiteGeometry()`	Update a drop site's geometry
`OlDnDChangeDropSitePreviewHints()`	Update a drop site's preview hints
`OlDnDDestroyDropSite()`	Delete an existing drop site
`OlDnDQueryDropSiteInfo()`	Query a drop site's geometry
`OlDnDGetWindowOfDropSite()`	Get a window associated with a drop site
`OlDnDGetWidgetOfDropSite()`	Get a widget associated with a drop site
`OlDnDGetDropSitesOfWidget()`	Get the drop sites for a widget
`OlDnDGetDropSitesOfWindow()`	Get the drop sites for a window

Message Functions

`OlDnDDeliverPreviewMessage()`
`OlDnDDeliverTriggerMessage()`
`OlDnDPreviewAndAnimate()`

Handshake Functions

`OlDnDBeginSelectionTransaction()`
`OlDnDEndSelectionTransaction()`
`OlDnDErrorDuringSelectionTransaction()`

Selection Functions

`OlDnDAllocTransientAtom()`
`OlDnDDisownSelection()`
`OlDnDFreeTransientAtom()`
`OlDnDOwnSelection()`
`OlDnDOwnSelectionIncremental()`
`OlDnDGetCurrentSelectionsForWidget()`

General Purpose Functions

`OlGrabDragPointer()`
`OlDnDDragAndDrop()`
`OlUngrabDragPointer()`
`OlDnDDragNDropDone()`
`OlDnDWidgetConfiguredInHier()`

Data Structures

The following data structures are used by several Drag and Drop functions.

OlDnDDropSiteID

```
#include <Xol/OlDnDVCX.h>

typedef struct oldnd_drop_site *OlDnDDropSiteID;
```

`OlDnDDropSiteID` is an opaque reference to a particular instance of a drop site.

OlDnDSiteRect

```
#include <Xol/OlDnDVCX.h>

typedef XRectangle OlDnDSiteRect, *OlDnDSiteRectPtr;
```

The `OlDnDSiteRect` structure describes the drop site rectangle. Drop sites can include multiple rectangles.

OlDnDSitePreviewHints

```
#include <Xol/OlDnDVCX.h>

typedef enum oldnd_site_preview_hints {
     OlDnDSitePreviewNone,
     OlDnDSitePreviewEnterLeave =    (1 << 0),
     OlDnDSitePreviewMotion =        (1 << 1),
     OlDnDSitePreviewBoth =          (OlDnDSitePreviewEnterLeave |
                                      OlDnDSitePreviewMotion),
     OlDnDSitePreviewDefaultSite =   (1 << 2),
     OlDnDSitePreviewForwarded =     (1 << 3),
     OlDnDSitePreviewInsensitive =   (1 << 4)
} OlDnDSitePreviewHints;
```

The `OlDnDSitePreviewHints` enumerated type specifies the conditions under which the drop site is interested in receiving notification through its preview callback:

`OlDnDSitePreviewNone`	The drop site does no previewing; its callback will not be invoked.
`OlDnDSitePreviewEnterLeave`	The drop site Preview Message Notify Callback will be invoked for Enter/Leave events.

`OlDnDSitePreviewMotion`	The drop site Preview Message Notify Callback will be invoked for Motion events.
`OlDnDSitePreviewBoth`	The drop site Preview Message Notify Callback will be invoked for Enter/Leave and Motion events.
`OlDnDSitePreviewDefaultSite`	The drop site is the default site for drop site forwarding on this application shell. A default drop site is the site nominated to receive drops forwarded by the window manager decorations or icons.
`OlDnDSitePreviewForwarded`	The drop site is acting as a "proxy" on behalf of some other object or client.
`OlDnDSitePreviewInsensitive`	The drop site is currently "insensitive"; this information will be passed to the source's Preview Animate Callback so it can animate the cursor appropriately as it passes over the drop site.

Notify Procedure Prototypes

OlDnDPreviewAnimateCbP

```
#include <Xol/OlDnDVCX.h>

typedef void (*OlDnDPreviewAnimateCbP)(
    Widget      widget,
    int         eventcode,
    Time        timestamp,
    Boolean     insensitive,
    XtPointer   client_data);
```

eventcode Event code: `LeaveNotify`, `EnterNotify`, or `MotionNotify`.

insensitive TRUE means the drop site under the cursor is insensitive.

client_data Application-defined data passed to `OlDnDDragAndDrop()` when the drag and drop operation began.

`OlDnDPreviewAnimateCbP` is the function prototype for the source's Preview and Animate Callback.

OlDnDPMNotifyProc

```
#include <Xol/OlDnDVCX.h>

typedef void (*OlDnDPMNotifyProc)(
    Widget          widget,
    Window          window,
    Position        root_x,
    Position        root_y,
    int             detail,
    Time            timestamp,
    OlDnDDropSiteID drop_site,
    Boolean         forwarded,
    XtPointer       client_data);
```

root_x The root-relative x-coordinate of the preview "event"

root_y The root-relative y-coordinate of the preview "event"

timestamp The time of the preview "event"

drop_site The ID of the drop site on which the preview occurred

forwarded TRUE means the drop has been forwarded to this target from another drop site

client_data Application-defined data passed to the toolkit when the drop site was registered.

The other arguments to this function are described in "Common Arguments" on page 114.

`OlDnDPMNotifyProc` is the function prototype for the drop site Preview Message Notify Procedure. This procedure is associated with a particular drop site at the time of registration of the drop site with the toolkit. It may be invoked by the toolkit during a "drag" operation when the pointer enters, leaves, or moves across the drop site in a manner consistent with the Preview Hints (see "OlDnDSitePreviewHints" on page 117) of the drop site.

OlDnDProtocolActionCbP

```
#include <Xol/OlDnDVCX.h>

typedef void (*OlDnDProtocolActionCbP) (
     Widget               widget,
     Atom                 selection,
     OlDnDProtocolAction  protocol_action,
     Boolean              flag,
     XtPointer            client_data);
```

protocol_action	An enumerated type indicating the protocol action that has occurred.
success	Success/failure flag; if TRUE, the protocol notification was successfully received.

The other arguments to this function are described in "Common Arguments" on page 114.

The `OlDnDProtocolAction` enumerated type is defined as:

```
typedef enum oldnd_protocol_action {
    OlDnDSelectionTransactionBegins,
    OlDnDSelectionTransactionEnds,
    OlDnDSelectionTransactionError,
    OlDnDDragNDropTransactionDone
} OlDnDProtocolAction;
```

`OlDnDProtocolActionCbP` the function prototype for a Protocol Action Callback Procedure. This callback will be invoked by the toolkit to inform the caller of the success or failure of these selection transaction handshake functions. It is supplied by the requester of a selection as an argument to the selection transaction handshake functions (see "OlDnDBeginSelectionTransaction" on page 128, "OlDnDEndSelectionTransaction" on page 130, "OlDnDErrorDuringSelectionTransaction" on page 131, and "OlDnDDragNDropDone" on page 129).

OlDnDTransactionStateCallback

```
#include <Xol/OlDnDVCX.h>

typedef void (*OlDnDTransactionStateCallback) (
    Widget                  widget,
    Atom                    selection,
    OlDnDTransactionState   state,
    Time                    timestamp,
    XtPointer               client_data);
```

widget	The selection holder.
selection	The selection atom
state	The protocol event that has occurred
timestamp	When the event occurred.
client_data	Application-defined data passed to `OlDnDOwnSelection()` or `OlDnDOwnSelectionIncremental()`.

The `OlDnDTransactionState` enumerated type is defined as:

```
typedef enum oldnd_transaction_state {
    OlDnDTransactionBegins,
    OlDnDTransactionEnds,
    OlDnDTransactionDone,
    OlDnDTransactionRequestorError,
    OlDnDTransactionRequestorWindowDeath,
    OlDnDTransactionTimeout,
} OlDnDTransactionState;
```

`OlDnDOwnSelection()` and `OlDnDOwnSelectionIncremental()` have an `OlDnDTransactionStateCallback()` function pointer as one of their parameters. This function is invoked as a result of drag and drop protocol events during the drag and drop selection transaction.

The callback is invoked with the following `OlDnDTransactionState` values when the requester of the selection (the destination drop site) invokes the following functions:

Function Invoked By Destination	OlDnDTransactionState Value
`OlDnDBeginSelectionTransaction()`	`OlDnDTransactionBegins`
`OlDnDEndSelectionTransaction()`	`OlDnDTransactionEnds`
`OlDnDDragNDropDone()`	`OlDnDTransactionDone`
`OlDnDErrorDuringSelectionTransaction()`	`OlDnDTransactionRequestorError`

If the requesting client is lost during the selection transfer because its window dies, the state callback will be invoked with a state value of `OlDnDTransactionRequestorWindowDeath.`

OlDnDTMNotifyProc

```
#include <Xol/OlDnDVCX.h>

typedef void (*OlDnDTMNotifyProc)(
    Widget                 widget,
    Window                 window,
    Position               root_x,
    Position               root_y,
    Atom                   selection,
    Time                   timestamp,
    OlDnDDropSiteID        dropsite,
    OlDnDTriggerOperation  operation,
```

```
    Boolean             send_done,
    Boolean             forwarded,
    XtPointer           client_data);
```

root_x The root-relative x-coordinate at which the drop occurred

root_y The root-relative y-coordinate at which the drop occurred

timestamp The timestamp of the trigger message

dropsite The ID of the drop site on which the drop occurred

send_done If TRUE, the selection holder expects to be notified at the end of the selection transaction that it has been completed and that no further transactions associated with this drop will occur. This notification is achieved by calling `OlDnDDragNDropDone()` when the selection transaction is completed successfully.

forwarded If TRUE, the site rectangle dropped upon is a forwarded site rectangle associated by a third party (such as a window manager) with the default drop site of a top level window.

client_data Application-defined data passed to the toolkit when the drop site was registered.

The other arguments to this function are described in "Common Arguments" on page 114.

`OlDnDTMNotifyProc` is the function prototype for the drop operation Trigger Message Notify Procedure. This notify procedure, associated with a particular drop site at registration, is invoked when a drop operation occurs on its associated drop site.

Source Functions

OlDnDAllocTransientAtom

```
#include <Xol/OlDnDVCX.h>

Atom OlDnDAllocTransientAtom(
    Widget widget);
```

widget The ID of the widget that will own the transient atom returned by this call.

`OlDnDAllocTransientAtom()` allocates a reusable "transient" atom suitable for use in a drag and drop selection transaction for this widget.

OlDnDClearDragState

```
#include <Xol/OlDnDVCX.h>

void OlDnDClearDragState(
    Widget widget);
```

widget The widget ID of the selection holder

`OlDnDClearDragState()` is called upon completion of the previewing phase of a drag and drop gesture to clear internal state within the drag and drop system. This function is called implicitly by `OlDnDDragAndDrop()` and therefore is not ordinarily called directly by the OLIT programmer.

OlDnDDeliverPreviewMessage

```
#include <Xol/OlDnDVCX.h>

Boolean OlDnDDeliverPreviewMessage(
    Widget      widget,
    Window      root,
    Position    rootx,
    Position    rooty,
    Time        timestamp);
```

widget The widget ID of the selection owner.

The other arguments to this function are described in "Common Arguments" on page 114.

`OlDnDDeliverPreviewMessage()` attempts to deliver Enter, Leave, and Motion events to any drop sites currently under the (*rootx*,*rooty*) position on the root window specified. `OlDnDDeliverPreviewMessage()` returns TRUE if it finds a drop site to deliver an event to; otherwise, it returns FALSE.

OlDnDDeliverTriggerMessage

```
#include <Xol/OlDnDVCX.h>

Boolean OlDnDDeliverTriggerMessage(
    Widget                 widget,
    Window                 root,
    Position               rootx,
    Position               rooty,
    Atom                   selection,
    OlDnDTriggerOperation  operation,
    Time                   timestamp);
```

widget The widget ID of the selection holder

The other arguments to this function are described in "Common Arguments" on page 114.

`OlDnDDeliverTriggerMessage()` is called by the dragging client to deliver a trigger message to a target drop site on the root window at the coordinates specified.

The calling client is responsible for establishing a timeout period. If the drop target doesn't send selection conversion requests during this period, it should take appropriate action. `OlDnDDeliverTriggerMessage()` returns TRUE if it finds a drop site to dispatch a trigger message to at the root (x,y); otherwise, it returns FALSE.

OlDnDDisownSelection

```
#include <Xol/OlDnDVCX.h>

void OlDnDDisownSelection(
    Widget  widget,
    Atom    selection,
    Time    timestamp);
```

`OlDnDDisownSelection()` is identical in semantics to the Xt function `XtDisownSelection()`. The source widget should call this function to relinquish ownership of the selection when the drag and drop operation has been completed.

OlDnDDragAndDrop

```
#include <Xol/OlDnDVCX.h>

Boolean OlDnDDragAndDrop(
    Widget                  widget,
    Window                  *window,
    Position                *x,
    Position                *y,
    OlDnDDragDropInfoPtr    drop_info,
    OlDnDPreviewAnimateCbP  proc,
    XtPointer               client_data);
```

widget The ID of the source widget initiating the drag and drop operation

window Returns the ID of the window containing the cursor (pointer)

x The x-coordinate of the cursor relative to the containing window

y The y-coordinate of the cursor relative to the containing window

drop_info A pointer to a structure of type `OlDnDDragNDropInfo` containing information about the location of the drop, root-relative:

```
typedef struct _ol_dnd_root_info {
     Window    root_window;
     Position  root_x;
     Position  root_y;
     Time      drop_timestamp;
} OlDnDDragNDropInfo, *OlDnDDragNDropInfoPtr;
```

proc The animate function that is called when the cursor enters a drop site.

client_data Application-defined data to be passed to the animate callback.

`OlDnDDragAndDrop()` provides a simple interface for processing the mouse and keyboard events during a drop and drop operation. Before calling this function, you should call `OlGrabDragPointer()` or `XGrabPointer()` to effectively grab pointer events.

`OlDnDDragAndDrop()` issues an `XGrabKeyboard()` to obtain keystrokes during the drag operation. It then inserts a raw event handler on the widget specified for the pointer and key events and initializes the drag and drop system with `OlDnDInitializeDragState()`. Then it proceeds to process the event stream, delivering preview messages where appropriate via `OlDnDDeliverPreviewMessage()` until the drag completes or is aborted. The function returns the x, y location and the window that the pointer was in when the operation completed. It also returns the necessary root information.

OlDnDFreeTransientAtom

```
#include <Xol/OlDnDVCX.h>

void OlDnDFreeTransientAtom(
    Widget  widget,
    Atom    transient);
```

widget The widget with which the transient atom was associated.

transient The selection atom.

`OlDnDFreeTransientAtom()` frees the transient atom obtained from `OlDnDAllocTransientAtom()`.

OlDnDInitializeDragState

```
#include <Xol/OlDnDVCX.h>

Boolean OlDnDInitializeDragState(
    Widget widget);
```

widget The ID of the source widget initiating the drag and drop operation

`OlDnDInitializeDragState()` is called prior to commencing delivery of preview messages to cause the drag and drop system to download drop site previewing information from the OPEN LOOK Window Manager. It returns TRUE if the download was successful. Otherwise, it returns FALSE. This function is called implicitly by `OlDnDDragAndDrop()` and therefore is not ordinarily called directly by the OLIT programmer.

OlDnDOwnSelection

```
#include <Xol/OlDnDVCX.h>

Boolean OlDnDOwnSelection(
    Widget                          widget,
    Atom                            selection,
    Time                            timestamp,
    XtConvertSelectionProc          convert_proc,
    XtLoseSelectionProc             lose_proc,
    XtSelectionDoneProc             done_proc,
    OlDnDTransactionStateCallback   state_cb,
    XtPointer                       client_data);
```

`OlDnDOwnSelection()` is identical in semantics to the Xt function `XtOwnSelection()` except for the additional parameters *state_cb* and *client_data* (see "OlDnDTransactionStateCallback" on page 120).

OlDnDOwnSelectionIncremental

```
#include <Xol/OlDnDVCX.h>

Boolean OlDnDOwnSelectionIncremental(
    Widget                          widget,
    Atom                            selection,
    Time                            timestamp,
    XtConvertSelectionIncrProc      convert_incr_proc,
    XtLoseSelectionIncrProc         lose_incr_proc,
    XtSelectionDoneIncrProc         done_incr_proc,
```

```
    XtCancelConvertSelectionProc      cancel_proc,
    XtPointer                         client_data,
    OlDnDTransactionStateCallback     state_cb);
```

`OlDnDOwnSelectionIncremental()` function is identical in semantics to the Xt function `XtOwnSelectionIncremental()` except for the additional parameter *state_cb* (see "OlDnDTransactionStateCallback" on page 120).

OlDnDPreviewAndAnimate

```
#include <Xol/OlDnDVCX.h>

Boolean OlDnDPreviewAnimate(
    Widget                    widget,
    Window                    root,
    Position                  rootx,
    Position                  rooty,
    Time                      timestamp,
    OlDnDPreviewAnimateCbP    animate_proc,
    XtPointer                 client_data);
```

animate_proc The animate callback function.

