Input, Focus, and Keyboard Navigation

The X server communicates with clients by means of various classes of *events*. Among these are events denoting input from the keyboard and mouse (and, in some X extensions, input from other devices). Each event is associated with a window, and the X server sends the event to any client that has expressed interest in events of that type on that window.

In the simplest case, when a keyboard or pointer event occurs, the X server sends the event to the client that has expressed interest in events of that type on the window that contains the pointer. If no such client exists, the server searches up the window's hierarchy until it finds a client that has expressed interest in events of that type on an ancestor window. In many cases, however, event processing is more complex:

- A client can *grab* a pointer button or key, the pointer or keyboard, or the entire server. The grabbing client then receives the relevant events for the duration of the grab.
- A client can set the *keyboard focus* to some window. Keyboard events that would normally be reported to this window or one of its inferiors are reported as usual, but other events are reported with respect to the focus window. Window managers typically use this technique to implement a "click–to–type" interaction style, in which the user clicks the pointer on some window, and that window retains the keyboard focus regardless of the position of the pointer. Other clients, often in cooperation with the window manager, can set the focus to a particular window within the application hierarchy.

To insulate applications from the complexities of X event handling, Xt and Motif have developed higher–level facilities based on widgets:

- Motif supplies a VendorShell resource, **XmNkeyboardFocusPolicy**, to allow a user or application to control the model of keyboard focus in the VendorShell and its descendants. Keyboard focus can be with the widget the contains the pointer or with the widget in which the user presses `Btn1`.
- In the click–to–type model, the user can also use keys to navigate from widget to widget. Motif provides a model of *tab groups*, which are widgets or sets of widgets to which the user moves by using `osfNextField` and `osfPrevField`. Within a tab group, the user traverses between widgets by using `osfUp`, `osfDown`, `osfLeft`, and `osfRight`. Motif supplies resources to control whether or not a widget constitutes a tab group and whether or not the user can traverse to it using the keyboard. Motif also has a general routine, **XmProcessTraversal**, for use by the application in moving keyboard focus to a widget or tab group. The Motif menu system has a specialized traversal mechanism.
- Xt provides the basic event–dispatching loop used by most applications. Xt takes events out of the application's queue and dispatches them to the appropriate widget, usually the widget that has input focus. Xt usually invokes an *action* associated with the particular event through a table of *translations* from event specifications to action routines. The action, in turn, often invokes a callback list. An application primarily responds to events by means of its callback routines. At a lower level, it can also provide its own *event handler*, a routine invoked by the Xt dispatching loop when the widget receives events of the specified type.
- Motif and Xt provide *mnemonics* and *accelerators*, which are shortcuts for taking actions associated with a widget when the widget does not have input focus. A *mnemonic* is a keysym for a key that activates a visible button in a menu. An *accelerator* is a description for an event that invokes an action routine through a translation.

Focus Models

Motif provides two models for determining which widget within an application receives keyboard events. The focus model is determined for all descendants of a VendorShell by the value of the VendorShell resource **XmNkeyboardFocusPolicy**:

- When the value is **XmEXPLICIT**, the widget under the pointer does not necessarily receive keyboard events. The user must take an action other than moving the pointer to transfer keyboard focus to a widget. The user can usually transfer focus to a widget by pressing `Btn1` on that widget or by using a keyboard navigation action to traverse to the widget.
  When the value is **XmEXPLICIT**, a widget must be *traversable* to receive keyboard events. In general, a widget is traversable when its **XmNsensitive**, **XmNancestorSensitive**, and **XmNtraversalOn** resources are True and when the widget and its ancestors are managed, realized, mapped, and viewable. See Section 13.2 for more information.
- When the value is **XmPOINTER**, the widget under the pointer receives keyboard events, unless that widget is insensitive. Keyboard navigation operations are not available. However, the user can still use the keyboard to traverse a menu system. Pressing `osfMenuBar` moves focus to the MenuBar, and `osfMenu` posts a PopupMenu if available. When the user posts a menu by using `osfMenu` or `Btn1Up`, `osfActivate`, or `osfSelect` in a CascadeButton, keyboard navigation operations are available in the menu until the menu is unposted. When the user exits the menu system, keyboard focus returns to the widget under the pointer.

MWM provides two parallel focus models for determining which top−level window receives keyboard events. The focus model is determined by the value of the **mwm** resource *keyboardFocusPolicy*:

- When the value is "explicit", the window under the pointer does not necessarily receive keyboard events. The user must take an action other than moving the pointer to transfer keyboard focus to a window. The user can usually transfer focus to a window by pressing `osfSelect` on that window or by using `AltF6`, `AltTab`, `AltShiftTab`, `AltShiftF6`, `AltShiftEsc`, or `AltEsc` to traverse to the window.
- When the value is "pointer", the widget under the pointer receives keyboard events. Keyboard window navigation operations are not available.

When the focus policy is "explicit", four Boolean **mwm** resources can be set to True to allow a window to receive keyboard focus automatically at specified times:

autoKeyFocus

When the window with focus is iconified or unmapped (gives focus to the window that last had it)

deiconifyKeyFocus

When the window is iconified

raiseKeyFocus

When the window is raised to the top of the stack

startupKeyFocus

When the window is mapped

Controlling Keyboard Navigation

In order to receive keyboard focus when the shell's **XmNkeyboardFocusPolicy** is **XmEXPLICIT**, a widget or gadget must meet the following conditions:

- The widget and its ancestors must not be in the process of being destroyed.
- The widget and its ancestors must be *sensitive*. A widget is sensitive when its **XmNsensitive** and **XmNancestorSensitive** resources are both True.
- The **XmNtraversalOn** resource for the widget and its ancestors must be True.
- The widget must be viewable. This means that the widget and its ancestors must be managed, realized, and (except for gadgets) mapped. Furthermore, in general, some part of the widget's rectangular area must be unobscured by the widget's ancestors.
  In a ScrolledWindow with an **XmNscrollingPolicy** of **XmAUTOMATIC**, a widget that is obscured because it is not within the clip window may be traversable if some part of the widget is within the work area and if an **XmNtraverseObscuredCallback** routine can make the widget unobscured by scrolling the window.

Most managers cannot receive focus even if they meet all these conditions. In general only primitives and gadgets are eligible to receive focus. A DrawingArea can receive focus if it meets the conditions above and if, in addition, it has no child whose **XmNtraversalOn** resource is True.

XmGetFocusWidget takes a widget argument that identifies a widget hierarchy, up to the nearest shell ancestor. It returns the widget in that hierarchy that has keyboard focus or that last had focus when the user navigated away from that hierarchy.

An application can use **XmIsTraversable** and **XmGetVisibility** to determine whether a widget is eligible to receive focus. **XmIsTraversable** returns True if the widget argument meets all the conditions described in this section. Otherwise, it returns False. This routine generally returns False if the widget argument is a composite, even if it has traversable children.

> **Note:** When all widgets in a shell hierarchy have been made untraversable, they are considered to have lost focus. When a widget in this hierarchy is made traversable again, it regains focus.

XmGetVisibility returns a value indicating the visibility of the widget argument:

XmVISIBILITY_FULLY_OBSCURED

> The widget is completely obscured by its ancestors or is not visible for some other reason (such as being unmapped or unrealized).

XmVISIBILITY_PARTIALLY_OBSCURED

> Some part of the widget's rectangular area is obscured by its ancestors.

XmVISIBILITY_UNOBSCURED

> None of the widget's rectangular area is obscured by its ancestors.

Note that a fully obscured widget may be traversable if it is inside the work area of an automatic ScrolledWindow with an **XmNtraverseObscuredCallback** list. See Section 13.2.5 for more information.

