Menus and Options

A menu is a widget that allows the user to make a choice among actions or states. When the menu is visible, the user makes a choice by activating a button in the menu, usually by pressing `Btn1`, `osfSelect`, or `osfActivate` on the button. Some buttons also have mnemonics that allow the user to activate them by pressing the mnemonic keys when the menu is visible. Buttons can also have accelerators, which activate the buttons whether or not the menu is visible.

Like any other widget, a menu must be controlled by some parent widget. However, some types of Motif menu may be shared by more than one widget. (Only one widget will be the parent, but the same menu can be accessed from any other widget on the menu's "post–from" list.)

Motif has four basic kinds of menu:

- MenuBar. This menu is normally always managed within some component of an application, often the MainWindow. It usually consists of a row of CascadeButtons. When the user activates a button in the menu, a PulldownMenu menu appears with one set of top–level choices that apply to the application component.
- PopupMenu. This menu contains a set of choices that apply to a component of the application. The menu is not visible until the user takes an action that posts it, usually pressing `Btn3` in the associated component or pressing `osfMenu` when the component has keyboard focus. A PopupMenu can contain buttons that take action or change state directly. It can also contain CascadeButtons that cause PulldownMenus to appear.
- PulldownMenu. This menu is associated with a CascadeButton in a MenuBar, a PopupMenu, or another PulldownMenu. The menu is not visible until the user posts it by activating the associated CascadeButton. Like a PopupMenu, a PulldownMenu can contain buttons that take action or change state directly. It can also contain CascadeButtons that cause other PulldownMenus to appear.
- OptionMenu. This menu allows the user to choose among one set of choices, usually mutually exclusive attributes or states. It consists of a label, a selection area, and a PulldownMenu. The selection area is a CascadeButtonGadget whose label shows the currently selected option. The PulldownMenu contains the set of options. The user posts the PulldownMenu by activating the CascadeButtonGadget or by pressing *MAlt* along with a mnemonic. When the user activates a button in the PulldownMenu, that button becomes the newly selected option.

RowColumn is the widget that Motif uses as a menu. A RowColumn can also be a nonmenu WorkArea. One use for a WorkArea is to contain a set of ToggleButtons constituting a RadioBox or a CheckBox. When the user selects a ToggleButton, its state changes from on to off or from off to on. In a RadioBox, only one ToggleButton at a time can be on; in a CheckBox, more than one ToggleButton can be on.

RowColumn performs special geometry management to align and lay out its children in a variety of ways. An application can use a RowColumn WorkArea to take advantage of the RowColumn geometry management for a set of widgets. For details see Chapter 14.

In addition to menus, users can also select choices by interacting with a ComboBox or a SpinBox.

A ComboBox is a combination of a List child and a TextField child. This combination gives users two ways to select a choice. That is, the user can either click on a displayed item in the List or type the choice directly into the TextField.

A SpinBox displays a combination of arrows and one or more textual widgets (usually Labels and TextFields). The user makes selections by clicking on the arrows. The arrow clicking increments or decrements the values displayed in the textual widgets. For example, a SpinBox would be an excellent widget to use when the user needs to select a date. The user could click on SpinBox arrows to adjust month and day fields.

Menu Components: Buttons, RowColumn, MenuShell

A menu is a three−level hierarchy:

- Buttons represent the menu selections.
- A RowColumn widget is the manager that contains the buttons.
- A MenuShell envelops each PulldownMenu and PopupMenu.

Buttons

The user makes a choice in a menu by activating one of the buttons in the menu. CascadeButtons, PushButtons, and ToggleButtons and their gadget variants are most commonly used in menus.

> **Note:** Motif does not support DrawnButtons or ArrowButtons in menus, though they can appear in a RowColumn WorkArea. To give a menu button a distinctive appearance, use a PushButton with a label type of **XmPIXMAP** and supply **XmNlabelPixmap** and **XmNlabelInsensitivePixmap** resources.

The application learns of the user's choice through the appropriate button callback lists:

- When the user activates a CascadeButton, the button calls the **XmNcascadingCallback** callbacks. If the button has an attached PulldownMenu after these callbacks return, the button posts the menu. Otherwise, the button calls the **XmNactivateCallback** callbacks.
- When the user activates a PushButton, the button calls the **XmNactivateCallback** callbacks.
- When the user activates a ToggleButton, the button calls the **XmNvalueChangedCallback** callbacks.

Buttons in a menu have translations and actions that arm, disarm, and activate the buttons. These actions also post and unpost menus in the hierarchy at appropriate times. The buttons inherit menu traversal translations and actions from **XmLabel**. These actions allow the user to move from button to button within a menu and from menu to menu within the menu hierarchy.

Use the **XmNenableEtchedInMenu** resource of **XmDisplay** to help control the shadowing of activated menu buttons.