This function is called implicitly by `OlDnDDragAndDrop()` and therefore is not ordinarily called by the OLIT programmer directly.

OlGrabDragPointer

```
#include <Xol/OpenLook.h>

void OlGrabDragPointer(
    Widget      w,
    Cursor      cursor,
    Window      confine_to_window);
```

w The ID of the source widget initiating the drag and drop operation.

cursor The cursor to be displayed.

confine_to_window Specifies the window to confine the drag pointer to, or `None`. `None` is typically what is desired for a drag and drop operation.

`OlGrabDragPointer()` effects an active grab of the mouse pointer. This function is normally called after a mouse drag operation has begun and prior to calling the `OlDnDDragAndDrop()` procedure, which is used to monitor the drag operation.

`OlGrabDragPointer()` does not return until it has successfully grabbed the drag pointer. If another widget in this client application has already grabbed the pointer, calling this function overrides any such previous grab. If another client application has already grabbed the pointer, this function blocks until the other client ungrabs the pointer and this client subsequently grabs the pointer.

OlUngrabDragPointer

```
#include <Xol/OpenLook.h>

void OlUngrabDragPointer(
    Widget w);
```

`OlUngrabDragPointer()` relinquishes the active pointer grab that was initiated by the `OlGrabDragPointer()` procedure. It simply ungrabs the pointer.

For `OlUngrabDragPointer()` to succeed, the widget passed to it must be on the same display as the widget used to grab the pointer.

Destination Functions

OlDnDBeginSelectionTransaction

```
#include <Xol/OlDnDVCX.h>

void OlDnDBeginSelectionTransaction(
    Widget                  widget,
    Atom                    selection,
    Time                    timestamp,
    OlDnDProtocolActionCbP  proc,
    XtPointer               client_data);
```

widget The requesting widget or the drop site owner.

selection The selection atom passed in the trigger notify function.

timestamp The server timestamp for the current time.

proc The callback to inform the requester whether the selection owner has successfully received the begin notification. When this callback is invoked, the *protocol_action* argument (see "OlDnDProtocolActionCbP" on page 119) is set to `OlDnDSelectionTransactionBegins`.

client_data Application-defined data to be passed to *proc*.

`OlDnDBeginSelectionTransaction()` is used in conjunction with the `OlDnDEndSelectionTransaction()` function to provide a positive handshake indicating a selection transaction. It invokes the selection holder's transaction state callback (specified by the `OlDnDOwnSelection()` and `OlDnDOwnSelectionIncremental()` functions) with a transaction state parameter value of `OlDnDTransactionBegins`.

OlDnDChangeDropSitePreviewHints

```
#include <Xol/OlDnDVCX.h>

Boolean OlDnDChangeDropSitePreviewHints(
    OlDnDDropSiteID        dropsiteid;
    OlDnDSitePreviewHints  preview_hints);
```

The arguments to this function are described in "Common Arguments" on page 114.

`OlDnDUpdateSitePreviewHints()` updates a drop site's preview hints. During the lifetime of a drop site it may be necessary to alter the nature of its previewing interest. Use `OlDnDUpdateSitePreviewHints()` to overwrite the existing preview hints for a drop site and update the drop site interest list appropriately.

OlDnDDestroyDropSite

```
#include <Xol/OlDnDVCX.h>

void OlDnDDestroyDropSite(
    OlDnDDropSiteID dropsiteid);
```

dropsiteid The ID of the drop site

`OlDnDDestroyDropSite()` explicitly destroys a drop site. When a drop site's widget or window is destroyed, all drop sites associated with that widget or window are automatically destroyed.

OlDnDDragNDropDone

```
#include <Xol/OlDnDVCX.h>

void OlDnDDragNDropDone(
    Widget                  widget,
    Atom                    selection,
```

```
    Time                        timestamp,
    OlDnDProtocolActionCbP      proc,
    XtPointer                   client_data);
```

proc The callback to inform the requester whether the selection owner has successfully received the done notification. When it is invoked, the *protocol_action* argument (see "OlDnDProtocolActionCbP" on page 119) is set to `OlDnDDragNDropTransactionDone`.

client_data Application-defined data to be passed to *proc*.

The other arguments to this function are described in "Common Arguments" on page 114.

`OlDnDDragNDropDone()` is called to inform the source (selection holder) of the completion of the drag and drop operation; this is a notification to the source that it may clean up any state associated with the selection atom, as described in "Cleanup" on page 114. It invokes the source's transaction state callback (registered with `OlDnDOwnSelection()` or `OlDnDOwnSelectionIncremental()`) with a transaction state parameter value of `OlDnDTransactionDone`.

OlDnDEndSelectionTransaction

```
#include <Xol/OlDnDVCX.h>

void OlDnDEndSelectionTransaction(
    Widget                      widget,
    Atom                        selection,
    Time                        timestamp,
    OlDnDProtocolActionCbP      proc,
    XtPointer                   client_data);
```

proc The callback to inform the requester whether the selection owner has successfully received the end notification. When it is invoked, the *protocol_action* (see "OlDnDProtocolActionCbP" on page 119) argument is set to `OlDnDSelectionTransactionEnds`.

client_data Application-defined data to be passed to *proc.*

The other arguments to this function are described in "Common Arguments" on page 114.

`OlDnDEndSelectionTransaction()` provides a positive handshake between the selection requester and holder.

It invokes the selection holder's transaction state callback (registered with the `OlDnDOwnSelection()` and `OlDnDOwnSelectionIncremental()` functions) with a transaction state parameter value of `OlDnDTransactionEnds`.

OlDnDErrorDuringSelectionTransaction

```
#include <Xol/OlDnDVCX.h>

void OlDnDErrorDuringSelectionTransaction(
    Widget                  widget,
    Atom                    selection,
    Time                    timestamp,
    OlDnDProtocolActionCbP  proc,
    XtPointer               client_data);
```

proc — The callback to inform the requester whether the selection owner has successfully received the error notification. When it is invoked, the *protocol_action* (see `OlDnDProtocolActionCbP`) argument is set to `OlDnDSelectionTransactionError`.

client_data — Application-defined data to be passed to *proc*.

The other arguments to this function are described in "Common Arguments" on page 114.

`OlDnDErrorDuringSelectionTransaction()` can be called at any time during the selection transfer by the requester to inform the selection holder that there is an error. The subsequent behavior of the holder is undefined by this protocol. `OlDnDErrorDuringSelectionTransaction()` invokes the selection holder's transaction state callback (registered with `OlDnDOwnSelection()` or `OlDnDOwnSelectionIncremental()`) with a transaction state parameter value of `OlDnDTransactionRequestorError`.

OlDnDGetCurrentSelectionsForWidget

```
#include <Xol/OlDnDVCX.h>

Boolean OlDnDGetCurrentSelectionsForWidget(
    Widget      widget,
    Atom        **atoms_return,
    Cardinal    *num_sites_return);
```

widget	The ID of the widget being investigated.
atoms_return	Points to an array of atoms currently held as selections by the widget.
num_sites_return	Points to a variable containing the number of atoms returned.

`OlDnDGetCurrentSelectionsForWidget()` returns a list of atoms currently held as drag and drop selections for the specified widget. If `OlDnDGetCurrentSelectionsForWidget()` finds any, it returns TRUE; otherwise, it returns FALSE.

The caller must call `XtFree()` on the pointer returned in the *atoms_return* parameter to free the storage allocated when it is no longer required.

OlDnDGetDropSitesOfWidget

```
#include <Xol/OlDnDVCX.h>

OlDnDDropSiteID *OlDnDGetDropSitesOfWidget(
    Widget          widget,
    Cardinal        *num_sites_return);
```

widget	The widget associated with the owner of the drop site.
num_sites_return	A pointer to a variable into which the function will return the number of drop sites.

`OlDnDGetDropSitesOfWidget()` obtains the currently registered list of drop sites for a particular widget instance. The function returns a pointer to an `OlDnDDropSiteID` array that is an enumeration of the drop sites currently registered for the widget. Clients should use `XtFree()` on this return value to deallocate the array when it is no longer needed. If there are no drop sites registered or the function fails, `OlDnDGetDropSitesOfWidget()` returns NULL.

OlDnDGetDropSitesOfWindow

```
#include <Xol/OlDnDVCX.h>

OlDnDDropSiteID *OlDnDGetDropSitesOfWindow(
    Display     *dpy,
    Window      window,
    Cardinal    *num_sites_return);
```

dpy	The display pointer.
window	The window associated with the owner of the drop site.
num_sites_return	A pointer to a variable into which the function will return the number of drop sites.

`OlDnDGetDropSitesOfWindow()` obtains the currently registered list of drop sites for a particular window. The function returns a pointer to an `OlDnDDropSiteID` array that is an enumeration of the drop sites currently registered for the window. Clients should use `XtFree()` on this return value to deallocate the array when it is no longer needed. If there are no drop sites registered or the function fails, `OlDnDGetDropSitesOfWindow()` returns NULL.

OlDnDGetWidgetOfDropSite

```
#include <Xol/OlDnDVCX.h>

Widget OlDnDGetWidgetOfDropSite(
    OlDnDDropSiteID dropsiteid);
```

`OlDnDGetWidgetOfDropSite()` returns the ID of the widget associated with the drop site, specified by the *dropsiteid* argument. If the drop site was registered with `OlDnDRegisterWindowDropSite()`, `OlDnDGetWidgetOfDropSite()` returns the ID of the widget that is the most immediate ancestor of the associated window.

OlDnDGetWindowOfDropSite

```
#include <Xol/OlDnDVCX.h>

Window OlDnDGetWindowOfDropSite(
    OlDnDDropSiteID dropsiteid);
```

`OlDnDGetWindowOfDropSite()` returns the window ID that the drop site specified by the *dropsiteid* argument is associated with. If the drop site was registered with a gadget, then `OlDnDGetWindowOfDropSite()` returns the window ID of the gadget's windowed parent.

OlDnDQueryDropSiteInfo

```
#include <Xol/OlDnDVCX.h>

Boolean OlDnDQueryDropSiteInfo(
    OlDnDDropSiteID           dropsiteid,
    Widget                    *widget,
    Window                    *window,
    OlDnDSitePreviewHints     *preview_hints,
    OlDnDSiteRectPtr          *site_rects,
    unsigned int              *num_rects,
    Boolean                   *on_interest);
```

widget The address of a variable of type `Widget` that returns the ID of the widget that owns the drop site. Set this parameter to NULL if no query on the widget ID is required. If the drop site was registered with `OlDnDRegisterWindowDropSite()`, this is the widget ID of the associated window's most immediate ancestor.

window The address of a variable of type `Window` that returns the ID of the window that owns the drop site. Set this parameter to NULL if no query on the window ID is required. For gadgets, this is the window ID of its windowed ancestor.

preview_hints The address of a variable of type `OlDnDSitePreviewHints` that returns the current hints for the drop site. Set this parameter to NULL if no query on the preview hints is required.

site_rects The address of a variable of type `OlDnDSiteRectPtr` that returns a pointer to an array that contains the current geometry of the drop site. This parameter may be set to NULL if no query on the site geometry is required. Clients must use `XtFree()` to deallocate the memory used by the array when they no longer require it.

num_rects The address of an `unsigned int` variable that returns the number of `OlDnDSiteRect` structures specified for the drop site. Set this parameter to NULL if no query on the number of rectangles.

on_interest The address of a `Boolean` variable that returns a value indicating whether the drop site is currently active (TRUE) or inactive (FALSE). Set this parameter to NULL if this value is not required.

`OlDnDQueryDropSiteInfo()` retrieves information about the drop site specified by the *dropsiteid* argument. The function returns TRUE if the query was successful; otherwise, it returns FALSE.

OlDnDRegisterWidgetDropSite

```
#include <Xol/OlDnDVCX.h>

OlDnDDropSiteID OlDnDRegisterWidgetDropSite(
    Widget                 widget,
    OlDnDSitePreviewHints  preview_hints,
    OlDnDSiteRectPtr       site_rects,
    unsigned int           num_sites,
    OlDnDTMNotifyProc      tmnotify,
    OlDnDPMNotifyProc      pmnotify,
    Boolean                on_interest,
    XtPointer              client_data);
```

client_data Application-defined data that is passed to the *tmnotify* and *pmnotify* functions when they are called.

on_interest Specifies whether the drop site is active (i.e., "interested" in responding to drops). TRUE means the drop site is active, FALSE means it is inactive. An inactive drop site is ignored during a Drag and Drop operation; its Preview Message Notify Procedure is not called when the cursor passes over it, nor is the source's Preview Animate Callback. This drop site attribute may be changed at any time during the existence of the drop site using the function `OlDnDSetDropSiteInterest()`.

The other arguments to this function are described in "Common Arguments" on page 114.

`OlDnDRegisterWidgetDropSite()` creates a drop site associated with a particular widget. The widget must be realized; that is, it must have a window associated with it before you can create a drop site for it. Gadgets can support drop sites and use their windowed ancestor's window in association with the registered drop site. Drop sites are automatically destroyed when their owning widgets die.

OlDnDRegisterWindowDropSite

```
#include <Xol/OlDnDVCX.h>

OlDnDDropSiteID OlDnDRegisterWindowDropSite(
    Display                *dpy,
    Window                 window,
    OlDnDSitePreviewHints  preview_hints,
    OlDnDSiteRectPtr       site_rects,
    unsigned int           num_sites,
    OlDnDTMNotifyProc      tmnotify,
    OlDnDPMNotifyProc      pmnotify,
    Boolean                on_interest,
    XtPointer              client_data);
```

dpy The display pointer.

client_data Application-defined data that is passed to the *tmnotify* and *pmnotify* functions when they are called.

The other arguments to this function are described in "Common Arguments" on page 114.

`OlDnDRegisterWindowDropSite()` registers a window-based drop site. It creates a drop site associated with a particular X Window and is useful for toolkit applications that mix "raw" X windows with widgets. Drop sites are automatically destroyed when their owning windows die. The window must be an inferior of a widget's window.

OlDnDSetDropSiteInterest

```
#include <Xol/OlDnDVCX.h>

void OlDnDSetDropSiteInterest(
    OlDnDDropSiteID    dropsiteid,
    Boolean            on_interest);
```

on_interest TRUE means the drop site is made active, FALSE means the drop site is made inactive.

`OlDnDSetDropSiteInterest()` activates or inactivates a drop site by exporting its existence. Active drop sites respond to drops. Inactive drop sites do not respond to drops.

OlDnDSetInterestInWidgetHier

```
#include <Xol/OlDnDVCX.h>

void OlDnDSetInterestInWidgetHier(
    Widget      widget,
    Boolean     on_interest);
```

on_interest TRUE means the drop sites are made active, FALSE means the drop sites are made inactive.

`OlDnDSetInterestInWidgetHier()` activates or inactivates all drop sites belonging to this widget and its children.

OlDnDUpdateDropSiteGeometry

```
#include <Xol/OlDnDVCX.h>

Boolean OlDnDUpdateDropSiteGeometry(
    OlDnDDropSiteID    dropsiteid,
    OlDnDSiteRectPtr   site_rects,
    unsigned int       num_sites);
```

dropsiteid The ID of the drop site to be updated.

site_rects The new list of site rectangles for the drop site.

num_sites The number of rectangles in the new rectangle list.

`OlDnDUpdateDropSiteGeometry()` alters the geometry of a drop site. Changes in the geometry of a drop site are caused by changes in the geometry of the widget or window that owns the drop site. To reduce client-server traffic, the toolkit does not automatically track changes in windows that own drop sites. The creator of a drop site is responsible for maintaining the geometry of the site to reflect any changes in the widget or window that owns the site.

OlDnDWidgetConfiguredInHier

```
#include <Xol/OlDnDVCX.h>

void OlDnDWidgetConfiguredInHier(
    Widget widget);
```

This function is primarily for use by developers of `Composite` widgets.

Since drop sites are separate from the server window hierarchy, drop site owners must attempt to maintain their drop sites clipped to their visible region(s), as defined by the server window hierarchy associated with the widget hierarchy that contains the drop sites.

In order to achieve this clipping, `Composite` widgets and their subclasses must inform the Drag and Drop system that they have configured some widgets in their subtree, as a result of a call to that `Composite` widget's `ChangeManaged()` or `GeometryManager()` methods (hence potentially changing the visible region(s) of drop sites in that subtree). Calling this function will cause the Drag and Drop system to recalculate the clipping region(s) of any drop sites under the configuring widget in the widget hierarchy.

In order to eliminate multiple recalculations of drop site clipping region(s) due to configures propagating down a widget hierarchy, a mechanism exists to suppress such multiple calculations; developers should take advantage of this in order to optimize performance.