Sensitivity

Unless a widget is sensitive, Xt does not dispatch keyboard or pointer events to the widget. An insensitive widget, therefore, cannot receive keyboard focus.

A widget can be sensitive only when all its ancestors are sensitive. Two Boolean resources determine sensitivity: **XmNsensitive** and **XmNancestorSensitive**. **XmNsensitive** indicates whether the widget itself is sensitive, and **XmNancestorSensitive** indicates whether all ancestors are sensitive.

An application uses the function **XtIsSensitive** to find out whether a widget is sensitive. This function returns True when **XmNsensitive** and **XmNancestorSensitive** are both True; otherwise, it returns False.

The function **XtSetSensitive** changes the sensitivity of a widget. With an argument of False, this function sets **XmNsensitive** to False and sets each child's **XmNancestorSensitive** to False. With an argument of True, this function sets **XmNsensitive** to True and, if the widget's **XmNancestorSensitive** is also True, it sets each child's **XmNancestorSensitive** to True. The function then recursively descends the widget tree. For each descendant whose **XmNsensitive** and **XmNancestorSensitive** are both True, it sets **XmNancestorSensitive** to True for that widget's

children. Otherwise, it sets **XmNancestorSensitive** to False for the descendant widget's children.

In this way, **XtSetSensitive** ensures that each widget's **XmNancestorSensitive** is True only when the parent's **XmNsensitive** and **XmNancestorSensitive** are both True. In other words, the widget is sensitive only when it and all its ancestors are sensitive. To maintain this relation, an application should always use **XtSetSensitive** to change a widget's sensitivity instead of calling **XtSetValues** on the widget's resources.

Note that **XtSetSensitive** does not modify any resources for pop−up children. If the parent widget is insensitive when a pop−up child is created, the child's **XmNancestorSensitive** will be False. **XtSetSensitive** on the parent widget will not change this value, and the child will remain insensitive. To avoid this problem, an application that creates a DialogShell or a MenuShell should either ensure that the parent is sensitive when the child is created, or specify a value of True for the child's **XmNancestorSensitive**. One way to do this is in a resource file:

```
*XmMenuShell.ancestorSensitive:   True
*XmDialogShell.ancestorSensitive: True
```

When a widget or gadget is insensitive, Motif indicates the insensitivity to the user by stippling or graying the widget.

XmNtraversalOn

XmNtraversalOn determines whether or not a widget is eligible to receive keyboard focus when **XmNkeyboardFocusPolicy** is **XmEXPLICIT**. When **XmNtraversalOn** is False and **XmNkeyboardFocusPolicy** is **XmEXPLICIT**, it is not possible for the user to give keyboard focus to the widget, even if the widget is sensitive and viewable. **XmNtraversalOn** has no effect when **XmNkeyboardFocusPolicy** is **XmPOINTER**.

The default value for **XmNtraversalOn** is True for most Motif widgets. Following are the exceptions:

- Separator and SeparatorGadget, where **XmNtraversalOn** is forced to False
- ScrollBar, where **XmNtraversalOn** defaults to True when it is the child of a ScrolledWindow whose **XmNscrollingPolicy** is **XmAUTOMATIC** and to False otherwise
- Label and LabelGadget, where **XmNtraversalOn** is forced to False inside menus and defaults to False otherwise
- RowColumn, where **XmNtraversalOn** defaults to True in a WorkArea and is not applicable otherwise

Tab Groups

A tab group is a collection of traversable widgets or a single widget that contains a collection of traversable elements. When the shell's **XmNkeyboardFocusPolicy** is **XmEXPLICIT** and the Display's **XmNenableButtonTab** is False, the user traverses to a tab group by using `osfNextField` or `osfPrevField`. Within a tab group, when the focus is on a non−tab−group widget or an element, the user traverses to another non−tab−group widget or another element by using `osfUp`, `osfDown`, `osfLeft`, or `osfRight`.

A tab group is always represented by a widget or gadget. When the group is a collection of widgets, the tab group is typically the manager that is the parent of the widgets. When the group is a single widget like List or Text, the tab group is that widget itself.

The arrow keys do not traverse to tab groups or to non−tab−group widgets or elements outside the current tab group.

The Display resource **XmNenableButtonTab** influences the meaning of `osfNextField` and `osfPrevField`. In brief, setting **XmNenableButtonTab** to True lets a user navigate through an entire application using only one key. To examine **XmNenableButtonTab** more closely, consider an application containing the two tab groups shown in Figure 13−1. As the figure shows, each tab group contains three widgets. You can assume that the **XmNlayoutDirection** resource is set to **XmLEFT_TO_RIGHT_TOP_TO_BOTTOM**.

Figure 1 An Application With Two Tab Groups

Suppose that **XmNenableButtonTab** is False. Further suppose Widget B of Tab Group 1 has traversal focus. If **XmNenableButtonTab** is False, then an `osfNextField` event will send traversal focus to the first widget in Tab Group 1 (probably Widget D). That is, the application responds to the `osfNextField` event by traversing to the next tab group. If the user provides a second `osfNextField` event, Motif traverses back to the first widget in Tab Group 1 (probably Widget A).

Now suppose that **XmNenableButtonTab** is True and that traversal focus is at Widget B. Whenever the user provides an `osfNextField` event, Motif traverses to the next traversable widget in the application, whether or not that field is in the current tab group. Therefore, pressing `osfNextField` six times would send traversal focus to Widget C, then Widget D, then Widget E, then Widget F, then Widget A, and then back to Widget B.

To be eligible for traversal, a tab group must meet all the conditions discussed in Section 13.2, except that a manager that is a tab group and meets the other conditions is eligible for traversal as long as it contains a descendant that can receive focus. If the tab group does not meet these conditions, the `osfNextField` and `osfPrevField` actions ignore the tab group.

Within a tab group, non−tab−group widgets must also meet all the conditions discussed in Section 13.2 to be eligible for traversal. If they do not meet these conditions, the arrow key actions ignore the widgets.

Whether or not a widget is a tab group is determined by the value of the **XmNnavigationType** resource. The two primary values for this resource are **XmTAB_GROUP**, which indicates that the widget is a tab group, and **XmNONE**, which indicates that it is not.

When the user traverses to the next or previous tab group, the direction of the traversal is usually determined by the relative locations of the current and target groups. In a left−to−right language environment, traversal to each subsequent tab group proceeds from left to right and top to bottom. At the bottom right, traversal wraps to the tab group at the top left. Traversal to previous tab groups proceeds in the opposite direction.

The application can control the order of traversal by specifying an **XmNnavigationType** of **XmEXCLUSIVE_TAB_GROUP** for a widget in the hierarchy. When any widget in a hierarchy has an **XmNnavigationType** of **XmEXCLUSIVE_TAB_GROUP**, `osfNextField` and `osfPrevField` do not move to any widgets in that hierarchy that have been designated tab groups by means of an **XmNnavigationType** of **XmTAB_GROUP**. But `osfNextField` and `osfPrevField` do move to widgets whose **XmNnavigationType** is **XmSTICKY_TAB_GROUP**, even if some widgets are exclusive tab groups. Thus, an application that uses **XmEXCLUSIVE_TAB_GROUP** to control traversal must be sure that all tab groups have an **XmNnavigationType** of either **XmEXCLUSIVE_TAB_GROUP** or **XmSTICKY_TAB_GROUP**.