The following is an example of the usage of this function in a simple geometry manager:

```
static XtGeometryResult GeometryManager(
        Widget              requester;
        XtWidgetGeometry    *request,
        XtWidgetGeometry    *reply);
{
  CompositeWidget  comp = requester->core.parent;
  Widget           vendor = comp;
  Arg              args[2];
  Boolean          configured_others = False;

XtSetArg(args[0], XtNconfiguringWidget, (XtPointer)requester);
XtSetArg(args[1], XtNdisableDSClipping, True);

while (!XtIsVendorShell(w))
        vendor = vendor->core.parent;

XtSetValues(vendor, args, XtNumber(args));
/* Disable clipping in my subtree while I configure. */
/* Consider the geometry request received and then maybe
 * configure one or more of the managed set and/or perhaps
 * request that my parent reconfigure me as a result of the
 * request being made by the requester widget.
 *
```

```
 * Set configured_others True if the Composite made a successful
 * geometry request to its parent, or if it moved siblings of
 * the requester */

if ((request->request_mode & CWX) == CWX)
        requester->core.x = request->x;
if ((request->request_mode & CWY) == CWY)
        requester->core.y = request->y;
if ((request->request_mode & CWWidth) == CWWidth)
        requester->core.width = request->width;
if ((request->request_mode & CWHeight) == CWHeight)
        requester->core.height = request->height;

XtSetArg(args[1], XtNdisableDSClipping, False);
XtSetValues(vendor, args, XtNumber(args));

/* enable clipping again .... */
/* Inform the drag and drop system to clip any drop sites in
 * the widget hierarchy under the configuring widget. */

if (configured_others)
        OlDnDWidgetConfiguredInHier((Widget)comp);
else
        OlDnDWidgetConfiguredInHier(requester);

return XtGeometryYes;
}
```

See Also

"Cursor and Pixmap Functions" on page 99,
"DropTarget Widget" on page 266.

5

Dynamic Resource Functions

Dynamic Resource Functions

OLIT supports Dynamic Resources. This means that for selected resource classes (those resources that include a D in the "Access" column), OLIT will detect dynamic changes in the server's resource database (by looking for updates on the RESOURCE_MANAGER property of the RootWindow) and automatically update the resource inside the widget.

The following routines allow applications to use this Dynamic Resource functionality.

Note – The `OlGetApplicationResources()` and `LookupOlColors()` routines previously included with the dynamic settings functions are no longer supported.

OlCallDynamicCallbacks

```
#include <Xol/Dynamic.h>

void OlCallDynamicCallbacks(void)
```

`OlCallDynamicCallbacks()` triggers the calling of the functions registered on the dynamic callback list. This procedure is called automatically whenever the RESOURCE_MANAGER property of the RootWindow is updated. It may also be called to force a synchronization of the dynamic settings.

OlRegisterDynamicCallback

```
#include <Xol/Dynamic.h>

void OlRegisterDynamicCallback(
    OlDynamicCallbackProc     CB,
    XtPointer                 data);
```

`OlRegisterDynamicCallback()` adds a function to the list of registered callbacks to be called whenever the procedure `OlCallDynamicCallbacks()` is invoked. `OlCallDynamicCallbacks()` is invoked whenever the RESOURCE_MANAGER property of the Root Window is updated. `OlCallDynamicCallbacks()` may also be called directly by either the

application or other routines in the widget libraries. The callbacks registered are guaranteed to be called in first-in-first-out (FIFO) order of registration and will be called as:

```
(*CB)(data);
```

OlUnregisterDynamicCallback

```
#include <Xol/Dynamic.h>

int OlUnregisterDynamicCallback(
    OlDynamicCallbackProc CB
    XtPointer             data);
```

`OlUnregisterDynamicCallback()` removes a function from the list of registered callbacks to be called whenever `OlCallDynamicCallbacks()` is invoked. It returns zero if the dynamic callback cannot be removed; otherwise, it returns 1.

5

Error Functions

Error Functions

The following functions provide error and warning message services.

Most programs should not use `OlError()` and `OlWarning()` since they do not allow for customization or internationalization.

The `OpenLook.h` header does not include `stdarg.h` or `varargs.h`. An application using `OlSetVaDisplayErrorMsgHandler()` or `OlSetVaDisplayWarningMsgHandler()` should include one of these two headers before including `OpenLook.h` to ensure the correct function prototype will be used for the application's error/warning handler.

OlError

```
#include <Xol/OpenLook.h>

void OlError(
    String msg);
```

`OlError()` writes a string to *stderr* and then exits.

OlWarning

```
#include <Xol/OpenLook.h>

void OlWarning(
    String msg);
```

`OlWarning()` writes a string to *stderr* and then returns.

OlVaDisplayErrorMsg

```
#include <Xol/OpenLook.h>

void OlVaDisplayErrorMsg(
    Display     *dpy,
    String      name,
    String      type,
    String      class,
    String      default_msg,
    ... );
```

`OlVaDisplayErrorMsg()` writes an error message to *stderr* and exits. The error message is looked up in the error database by calling `XtAppGetErrorDatabaseText()` using the *name*, *type*, and *class* arguments.

If no message is found in the error database, the *default_msg* string is used. The application context is determined by calling `XtDisplayToApplicationContext()` with the supplied `Display` pointer. If the display pointer is NULL, the display created at application startup is used to determine the application context.

OlVaDisplayWarningMsg

```
#include <Xol/OpenLook.h>

void OlVaDisplayWarningMsg(
    Display     *dpy,
    String      name,
    String      type,
    String      class,
    String      default_msg,
    ... );
```

`OlVaDisplayWarningMsg()` has the same semantics as `OlVaDisplayErrorMsg()`, except that it returns instead of exiting.

OlSetErrorHandler

```
#include <Xol/OpenLook.h>

OlErrorHandler OlSetErrorHandler(
    OlErrorHandler handler);
```

`OlSetErrorHandler()`, `OlSetWarningHandler()`, `OlSetVaDisplayErrorMsgHandler()`, and `OlSetVaDisplayWarningMsgHandler()` allow an application to override the various warning and error handlers. These routines return a pointer to the previous handler. If NULL is supplied to any of these routines, the default handler will be used. Application-supplied error handlers should do the same since continuation of an application will result in undefined behavior.

OlSetWarningHandler

```
#include <Xol/OpenLook.h>

OlWarningHandler OlSetWarningHandler(
    OlWarningHandler handler);
```

See `OlSetErrorHandler()` above.

OlSetVaDisplayErrorMsgHandler

```
#include <Xol/OpenLook.h>

OlVaDisplayErrorMsgHandler OlSetVaDisplayErrorMsgHandler(
    OlVaDisplayErrorMsgHandler handler,
    ... );
```

See `OlSetErrorHandler()` above.

OlSetVaDisplayWarningMsgHandler

```
#include <Xol/OpenLook.h>

OlVaDisplayWarningMsgHandler OlSetVaDisplayWarningMsgHandler(
    OlVaDisplayWarningMsgHandler handler,
    ... );
```

See `OlSetErrorHandler()` above.

OlErrorHandler

```
#include <Xol/OpenLook.h>

typedef void (*OlErrorHandler)(
    String msg);
```

OlWarningHandler

```
#include <Xol/OpenLook.h>

typedef void (*OlWarningHandler)(
    String msg);
```

OlVaDisplayErrorMsgHandler

```
#include <Xol/OpenLook.h>

typedef void (*OlVaDisplayErrorMsgHandler)(
    Display    *dpy,
    String     name,
    String     type,
    String     class,
    String     default_msg,
    ... );
```

OlVaDisplayWarningMsgHandler

```
#include <Xol/OpenLook.h>

typedef void (*OlVaDisplayWarningMsgHandler)(
    Display     *dpy,
    String      name,
    String      type,
    String      class,
    String      default_msg,
    ... );
```

5

Help Function

Help Function

The following function is used to register help.

OlRegisterHelp

```
#include <Xol/OpenLook.h>

void OlRegisterHelp(
    OlDefine    id_type,
    XtPointer   id,
    String      tag,
    OlDefine    source_type,
    XtPointer   source);
```

`OlRegisterHelp()` associates help information with either a widget instance or a widget class. The widget ID or widget class pointer is given in *id*, and *id_type* identifies whether it is a widget or a widget class using one of the values OL_WIDGET_HELP or OL_CLASS_HELP, respectively. Use OL_WIDGET_HELP to register help on gadgets. The other arguments are explained in "Format of Help" on page 147.

The *tag* value is shown in the title of the help window, as follows:

app-name:*tag* `Help`

where *app-name* is the name of the application. The *tag* can be null, in which case only *app-name:* `Help` is printed.

Help for Flat Widgets

To set the same help message for all items in a flat widget container, use the `OlRegisterHelp()` routine with *id_type* set to OL_WIDGET_HELP. To register help for individual items in a flat widget container, use `OlRegisterHelp()` with *id_type* set to OL_FLAT_HELP. Use the following structure to specify the object that gets the help message and pass `OlRegisterHelp()` a pointer to it in the *id* parameter:

```
typedef struct {
    Widget      widget;
    Cardinal    item_index;
} OlFlatHelpId;
```

Format of Help

The help message is identified in *source; source_type* identifies the form of the help message as one of the following:

OL_STRING_SOURCE

The *source* is of type `String` and contains text with embedded newlines. `OlRegisterHelp()` does not copy this source; the application is expected to maintain the original as long as it is registered.

OL_DISK_SOURCE

The *source* is also of type `String`, but contains the name of a file that contains the help text. `OlRegisterHelp()` does not copy this filename; the application is expected to maintain the original as long as it is registered. The file content is expected to be text with embedded newlines.

OL_INDIRECT_SOURCE

The *source* is of type `void(*)()` and is a pointer to an application-defined routine to be called by OLIT. This routine is called after HELP has been clicked. The application is expected to define the type of the help source in the routine; after it has returned, the help information will be displayed.

The routine is called as follows:

```
(*source)(id_type, id, src_x, src_y, &source_type, &source);
```

id_type and *id*	The values for the widget class or widget instance that was under the pointer when HELP was pressed. These are the same values previously registered with `OlRegisterHelp()`.
src_x and *src_y*	The coordinates of the pointer when HELP was pressed. These are relative to the upper-left corner of the window.
source_type and *source*	Pointers to values the application's routine should set for the help source it wants to display. The only *source_type* values accepted are OL_STRING_SOURCE and OL_DISK_SOURCE.

OL_TRANSPARENT_SOURCE

The *source* is of type `void(*)()` and is a pointer to an application-defined routine. The routine is called after HELP has been invoked. The application is expected to handle the HELP event completely. This might be used by an application that does not want the standard help window (for example, `xterm`(1) simply generates an escape sequence).

The routine is called as follows:

```
(*source)(id_type, id, src_x, src_y);
```

id_type and *id* — The values for the widget class or widget instance that was under the pointer when HELP was pressed. These are the same values registered with `OlRegisterHelp()`.

src_x and *src_y* — The coordinates of the pointer when HELP was pressed. These are relative to the upper-left corner of the window.

The help window is automatically popped up for the OL_STRING_SOURCE, OL_DISK_SOURCE, and OL_INDIRECT_SOURCE help sources. (It is popped up after the application routine returns for the OL_INDIRECT_SOURCE help source.) The application is responsible for popping up a help window (if needed) for the OL_TRANSPARENT_SOURCE help source.

Handling the Help Key Event

When the user clicks HELP, if the event occurs within a widget or window registered with the `OlRegisterHelp()` routine, the corresponding help message is automatically displayed (for source types OL_STRING_SOURCE and OL_DISK_SOURCE) or the application routine is called (for source types OL_INDIRECT_SOURCE and OL_TRANSPARENT_SOURCE). If the event occurs elsewhere, a default help message is displayed.

If the help key is pressed on a widget, the help routine looks for help registered on that widget of type OL_WIDGET_HELP. If no help is found, the help routine searches up the widget tree (i.e., goes to the widget's parents up to a widget which is a subclass of shell) looking for the first widget that has help of type OL_WIDGET_HELP registered. If it finds help registered on one of the ancestors of the original widget, the help message for that widget will be used. If help is not found, the help routine looks for help of type OL_CLASS_HELP on the original widget. If no help is found, the default message is used.

The use of `OlRegisterHelp()` is considered across all applications. In other words, even though a regular application does not register help for the root window (the "workspace"), it does not mean that pressing HELP on the root window causes a default message. Another application (typically the workspace manager) may have registered the help.

Separate Help per Application

An application will have, at most, one help message displayed. However, several applications can display their separate help messages simultaneously, in different help windows.

Displaying the Help Message

A help source of type OL_STRING_SOURCE and OL_DISK_SOURCE is displayed in a help window that is 50 ens wide and 10 lines tall. (An en is $S/2$ points, where S is the current point size.)

Lines longer than the help window width are wrapped at the space(s) between words, or at the nearest character boundary if there is no space at which to wrap. Lines are also wrapped at embedded newlines, regardless of their lengths.

Only spaces and newlines are recognized for format control; all other non-printable characters are silently ignored.

Up to ten lines of the message are visible at once. Messages longer than ten lines have a scrollbar control that allows scrolling non-visible lines into view.

Static Variables

The *tag* and *source* values should be statically defined (or allocated and not freed). Using automatic variables here will almost always fail.

5

Input Focus Functions

Input Focus Functions

Each of these utility routines works with widgets or gadgets to manipulate input focus.

OlCallAcceptFocus

```
#include <Xol/OpenLook.h>

Boolean OlCallAcceptFocus(
    Widget      w,
    Time        time);
```

`OlCallAcceptFocus()` sets the focus to a specified widget. If widget *w* currently is capable of accepting input focus, `OlCallAcceptFocus()` assigns focus to *w* and it returns TRUE; otherwise, it returns FALSE. See the `XtCallAcceptFocus()` function in the *Xt Intrinsics Reference Manual* for further details about the *time* argument.

Note – `OlCallAcceptFocus()` will be declared obsolete in a future version of OLIT. You should use the `XtCallAcceptFocus()` for all new applications.

OlCanAcceptFocus

```
#include <Xol/OpenLook.h>

Boolean OlCanAcceptFocus(
    Widget      w,
    Time        time);
```

`OlCanAcceptFocus()` tests whether a widget can accept focus. If it can accept focus, it returns TRUE; otherwise, it returns FALSE. Acceptance of focus is determined by all of the following being true:

- The widget is not being destroyed.
- The widget is managed.
- The widget is mapped when managed (if it is not a gadget).
- The widget is realized, or for a gadget, the gadget's parent are realized.
- The widget and its ancestors are sensitive.

- A query for the widget's `Window` attributes is successful and the widget's window is viewable (i.e., the window and all its ancestor windows are mapped).
- The `XtNmouseless` resource is TRUE or the widget is a shell or text input widget.

OlSetInputFocus

```
#include <Xol/OpenLook.h>

void OlSetInputFocus(
    Widget      w,
    int         revert_to,
    Time        time);
```

`OlSetInputFocus()` sets focus to a widget. Applications should use this routine instead of `XSetInputFocus()`; see the description of `XSetInputFocus()` in the *XLib Reference Manual* for further details about the *revert_to* and *time* arguments. If `XtNmouseless` is FALSE, `OlSetInputFocus()` is ignored unless the widget is a text input or shell widget.

OlGetCurrentFocusWidget

```
#include <Xol/OpenLook.h>

Widget OlGetCurrentFocusWidget(
    Widget w);
```

`OlGetCurrentFocusWidget()` returns the widget that currently has focus in the window group of the specified widget. If no widget in the window group has focus, `OlGetCurrentFocusWidget()` returns NULL.

OlHasFocus

```
#include <Xol/OpenLook.h>

Boolean OlHasFocus(
    Widget w);
```

`OlHasFocus()` returns TRUE if the specified widget has focus. `OlHasFocus()` simply calls `OlGetCurrentFocusWidget()` and compares its return value to the supplied widget.

OlMoveFocus

```
#include <Xol/OpenLook.h>

Widget OlMoveFocus(
    Widget          w,
    OlVirtualName   direction,
    Time            time);
```

`OlMoveFocus()` moves the input focus relative to the widget *w*, as indicated by *direction*, and returns the new focus widget. It calls `OlCallAcceptFocus()` to move the input focus. If `OlCallAcceptFocus()` is unable to move input focus, `OlMoveFocus()` returns NULL. It will also return NULL if `XtNmouseless` is set to FALSE and the widget is not a text input widget. When moving input focus between widgets contained within an Exclusives or Nonexclusives widget, valid values for direction are shown in the following list. For the OL_MULTI directions below, the value of *m* is the value of the toolkit resource `XtNmultiObjectCount`. See the description of `XSetInputFocus()` in the *XLib Reference Manual* for further details about the *time* argument.

OL_IMMEDIATE — Set focus to the next widget that will accept it, starting with *w*.

OL_MOVERIGHT — Set focus to the widget in the next column (and same row) that will accept it, starting with the first column after *w's* column. If *w* is located in the extreme right column, focus is set to the widget in the extreme left column of the same row.

OL_MOVELEFT — Set focus to the widget in the previous column (and same row) that will accept it, starting with the first column before *w's* column. If *w* is located on the extreme left column, focus is set to the widget in the extreme right column of the same row.

OL_MOVEUP — Set focus to the widget in the previous row (and same column) that will accept it, starting with the first row before *w*'s row. If *w* is located in the top row, focus is set to the widget in the bottom row of the same column.

OL_MOVEDOWN — Set focus to the widget in the next row (and same column) that will accept it, starting with the first row after *w*'s row. If *w* is located in the bottom row, focus is set to the widget in the top row of the same column.

OL_MULTIRIGHT — Set focus to the widget in the next column (and same row) that will accept it, starting with the first column *m* columns after *w's* column. If *m* is greater than the number of objects

between *w* and the extreme right column, focus is set to the widget in the extreme left column of the same row.

OL_MULTILEFT — Set focus to the widget in the previous column (and same row) that will accept it, starting with the first column *m* columns before *w's* column. If *m* is greater than the number of objects between *w* and the extreme left column, focus is set to the widget in the extreme right column of the same row.

OL_MULTIUP — Set focus to the widget in the previous row (and same column) that will accept it, starting with the first row *m* rows before *w*'s row. If *m* is greater than the number of objects between *w* and the extreme top row, focus is set to the widget in the extreme bottom row of the same column.

OL_MULTIDOWN — Set focus to the widget in the next row (and same column) that will accept it, starting with the first row *m* rows after *w*'s row. If *m* is greater than the number of objects between *w* and the extreme bottom row, focus is set to the widget in the extreme top row of the same column.

When moving between widgets in a base window or popup window, focus is moved according to the order defined by the *traversal list*. The default traversal order is determined by the order in which widgets are created. Valid values for *direction* are:

OL_IMMEDIATE — Set focus to the next object that will accept it, starting with w.

OL_NEXTFIELD, OL_MOVERIGHT, OL_MOVEDOWN — Set focus to the next object that will accept it, starting with the first object after *w*.

OL_PREVFIELD, OL_MOVELEFT, OL_MOVEUP — Set focus to the next object that will accept it, starting with the first object before *w*. (The list is searched in reverse order.)

OL_MULTIRIGHT, OL_MULTIDOWN — Set focus to the next object that will accept it, starting with the first *m* objects after *w*.

OL_MULTILEFT, OL_MULTIUP — Set focus to the next object that will accept it, starting with the first *m* objects before *w*. (The list is searched in reverse order.)

Multiple Visual Functions

The following functions are used to work with multiple visuals.

A visual is specified by a *depth* (for example, 8 bits) and a *visual class* (for example, `PseudoColor`).

A shell widget or a DrawArea widget can have a nondefault visual. Other widgets use the visuals of their nearest shell or DrawArea ancestor. An application in which eligible widgets have nondefault visuals is termed a multi-visual application.

You must specify the visual class when you specify the depth, or the depth will be ignored. Specifically, you should use the `XtVaTypedArg` interface with `XtVaCreateManagedWidget()`, rather than `XMatchVisualinfo()`. (The argument list interface implicitly invokes the resource converter, while `XMatchVisualinfo()` does not. Trying to set the depth without also setting the visual class and running the resource converter can create problems.)

For example, in creating a DrawArea widget using this interface, you might use something like:

```
drawarea = XtVaCreateManagedWidget("drawarea",
        drawAreaWidgetClass, toplevel,
        XtVaTypedArg, XtNvisual, XtRString,
        VisualClassName, sizeof(VisualClassName),
        XtNlayout, OL_IGNORE,
        XtNheight, Height,
        XtNwidth, Width,
        NULL);
```

Each multiple visual function returns a characteristic of either a widget or gadget.

Multiple visuals are meant to run on machines with hardware colormaps; otherwise, serious flashing results when the mouse pointer moves between applications or widgets with different visuals.

OlBlackPixel

```
#include <Xol/OpenLook.h>

Pixel OlBlackPixel(
     Widget w);
```

`OlBlackPixel()` returns the black pixel for the colormap associated with the given widget. Use this function instead of the macro `BlackPixel()`, in OLIT applications that use multiple colormaps and/or multiple visuals.

OlColormapOfObject

```
#include <Xol/OpenLook.h>

Colormap OlColormapOfObject(
    Widget      object);
```

`OlColormapOfObject()` obtains the colormap associated with the *object*.

OlDepthOfObject

```
#include <Xol/OpenLook.h>

int OlDepthOfObject(
    Widget      object);
```

`OlDepthOfObject()` obtains the depth associated with the *object*.

OlInternAtom

```
#include <Xol/RootShell.h>

Atom OlInternAtom(
    Display     *dpy,
    String      atom_name);
```

`OlInternAtom()` uses the Intrinsics `XtRString`-to-`XtRAtom` resource converter and the converter cache to store Atoms in the resource cache on a per display basis.

You should use this function to cache Atoms across displays, especially for applications using multiple displays.

For efficient use of the resource converter cache, the string *atom_name* should be the same physical string for each invocation. For example:

```
OlInternAtom(dpy, "foo");
OlInternAtom(dpy, "foo");
```

results in *two* entries in the resource converter cache, while the following results in only one cache entry:

```
char *foo = "foo";
OlInternAtom(dpy, foo);
OlInternAtom(dpy, foo);
```

OlVisualOfObject

```
#include <Xol/OpenLook.h>

Visual *OlVisualOfObject(
    Widget      object);
```

`OlVisualOfObject()` obtains the visual associated with the *object.*

OlWhitePixel

```
#include <Xol/OpenLook.h>

Pixel OlWhitePixel(
     Widget w);
```

`OlWhitePixel()` returns the white pixel for the colormap associated with the given widget. Use this function instead of the macro `WhitePixel()`, in OLIT applications that use multiple colormaps and/or multiple visuals.

Packed Widget Function

The following function creates a widget (sub)tree in one call.

OlCreatePackedWidgetList

```
#include <Xol/OpenLook.h>

Widget OlCreatePackedWidgetList(
    OlPackedWidgetList     *pw_list,
    Cardinal               num_pw);
```

`OlCreatePackedWidgetList()` and its associated `OlPackedWidget` structure allow an application to create a widget tree or subtree in one call.

pw_list A pointer to an `OlPackedWidget` array. It creates widgets starting from the first element in the array.

num_pw The number of elements in the array *pw_list*

`OlCreatePackedWidgetList()` returns the widget ID of the first element in the array *pw_list*.

The `OlPackedWidget` structure contains all the information needed to create a new widget. It is defined as:

```
typedef struct {
    Widget          widget;
    String          name;
    WidgetClass     *class_ptr;
    Widget          *parent_ptr;
    String          descendant;
    ArgList         resources;
    Cardinal        num_resources;
    Boolean         managed;
} OlPackedWidget, *OlPackedWidgetList;
```

widget Contains the ID of the newly created widget.

name The name of the widget that will be created.

class_ptr A pointer to the `WidgetClass` pointer for the new widget. This gives the class of widget to create. It is a pointer to the pointer because typically the pointer itself is an external value that is not suitable for using in an array initialization.

parent_ptr A pointer to the widget ID of the intended parent of the new widget *or* the ID of an indirect widget that "knows who the parent is" (see below). This value may point to a *widget* member in another PackedWidget item; if the parent is an indirect widget, it must appear earlier in the list.

descendant The name of a resource available in the widget identified by *parent_ptr.* The value of this resource is the ID of the real parent for the new widget. If the *descendant* value is not zero, *parent* is expected to identify an indirect parent that is interrogated for the ID of the real parent. If this value is zero, *parent* is expected to identify the real parent.

resources The resource array to use when creating the new widget.

num_resources The number of resources in the array.

managed TRUE if the new widget should be managed when created, FALSE otherwise.

5

Pixel Conversion Functions

Pixel Conversion Functions

The following routines convert pixel dimensions to other measurements.

```
#include <Xol/OpenLook.h>

Screen *OlDefaultScreen;
Display *OlDefaultDisplay;

Axis axis;
Screen screen;

OlMMToPixel(axis, millimeters);
Ol_MMToPixel(axis, millimeters);

OlPointToPixel(axis, points);
Ol_PointToPixel(axis, points);

OlScreenMMToPixel(axis, millimeters, screen);
Ol_ScreenMMToPixel(axis, millimeters, screen);

OlScreenPointToPixel(axis, points, screen);
Ol_ScreenPointToPixel(axis, points, screen);

OlPixelToMM(axis, pixels);
Ol_PixelToMM(axis, pixels);

OlPixelToPoint(axis, pixels);
Ol_PixelToPoint(axis, pixels);

OlScreenPixelToPoint(axis, pixels, screen);
Ol_ScreenPixelToPoint(axis, pixels, screen);

OlScreenPixelToMM(axis, pixels, screen);
Ol_ScreenPixelToMM(axis, pixels, screen);
```

All the X-based OPEN LOOK widgets refer to pixels in coordinates and dimensions for compatibility with other X Window System widgets.

This puts the burden on the application programmer to convert between externally useful measures, such as points or millimeters, and pixels as applied to the screen at hand. These routines examine the data structures that describe the physical dimensions and the pixel resolution of a screen and convert among millimeters, points, and pixels for that screen.

Screen Selection

The shorter forms of these routines (the ones without the word Screen in their names) work for the default screen. This is the screen that is active when the X-Toolkit Intrinsics are started. The longer forms of these routines take a `Screen *` type argument that refers to a particular screen. The macros `OlDefaultScreen` and `OlDefaultDisplay` identify the current screen and display being used by the Intrinsics.

Use After Toolkit Initialization

These routines make use of data structures that are initialized when the Toolkit is initialized (see Initialization and Activation Functions on page 92). Therefore, using them before toolkit initialization (for example, as an initial value to a statically defined variable) will result in a run-time error.

Axis Argument

The first argument of all the routines is the direction in which the measurement is made. This is necessary because not all screens have equivalent resolution in the horizontal and vertical axes. The *axis* argument can take one of the two values: OL_HORIZONTAL or OL_VERTICAL. These routines are not directly usable in computing a diagonal measure. (Find the diagonal with the Pythagorean Theorem: $a^2 + b^2 = c^2$).

Implemented as Macros

All these routines are implemented as macros, so they can take any reasonable type value for the millimeters, points, and pixels. The macros cast the values into the proper type needed for the conversion. However, only a single type value can be "returned."

The routines without an underscore in their names produce values of type `int` (the values are rounded to the nearest integer). The routines with an underscore in their names produce values of type `double` (these values have not been rounded, leaving it up to the application to round up, round down, or truncate as needed). Given the small size of the units involved, the integer-returning routines should be sufficient for many applications.

Because these routines are implemented as macros, there are no function addresses available.

Protocol Function

OlWMProtocolAction

`OlWMProtocolAction()` simulates a response to any window manager's protocol messages.

```
#include <Xol/OpenLook.h>

void OlWMProtocolAction(
    Widget                  w,
    OlWMProtocolVerify      *st,
    OlDefine                action)
```

The *w* parameter must be a widget that is a subclass of `VendorShell`. Otherwise, no action will be taken.

The `OlWMProtocolVerify` structure is defined as follows:

```
typedef struct {
    unsigned long  msgtype;
    XEvent         *xevent;
} OlWMProtocolVerify;
```

Its *msgtype* field is an integer constant indicating the type of protocol message that invoked the callback; it will be one of the following values:

OL_WM_TAKE_FOCUS
OL_WM_SAVE_YOURSELF
OL_WM_DELETE_WINDOW

The *action* parameter can be:

OL_QUIT	Quit the application immediately.
OL_DEFAULTACTION	Perform the action that is appropriate for each subclass of `VendorShell`.
OL_DESTROY	Destroy the shell widget.
OL_DISMISS	Dismiss or unmap the shell widget.

Regular Expression Functions

The following functions scan strings using a form of regular expressions. Unlike the regular expressions supported by `ed`(1) or `egrep`(1), these functions use a regular expression notation consisting of:

Table 5-2 Regular Expression Notation

Element	Meaning
c	Match the character *c*
[<*set*>]	Match any character in <*set*> (where <*set*> is one or more characters concatenated into a string; range expressions, such as `[a-z]`, are not supported)
[!<*set*>]	Match any character not in <*set*>
*	Match any character(s) (one or more)
^	When the circumflex is the first character in the regular expression, the match must start at *curp*

streexp

```
#include <Xol/regexp.h>

char *streexp(void);
```

The `streexp()` function returns the pointer of the last character in a match found following a `strexp()` or `strrexp()` function call.

strexp

```
#include <Xol/regexp.h>

char *strexp(
    char    *string,
    char    *curp,
    char    *expression);
```

The `strexp()` function performs a regular expression forward scan of *string* for *expression* starting at *curp*.

NULL is returned if *expression* cannot be found in *string*; otherwise, a pointer to the first character in the substring that matches *expression* is returned. The `streexp()` function can be used to get the pointer to the last character in the match.

strrexp

```
#include <Xol/regexp.h>

char *strrexp(
    char    *string,
    char    *curp,
    char    *expression);
```

The `strrexp()` function performs a regular expression backward scan of *string* for *expression* starting at *curp*.

NULL is returned if *expression* cannot be found in *string*; otherwise, a pointer to the first character in the substring that matches *expression* is returned. The `streexp()` function can be used to get the pointer to the last character in the match.

See Also

"Buffer Functions" on page 95,
"TextField Functions" on page 686.

Text Buffer Functions

A *TextBuffer* is a data structure used by every single-byte TextEdit widget to store and manipulate its data. For internationalized TextBuffers, see page 176. The functions in this section can be used to manipulate a TextBuffer.

TextLocation Structure

A number of the functions in this section refer to a `TextLocation` structure. It is defined as follows:

```
typedef struct _TextLocation {
    TextLine            line;
    TextPosition        offset;
    BufferElement       *buffer;
} TextLocation;
```

AllocateTextBuffer

```
#include <Xol/textbuff.h>

TextBuffer *AllocateTextBuffer(
    char                    *filename,
    TextUpdateFunction      f,
    XtPointer               d);
```

`AllocateTextBuffer()` allocates a new TextBuffer structure, initializes the members of the structure, does one more step described below, and returns a pointer to the newly allocated structure.

It is possible to register one or more text update functions (of type `TextUpdateFunction`) with a TextBuffer. As the name suggests, the text update functions are invoked by the toolkit when the TextBuffer is updated. In the course of registering a text update function, a possibly NULL client data (of type `XtPointer`) must be provided with the function. The client data is passed to the associated text update function when the function is invoked by the toolkit. See "ReplaceBlockInTextBuffer" on page 172 for more details of the `TextUpdateFunction`.

The argument *f* above is a text update function that together with its associated client data *d* is registered with the newly allocated TextBuffer, before `AllocateTextBuffer()` returns. The programmer must use

`FreeTextBuffer()` function to free the TextBuffer. The *filename* argument is used by the `SaveTextBuffer()` function (see page 174) if it is called with a NULL *filename* argument.

BackwardScanTextBuffer

```
#include <Xol/textbuff.h>
ScanResult BackwardScanTextBuffer(
    TextBuffer      *text,
    char            *exp,
    TextLocation    *location);
```

`BackwardScanTextBuffer()` scans towards the beginning of the buffer for a given *exp*ression in the TextBuffer starting at *location*. The *exp* string is interpreted as described in "Regular Expression Functions" on page 161. A `ScanResult` is returned, which indicates:

SCAN_NOTFOUND	The scan wrapped without finding a match.
SCAN_WRAPPED	A match was found at a location after the start location.
SCAN_FOUND	A match was found at a location before the start location.
SCAN_INVALID	Either the *location* or the *exp* was invalid.

CopyTextBufferBlock

```
#include <Xol/textbuff.h>

int CopyTextBufferBlock(
    TextBuffer      *text,
    char            *buffer,
    TextPosition    start_position,
    TextPosition    end_position);
```

`CopyTextBufferBlock()` copies a text block from the *text* TextBuffer into *buffer*. The block is defined as the characters between *start_position* and *end_position* inclusive. It returns the number of bytes copied; if the parameters are invalid, the return value is zero.

Note – The storage for the copy is allocated by the caller. It is the responsibility of the caller to ensure that enough storage is allocated to copy *end_position* – *start_position* + 1 bytes.

EndCurrentTextBufferWord

```
#include <Xol/textbuff.h>

TextLocation EndCurrentTextBufferWord(
    TextBuffer      *textBuffer,
    TextLocation    current);
```

`EndCurrentTextBufferWord()` locates the end of a word in the TextBuffer relative to a given *current* location. The function returns the location of the end of the current word. Note: this return value will equal the given current value if the current location is already at the end of a word.

FreeTextBuffer

```
#include <Xol/textbuff.h>

void FreeTextBuffer(
    TextBuffer              *text,
    TextUpdateFunction      f,
    XtPointer               d);
```

`FreeTextBuffer()` deallocates storage associated with a given TextBuffer. Note: the storage is not actually freed if the TextBuffer is still associated with other update function/data pairs. See "ReplaceBlockInTextBuffer" on page 172 for more details of the `TextUpdateFunction`.