When any widget in a hierarchy has an **XmNnavigationType** of **XmEXCLUSIVE_TAB_GROUP**, traversal to subsequent tab groups does not depend on the relative locations of the groups. Instead, it proceeds to widgets in the order in which their **XmNnavigationType** resources were specified as **XmEXCLUSIVE_TAB_GROUP** or **XmSTICKY_TAB_GROUP**, either by creating the widgets with that value or by calling **XtSetValues**. That is, traversal proceeds to the widget whose **XmNnavigationType** was next specified to be **XmEXCLUSIVE_TAB_GROUP**

or **XmSTICKY_TAB_GROUP**. Traversal to previous tab groups proceeds in the opposite direction.

Within a tab group whose **XmNnavigationType** is **XmEXCLUSIVE_TAB_GROUP**, the arrow keys do not behave the same way as they would if the **XmNnavigationType** were either **XmTAB_GROUP** or **XmSTICKY_TAB_GROUP**. With **XmTAB_GROUP** or **XmSTICKY_TAB_GROUP**, the direction of traversal using the arrow keys depends on the relative locations of the tab group's children. Pressing `osfRight` moves to the next traversable child to the right of the child with the focus; `osfDown` moves to the next traversable child below the child with the focus; and so on.

With **XmEXCLUSIVE_TAB_GROUP**, traversal using the arrow keys depends on the order of the tab group's list of children, not on the relative locations of the children. Pressing `osfRight` has the same effect as `osfDown`: both move to the next traversable child in the tab group's list of children. Pressing `osfLeft` has the same effect as `osfUp`: both move to the previous traversable child in the tab group's list of children.

There are three principal differences between **XmEXCLUSIVE_TAB_GROUP** and **XmSTICKY_TAB_GROUP**:

- **XmEXCLUSIVE_TAB_GROUP** has the effect of disabling traversal to tab groups that have an **XmNnavigationType** of **XmTAB_GROUP**. **XmSTICKY_TAB_GROUP** does not; it simply ensures that traversal to that tab group is possible, even when some widget in the hierarchy has an **XmNnavigationType** of **XmEXCLUSIVE_TAB_GROUP**.
- **XmEXCLUSIVE_TAB_GROUP** changes the order of traversal of tab groups within the widget hierarchy. **XmSTICKY_TAB_GROUP** does not.
- **XmEXCLUSIVE_TAB_GROUP** changes the order of traversal of widgets inside the tab group. **XmSTICKY_TAB_GROUP** does not.

The function **XmAddTabGroup** has the same effect as calling **XtSetValues** with an **XmNnavigationType** of **XmEXCLUSIVE_TAB_GROUP**. The function **XmRemoveTabGroup** has the same effect as calling **XtSetValues** with an **XmNnavigationType** of **XmNONE**. **XmAddTabGroup** and **XmRemoveTabGroup** are obsolete and exist for compatibility with earlier releases of Motif.

All Motif managers except RowColumn have a default **XmNnavigationType** of **XmTAB_GROUP**. In RowColumn, **XmNnavigationType** is not applicable for MenuBars, PulldownMenus, and PopupMenus. For a WorkArea the default is **XmTAB_GROUP**, and for an OptionMenu the default is **XmNONE**.

All Motif primitives except List, ScrollBar, Text, and TextField have a default **XmNnavigationType** of **XmNONE**. The default for List, Text, and TextField is **XmTAB_GROUP**, and the default for ScrollBar is **XmSTICKY_TAB_GROUP**. These are all controls that have their own internal navigation.

Motif sets the navigation type of widgets in some situations. In particular:

- The child of a shell always behaves as a tab group, no matter what the value of its **XmNnavigationType**.
- Panes and sashes inside PanedWindows have a default **XmNnavigationType** of **XmTAB_GROUP**. If the **XmNnavigationType** of a pane is **XmNONE** when the pane is created, Motif sets the value of that resource to **XmTAB_GROUP**.
- SelectionBox and its subclasses set the **XmNnavigationType** of their automatically created List and Text children to **XmSTICKY_TAB_GROUP**.

The function **XmGetTabGroup** returns the tab group that contains a widget. If the widget itself is a tab group or a shell, it returns that widget. If neither the widget nor any ancestor up to the nearest shell is a tab group, it returns the nearest ancestor that is a shell. Otherwise, it returns the nearest ancestor that is a tab group.

Controlling Tab Group Traversal Order

By default, `osfNextField` and `osfPrevField` traverse to successive tab groups in order of layout, from left to right and top to bottom, within a parent tab group, before proceeding in layout order to the next tab group that is a sibling of the parent. Traversal order changes when any widget in a shell hierarchy has an **XmNnavigationType** of **XmEXCLUSIVE_TAB_GROUP**. In this case, `osfNextField` and `osfPrevField` traverse only to widgets in the hierarchy whose **XmNnavigationType** is either **XmEXCLUSIVE_TAB_GROUP** or **XmSTICKY_TAB_GROUP**. The traversal order is the order in which the widgets' **XmNnavigationType** was specified to be either **XmEXCLUSIVE_TAB_GROUP** or **XmSTICKY_TAB_GROUP**.

This mechanism gives an application the means to control tab group traversal order. An application must do the following:

- Ensure that at least one widget in the shell hierarchy has an **XmNnavigationType** of **XmEXCLUSIVE_TAB_GROUP**
- Ensure that all widgets that the application wants to be tab groups have an **XmNnavigationType** of either **XmEXCLUSIVE_TAB_GROUP** or **XmSTICKY_TAB_GROUP**
- Specify values for the tab groups' **XmNnavigationType**, using either creation argument lists or **XtSetValues**, in the order in which the tab groups are to be traversed

Note that, when a tab group has an **XmNnavigationType** of **XmEXCLUSIVE_TAB_GROUP**, traversal to non−tab−group widgets inside that tab group proceeds in the order in which the children appear in their parents' **XmNchildren** lists. If the application wants to specify the order of tab group traversal but still wants traversal of non−tab−group widgets to proceed according to layout, it should select one widget in the hierarchy to have an **XmNnavigationType** of **XmEXCLUSIVE_TAB_GROUP**. This tab group should contain no non−tab−group widgets. For example, it could be the MainWindow if the MainWindow contains only tab groups, or it could be a primitive tab group, such as List or Text. The application should then specify an **XmNnavigationType** of **XmSTICKY_TAB_GROUP** for all other tab groups in the hierarchy.

Initial Focus

A tab group may contain any combination of tab group and non−tab−group widgets. A tab group that contains other widgets cannot receive focus itself. When the user traverses to a composite tab group, Motif gives focus to some widget within the tab group.

Motif uses the Manager resource **XmNinitialFocus** in determining which widget receives focus. The value of **XmNinitialFocus** is a widget that meets the following conditions:

- The widget must be either a tab group or a non−tab−group widget that can receive keyboard focus. In general, a widget can receive keyboard focus when it is a primitive, a gadget, or a manager (such as a DrawingArea with no traversable children) that acts as a primitive.
- The widget must not be a descendant of a tab group that is itself a descendant of the manager. That is, the widget cannot be contained within a tab group that is nested inside the manager.
- The widget and its ancestors must have a value of True for their **XmNtraversalOn** resources.

If the widget does not meet these conditions, **XmNinitialFocus** is treated as if the value were NULL.

Motif uses **XmNinitialFocus** to determine which widget receives focus in these situations:

- When the manager is the child of a shell and the shell hierarchy receives focus for the first time
- When focus is inside the shell hierarchy, the manager is a composite tab group, and the user traverses to the manager using the keyboard

Motif then determines focus as follows:

- If **XmNinitialFocus** is a traversable non−tab−group widget, that widget receives focus.
- If **XmNinitialFocus** is a traversable tab group, that tab group receives focus. If that tab group is a composite with descendant tab groups or traversable non−tab−group widgets, these procedures are used recursively to assign focus to a descendant of that tab group.
- If **XmNinitialFocus** is NULL, the first traversable non−tab−group widget that is not contained within a nested tab group receives focus.
- If **XmNinitialFocus** is NULL and no traversable non−tab−group widget exists, the first traversable tab group that is not contained within a nested tab group receives focus. If that tab group is a composite with descendant tab groups or traversable non−tab−group widgets, these procedures are used recursively to assign focus to a descendant of that tab group.