ForwardScanTextBuffer

```
#include <Xol/textbuff.h>
ScanResult ForwardScanTextBuffer(TextBuffer *text,
    char                *exp,
    TextLocation        *location);
```

`ForwardScanTextBuffer()` scans towards the end of the buffer for a given *exp*ression in the TextBuffer starting at *location*. The *exp* string is interpreted as described in "Regular Expression Functions" on page 161. A `ScanResult` is returned, which indicates:

SCAN_NOTFOUND	The scan wrapped without finding a match.
SCAN_WRAPPED	A match was found at a location before the start location.
SCAN_FOUND	A match was found at a location after the start location.
SCAN_INVALID	Either the location or the expression was invalid.

GetTextBufferBlock

```
#include <Xol/textbuff.h>

char *GetTextBufferBlock(
    TextBuffer      *text,
    TextLocation    start_location,
    TextLocation    end_location);
```

`GetTextBufferBlock()` retrieves a text block from the *text* TextBuffer. The block is defined as the characters between *start_location* and *end_location* inclusive. It returns a pointer to a string containing the copy. If the parameters are invalid, NULL is returned.

Note – The storage for the copy is allocated by this routine. It is the responsibility of the caller to free this storage when it becomes dispensable.

GetTextBufferBuffer

```
#include <Xol/textbuff.h>

Buffer *GetTextBufferBuffer(
    TextBuffer      *text,
    TextLine        line);
```

`GetTextBufferBuffer()` retrieves a pointer to the Buffer stored in TextBuffer *text* for *line*. This pointer is volatile; subsequent calls to any TextBuffer routine may make it invalid. If a more permanent copy of this Buffer is required, the `CopyTextBufferBlock()` function (see page 164) can be used to create a private copy of it.

GetTextBufferChar

```
#include <Xol/textbuff.h>

int GetTextBufferChar(
    TextBuffer      *text,
    TextLocation    location);
```

`GetTextBufferChar()` retrieves a character stored in the *text* TextBuffer at *location*. It returns either the character itself or EOF if location is outside the range of valid locations within the TextBuffer.

GetTextBufferLine

```
#include <Xol/textbuff.h>

char *GetTextBufferLine(
    TextBuffer      *text,
    TextLine        lineindex);
```

`GetTextBufferLine()` retrieves the contents of string containing the copy of the contents of the line or NULL if the *lineindex* is outside the range of valid lines in *text*.

Note – The storage for the copy is allocated by this routine. It is the responsibility of the caller to free this storage when it becomes dispensable.

GetTextBufferLocation

```
#include <Xol/textbuff.h>

char *GetTextBufferLocation(
    TextBuffer      *text,
    TextLine        line_number,
    TextLocation    *location);
```

`GetTextBufferLocation()` retrieves the contents of the given line within the TextBuffer. It returns a pointer to the character string. If the line number is invalid, a NULL pointer is returned. If a non-NULL `TextLocation` pointer is supplied in the argument list, the contents of this structure are modified to reflect the values corresponding to the given line.

IncrementTextBufferLocation

```
#include <Xol/textbuff.h>

TextLocation IncrementTextBufferLocation(
    TextBuffer      *text,
    TextLocation    location,
    TextLine        line,
    TextPosition    offset);
```

`IncrementTextBufferLocation()` increments a *location* by either *line* lines and/or *offset* characters. It returns the new *location*. If *line* or *offset* are negative, the function performs a decrement operation. If the starting location or the resulting location is invalid, the starting location is returned without modification; otherwise, the new location is returned.

LastTextBufferLocation

```
#include <Xol/textbuff.h>

TextLocation LastTextBufferLocation(
    TextBuffer     *text);
```

`LastTextBufferLocation()` returns the last valid `TextLocation` in the TextBuffer associated with *text*.

LastTextBufferPosition

```
#include <Xol/textbuff.h>

TextPosition LastTextBufferPosition(
    TextBuffer     *text);
```

`LastTextBufferPosition()` returns the last valid `TextPosition` in the TextBuffer associated with *text*.

LineOfPosition

```
#include <Xol/textbuff.h>

int LineOfPosition(
    TextBuffer     *text,
    TextPosition   position);
```

The `LineOfPosition()` function returns the line number in which *position* occurs. If *position* is invalid, it returns EOF.

LocationOfPosition

```
#include <Xol/textbuff.h>

TextLocation LocationOfPosition(
    TextBuffer     *text,
    TextPosition   position);
```

`LocationOfPosition()` translates a *position* in the *text* TextBuffer to a `TextLocation` (see page 163). It returns the translated `TextLocation`. If the *position* is invalid, the *buffer* pointer in the `TextLocation` struct is set to NULL and the *line* and *offset* members in the `TextLocation` struct are set the last valid location in the TextBuffer; otherwise, *buffer* is set to a non-NULL (though useless) value.

NextLocation

```
#include <Xol/textbuff.h>

TextLocation NextLocation(
    TextBuffer      *textBuffer,
    TextLocation    current);
```

`NextLocation()` returns the `TextLocation` that follows the given *current* location in a TextBuffer. If the current location points to the end of the TextBuffer, this function wraps to the beginning of the TextBuffer.

NextTextBufferWord

```
#include <Xol/textbuff.h>

TextLocation NextTextBufferWord(
    TextBuffer      *textBuffer,
    TextLocation    current);
```

`NextTextBufferWord()` locates the beginning of the next word from a given *current* location in a TextBuffer. If the current location is within the last word in the TextBuffer, the function wraps to the beginning of the TextBuffer.

PositionOfLine

```
#include <Xol/textbuff.h>

TextPosition PositionOfLine(
    TextBuffer      *text,
    TextLine        lineindex);
```

The `PositionOfLine()` function returns the `TextPosition` corresponding to *lineindex*. If *lineindex* is invalid, it returns EOF.

PositionOfLocation

```
#include <Xol/textbuff.h>

TextPosition PositionOfLocation(
    TextBuffer      *text,
    TextLocation    location);
```

The `PositionOfLocation()` function returns the `TextPosition` corresponding to *location*. If *location* is invalid, it returns EOF.

PreviousLocation

```
#include <Xol/textbuff.h>

TextLocation PreviousLocation(
    TextBuffer      *textBuffer,
    TextLocation    current);
```

The `PreviousLocation()` function returns the `TextLocation` (see page 163) that precedes the given *current* location in a TextBuffer. If the current location points to the beginning of the TextBuffer, this function wraps to the end of the TextBuffer.

PreviousTextBufferWord

```
#include <Xol/textbuff.h>

TextLocation PreviousTextBufferWord(
    TextBuffer      *textBuffer,
    TextLocation    current);
```

`PreviousTextBufferWord()` locates the beginning of a word in a TextBuffer relative to a given *current* location. It returns the location of the beginning of the word that precedes the given current location. If the *current* location is within a word, this function returns beginning of the current word.

ReadFileIntoTextBuffer

```
#include <Xol/textbuff.h>

TextBuffer *ReadFileIntoTextBuffer(
    char                    *filename,
    TextUpdateFunction      f,
    XtPointer               d);
```

`ReadFileIntoTextBuffer()` allocates a new TextBuffer and reads the file denoted by the given filename into it. The supplied text update function *f* and the client data *d* are associated with the newly allocated TextBuffer. The function returns a pointer to this TextBuffer. See "ReplaceBlockInTextBuffer" on page 172 for more details of the `TextUpdateFunction`.

ReadStringIntoTextBuffer

```
#include <Xol/textbuff.h>

TextBuffer *ReadStringIntoTextBuffer(
    char                    *string,
    TextUpdateFunction      f,
    XtPointer               d);
```

`ReadStringIntoTextBuffer()` allocates a new TextBuffer and copies the given *string* into it. The supplied `TextUpdateFunction` and data pointer are associated with this TextBuffer. The function returns a pointer to this TextBuffer. See "ReplaceBlockInTextBuffer" on page 172 for more details of the `TextUpdateFunction`.

RegisterTextBufferScanFunctions

```
#include <Xol/textbuff.h>

void RegisterTextBufferScanFunctions(
    char *(*forward)(),
    char *(*backward)());
```

`RegisterTextBufferScanFunctions()` provides the capability to replace the default scan functions used by the `ForwardScanTextBuffer()` and `BackwardScanTextBuffer()` functions. These functions are called as:

```
(*forward)(string, curp, expression);
(*backward)(string, curp, expression);
```

and are responsible for returning either a pointer to the beginning of a match for the expression or NULL. Calling `RegisterTextBufferScanFunctions()` with NULL function pointers reinstates the default regular expression facility, as described in "Regular Expression Functions" on page 161.

RegisterTextBufferWordDefinition

```
#include <Xol/textbuff.h>

void RegisterTextBufferWordDefinition(
    int (*word_definition)());
```

`RegisterTextBufferWordDefinition()` provides the capability to replace the default word definition function used by the TextBuffer functions in this section. This function is called as:

```
(*word_definition)(c);
```

The function is responsible for returning nonzero if the character *c* is considered a character that can occur in a word, and zero otherwise. Calling `RegisterTextBufferWordDefinition()` with NULL reinstates the default word definition, which allows the following set of characters: a-z, A-Z, 0-9_

RegisterTextBufferUpdate

```
#include <Xol/textbuff.h>

void RegisterTextBufferUpdate(
    TextBuffer              *text,
    TextUpdateFunction      f,
    XtPointer               d);
```

`RegisterTextBufferUpdate()` associates the `TextUpdateFunction` *f* and data pointer *d* with the given TextBuffer *text*. This update function will be called whenever an update operation is performed on the TextBuffer. See "ReplaceBlockInTextBuffer" on page 172 for more details of the `TextUpdateFunction`.

Note – Calling `RegisterTextBufferUpdate()` increments a reference count mechanism used to determine when to actually free the TextBuffer. Calling the function with a NULL value for the function circumvents this mechanism.

ReplaceBlockInTextBuffer

```
#include <Xol/textbuff.h>

EditResult ReplaceBlockInTextBuffer(
    TextBuffer              *text,
    TextLocation            *startloc,
    TextLocation            *endloc,
    char                    *string,
    TextUpdateFunction      f,
    XtPointer               d);
```

`ReplaceBlockInTextBuffer()` updates the contents of the TextBuffer *text*. The characters stored between *startloc* (inclusive) and *endloc* (exclusive) are deleted and the string is inserted after *startloc*. If the edit succeeds and if `TextUpdateFunction` *f* is associated with TextBuffer text, then *f* is called with the following parameters:

```
(*f)(XtPointer d, TextBuffer *text, EDIT_SUCCESS)
```

All the other text update functions associated with TextBuffer text are called with the following parameters:

```
(XtPointer d, TextBuffer *text, EDIT_FAILURE)
```

`ReplaceBlockInTextBuffer()` stores the details of the editing operation it performs in *text->deleted* and *text->insert* `TextUndoItem` structures. The contents of these structures may be used for implementing an Undo mechanism. The hints provided in *text->deleted.hint* and *text->insert.hint* are an inclusive OR of:

```
#define TEXT_BUFFER_NOP                  (0)
#define TEXT_BUFFER_DELETE_START_LINE    (1L<<0)
#define TEXT_BUFFER_DELETE_START_CHARS   (1L<<1)
#define TEXT_BUFFER_DELETE_END_LINE      (1L<<2)
#define TEXT_BUFFER_DELETE_END_CHARS     (1L<<3)
#define TEXT_BUFFER_DELETE_JOIN_LINE     (1L<<4)
#define TEXT_BUFFER_DELETE_SIMPLE        (1L<<5)
#define TEXT_BUFFER_INSERT_SPLIT_LINE    (1L<<6)
#define TEXT_BUFFER_INSERT_LINE          (1L<<7)
#define TEXT_BUFFER_INSERT_CHARS         (1L<<8)
```

The meaning of each of these values is described below:

TEXT_BUFFER_NOP	No edit operation.
TEXT_BUFFER_DELETE_START_LINE	The deleted block started at beginning of some line.
TEXT_BUFFER_DELETE_START_CHARS	The deleted block did not start at the beginning of some line.
TEXT_BUFFER_DELETE_END_LINE	The end of the deleted block coincided with the end of some line.
TEXT_BUFFER_DELETE_END_CHARS	Some characters were deleted from the end of some line.
TEXT_BUFFER_DELETE_JOIN_LINE	Some characters were deleted and two lines were joined into a single line.
TEXT_BUFFER_DELETE_SIMPLE	The whole of the deleted block was confined to a single line.

TEXT_BUFFER_INSERT_SPLIT_LINE	One line was split into two lines and some characters were inserted at the split location.
TEXT_BUFFER_INSERT_LINE	A line was inserted without spliting an existing line.
TEXT_BUFFER_INSERT_CHARS	Some characters were inserted at the beginning of some existing line.

ReplaceCharInTextBuffer

```
#include <Xol/textbuff.h>

EditResult ReplaceCharInTextBuffer(
    TextBuffer              *text,
    TextLocation            *location,
    int                     c,
    TextUpdateFunction      f,
    XtPointer               d);
```

`ReplaceCharInTextBuffer()` replaces the character in the TextBuffer *text* at *location* with the character *c*. Everything described in "ReplaceBlockInTextBuffer" on page 172 about text update functions also applies to this function.

SaveTextBuffer

```
#include <Xol/textbuff.h>

SaveResult SaveTextBuffer(
    TextBuffer      *text,
    char            *filename);
```

`SaveTextBuffer()` writes the contents of the *text* TextBuffer to the file *filename*. If *filename* is NULL, it uses the *filename* argument that was given to the `AllocateTextBuffer()` function (see page 163).

`SaveTextBuffer()` returns a `SaveResult`, which can be SAVE_FAILURE or SAVE_SUCCESS.

StartCurrentTextBufferWord

```
#include <Xol/textbuff.h>

TextLocation StartCurrentTextBufferWord(
    TextBuffer     *textBuffer,
    TextLocation    current);
```

`StartCurrentTextBufferWord()` locates the beginning of a word in the TextBuffer relative to a given *current* location. The function returns the location of the beginning of the current word. Note: this return value will equal the given current value if the current location is the beginning of a word.

UnregisterTextBufferUpdate

```
#include <Xol/textbuff.h>

int UnregisterTextBufferUpdate(
    TextBuffer              *text,
    TextUpdateFunction      f,
    XtPointer               d);
```

The `UnregisterTextBufferUpdate()` function disassociates the `TextUpdateFunction` *f* and data pointer *d* with the given TextBuffer *text*. If the function/data pointer pair is not associated with the given TextBuffer, zero is returned; otherwise, the association is dissolved and one is returned. See "ReplaceBlockInTextBuffer" on page 172 for more details of the `TextUpdateFunction`.

TextBuffer Macros

The macros described in "Buffer Macros" on page 95 can also be used with the text buffer functions in this section.

See Also

Buffer Functions on page 95,
Regular Expression Functions on page 161,
"Text Buffer Functions for Internationalization" on page 176.

5

Text Buffer Functions for Internationalization

The text buffer functions in this section provide multibyte equivalents to the single-byte OLIT text buffer functions in the previous section.

OlAllocateTextBuffer

```
#include <Xol/Oltextbuff.h>

OlTextBufferPtr OlAllocateTextBuffer(
    OlStrRep              strrep,
    char                  *filename,
    TextUpdateFunction    update_func,
    XtPointer             data);
```

Arguments

strrep	Specifies the text format
filename	Specifies the filename
update_func	The update function
data	Client data

`OlAllocateTextBuffer()` allocates a new TextBuffer structure, initializes the members of the structure, does one more step described below, and returns a pointer to the newly allocated structure.

It is possible to register one or more text update functions (of type `TextUpdateFunction`) with a `TextBuffer`. As the name suggests, the text update functions are invoked by the toolkit when the `TextBuffer` is updated. In the course of registering a text update function, a client data (of type `XtPointer`) must be provided with the function; this client data can be NULL. The client data is passed to the associated text update function when the function is invoked by the toolkit (See "OlReplaceBlockInTextBuffer" on page 197).

The argument *update_func* is a text update function that, together with its associated client data *data*, is registered with the newly allocated TextBuffer, before `AllocateTextBuffer()` returns. The programmer must use `OlFreeTextBuffer()` function to free the TextBuffer.

The *strrep* argument can have the following values:

Value	Meaning
OL_SB_STR_REP	Single-byte character representation
OL_WC_STR_REP	Wide character representation
OL_MB_STR_REP	Multibyte character representation

See Also

"OlFreeTextBuffer" on page 180,
"OlReadFileIntoTextBuffer" on page 193,
"OlReadStringIntoTextBuffer" on page 194.

OlBackwardScanTextBuffer

```
#include <Xol/Oltextbuff.h>

ScanResult OlBackwardScanTextBuffer(
    OlTextBufferPtr    text,
    OlStr              exp,
    TextLocation       *location);
```

Arguments

text	The text buffer
exp	The expression to scan for
location	The location to start scanning at

`OlBackwardScanTextBuffer()` scans towards the beginning of the buffer for a given expression in the `OlTextBuffer` starting at *location*. The `ScanResult` can have the following values:

SCAN_NOTFOUND	The scan wrapped without finding a match.
SCAN_WRAPPED	A match was found at a location after the start location
SCAN_FOUND	A match was found at a location before the start location
SCAN_INVALID	Either the location or the expression was invalid

See Also

"OlForwardScanTextBuffer" on page 179.