If a shell hierarchy regains focus after losing it, focus returns to the widget that had the focus at the time it left the hierarchy.

The use of **XmNinitialFocus** is undefined if the manager is a MenuBar, PulldownMenu, PopupMenu, or OptionMenu.

Traversing to Obscured Widgets

In general, a widget is not eligible to receive focus unless some part of its rectangular area is unobscured by its ancestors. However, it may be possible to traverse to a widget that is a descendant of a ScrolledWindow whose **XmNscrollingPolicy** is **XmAUTOMATIC**, even if that widget is not within the ScrolledWindow's clip window. Traversal to such a widget is possible under the following conditions:

- Some part of the widget's rectangular area is within the bounds of the ScrolledWindow's work window.
- The ScrolledWindow's clip window is completely unobscured by its ancestors. If the ScrolledWindow is a descendant of another ScrolledWindow, it must be unobscured by the ancestor's work window but may be outside the ancestor's clip window.
- The ScrolledWindow has a procedure on its **XmNtraverseObscuredCallback** list that can bring some part of the widget's rectangular area into the clip window.
- The widget meets the other conditions for receiving focus described in Section 13.2.

Whenever the user attempts to traverse to such a widget and the widget is partially or fully obscured by the clip window, Motif calls the ScrolledWindow's **XmNtraverseObscuredCallback** procedures. If the ScrolledWindow has one or more ancestor ScrolledWindows, Motif calls the **XmNtraverseObscuredCallback** list for each ScrolledWindow whose clip window obscures the traversal target, from the lowest level of the hierarchy to the highest. The **XmNtraverseObscuredCallback** procedure can try to bring the widget into the clip window if necessary, usually by calling **XmScrollVisible**. If the target widget is traversable after the **XmNtraverseObscuredCallback** procedures are invoked, that widget receives focus.

A procedure can determine the visibility of a widget by calling **XmGetVisibility**.

XmProcessTraversal

The principal routine for traversing to a widget is **XmProcessTraversal**. Motif uses this routine to effect traversal when the user presses an arrow key, `osfNextField`, or `osfPrevField`. An application can use **XmProcessTraversal** to implement its own traversal actions.

XmProcessTraversal takes two arguments, a widget and a constant specifying a traversal action. The routine uses the widget argument to identify the hierarchy that contains the widget and that has its root at the nearest shell. If that shell does not currently have the focus, any changes to the element with focus within that shell will not occur until the next time the shell receives focus.

The traversal action argument identifies one of three kinds of action to take. The following descriptions of these actions refer to traversable non−tab−group widgets and traversable tab groups. A traversable non−tab−group widget is a widget that is not a tab group and that meets all the conditions for receiving focus discussed in Section 13.2. A traversable tab group is a tab group widget that meets the same conditions, except that a manager that is a tab group and meets the other conditions is also traversable as long as it contains a descendant that can receive focus.

The routine begins the traversal action from the widget in the hierarchy that currently has keyboard focus or that last had focus when the user traversed away from the shell hierarchy.

Note that **XmProcessTraversal** cannot be called recursively. In particular, an application cannot call this routine from an **XmNfocusCallback** or **XmNlosingFocusCallback** procedure.

The descriptions in the following three subsections all assume that **XmNlayoutDirection** is set to **XmLEFT_TO_RIGHT_TOP_TO_BOTTOM**. Later in this section, we will explore the influence of **XmNlayoutDirection** in greater detail.

Traversal to a Non−Tab−Group Widget

This kind of traversal is possible only when the widget that currently has focus is not a tab group. Also, these actions do not move focus from one tab group to another. The actions first determine the containing tab group. This is the tab group containing the widget that currently has focus. The actions traverse only to a non−tab−group widget within the containing tab group.

XmTRAVERSE_RIGHT

If the **XmNnavigationType** of the containing tab group is not **XmEXCLUSIVE_TAB_GROUP**, focus moves to the next traversable non−tab−group widget to the right of the widget that currently has focus. At the right side of the tab group, this action wraps to the non−tab−group widget at the left side and next toward the bottom. At the lower right corner of the tab group, this action wraps to the non−tab−group widget at the upper left.

If the **XmNnavigationType** of the containing tab group is **XmEXCLUSIVE_TAB_GROUP**, focus moves to the next traversable non−tab−group widget in the tab group, proceeding in the order in which the widgets appear in their parents' **XmNchildren** lists. After the last widget in the tab group, this action wraps to the first non−tab−group widget.

XmTRAVERSE_LEFT

If the **XmNnavigationType** of the containing tab group is not **XmEXCLUSIVE_TAB_GROUP**, focus moves to the next traversable non−tab−group widget to the left of the widget that currently has focus. At the left side of the tab group, this action wraps to the non−tab−group widget at the right side and next toward the top. At the upper left corner of the tab group, this action wraps to the non−tab−group widget at the lower right.

If the **XmNnavigationType** of the containing tab group is **XmEXCLUSIVE_TAB_GROUP**, focus moves to the previous traversable non−tab−group widget in the tab group, proceeding in the reverse order in which the widgets appear in their parents' **XmNchildren** lists. After the first widget in the tab group, this action wraps to the last non−tab−group widget.

XmTRAVERSE_DOWN

If the **XmNnavigationType** of the containing tab group is not **XmEXCLUSIVE_TAB_GROUP**, focus moves to the next traversable non−tab−group widget below the widget that currently has focus. At the bottom of the tab group, this action wraps to the non−tab−group widget at the top and next toward the right. At the lower right corner of the tab group, this action wraps to the non−tab−group widget at the upper left.

If the **XmNnavigationType** of the containing tab group is **XmEXCLUSIVE_TAB_GROUP**, focus moves to the next traversable non−tab−group widget in the tab group, proceeding in the order in which the widgets appear in their parents' **XmNchildren** lists. After the last widget in the tab group, this action wraps to the first non−tab−group widget.

XmTRAVERSE_UP

If the **XmNnavigationType** of the containing tab group is not **XmEXCLUSIVE_TAB_GROUP**, focus moves to the next traversable non−tab−group widget above the widget that currently has focus. At the top of the tab group, this action wraps to the non−tab−group widget at the bottom and next toward the left. At the upper left corner of the tab group, this action wraps to the non−tab−group widget at the lower right.

If the **XmNnavigationType** of the containing tab group is **XmEXCLUSIVE_TAB_GROUP**, focus moves to the previous traversable non−tab−group widget in the tab group, proceeding in the reverse order in which the widgets appear in their parents' **XmNchildren** lists. After the first widget in the tab group, this action wraps to the last non−tab−group widget.

XmTRAVERSE_NEXT

Focus moves to the next traversable non−tab−group widget in the tab group, proceeding in the order in which the widgets appear in their parents' **XmNchildren** lists. After the last widget in the tab group, this action wraps to the first non−tab−group widget.

XmTRAVERSE_PREV

Focus moves to the previous traversable non−tab−group widget in the tab group, proceeding in the reverse order in which the widgets appear in their parents' **XmNchildren** lists. After the first widget in the tab group, this action wraps to the last non−tab−group widget.

XmTRAVERSE_HOME

If the **XmNnavigationType** of the containing tab group is not **XmEXCLUSIVE_TAB_GROUP**, focus moves to the first traversable non−tab−group widget at the top left corner of the tab group.