OlCopyTextBufferBlock

```
#include <Xol/Oltextbuff.h>

int OlCopyTextBufferBlock(
    OlTextBufferPtr    text,
    OlStr              outbuffer,
    int                num_bytes,
    TextPosition       start_position,
    TextPosition       end_position);
```

Arguments

text	The text buffer
outbuffer	The buffer to output the text to
num_bytes	Size of the text block
start_position	Beginning of text block
end_position	End of text block

`OlCopyTextBufferBlock()` retrieves a text block from the `OlTextBuffer`. The block is defined as the characters between *start_position* and *end_position* inclusive. If *num_bytes* is not sufficient, `OlCopyTextBufferBlock()` returns –1; otherwise, it returns actual bytes used.

Note – The storage for the copy is allocated by the caller. It is the responsibility of the caller to ensure that enough storage is allocated to copy (*end_position – start_position*) + (*bytes to store null character*).

See Also

"OlAllocateTextBuffer" on page 176,
"OlRegisterTextBufferUpdate" on page 197,
"OlGetTextBufferCharAtLoc" on page 182,
"OlGetTextBufferLine" on page 183.

OlEndCurrentTextBufferWord

```
#include <Xol/Oltextbuff.h>

TextLocation *OlEndCurrentTextBufferWord(
    OlTextBufferPtr    text,
    TextLocation       *current);
```

Arguments

text	The text buffer
current	Specifies current location

`OlEndCurrentTextBufferWord()` locates the end of a word in the `OlTextBuffer` relative to a given current location. It returns the location of the end of the current word. The return value will equal the given current value if the current location is already at the end of a word. If the location is not in a word, it returns the end of the "not word" region it is in.

Note – The location passed to `OlEndCurrentTextBufferWord()` is modified. It contains the end of the current buffer word (or "not word") at the end of the call.

See Also

"OlPreviousTextBufferWord" on page 192,
"OlNextTextBufferWord" on page 189.

OlForwardScanTextBuffer

```
#include <Xol/Oltextbuff.h>

ScanResult OlForwardScanTextBuffer(
    OlTextBufferPtr    text,
    OlStr              exp,
    TextLocation       *location);
```

Arguments

text	The text buffer
exp	Specifies the expression to scan for
location	The location to start scanning at

`OlForwardScanTextBuffer()` scans towards the end of the buffer, for a given expression in the `OlTextBuffer` starting at `location`. A ScanResult is returned, which indicates the following:

SCAN_NOTFOUND	The scan wrapped without finding a match.
SCAN_WRAPPED	A match was found at a location before the start location.
SCAN_FOUND	A match was found at a location after the start location.
SCAN_INVALID	Either the location or the expression was invalid

See Also

"OlBackwardScanTextBuffer" on page 177.

OlFreeTextBuffer

```
#include <Xol/Oltextbuff.h>

void OlFreeTextBuffer(
    OlTextBufferPtr         text,
    TextUpdateFunction      update_func,
    XtPointer               data);
```

Arguments

text	The text buffer to free
exp	The update function
data	Data

`OlFreeTextBuffer()` deallocates storage associated with a given `OlTextBuffer`. See "OlReplaceBlockInTextBuffer" on page 197 for more details of the TextUpdateFunction.

Note – The storage is not actually freed if the `OlTextBuffer` is still associated with other update function/data pairs.

See Also

"OlAllocateTextBuffer" on page 176,
"OlRegisterTextBufferUpdate" on page 197.

OlGetTextBufferBlock

```
#include <Xol/Oltextbuff.h>

OlStr OlGetTextBufferBlock(
    OlTextBufferPtr     text,
    TextLocation        *start_location,
    TextLocation        *end_location);
```

Arguments

text	The text buffer to retrieve block from
start_location	Start of the text block
end_location	End of the text block

`OlGetTextBufferBlock()` retrieves a text block from the *text* TextBuffer. The block is defined as the characters between *start_location* and *end_location* inclusive. It returns a pointer to a string containing the copy. If the parameters are invalid NULL is returned.

Note – The storage for the copy is allocated by this routine. It is the responsibility of the caller to free this storage when it becomes dispensable.

See Also

"OlAllocateTextBuffer" on page 176,
"OlRegisterTextBufferUpdate" on page 197,
"OlGetTextBufferCharAtLoc" on page 182,
"OlGetTextBufferLine" on page 183.

OlGetTextBufferBuffer

```
#include <Xol/Oltextbuff.h>

Buffer *OlGetTextBufferBuffer(
    OlTextBufferPtr    text,
    TextLine           line);
```

Arguments

text	The text buffer
line	The line to retrieve a pointer for

`OlGetTextBufferBuffer()` retrieves a pointer to the Buffer stored in `OlTextBuffer` text for a line. This pointer is volatile; subsequent calls to any `OlTextBuffer` routine may make it invalid. If a more permanent copy of this Buffer is required the buffer utility, `CopyBuffer()` can be used to create a private copy of it.

See Also

"OlGetTextBufferBlock" on page 180,
"OlAllocateTextBuffer" on page 176,
"OlRegisterTextBufferUpdate" on page 197.

OlGetTextBufferCharAtLoc

```
#include <Xol/Oltextbuff.h>

OlStr OlGetTextBufferCharAtLoc(
    OlTextBufferPtr    text,
    TextLocation       *location);
```

Arguments

text	The text buffer
location	The location at which to get a character

`OlGetTextBufferCharAtLoc()` retrieves a character stored in the `OlTextBuffer` at *location*. It returns either the pointer to the character itself or NULL if *location* is outside the range of valid locations within the `OlTextBuffer`.

See Also

"OlAllocateTextBuffer" on page 176,
"OlRegisterTextBufferUpdate" on page 197,
"OlGetTextBufferBlock" on page 180,
"OlGetTextBufferLine" on page 183.

OlGetTextBufferFileName

```
#include <Xol/Oltextbuff.h>

String OlGetTextBufferFileName(
    OlTextBufferPtr text);
```

Arguments

text	The text buffer for which to get a filename.

`OlGetTextBufferFileName()` returns the file name associated with the buffer. Otherwise, it returns NULL.

OlGetTextBufferLine

```
#include <Xol/Oltextbuff.h>

OlStr OlGetTextBufferLine(
    OlTextBufferPtr    text,
    TextLine           lineindex);
```

Arguments

text	The text buffer
lineindex	Index of the line to retrieve the contents for

`OlGetTextBufferLine()` retrieves the contents of line from the `OlTextBuffer`. It returns a pointer to a string containing the copy of the contents of the line or NULL if the line is outside the range of valid lines in text.

Note – The storage for the copy is allocated by this routine. It is the responsibility of the caller to free this storage when it becomes dispensable.

See Also

"OlAllocateTextBuffer" on page 176,
"OlRegisterTextBufferUpdate" on page 197,
"OlGetTextBufferCharAtLoc" on page 182,
"OlGetTextBufferBlock" on page 180.

OlGetTextUndoDeleteItem

```
#include <Xol/Oltextbuff.h>

OlTextUndoItem OlGetTextUndoDeleteItem(
    OlTextBufferPtr text);
```

Arguments

text	The text buffer for which to get the undo delete item.

`OlGetTextUndoDeleteItem()` returns a `OlTextUndoItem` struct containing the value of "deleted" undo item. A copy of the deleted string is provided in the returned struct.

OlGetTextUndoInsertItem

```
#include <Xol/Oltextbuff.h>

OlTextUndoItem OlGetTextUndoInsertItem(
    OlTextBufferPtr text);
```

Arguments

text	The text buffer for which to get the undo insert item

`OlGetTextUndoInsertItem()` returns a `OlTextUndoItem` struct containing the value of "insert" undo item. A copy of the insert string is provided in the returned struct.

OlIncrementTextBufferLocation

```
#include <Xol/Oltextbuff.h>

TextLocation *OlIncrementTextBufferLocation(
    OlTextBufferPtr    text,
    TextLocation       *location,
    TextLine           line,
    TextPosition       offset);
```

Arguments

text	The text buffer
location	The location to increment
line	The lines to increment by
offset	The character offset to increment by

`OlIncrementTextBufferLocation()` increments a location by lines and/or offset characters. It returns a pointer to the modified starting *location*. If *line* or *offset* are negative, the function performs a decrement operation. If the starting location is NULL, NULL is returned. If the starting location or the resulting location is invalid the pointer to the starting location is returned without modification; otherwise, the starting location is modified and a pointer to the starting location is returned.

Note – Do not expect the location passed to this function to remain unchanged.

See Also

"OlNextLocation" on page 189,
"OlPreviousLocation" on page 192.

OlIsTextBufferEmpty

```
#include <Xol/Oltextbuff.h>

Boolean OlIsTextBufferEmpty(
    OlTextBufferPtr text);
```

Arguments

text	The text buffer to test

`OlIsTextBufferEmpty()` returns TRUE if the text buffer is empty. Otherwise, it returns FALSE.

OlIsTextBufferModified

```
#include <Xol/Oltextbuff.h>

Boolean OlIsTextBufferModified(
    OlTextBufferPtr text);
```

Arguments

text	The text buffer

`OlIsTextBufferModified()` returns TRUE if the text buffer has been modified since the last save. Otherwise, it returns FALSE.

OlLastCharInTextBufferLine

```
#include <Xol/Oltextbuff.h>

int OlLastCharInTextBufferLine(
    OlTextBufferPtr    text,
    TextLine           line);
```

Arguments

text	The text buffer
line	The line to find the last character offset for

`OlLastCharInTextBufferLine()` returns the character offset of the last character in the text buffer line.

OlLastTextBufferLine

```
#include <Xol/Oltextbuff.h>

TextLine OlLastTextBufferLine(
    OlTextBufferPtr text);
```

Arguments

text	The text buffer for which to find the last line number

`OlLastTextBufferLine()` returns the last text buffer line number.

OlLastTextBufferLocation

```
#include <Xol/Oltextbuff.h>

TextPosition OlLastTextBufferLocation(
    OlTextBufferPtr    text);
    TextLocation       *last);
```

Arguments

text	The text buffer
last	Address to return the last location to

`OlLastTextBufferLocation()` returns the pointer to the last valid `TextLocation` in the `OlTextBuffer` associated with text. If the *last* argument is NULL, space for last `TextLocation` is allocated; otherwise, the *last* argument contains the last valid `TextLocation`.

See Also

"OlLastTextBufferPosition" on page 187.

OlLastTextBufferPosition

```
#include <Xol/Oltextbuff.h>

TextPosition OlLastTextBufferPosition(
    OlTextBufferPtr text);
```

Arguments

text	The text buffer

`OlLastTextBufferPosition()` returns the last valid `TextPosition` in the `OlTextBuffer` associated with text.

See Also

"OlLastTextBufferLocation" on page 186.

OlLineOfPosition

```
#include <Xol/Oltextbuff.h>

TextLine OlLineOfPosition(
    OlTextBufferPtr    text,
    TextPosition       position);
```

Arguments

text	The text buffer

`OlLineOfPosition()` translates a position in the `OlTextBuffer` to a line index. It returns the translated line index, or EOF if the position is invalid.

See Also

"OlLineOfPosition" on page 187,
"OlPositionOfLocation" on page 191,
"OlLocationOfPosition" on page 188.

OlLinesInTextBuffer

```
#include <Xol/Oltextbuff.h>

int OlLinesInTextBuffer(
    OlTextBufferPtr text);
```

Arguments

text	The text buffer to get the number of lines for

`OlLinesInTextBuffer()` returns the number of lines in the given `OlTextBuffer`.

OlLocationOfPosition

```
#include <Xol/Oltextbuff.h>

TextLocation *OlLocationOfPosition(
    OlTextBufferPtr    text,
    TextPosition       position,
    TextLocation       *location);
```

Arguments

text	The text buffer
position	The position to translate to a `TextLocation`
location	The address to store the translated `TextLocation`

`OlLocationOfPosition()` translates a position in the `OlTextBuffer` to a `TextLocation`. It expects a pointer to a `TextLocation` in the *location* argument to deposit the translated `TextLocation`. If the *location* argument is NULL, it allocates space. It returns a pointer to the allocated `TextLocation` or the passed `TextLocation`, with the translated value deposited in it. If the position is invalid, the Buffer pointer buffer is set to NULL and the *line* and *offset* members are set to the last valid location in the `OlTextBuffer`; otherwise, the buffer is set to a non-NULL (though useless) value.

Note – The storage space for `TextLocation`, if not provided by the caller, is allocated by this function.

See Also

"OlLineOfPosition" on page 187,
"OlPositionOfLocation" on page 191,
"OlLocationOfPosition" on page 188.

OlNextLocation

```
#include <Xol/Oltextbuff.h>

TextLocation *OlNextLocation(
    OlTextBufferPtr    text,
    TextLocation       *current);
```

Arguments

text	The text buffer
current	The current location

`OlNextLocation()` returns the pointer to the `TextLocation` that follows the given current location in an `OlTextBuffer`. If the current location points to the end of the `OlTextBuffer`, this function wraps to the beginning of the `OlTextBuffer`.

Note – The location passed to this function is modified. It contains the next location at the end of the call.

See Also

"OlPreviousLocation" on page 192.

OlNextTextBufferWord

```
#include <Xol/Oltextbuff.h>

TextLocation *OlNextTextBufferWord(
    OlTextBufferPtr    text,
    TextLocation       *current);
```

Arguments

text	The text buffer
current	The current location

`OlNextTextBufferWord()` locates the beginning of the next word from a given current location in an `OlTextBuffer`. If the current location is within the last word in the `OlTextBuffer`, the function wraps to the beginning of the `OlTextBuffer`.

Note – The location passed to this function is modified. It contains the start of the next buffer word at the end of the call.

See Also

"OlPreviousTextBufferWord" on page 192,
"OlStartCurrentTextBufferWord" on page 201.

OlNumBytesInTextBufferLine

```
#include <Xol/Oltextbuff.h>

int OlNumBytesInTextBufferLine(
    OlTextBufferPtr    text,
    TextLine           line);
```

Arguments

text	The text buffer
line	The line to get the number of bytes for

`OlNumBytesInTextBufferLine()` returns the number of bytes in *line*.

OlNumCharsInTextBufferLine

```
#include <Xol/Oltextbuff.h>

int OlNumCharsInTextBufferLine(
    OlTextBufferPtr    text,
    TextLine           line);
```

Arguments

text	The text buffer
line	The line for which to get the number of chars

`OlNumCharsInTextBufferLine()` returns the number of characters in the specified *line*.

OlNumUnitsInTextBufferLine

```
#include <Xol/Oltextbuff.h>

int OlNumUnitsInTextBufferLine(
    OlTextBufferPtr    text,
    TextLine           line);
```

Arguments

text	The text buffer
line	The line for which to get the number of units

`OlNumCharsInTextBufferLine()` returns the number of units in the specified *line*.

OlPositionOfLine()

```
#include <Xol/Oltextbuff.h>

TextPosition OlPositionOfLine(
    OlTextBufferPtr    text,
    TextLine           lineindex);
```

Arguments

text	The text buffer
lineindex	The line index to translate into a text position

`OlPositionOfLine()` translates a *lineindex* in the `OlTextBuffer` to a `TextPosition`. It returns the translated `TextPosition` or EOF if the *lineindex* is invalid.

OlPositionOfLocation

```
#include <Xol/Oltextbuff.h>

TextPosition OlPositionOfLocation(
    OlTextBufferPtr    text,
    TextLocation       *location);
```

Arguments

text	The text buffer
location	The location to translate to a TextPosition

`OlPositionOfLocation()` translates a location in the `OlTextBuffer` to a `TextPosition`. It returns the translated `TextPosition` or EOF if the *location* is invalid.

OlPreviousLocation

```
#include <Xol/Oltextbuff.h>

TextLocation *OlPreviousLocation(
    OlTextBufferPtr    text,
    TextLocation       *current);
```

Arguments

text	The text buffer
current	The current location

`OlPreviousLocation()` function the pointer to the `TextLocation` that precedes the given current location in a `OlTextBuffer`. If the current location points to the beginning of the `OlTextBuffer`, this function wraps to the end of the `OlTextBuffer`.

Note – The current location is modified. It contains the previous location at the end of the call.

See Also

"OlNextLocation" on page 189.

OlPreviousTextBufferWord

```
#include <Xol/Oltextbuff.h>

TextLocation *OlPreviousTextBufferWord(
    OlTextBufferPtr    text,
    TextLocation       *current);
```

Arguments

text	The text buffer
current	The current location

`OlPreviousTextBufferWord()` locates the beginning of a word in a `OlTextBuffer` relative to a given current location. It returns the location of the beginning of the word that precedes the given *current* location. If the current location is within a word, this function will skip over the current word. If the current word is the first word in the `OlTextBuffer`, the function wraps to the end of the `OlTextBuffer`.

Note – The location passed to this function is modified. It contains the start of the previous buffer word at the end of the call.