If the **XmNnavigationType** of the containing tab group is **XmEXCLUSIVE_TAB_GROUP**, focus moves to the first traversable non−tab−group widget in the tab group, according to the order in which the widgets appear in their parents' **XmNchildren** lists.

Traversal to a Tab Group

The following actions begin by determining the current widget hierarchy and the containing tab group. The current widget hierarchy is the widget hierarchy whose root is the nearest shell ancestor of the widget that currently has focus. The containing tab group is is the tab group containing the widget that currently has focus.

XmTRAVERSE_NEXT_TAB_GROUP

If no tab group in the current widget hierarchy has a value of **XmEXCLUSIVE_TAB_GROUP** for **XmNnavigationType**, focus goes to the next traversable tab group that is to the right of the widget with current focus and is within the containing tab group. At the right side of the containing tab group, this action wraps to the tab group at the left side and next toward the bottom. At the lower right corner of the containing tab group, this action recursively moves up one level in the hierarchy. Focus then goes to the next traversable tab group that is to the right of the original containing tab group and is within the tab group that contains that one. At the lower right corner of the topmost tab group in the hierarchy, this action wraps to the first traversable tab group at the upper left corner of the topmost tab group.

If any tab group in the current widget hierarchy has a value of **XmEXCLUSIVE_TAB_GROUP** for **XmNnavigationType**, focus goes to the next traversable tab group in the hierarchy, in the order in which the **XmNnavigationType** resources of the tab groups were set to **XmEXCLUSIVE_TAB_GROUP** or **XmSTICKY_TAB_GROUP**. After the last tab group in the hierarchy, this action wraps to the first tab group.

XmTRAVERSE_PREV_TAB_GROUP

If no tab group in the current widget hierarchy has a value of **XmEXCLUSIVE_TAB_GROUP** for **XmNnavigationType**, focus goes to the next traversable tab group that is to the left of the widget with current focus and is within the containing tab group. At the left side of the containing tab group, this action wraps to the tab group at the right side and next toward the top. At the upper left corner of the containing tab group, this action recursively moves up one level in the hierarchy. Focus then goes to the next traversable tab group that is to the left of the original containing tab group and is within the tab group that contains that one. At the upper left corner of the topmost tab group in the hierarchy, this action wraps to the first traversable tab group at the lower right corner of the topmost tab group.

If any tab group in the current widget hierarchy has a value of **XmEXCLUSIVE_TAB_GROUP** for **XmNnavigationType**, focus goes to the previous traversable tab group in the hierarchy, in the reverse order in which the **XmNnavigationType** resources of the tab groups were set to either **XmEXCLUSIVE_TAB_GROUP** or **XmSTICKY_TAB_GROUP**. After the first tab group in the hierarchy, this action wraps to the last tab group.

Traversal to Any Widget

In the following case, the widget argument is the widget to which **XmProcessTraversal** tries to give focus.

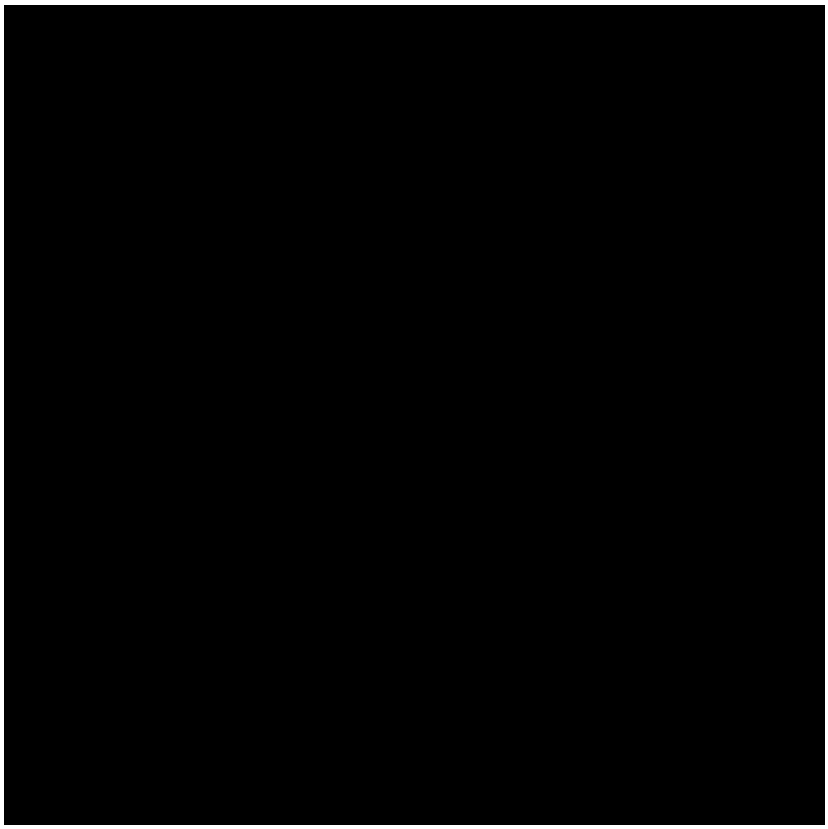
XmTRAVERSE_CURRENT

Focus goes to the widget argument if that widget is a traversable non−tab−group widget or tab group.

Effect of XmNlayoutDirection on XmProcessTraversal

The descriptions in the preceding subsections all assumed that **XmNlayoutDirection** was set to **XmLEFT_TO_RIGHT_TOP_TO_BOTTOM**. When **XmNlayoutDirection** is set to some other direction, the navigation rules are somewhat different. To help explain these differences, consider the application shown in Figure 13−2. This application consists of one tab group. This one tab group consists of 12 widgets.

Figure 2 An Application With 12 Widgets in One Tab Group



Given the application shown in Figure 13−2, Table 13−1 summarizes what happens when the application calls **XmProcessTraversal** with a direction argument of **XmTRAVERSE_HOME**.

Table 1 Effect of XmNlayoutDirection on XmTRAVERSE_HOME

XmNlayoutDirection =	*Home Widget*
XmLEFT_TO_RIGHT_TOP_TO_BOTTOM	A
XmRIGHT_TO_LEFT_TOP_TO_BOTTOM	B
XmLEFT_TO_RIGHT_BOTTOM_TO_TOP	K
XmRIGHT_TO_LEFT_BOTTOM_TO_TOP	L
XmTOP_TO_BOTTOM_LEFT_TO_RIGHT	C
XmTOP_TO_BOTTOM_RIGHT_TO_LEFT	F
XmBOTTOM_TO_TOP_LEFT_TO_RIGHT	G
XmBOTTOM_TO_TOP_RIGHT_TO_LEFT	J

Table 13–2 summarizes the influence of **XmNlayoutDirection** on **XmProcessTraversal** in the application shown in Figure 13–2. For example, suppose that the second argument to **XmProcessTraversal** is **XmTRAVERSE_RIGHT** and that the **XmNlayoutDirection** resource is set to **XmRIGHT_TO_LEFT_TOP_TO_BOTTOM**. If **XmNinitialFocus** for the tab group manager is NULL, Table 13–2 says that initial focus will default to widget B. When **XmProcessTraversal** is called again with **XmTRAVERSE_RIGHT** as its direction argument, the application will navigate to widget K, then to widget L, and so on. After navigating through all 12 widgets, the application will navigate around to the starting widget, which in this case is widget B.

One way to understand the navigation rules is to ask yourself whether the application is trying to traverse forwards or traverse backwards. To understand forwards and backwards, suppose that **XmNlayoutDirection** is set to **XmLEFT_TO_RIGHT_TOP_TO_BOTTOM**. In this case

- **XmTRAVERSE_RIGHT** and **XmTRAVERSE_DOWN** are forward traversals (with the flow)
- **XmTRAVERSE_LEFT** and **XmTRAVERSE_UP** are backwards traversals (against the flow)

Understanding forwards and backwards helps you understand what Motif does when the application wants to traverse past the last widget in a particular direction. For example, what happens when the application wants to traverse right but there are no widgets to the right of the current widget? If the traversal is forwards, then upon reaching the last widget in given direction, Motif navigates towards the direction specified by **XmNlayoutDirection**. If the traversal is backwards, Motif navigates away from the direction specified by **XmNlayoutDirection**.

Table 2 Effect of XmNlayoutDirection on XmProcessTraversal

XmProcessTraversal =	XmNlayoutDirection	*Sequence*
XmTRAVERSE_RIGHT	**XmLEFT_TO_RIGHT_TOP_TO_BOTTOM**	A,B,C,D,E,F,G,H,I,J,K,L
	XmRIGHT_TO_LEFT_TOP_TO_BOTTOM	B,K,L,G,H,I,J,C,D,E,F,A
	XmLEFT_TO_RIGHT_BOTTOM_TO_TOP	K,L,G,H,I,J,C,D,E,F,A,B
	XmRIGHT_TO_LEFT_BOTTOM_TO_TOP	L,A,B,C,D,E,F,G,H,I,J,K
	XmTOP_TO_BOTTOM_LEFT_TO_RIGHT	C,D,E,F,G,H,I,J,K,L,A,B
	XmTOP_TO_BOTTOM_RIGHT_TO_LEFT	F,A,B,K,L,G,H,I,J,C,D,E
	XmBOTTOM_TO_TOP_LEFT_TO_RIGHT	G,H,I,J,C,D,E,F,A,B,K,L
	XmBOTTOM_TO_TOP_RIGHT_TO_LEFT	J,K,L,A,B,C,D,E,F,G,H,I
XmTRAVERSE_LEFT	**XmLEFT_TO_RIGHT_TOP_TO_BOTTOM**	A,L,K,J,I,H,G,F,E,D,C,B
	XmRIGHT_TO_LEFT_TOP_TO_BOTTOM	B,A,F,E,D,C,J,I,H,G,L,K
	XmLEFT_TO_RIGHT_BOTTOM_TO_TOP	K,B,A,F,E,D,C,J,I,H,G,L
	XmRIGHT_TO_LEFT_BOTTOM_TO_TOP	L,K,J,I,H,G,F,E,D,C,B,A
	XmTOP_TO_BOTTOM_LEFT_TO_RIGHT	C,B,A,L,K,J,I,H,G,F,E,D
	XmTOP_TO_BOTTOM_RIGHT_TO_LEFT	F,E,D,C,J,I,H,G,L,K,B,A
	XmBOTTOM_TO_TOP_LEFT_TO_RIGHT	G,L,K,B,A,F,E,D,C,J,I,H
	XmBOTTOM_TO_TOP_RIGHT_TO_LEFT	J,I,H,G,F,E,D,C,B,A,L,K
XmTRAVERSE_DOWN	**XmLEFT_TO_RIGHT_TOP_TO_BOTTOM**	A,D,H,K,B,E,I,L,F,J,C,G
	XmRIGHT_TO_LEFT_TOP_TO_BOTTOM	B,E,I,L,A,D,H,K,C,G,F,J
	XmLEFT_TO_RIGHT_BOTTOM_TO_TOP	K,C,G,F,J,B,E,I,L,A,D,H
	XmRIGHT_TO_LEFT_BOTTOM_TO_TOP	L,F,J,C,G,A,D,H,K,B,E,I
	XmTOP_TO_BOTTOM_LEFT_TO_RIGHT	C,G,A,D,H,K,B,E,I,L,F,J
	XmTOP_TO_BOTTOM_RIGHT_TO_LEFT	F,J,B,E,I,L,A,D,H,K,C,G
	XmBOTTOM_TO_TOP_LEFT_TO_RIGHT	G,F,J,B,E,I,L,A,D,H,K,C
	XmBOTTOM_TO_TOP_RIGHT_TO_LEFT	J,C,G,A,D,H,K,B,E,I,L,F
XmTRAVERSE_UP	**XmLEFT_TO_RIGHT_TOP_TO_BOTTOM**	A,G,C,J,F,L,I,E,B,K,H,D
	XmRIGHT_TO_LEFT_TOP_TO_BOTTOM	B,J,F,G,C,K,H,D,A,L,I,E
	XmLEFT_TO_RIGHT_BOTTOM_TO_TOP	K,H,D,A,L,I,E,B,J,F,G,C
	XmRIGHT_TO_LEFT_BOTTOM_TO_TOP	L,I,E,B,K,H,D,A,G,C,J,F
	XmTOP_TO_BOTTOM_LEFT_TO_RIGHT	C,J,F,L,I,E,B,K,H,D,A,G
	XmTOP_TO_BOTTOM_RIGHT_TO_LEFT	F,G,C,K,H,D,A,L,I,E,B,J
	XmBOTTOM_TO_TOP_LEFT_TO_RIGHT	G,C,K,H,D,A,L,I,E,B,J,F
	XmBOTTOM_TO_TOP_RIGHT_TO_LEFT	J,F,L,I,E,B,K,H,D,A,G,C

Focus Callbacks

BulletinBoard, Text, and TextField have **XmNfocusCallback** callback lists. Motif invokes the procedures on these lists when these widgets receive keyboard focus. A callback procedure may change the widget's state to reflect the new focus, but it should not try to change the focus and, in particular, must not call **XmProcessTraversal**.

Text and TextField also have **XmNlosingFocusCallback** callback lists. The Text and TextField traversal actions invoke these procedures before traversing to another widget. The third argument to each procedure is a pointer to an **XmTextVerifyCallbackStruct** structure whose *reason* member is **XmCR_LOSING_FOCUS**. If a callback procedure sets the *doit* member of this structure to False, the traversal action does not carry out the traversal. In this way the application can prevent a user from traversing out of the widget by means of these actions.

Motif also invokes the **XmNlosingFocusCallback** procedures when the widget loses focus by some other means. For example, the user might click `Btn1` in another traversable widget, or when the shell's **XmNkeyboardFocusPolicy** is **XmPOINTER**, the user might move the pointer into another widget. In such cases, setting the *doit* member of the callback structure has no effect.

Translations and Actions

In Xt, the primary means of associating an input event with a widget−specific procedure is the combination of translations and actions. Each widget (but not gadget) instance contains a table of translations that maps event descriptions to procedure names. Each widget instance also has a table of actions that maps these procedure names to actual procedures. When a widget receives an input event, the Xt event−dispatching facility looks up the event in the translation table, looks up the associated procedure in the action table, and invokes the action procedure itself. This procedure usually takes some action to change the widget state and often invokes callback procedures.

Translation Table Format

An application or user specifies a translation table as a string whose format is defined in *X Toolkit Intrinsics−−C Language Interface*. In general, the table consists of individual translations separated by "\n". Each translation consists of an event description sequence, a colon, and one or more associated procedure names. Each procedure name also has a list of parameters within parentheses to be passed to the procedure when it is invoked as a result of that translation.

An event description in general consists of an optional list of modifiers, an event type within angle brackets (< and >), an optional repeat count within parentheses, and an optional event detail. Modifiers apply only to key, button, motion, enter, and leave events. If an exclamation point (!) precedes the modifiers, then the modifiers in the list and no others must be asserted for the action to be invoked. Otherwise, the modifiers in the list must be asserted, but others may be as well. A tilde (~) before any modifier means that that modifier must not be asserted. If the modifier list is empty, any modifiers may be asserted.