See Also

"OlPreviousTextBufferWord" on page 192

OlReadFileIntoTextBuffer

```
#include <Xol/Oltextbuff.h>

OlTextBufferPtr OlReadFileIntoTextBuffer(
    OlStrRep              strrep,
    char                  *filename,
    TextUpdateFunction    update_func,
    XtPointer             data);
```

Arguments

strrep	The string representation (OL_SB_STR_REP, OL_WC_STR_REP, or OL_MB_STR_REP)
filename	The file to be read
update_func	The update function
data	Data

`OlReadFileIntoTextBuffer()` reads the given file into a newly allocated `OlTextBuffer`. The supplied `TextUpdateFunction` and data pointer are associated with this `OlTextBuffer`. See "OlReplaceBlockInTextBuffer" on page 197 for more details of the `TextUpdateFunction`.

See Also

"OlReadStringIntoTextBuffer" on page 194.

OlReadStringIntoTextBuffer

```
#include <Xol/Oltextbuff.h>
OlTextBufferPtr OlReadStringIntoTextBuffer(
    OlStrRep              strrep,
    char                  *string,
    TextUpdateFunction    update_func,
    XtPointer             data);
```

Arguments

strrep	The string representation (OL_SB_STR_REP, OL_WC_STR_REP, or OL_MB_STR_REP
string	The string to be read
update_func	The update function
data	Data

`OlReadStringIntoTextBuffer()` copies the given string into a newly allocated `OlTextBuffer`. The supplied `TextUpdateFunction` and *data* pointer are associated with this `OlTextBuffer`. See "OlReplaceBlockInTextBuffer" on page 197 for more details of the `TextUpdateFunction`.

See Also

"OlReadFileIntoTextBuffer" on page 193.

OlRegisterAllTextBufferScanFunctions

```
#include <Xol/Oltextbuff.h>

void OlRegisterAllTextBufferScanFunctions(
    OlStrRep           strrep,
    OlStrScanDefFunc   forward_scan_func,
    OlStrScanDefFunc   backward_scan_func);
```

Arguments

strrep	The string representation (OL_SB_STR_REP, OL_WC_STR_REP, or OL_MB_STR_REP
forward_scan_func	The forward scan function to be used by all OlTextBuffers
backward_scan_func	The backward scan function to be used by all OlTextBuffers

The *forward_scan_func* and *backward_scan_func* arguments specify `OlStrScanDefFunc()` functions. `OlStrScanDefFunc` is defined as:

```
typedef XtPointer  (*OlStrScanDefFunc)(
    OlStr string,
    OlStr curp,
    OlStr expression);
```

`OlRegisterAllTextBufferWordDefinition()` provides the capability to replace the text buffer functions used by all `OlTextBuffers`.

OlRegisterAllTextBufferWordDefinition

```
#include <Xol/Oltextbuff.h>

void OlRegisterAllTextBufferWordDefinition(
    OlStrRep           strrep,
    OlStrWordDefFunc   word_definition_func);
```

Arguments

strrep	The string representation
word_definition_func	The word definition function used by all OlTextBuffers

The *word_definition_func* argument specifies an `OlStrWordDefFunc()`, which is defined as:

```
typedef Boolean    (*OlStrWordDefFunc)(OlStr rc);
```

`OlRegisterAllTextBufferWordDefinition()` provides the capability to replace the word definition function used by all `OlTextBuffers`. These functions are responsible for returning TRUE if the character that *rc* points to can occur in a word, and FALSE otherwise. Calling this function with NULL reinstates the default word definition function associated with the text format.

OlRegisterPerTextBufferScanFunctions

```
#include <Xol/Oltextbuff.h>

void OlRegisterPerTextBufferScanFunctions(
    OlTextBufferPtr    text,
    OlStrScanDefFunc   forward_scan_func,
    OlStrScanDefFunc   backward_scan_func);
```

Arguments

strrep	The string representation (OL_SB_STR_REP, OL_WC_STR_REP, or OL_MB_STR_REP)
forward_scan_func	The forward scan function used by OlTextBuffers
backward_scan_func	The backward scan function used by OlTextBuffers

The arguments *forward_scan_func* and *backward_scan_func* specify `OlStrScanDefFunc()` functions. `OlStrScanDefFunc` is defined as:

```
typedef XtPointer  (*OlStrScanDefFunc)(
    OlStr string,
    OlStr curp,
    OlStr expression);
```

`OlRegisterPerTextBufferScanFunctions()` provides the capability to replace the scan functions used by the `OlForwardScanTextBuffer()` and `OlBackwardScanTextBuffer()` functions, as applied to the passed `OlTextBuffer` only.

`OlRegisterAllTextBufferScanFunctions()` provides the capability to replace the scan functions used by the `OlForwardScanTextBuffer()` and `OlBackwardScanTextBuffer()` functions, as applied to all `OlTextBuffers`. These functions are responsible for returning either a pointer to the beginning of a match for the expression or NULL. Calling this procedure with NULL function pointers reinstates the default regular expression facility associated with the text format.

OlRegisterPerTextBufferWordDefinition

```
#include <Xol/Oltextbuff.h>

void OlRegisterPerTextBufferWordDefinition(
    OlTextBufferPtr    text,
    OlStrWordDefFunc   word_definition_func);
```

Arguments

text	The text buffer
word_definition_func	The new word definition function

The *word_definition_func* argument specifies an `OlStrWordDefFunc()`, which is defined as:

```
typedef Boolean        (*OlStrWordDefFunc)(OlStr rc);
```

`OlRegisterPerTextBufferWordDefinition()` provides the capability to replace the word definition function used by the passed `OlTextBuffer`.

OlRegisterTextBufferUpdate

```
#include <Xol/Oltextbuff.h>

void OlRegisterTextBufferUpdate(
    OlTextBufferPtr         text,
    TextUpdateFunction      update_func,
    XtPointer               data);
```

Arguments

text	The text buffer
update_func	The update function
data	Data

`OlRegisterTextBufferUpdate()` associates the `TextUpdateFunction` *update_func* and data pointer *data* with the given `OlTextBuffer` text. This update function will be called whenever an update operation is performed on the `OlTextBuffer`. See "OlReplaceBlockInTextBuffer" on page 197 for more details of the `TextUpdateFunction`.

Note – Calling this function increments a reference count mechanism used to determine when to actually free the `OlTextBuffer`. Calling the function with a NULL value for the function circumvents this mechanism.

See Also

"OlUnregisterTextBufferUpdate" on page 203,
"OlReadStringIntoTextBuffer" on page 194,
"OlReadFileIntoTextBuffer" on page 193.

OlReplaceBlockInTextBuffer

```
#include <Xol/Oltextbuff.h>

EditResult OlReplaceBlockInTextBuffer(
    OlTextBufferPtr         text,
    TextLocation            *startloc,
    TextLocation            *endloc,
    OlStr                   string,
```

```
TextUpdateFunction     update_func,
XtPointer              data);
```

Arguments

text	The text buffer
startloc	The start location
endloc	the end location
string	The string to replace the block with
update_func	The update function
data	Data

`OlReplaceBlockInTextBuffer()` updates the contents of the TextBuffer text. The characters stored between *startloc* (inclusive) and *endloc* (exclusive) are deleted and the string is inserted after *startloc*. If the edit succeeds and if `TextUpdateFunction` *update_func* is associated with TextBuffer text, then *update_func* is called with the following parameters:

```
(*update_func)(XtPointer d, TextBuffer *text, EDIT_SUCCESS)
```

All the other text update functions associated with TextBuffer text are called with the following parameters:

```
(XtPointer data, TextBuffer *text, EDIT_FAILURE)
```

`OlReplaceBlockInTextBuffer()` stores the details of the editing operation it performs in *text->deleted* and *text->insert* `OlTextUndoItem` structures. The contents of these structures may be used for implementing an Undo mechanism. The hints provided in *text->deleted.hint* and *text->insert.hint* are an inclusive OR of:

```
#define TEXT_BUFFER_NOP                    (0)
#define TEXT_BUFFER_DELETE_START_LINE      (1L<<0)
#define TEXT_BUFFER_DELETE_START_CHARS     (1L<<1)
#define TEXT_BUFFER_DELETE_END_LINE        (1L<<2)
#define TEXT_BUFFER_DELETE_END_CHARS       (1L<<3)
#define TEXT_BUFFER_DELETE_JOIN_LINE       (1L<<4)
#define TEXT_BUFFER_DELETE_SIMPLE          (1L<<5)
#define TEXT_BUFFER_INSERT_SPLIT_LINE      (1L<<6)
#define TEXT_BUFFER_INSERT_LINE            (1L<<7)
#define TEXT_BUFFER_INSERT_CHARS           (1L<<8)
```

The meaning of each of these values is described below:

TEXT_BUFFER_NOP	No edit operation.
TEXT_BUFFER_DELETE_START_LINE	The deleted block started at beginning of some line.
TEXT_BUFFER_DELETE_START_CHARS	The deleted block did not start at the beginning of some line.
TEXT_BUFFER_DELETE_END_LINE	The end of the deleted block coincided with the end of some line.
TEXT_BUFFER_DELETE_END_CHARS	Some characters were deleted from the end of some line.
TEXT_BUFFER_DELETE_JOIN_LINE	Some characters were deleted and two lines were joined into a single line.
TEXT_BUFFER_DELETE_SIMPLE	The whole of the deleted block was confined to a single line.
TEXT_BUFFER_INSERT_SPLIT_LINE	One line was split into two lines and some characters were inserted at the split location.
TEXT_BUFFER_INSERT_LINE	A line was inserted without spliting an existing line.
TEXT_BUFFER_INSERT_CHARS	Some characters were inserted at the beginning of some existing line.

See Also

"OlReplaceCharInTextBuffer" on page 199.

OlReplaceCharInTextBuffer

```
#include <Xol/Oltextbuff.h>

EditResult OlReplaceCharInTextBuffer(
    OlTextBufferPtr       text,
    TextLocation          *location,
    OlStr                 c,
    TextUpdateFunction    update_func,
    XtPointer             data);
```

Arguments

text	The text buffer
location	The location of text
c	replacement buffer
update_func	The update function
data	Data

`OlReplaceCharInTextBuffer()` replaces the character in the `OlTextBuffer`.

See Also

"OlReplaceBlockInTextBuffer" on page 197.

OlSaveTextBuffer

```
#include <Xol/Oltextbuff.h>

SaveResult OlSaveTextBuffer(
    OlTextBufferPtr    text,
    char               *filename);
```

Arguments

text	The text buffer
filename	The filename to write the text buffer to

`OlSaveTextBuffer()` writes the contents of the `OlTextBuffer` to the file *filename*. It returns a `SaveResult`, which can be SAVE_FAILURE or SAVE_SUCCESS.

OlSetTextUndoDeleteItem

```
#include <Xol/Oltextbuff.h>

void OlSetTextUndoDeleteItem(
    OlTextBufferPtr    text,
    OlTextUndoItem     text_undo_deleted);
```

Arguments

text	The text buffer
text_undo_deleted	The item for which the delete was undone

`OlSetTextUndoDeleteItem()` sets the "deleted" `OlTextUndoItem` of the `OlTextBuffer` to the value of the passed `OlTextUndoItem`. The "deleted" string is copied in. `OlTextUndoItem` is defined as:

```
typedef struct _OlTextUndoItem {
    OlStr           string;
    TextLocation    start;
    TextLocation    end;
    TextUndoHint    hint;
} OlTextUndoItem;
```

OlSetTextUndoInsertItem

```
#include <Xol/Oltextbuff.h>

void OlSetTextUndoInsertItem(
    OlTextBufferPtr     text,
    OlTextUndoItem      text_undo_insert);
```

Arguments

text	The text buffer
text_undo_insert	The item for which the insert was undone

`OlSetTextUndoInsertItem()` sets the "insert" `OlTextUndoItem` of the `OlTextBuffer` to the value of the passed `OlTextUndoItem`. The "insert" string is copied in.

OlStartCurrentTextBufferWord

```
#include <Xol/Oltextbuff.h>

TextLocation *OlStartCurrentTextBufferWord(
    OlTextBufferPtr     text,
    TextLocation        *current);
```

Arguments

text	The text buffer
current	The location

`OlStartCurrentTextBufferWord()` locates the beginning of a word in the `OlTextBuffer` relative to a given current location. It returns the location of the beginning of the current word.

Note – This return value will equal the given current value if the current location is the beginning of a word. If the location is not in a word, it returns the start of the "not word" region it is in. The location passed to this function is modified. It contains the start of the current buffer word (or "not word") at the end of the call.

See Also

"OlPreviousTextBufferWord" on page 192,
"OlNextTextBufferWord" on page 189.

OlTextEditOlTextBuffer

```
#include <Xol/buffutil.h>
#include <Xol/Oltextbuff.h>
#include <Xol/Dynamic.h>
#include <Xol/TextEdit.h>

OlTextBufferPtr OlTextEditOlTextBuffer(
    TextEditWidget     ctx);
```

`OlTextEditOlTextBuffer()` retrieves the `OlTextBufferPtr` associated with the TextEdit widget *ctx*. This buffer exists only when the value of `XtNtextFormat` for the widget is not OL_SB_STR_REP. This pointer can be used to access the facilities provided by the multibyte functions. In case `XtNtextFormat` is OL_SB_STR_REP, this function returns a NULL pointer.

OlUnitOffsetOfLocation

```
#include <Xol/Oltextbuff.h>

UnitPosition OlUnitOffsetOfLocation(
    OlTextBufferPtr    text,
    TextLocation       *loc);
```

Arguments

text	The text buffer
loc	The location

`OlUnitOffsetOfLocation()` returns the font offset corresponding to the `TextLocation` offset passed to it. The units are `char` for single-byte and multi-byte and `wchar_t` for wide character.

OlUnregisterTextBufferUpdate

```
#include <Xol/Oltextbuff.h>

int OlUnregisterTextBufferUpdate(
    OlTextBufferPtr         text,
    TextUpdateFunction      update_func,
    XtPointer               data);
```

Arguments

text	The text buffer
update_func	The update function to disassociate
data	Data

`OlUnregisterTextBufferUpdate()` disassociates the `TextUpdateFunction` and data pointer data with the given `OlTextBuffer` text. If the function/data pointer pair is not associated with the given `OlTextBuffer`, zero is returned; otherwise, the association is dissolved and 1 is returned. See "OlReplaceBlockInTextBuffer" on page 197 for more details of the `TextUpdateFunction`.

See Also

"OlRegisterTextBufferUpdate" on page 197,
"OlFreeTextBuffer" on page 180.

Text Selection Operations

The Caption, NumericField, StaticText, TextEdit, TextField, and TextLine widgets use the following operations to copy and move text.

Setting Insert Point

Clicking SELECT sets the insert point at the boundary between two characters or spaces nearest the pointer. This makes an inactive caret active and highlights the header of the main window (base window or popup window) containing the specific text widget, to show which window has the input focus. Any active selection on the screen is deselected.

Wipethrough Selection

Pressing and dragging SELECT marks the bounds of a new selection and highlights it, and deselects any other active selection on the screen. While SELECT is pressed, the active or inactive caret that marks the insert point is invisible, but when SELECT is released, the insert point is left at the position of the release. This does not make the insert point (caret) active if it is not already active.

The selection starts with the character where SELECT is pressed and extends to the character where SELECT is released. If the pointer moves outside the widget and the widget can scroll in that direction (i.e., there is a scrollbar for that direction), the widget scrolls additional text into the widget and adds it to the selection. The rate at which text scrolls into the widget is the same rate at which pressing SELECT on the arrows of the Scrollbar scrolls the widget.

Deletion of the New Selection

If new text is entered from the keyboard or pasted from the CLIPBOARD, it replaces the selection.

Adjusted Selection

Clicking SELECT, moving the pointer, and clicking ADJUST marks the bounds of a selection and highlights it. A subsequent click of ADJUST changes the end bound of the selection. The ADJUST may also follow a wipe-through selection. The selection starts with the character where SELECT was clicked and extends to the character where ADJUST is clicked. The insert point is moved to the position of the ADJUST. As above, deletion of the new selection is pending.

Multiclick Selection

Double-clicking SELECT selects the word nearest the pointer. In case of a tie, the word to the left is selected. Triple-clicking SELECT selects the entire line, and quadruple-clicking selects the entire content. The selection is highlighted and the insert point is left at the position of the multi-click.

- Copying Text – Using COPY copies any selected text to the CLIPBOARD and deselects it.
- Cutting Text – Using CUT moves any selected text to the CLIPBOARD and deletes it from the Input Field.
- Pasting Text – After setting the insert point, using PASTE copies text from the CLIPBOARD as though it were typed in, leaving the insert point at the end of the pasted text. This will replace any text currently selected in the widget. Note that the data on the CLIPBOARD may have come from outside the input field, but it must be text. If the CLIPBOARD is empty, the system beeps.