The detail field varies depending on the event type. The most common use is to identify the keysym for a **KeyPress** or **KeyRelease** event.

Event descriptions in a sequence are separated by commas. Mouse motion is discarded if it occurs between events in a sequence that does not include explicit motion events. This allows the following sort of translation to invoke an action even if the mouse moves between button press and release:

```
<Btn1Down>,<Btn1Up>: action()
```

Following are some important considerations in using translations:

- More specific events should always precede less specific events in the table:

```
Ctrl<Key>space: action_1()
<Key>space: action_2()
```

- Translations with event sequences that are noninitial subsequences of other translations are not invoked when the events occur as part of the longer sequence. For instance, *up_action()* in the following example would not be invoked on a button release that followed a button press:

```
<Btn1Down>,<Btn1Up>: click_action()
<Btn1Up>: up_action()
```

- Event descriptions that use a repeat count expand into longer sequences. For example, the following descriptions are more or less equivalent:

```
<Btn1Up>(2): double_click()
<Btn1Up>,<Btn1Down>,<Btn1Up>: double_click()
```

This result, combined with the implicit insertion of motion events between any two other events, means that motion translations cannot exist in a table with multiclick translations.

See *X Toolkit Intrinsics−−C Language Interface* for more information on the format of translation tables.

Using Translations

One translation table frequently needs to be merged with another. For example, a user may want to add one or more translations to a widget's default translations. A translation table may begin with one of three directives that specifies how the table is to be merged with an existing table:

#replace

The new translation table should completely replace any existing table. This is the default if no directive is specified.

#augment

The new translation table should be added to any existing table. If the two tables contain duplicate event descriptions, the translations in the existing table are used.

#override

The new translation table should be added to any existing table. If the two tables contain duplicate event descriptions, the translations in the new table are used.

A widget's translation table is the value of the Core **XmNtranslations** resource. The initial value is determined in the following way:

- If a non−NULL value is specified for **XmNtranslations** in the widget creation argument list, the widget class translations are merged with that value, in order, and the resulting table is used.
- Otherwise, the following tables are merged, in order, and the resulting table is used:
  - The widget class translations
  - The value of the *baseTranslations* resource from the resource database
  - The value of the **XmNtranslations** resource from the resource database or, if no value was specified, the default value for the widget's **XmNtranslations**

To take advantage of this initialization ordering, an application should usually provide any translations of its own by specifying a value for *baseTranslations* rather than **XmNtranslations** in an application class defaults file or a fallback resource list. This essentially reserves **XmNtranslations** to the user. The application can change the widget class translations by specifying *baseTranslations*, and the user can change the application's translations by specifying **XmNtranslations**.

As the value of a widget's **XmNtranslations**, a translation table must be in a parsed format rather than a string. The string−to−translation−table converter parses a resource string into a translation table. An application can also use **XtParseTranslationTable** to compile a translation table string into the parsed format. The application can then merge the parsed table with a widget's **XmNtranslations** in three ways:

- **XtAugmentTranslations** merges the parsed table in *#augment* mode
- **XtOverrideTranslations** merges the parsed table in *#override* mode
- **XtSetValues** of **XmNtranslations** replaces the existing value with the parsed table

Some Motif widgets merge additional translations in their *initialize* and *set_values* methods. This process may make it impossible for an application or user to override some translations by means of resource files. For example, for some widgets it may not be possible to change traversal translations in this way.

Actions

Each widget instance has a table that maps action procedure names, as they appear in translation tables, to actual action procedures. When an action is invoked through a translation, Xt looks up the action procedure name in this table and calls the associated procedure.

Each widget class may have its own action table. In addition, an application can use **XtAppAddActions** to add entries to an action table associated with the application context. Only one such table exists per application context. If a call to **XtAppAddActions** contains an action name that is already in the table, the action name becomes associated with the action procedure supplied in the call to **XtAppAddActions**, overriding the existing action.

Xt creates a widget's action table when the widget is realized. It uses actions from the following action tables, those listed first having highest precedence:

- The action tables for the widget's class and its superclasses, in subclass−to−superclass order
- The action tables for the parent's class and its superclasses, in subclass−to−superclass order, and so on up the widget hierarchy
- The application context action table (created by calls to **XtAppAddActions**)

This ordering means that an application cannot use **XtAppAddActions** to provide a new action procedure for an action name that is already registered by a widget class. To do that, the application must supply a translation that maps the event to an action name that is not registered by the class. The application must then call **XtAppAddActions** to supply a procedure for the action name.

An action procedure is a function of type **XtActionProc**. This function receives four arguments:

- The widget
- The event, or the last event of a sequence, that caused the procedure to be invoked
- A list of strings representing the parameters specified for this action in the translation table
- An integer representing the number of parameters in the parameter list

An application can use the parameter list to perform a number of related actions in a single action routine. For example, a widget might have the following translations:

```
c <Key> osfLeft: move-object(left) \n\
c <Key> osfRight: move-object(right) \n\
c <Key> osfUp: move-object(up) \n\
```

```
c <Key> osfDown: move-object(down)
```

The routine implementing the **move–object()** action is passed one of the strings "left", "right", "up", and "down" as the only item in the parameter list, depending on which key event invoked the action. The routine performs the action appropriate for this parameter.

Bindings for osf Keysyms

Motif maintains a client–side mechanism for mapping one set of keysyms to another set. This mapping allows Motif widgets and applications to use a single set of keysyms in translation tables and also allows applications and users to customize the keysyms used in the translations for the particular keyboard used with the display.

The names of keysyms eligible for use in translations in this way begin with the prefix "osf" and are referred to as *osf keysyms*. Motif maintains a mapping between these "virtual" keysyms and the "actual" keysyms that correspond to keys on a particular keyboard. When Xt receives a keyboard event, the function **XmTranslateKey** translates the keycode of the event to the appropriate osf keysym if a mapping exists for that keysym. Xt then dispatches the event to the appropriate action routine if a translation exists for that osf keysym.

The mapping between osf and actual keysyms is determined at application startup based on information obtained from one of the following sources, listed in order of precedence:

- *defaultVirtualBindings* application resource in the resource database.
- property on the root window, which can be set by **mwm** on startup, by the **xmbind** client, or on prior startup of a Motif application.
- *.motifbind* file in the user's home directory.
- default binding based on the vendor string and optionally the vendor release of the X server. Motif searches the file *xmbind.alias* in the user's home directory, the directory specified by the environment variable **XMBINDDIR**, or the directory `/usr/lib/Xm/bindings`.

The file *xmbind.alias* maps combinations of vendor strings and vendor release numbers to pathnames. Each pathname represents a file that contains keysym bindings for a particular vendor string and optional vendor release number. If Motif fails to find a bindings file for the current display, it uses a set of hard–coded fallback bindings.

The format of the *defaultVirtualBindings* resource is similar to that of a string specifying translations. Each binding consists of an osf keysym, a colon, a list of key event descriptions (with optional modifiers) for actual keysyms, and "\n". Use a comma to separate multiple key event descriptions. The format of a *.motifbind* file or a file containing vendor bindings is the same, except that each binding is on a separate line.

Following is an example of a specification for the *defaultVirtualBindings* resource in a resource file:

```
*defaultVirtualBindings: \
        osfBackSpace:       <Key>BackSpace       \n\
        osfInsert:       <Key>InsertChar      \n\
...
        osfDelete:       <Key>DeleteChar
```

The example specification above appears as follows in a *.motifbind* or vendor bindings file:

```
osfBackSpace:       <Key>BackSpace
osfInsert:       <Key>InsertChar
...
osfDelete:       <Key>DeleteChar
```

For more information, see the **VirtualBindings**(3) and **xmbind**(1) reference pages in the *Motif Programmer's Reference*.