Toolkit Resource Functions

OlGetApplicationValues

```
#include <Xol/OpenLook.h>

void OlGetApplicationValues(
    Widget      widget,
    ArgList     args,
    Cardinal    num_args);
```

`OlGetApplicationValues()` retrieves the value of any of the OLIT Toolkit Resources listed in Table 2-1 on page 7. OLIT toolkit resources have an application-wide scope. The *widget* argument is used to derive the screen and display. The *args* argument is a list of name/address pairs that contain the resource names and the addresses into which the resource values are to be stored. The *num_args* argument specifies the number of name/address pairs in *args*. If the resource name supplied in the *args* list is not recognized by the toolkit, the corresponding supplied address is not accessed by the toolkit. An application should query the value of an OLIT toolkit resource each time it needs it.

OlSetApplicationValues

```
#include <Xol/OpenLook.h>

void OlSetApplicationValues(
    Widget      widget,
    ArgList     args,
    Cardinal    num_args);
```

`OlSetApplicationValues()` sets the OLIT toolkit resources values. The *widget* and *num_args* arguments are used as in `OlGetApplicationValues()`. The *args* argument is a list of name/value pairs that contain the resource names and the values; as with `XtSetValues()`, if a resource name does not fit into an `XtArgVal`, the corresponding *args* value field contains a pointer to the resource value.

See Also

`XtGetValues()` and `XtSetValues()` in the *Xt Intrinsics Reference Manual*.

Virtual Event Functions

For all functions discussed here, the registration order determines the search order when doing a lookup.

LookupOlInputEvent

```
#include <Xol/Dynamic.h>

OlInputEvent LookupOlInputEvent(
    Widget      w,
    XEvent      *event,
    KeySym      *keysym,
    char        **buffer,
    int         *length);
```

`LookupOlInputEvent()` decodes the *event* for widget *w* to an `OlInputEvent`. See the table of OLIT Activation Types (Table 3-1 on page 65) for a list of the `OlInputEvent` values (listed in the "Activation Type" column) this function may return. The event passed should be a ButtonPress, ButtonRelease, or KeyPress event. `LookupOlInputEvent()` attempts to decode this event based on the settings of the OPEN LOOK defined dynamic mouse and keyboard settings.

If the event is a KeyPress, the function may return the *keysym*, *buffer*, and/or *length* of the buffer returned from a call to `XLookupString()`. It returns these values if non-NULL values are provided by the caller.

OlDetermineMouseAction

```
#include <Xol/Dynamic.h>

ButtonAction OlDetermineMouseAction(
    Widget      w,
    XEvent      *event);
```

`OlDetermineMouseAction()` determines the kind of mouse gesture that is being attempted: it will return one of the values MOUSE_CLICK, MOUSE_MULTI_CLICK, or MOUSE_MOVE. This function is normally called immediately upon receipt of a mouse button press event. It uses the current settings for the `XtNmouseDampingFactor` and `XtNmultiClickTimeout` resources to determine the kind of gesture being made.

`OlDetermineMouseAction()` performs an active pointer grab. This grab is released for the CLICK type actions, but not for MOUSE_MOVE. It is the responsibility of the caller to ungrab the pointer if the action is MOUSE_MOVE.

Example

```
static void ButtonConsumeCB (w, client_data, call_data)
widget          w;
XtPointer       client_data;
XtPointer       call_data;
{
Position        x, y;
OlVirtualEvent ve;

ve = (OlVirtualEvent) call_data;

switch (ve->virtual_name) {
    case OL_SELECT:
        switch(OlDetermineMouseAction(widget, event)) {
            case MOUSE_MOVE:
                OlGrabDragPointer(widget,
                    OlGetMoveCursor(XtScreen(widget), None);
                OlDragAndDrop(widget, &drop_window, &x, &y);
                DropOn(widget, drop_window, x, y, ....);
                OlUngrabDragPointer(widget);
                break;
            case MOUSE_CLICK:
                ClickSelect(widget, ....);
                break;
            case MOUSE_MULTI_CLICK:
                MultiClickSelect(widget, ....);
                break;
            }
        break;
    default:
        OlReplayBtnEvent(widget, NULL, event);
        break;
    }
}
```

OlReplayBtnEvent

```
#include <Xol/Dynamic.h>

void OlReplayBtnEvent(
    Widget      w,
    caddr_t     client_data,
    XEvent      *event);
```

`OlReplayBtnEvent()` replays a button press event to the next window (towards the root) that is interested in button events. This provides a means of propagating events up a window tree.

OlClassSearchIEDB

```
#include <Xol/OpenLook.h>

void OlClassSearchIEDB(
    WidgetClass             wc;
    OlVirtualEventTable     db);
```

`OlClassSearchIEDB()` registers a given database on a specific widget class. The *db* value was returned from a call to `OlCreateInputEventDB()`. Once a database is registered with a given widget class, the `OlLookupInputEvent()` procedure (if *db_flag* is OL_DEFAULT_IE or OLTEXT_IE) will include this database in the search stack if the given widget ID is a subclass of this widget class.

Example

```
/* To create a client application database */

    /* start with a big value to avoid */
    /* the "virtual_name" collision    */
#define OL_MY_BASE              1000
#define OL_MY_REDISPLAYKEY      OL_MY_BASE+2
#define OL_MY_SAVEPARTKEY       OL_MY_BASE+3

#define XtNmyDrawLineBtn        "myDrawLineBtn"
#define XtNmyDrawArcBtn         "myDrawArcBtn"
#define XtNmyRedisplayKey       "myRedisplayKey"
#define XtNmySavePartKey        "mySavePartKey"

static OlKeyOrBtnRec   OlMyBtnInfo[] = {
     /*name          default_value     virtual_name        */

};
```

```
static OlKeyOrBtnRec     OlMyKeyInfo[] = {
     /*name                default_value      virtual_name        */

     { XtNmyRedisplayKey, "c<F5>",           OL_MY_REDISPLAYKEY },
     { XtNmySavePartKey,  "c<F5>",           OL_MY_SAVEPARTKEY  },
};
static OlVirtualEventTable     OlMyDB;
...
OlMyDB = OlCreateInputEventDB(
               w,
               OlMyKeyInfo, XtNumber(OlMyKeyInfo),
               OlMyBtnInfo, XtNumber(OlMyBtnInfo)
     );
...
     /* assume: all stub widgets are interested in OlMyDB */
OlClassSearchIEDB(stubWidgetClass, OlMyDB);
     /* once this step is done, all stub widget instances */
     /* will receive the OlMyDB commands after a call to  */
     /* OlLookupInputEvent(), or in the XtNconsumeEvent */
     /* callback's OlVirtualEvent structure supplied with */
     /* the call_data field.                              */
```

OlClassSearchTextDB

```
#include <Xol/OpenLook.h>
void    OlClassSearchTextDB(
    WidgetClass wc);
```

`OlClassSearchTextDB()` registers the OPEN LOOK TEXT database on a specific widget class. Once the OPEN LOOK TEXT database is registered with a given widget class, the `OlLookupInputEvent()` procedure (if *db_flag* is OL_DEFAULT_IE or OLTEXT_IE) will include this database in the search stack if the given widget ID is a subclass of this widget class.

Example

```
    /* assume: all stub widgets are interested in the    */
    /*          OPEN LOOK TEXT database                  */
OlClassSearchTextDB(stubWidgetClass);
    /* once this step is done, all stub widget instances */
    /* will receive OPEN LOOK TEXT commands after a      */
```

```
    /* call to OlLookupInputEvent(), or in the        */
    /* XtNconsumeEvent callback's OlVirtualEvent      */
    /* structure supplied with the call_data field.   */
```

OlCreateInputEventDB

```
#include <Xol/OpenLook.h>

OlVirtualEventTable OlCreateInputEventDB(
    Widget              w,
    OlKeyOrBtnInfo      key_info,
    int                 num_key_info,
    OlKeyBtnInfo        btn_info,
    int                 num_btn_info);
```

`OlCreateInputEventDB()` creates a client specific Key and/or Button database. This function returns a database pointer if the call to this function is successful; otherwise, a NULL pointer is returned. Mapping for a new virtual command can be composed from the mappings of a previously defined virtual command. The returned value from this function is an opaque pointer (`OlVirtualEventTable`). A client application should use this pointer when registering and/or looking up this database.

```
    typedef struct _OlVirtualEventInfo *OlVirtualEventTable;
```

The *key_info* and *btn_info* parameters are pointers to an `OlKeyOrBtnRec` structure.

```
typedef struct {
    String          name;
    String          default_value; /* comma-separated string */
    OlVirtualName   virtual_name;
} OlKeyOrBtnRec, *OlKeyOrBtnInfo;
```

Note – A client application can create a Key-only database by specifying a NULL *btn_info.* The same applies to a Button-only database. Each virtual command can have two different bindings because the OLIT toolkit allows the alternate key or button sequence. The OLIT toolkit already has a set of predefined OPEN LOOK virtual names. It is important that the *virtual_name* value of a client application database starts with a big value to avoid a virtual name collision.

Example

```
    /* To create a client application database */
    /* start with a big value to avoid */
    /* the "virtual_name" collision    */
#define OL_MY_BASE            1000
#define OL_MY_REDISPLAYKEY    OL_MY_BASE+2
#define OL_MY_SAVEPARTKEY     OL_MY_BASE+3

#define XtNmyDrawLineBtn     "myDrawLineBtn"
#define XtNmyDrawArcBtn      "myDrawArcBtn"
#define XtNmyRedisplayKey    "myRedisplayKey"
#define XtNmySavePartKey     "mySavePartKey"

static OlKeyOrBtnRec     OlMyBtnInfo[] = {
     /*name             default_value      virtual_name        */

};

static OlKeyOrBtnRec     OlMyKeyInfo[] = {
     /*name               default_value      virtual_name        */
     { XtNmyRedisplayKey, "c<F5>",           OL_MY_REDISPLAYKEY },
     { XtNmySavePartKey,  "c<F5>",           OL_MY_SAVEPARTKEY  },
};

static OlVirtualEventTable     OlMyDB;

    ...
OlMyDB = OlCreateInputEventDB(
            w,
            OlMyKeyInfo, XtNumber(OlMyKeyInfo),
            OlMyBtnInfo, XtNumber(OlMyBtnInfo)
     );
    ...
```

OlLookupInputEvent

```
#include <Xol/OpenLook.h>

void OlLookupInputEvent(
    Widget              w,
    XEvent              *xevent,
    OlVirtualEvent      virtual_event_ret,
    XtPointer           db_flag);
```

`OlLookupInputEvent()` translates an X event to an OPEN LOOK virtual event. The X event (*xevent*) could be a KeyPress, ButtonPress, ButtonRelease, EnterNotify, LeaveNotify, or MotionNotify event. The procedure attempts to translate this event based on the setting of the OPEN LOOK-defined dynamic

databases. The *virtual_event_ret* parameter is a pointer to an `OlVirtualEventRec` structure in which the OPEN LOOK virtual event is returned:

```
typedef struct {
    Boolean         consumed;
    XEvent          *xevent;
    Modifiers       dont_care;
    OlVirtualName   virtual_name;
    KeySym          keysym;
    String          buffer;
    Cardinal        length;
    Cardinal        item_index;
} OlVirtualEventRec, *OlVirtualEvent;
```

(This structure is also used by the `XtNconsumeEvent` resource's callbacks.)

See the table of OLIT Activation Types (Table 3-1 on page 65) for a list of the values (listed in the "Activation Type" column) that may be returned in the *virtual_name* member of the *virtual_event_rec*. If the X event is a KeyPress, the *keysym*, *buffer*, and *length* information will be included in *virtual_event_ret*; `OlLookupInputEvent()` obtains these values from `XLookupString()`.

The (*w*, *db_flag*) pair determines the searching database(s). Valid values for the *db_flag* parameter are OL_DEFAULT_IE, OL_CORE_IE, and OL_TEXT_IE. If the *db_flag* value is not OL_DEFAULT_IE, then only the given database (for example, OL_TEXT_IE means: search the OPEN LOOK TEXT database) will be searched; otherwise, a search stack will be built. This stack is based on the widget information (*w*) and the registering order to determine the searching database(s). Once this stack is built, the procedure searches in a LIFO (Last In First Out) manner.

Most OLIT widgets have an `XtNconsumeEvent` callback. When this callback is called, the *call_data* field is a pointer to an `OlVirtualEventRec` structure that is filled in with the results of calling `OlLookupInputEvent()` with the *db_flag* set to OL_DEFAULT_IE.

Example

```
OlVirtualEventRec    ve;

     /* To look up the OPEN LOOK CORE database */
OlLookupInputEvent(w, xevent, &ve, OL_CORE_IE);
switch (ve.virtual_name) {
    case OL_UNKNOWN_INPUT:
```

```
        ...
    case OL_UNKNOWN_KEY_INPUT:
        ...
    case OL_ADJUST:
        printf ("pressed the adjustBtn\n");
        ...
    case OL_ADJUSTKEY:
        printf ("pressed the adjustKey\n");
        ...
}

    ...
OlVirtualEventRec     ve;
     /* To look up the OPEN LOOK TEXT database */
OlLookupInputEvent(w, xevent, &ve, OLTEXT_IE);
switch (ve.virtual_name) {
    ...
    case OL_DOCEND:
        printf ("pressed the docEndKey\n");
        ...
    case OL_LINEEND:
        printf ("pressed the lineEndKey\n");
        ...
}
    ...
OlVirtualEventRec     ve;
     /* To look up all possible databases */
     /* assume: "w" is a textfield widget */
OlLookupInputEvent(w, xevent, &ve, OL_DEFAULT_IE);
switch (ve.virtual_name) {
        ...
    case OL_ADJUST:
        printf ("pressed the adjustBtn\n");
        ...
    case OL_ADJUSTKEY:
        printf ("pressed the adjustKey\n");
        ...
    case OL_DOCEND:
        printf ("pressed the docEndKey\n");
        ...
    case OL_LINEEND:
        printf ("pressed the lineEndKey\n");
        ...
}
```

OlWidgetSearchIEDB

```
#include <Xol/OpenLook.h>

void OlWidgetSearchIEDB(
    Widget                w,
    OlVirtualEventTable   db);
```

`OlWidgetSearchIEDB()` registers a given database on a specific widget instance. The *db* value was returned from a call to `OlCreateInputEventDB()`. Once a database is registered with a given widget instance, the `OlLookupInputEvent()` procedure (if *db_flag* is OL_DEFAULT_IE or OL_TEXT_IE) will include this database in the search stack if the given widget ID is this widget instance.

Example

```
    /* To create a client application database */
    /* start with a big value to avoid         */
    /* the "virtual_name" collision            */
#define OL_MY_BASE            1000
#define OL_MY_REDISPLAYKEY    OL_MY_BASE+2
#define OL_MY_SAVEPARTKEY     OL_MY_BASE+3

#define XtNmyDrawLineBtn      "myDrawLineBtn"
#define XtNmyDrawArcBtn       "myDrawArcBtn"
#define XtNmyRedisplayKey     "myRedisplayKey"
#define XtNmySavePartKey      "mySavePartKey"

static OlKeyOrBtnRec     OlMyBtnInfo[] = {
     /*name             default_value      virtual_name        */

};

static OlKeyOrBtnRec     OlMyKeyInfo[] = {
    /*name              default_value      virtual_name        */
     { XtNmyRedisplayKey, "c<F5>",           OL_MY_REDISPLAYKEY },
     { XtNmySavePartKey,  "c<F5>",           OL_MY_SAVEPARTKEY  },
};

static OlVirtualEventTable     OlMyDB;
    ...
OlMyDB = OlCreateInputEventDB(
               w,
               OlMyKeyInfo, XtNumber(OlMyKeyInfo),
               OlMyBtnInfo, XtNumber(OlMyBtnInfo)
        );
    ...
```

```
     /* Assume: "w" is a stub widget that is interested in */
     /*          OlMyDB                                    */
OlWidgetSearchIEDB(w, OlMyDB);
     /* Once this step is done, this widget instance will  */
     /* receive OlMyDB commands after a call to            */
     /* OlLookupInputEvent(), or in the XtNconsumeEvent    */
     /* callback's OlVirtualEvent structure supplied with  */
     /* the call_data field.                               */
```

OlWidgetSearchTextDB

```
#include <Xol/OpenLook.h>

void OlWidgetSearchTextDB(
    OlVirtualEventTable w);
```

`OlWidgetSearchTextDB()` is used to register the OPEN LOOK TEXT database on a given widget instance.

Once the OPEN LOOK TEXT database is registered with a given widget instance, the `OlLookupInputEvent()` procedure (if *db_flag* is OL_DEFAULT_IE or OL_TEXT_IE) will include this database in the search stack if the given widget ID is this widget instance.

Example

```
    /* assume: "w" is a stub widget that is interested in */
    /*         the OPEN LOOK TEXT database                */
OlWidgetSearchTextDB(w);
    /* Once this step is done, this widget instance will  */
    /* receive OPEN LOOK TEXT commands after a call       */
    /* to OlLookupInputEvent(), or in the XtNconsumeEvent */
    /* callbacks OlVirtualEvent structure supplied with   */
    /* the call_data field.                               */
```

See Also

Chapter 3, "Activation Types."