Mnemonics and Accelerators

Sometimes it is desirable for an event received by one widget to activate an action in another. For example, the application may establish a shortcut for activating a button in a menu; the user can activate the menu item even when focus is not in the menu. Motif has two facilities, mnemonics and accelerators, for allowing events in one widget to invoke actions in another.

A mnemonic is a keysym that identifies a key the user can press to activate a menu item when the menu is posted. A button in a MenuBar, PulldownMenu, or PopupMenu can have a mnemonic. When the button is in a PulldownMenu or PopupMenu that is the most recently posted menu, the user activates the button by pressing the key associated with the mnemonic. When the button is in a MenuBar, the MenuBar must have focus for the mnemonic to activate the button. However, the user can activate the button from within the hierarchy that contains the MenuBar, even if the MenuBar does not have focus, by pressing the key while holding the `Alt` modifier.

An application or user supplies a mnemonic for a button by specifying a value for the Label or LabelGadget resource **XmNmnemonic**. When the button is displayed, Motif underlines the first character in the label string that exactly matches the mnemonic in the character set specified by **XmNmnemonicCharSet**. Although the mnemonic must match a character in the label string exactly in order to be underlined, the user can activate the mnemonic by pressing either the shifted or the unshifted key.

An accelerator allows the user to activate a menu item when focus is anywhere in the hierarchy containing the menu, even if the menu is not posted. Accelerators are supported only for PushButtons and ToggleButtons (or their gadget equivalents) in PulldownMenus and PopupMenus.

An application or user supplies an accelerator for a button by specifying a value for the Label or LabelGadget resource **XmNaccelerator**. The value is a string in the same format as an event description in a translation table, except that only **KeyPress** events are allowed. Thus, an accelerator can have a modifier like `Ctrl` or `Alt`. **XmNacceleratorText** is a compound string that describes the accelerator event, for example, "Ctrl+A". Motif displays the accelerator text to the side of the button's label string or pixmap.

The following example creates a button with a mnemonic and an accelerator:

```
n = 0;
XtSetArg(args[n], XmNmnemonic, XStringToKeysym("A"); n++;
XtSetArg(args[n], XmNaccelerator, "Ctrl<Key>A"); n++;
XtSetArg(args[n], XmNacceleratorText,
        XmStringCreateLocalized("Ctrl+A"); n++;
button1 = XmCreatePushButton(file_pane, "Answer", args, n);
```

Motif's button accelerators and mnemonics are supported only for buttons in certain menus. Xt has a more general facility, also called accelerators, for allowing events in one widget to invoke actions in another.

Xt accelerators are mappings of event descriptions to actions, in the same format as a translation table. An application or user supplies accelerators for a widget as the value of the Core resource **XmNaccelerators**. The accelerators map events to actions of this widget, called the source widget. The application must then install the accelerators on a destination widget, using **XtInstallAccelerators**. This routine takes two arguments: the source widget, whose **XmNaccelerators** resource contains the accelerator table; and the destination widget, where the accelerators are to be installed. When the user produces an event in the destination widget that maps to an accelerator in the table, the event invokes the corresponding action *in the source widget*.

XtInstallAccelerators merges the accelerators with the destination widget's existing translations (the value of **XmNtranslations**). Accelerators can be merged in either *#augment* mode, the default, or *#override* mode. An accelerator table may begin with an *#augment* directive or a *#override* directive. The *#replace* directive is ignored.

As with translations, accelerators must be in an internal format when they are the value of **XmNaccelerators**. A string−to−accelerator−table converter parses an accelerator table string from a resource file. An application can use **XtParseAcceleratorTable** to compile an accelerator table string explicitly.

Accelerators are often defined for a parent source widget and installed on one or more child destination widgets. The SelectionBox and FileSelectionBox widgets install accelerators, the value of **XmNtextAccelerators**, on their text children. The default accelerators bind `osfUp`, `osfDown`, `osfBeginLine`, `osfEndLine`, and `osfRestore` events in the Text widget to SelectionBox or FileSelectionBox actions that select an item in the List and replace the Text widget value with that List item.

Event Handlers

Many applications can implement their entire input processing by adding procedures to widget callback lists and by adding mnemonics and accelerators for menu buttons. Some applications change translations, accelerators, or actions. More rarely, an application needs finer control over event processing. Such an application can register an event handler with the Xt event dispatcher.

An event handler is a procedure that the Xt event dispatcher calls when the application receives events of one or more types. An event handler procedure is of type **XtEventHandler**. It receives four arguments: the widget for which the event arrived; any client data registered with the event handler; a pointer to the event; and a Boolean return argument telling the Xt dispatch facility whether or not to call the remaining event handlers registered for this event. This argument is initialized to True and should rarely be changed.

An application usually registers an event handler by using the function **XtAddEventHandler**. The arguments are the widget, an event mask, an indication whether or not the hander should be called for nonmaskable events, the procedure itself, and any client data to be passed to the event handler when it is called. The order in which event handlers are called is undefined when more than one handler exists for a given widget and event type. However, if the application registers the event handler by using **XtInsertEventHandler**, it can specify that the procedure is to be called either before or after all currently registered event handlers.

Motif requires an application to provide an event handler if it wants to post a PopupMenu on a button press. The call to **XtAddEventHandler** should specify **ButtonPressMask** as the event mask and the popup RowColumn as the client data. The event handler should use **XmMenuPosition** to position the menu at the x and y location of the button press event. It should then manage the RowColumn. If the button press matches the event specified by the RowColumn's **XmNmenuPost** resource, Motif posts the PopupMenu. See Chapter 6 for more information.

Guidelines for Grabs

Following is a summary of the restrictions on Motif programs regarding the posting and unposting of shell widgets and the use of grabs:

- The functions **XtAddGrab** and **XtRemoveGrab** cannot be used by a Motif application.
- The Intrinsics convenience popup callback functions **XtCallbackNonexclusive** and **XtCallbackExclusive** cannot be used by Motif applications.
- The functions **XtPopup**, **XtPopupSpringLoaded**, and **XtPopdown** can only be used on shell widgets created with the functions **XtCreatePopupShell** or **XtVaCreatePopupShell**. **XtMapWidget** and **XtUnmapWidget** should never be used on shells created in this manner.
- For Motif applications, the value of the second argument of **XtPopup** must always be **XtGrabNone**.
- Never use **XtManageChild** and **XtUnmanageChild** on shell widgets.
- Use **XtMapWidget** and **XtUnmapWidget** only on widgets with the **XmNmappedWhenManaged** resource set to **False**.
- By default, top level shells (those shells created with **XtAppCreateShell** and **XtVaAppCreateShell**) are mapped at the time that they are realized. Generally, only shells of classes **applicationShellWidgetClass** and **topLevelShellWidgetClass** are created in this manner. Except for iconification, these shells are generally expected to remain mapped until they are destroyed or until the application is terminated. (Transient shells should use the "popup" interfaces).
- For unconventional uses of these top level shells, control of posting and unposting can be achieved by creating these shells with the **XmNmappedWhenManaged** resource set to **False**, and then using **XtMapWidget** and **XtUnmapWidget** to post and unpost them. Note that neither the **XtSetValues** nor the **XtSetMappedWhenManaged** interfaces are effective for this resource on shell widgets; these widgets must be created with **XmNmappedWhenManaged** set to **False** in order to use **XtMapWidget** and **XtUnmapWidget**.