RowColumn

The parent of the buttons in a menu is a RowColumn widget. RowColumn interacts with its button children in these ways:

- In a menu (but not a WorkArea), it ensures that all children are CascadeButtons, PushButtons, ToggleButtons, Labels, or Separators (or their gadget variants). If the **XmNisHomogeneous** resource is True, it ensures that all children are of the class specified by **XmNentryClass**.
- It lays out its children and, if **XmNisAligned** is True, aligns the labels of children that are **XmLabel** or **XmLabelGadget** subclasses.
- It stores the widget ID of the last menu item selected in the **XmNmenuHistory** resource.
- It allows the application to supply a single callback list for all button children. If **XmNentryCallback** is not NULL, it disables the **XmNactivateCallback** and **XmNvalueChangedCallback** callbacks for its button children and arranges for the buttons to call the **XmNentryCallback** callbacks instead.
- If **XmNradioBehavior** is True, it ensures that only one ToggleButton at a time is normally selected. It also changes the default values for **XmNindicatorType** and **XmNvisibleWhenOff** for its ToggleButton children to the one−of−many, always−displayed style.
- It has additional resources for MenuBars and OptionMenus, described in the following sections.

In addition to **XmNentryCallback**, RowColumn also has **XmNmapCallback** and **XmNunmapCallback** callbacks. These callbacks apply only to PopupMenus and PulldownMenus. The **XmNmapCallback** callbacks are called just before the menu is posted, and the **XmNunmapCallback** callbacks are called just after the menu is unposted. They are useful for changing the menu to reflect the current state of the application. For example, an **XmNmapCallback** callback can use **XtSetSensitive** to make some menu items insensitive if they are not applicable in the current state of the program.

MenuShell

The windows associated with PopupMenus and PulldownMenus are top−level windows. That is, the parent window of such a menu is the root window of the screen, not the window associated with the parent widget. This allows the menu to appear anywhere on the screen without being clipped by the parent widget's window.

The parent widget of each PopupMenu and PulldownMenu RowColumn must be a MenuShell. It is actually the MenuShell's window that is the top−level window. **XmMenuShell** is a subclass of OverrideShell, so the window manager ignores MenuShell's windows.

A MenuShell is often invisible to the application. The Motif convenience routines for creating PopupMenus and PulldownMenus automatically create MenuShell parents for these menus. When a PulldownMenu is the child of a PopupMenu or another PulldownMenu, the child's MenuShell is actually the child of the parent's MenuShell. The convenience routines for creating PulldownMenus manage these relations automatically.

Motif arranges for the RowColumn's window to coincide with the MenuShell's window. Setting **XmNheight**, **XmNwidth**, or **XmNborderWidth** for either a MenuShell or its child sets that resource to the same value in both the parent and the child. For a child of a MenuShell, setting **XmNx** or **XmNy** sets the corresponding resource of the parent but does not change the child's position relative to the parent. **XtGetValues** for the child's **XmNx** or **XmNy** yields the value of the corresponding resource in the parent. The x and y coordinates of the child's upper left outside corner relative to the parent's upper left inside corner are both zero minus the value of **XmNborderWidth**.

To change any geometry−related resources of a PopupMenu or PulldownMenu, an application should always specify these resources for the RowColumn child, not the MenuShell parent.

If an application needs to create a MenuShell explicitly, it should create the MenuShell as a popup child of its parent (using **XtCreatePopupShell** or **XtVaCreatePopupShell**). All Motif convenience routines that create MenuShells do this automatically, and an application rarely needs to create a MenuShell directly.

MenuBar

All children of a MenuBar must be CascadeButtons or CascadeButtonGadgets. The MenuBar attempts to place its button children in a single row. If it does not have enough room, it tries to wrap the remaining children into additional rows.

An application should treat specially the button, if any, that pulls down a help menu. The application should set the MenuBar RowColumn's **XmNmenuHelpWidget** to the widget ID of this button. The MenuBar attempts to place this button at one of the lower corners of the MenuBar, as specified by the *Motif Style Guide*.

In a MenuBar, all buttons typically have associated PulldownMenus. Each PulldownMenu associated with a button in a MenuBar must be a child of the MenuBar. (More precisely, each PulldownMenu's MenuShell must be a child of the MenuBar.) Each button's **XmNsubMenuId** resource must be set to the widget ID of the associated PulldownMenu. Set **XmNsubMenuId** to the widget ID of the PulldownMenu RowColumn, not of the PulldownMenu's MenuShell.

The routines **XmCreateMenuBar**, **XmCreateSimpleMenuBar**, and **XmVaCreateSimpleMenuBar** all create MenuBars.

PopupMenu

A PopupMenu is normally invisible. When the user takes some action––usually pressing `Btn3` or `osfMenu`––in a widget that has a PopupMenu, the menu is posted. The user moves from item to item in the menu by dragging `Btn3` or, when keyboard traversal is enabled, by keyboard traversal actions. Motif unposts the menu when the user activates an item in the menu system (other than a CascadeButton), presses `osfCancel`, or releases or clicks `Btn3` outside a menu item.

A PopupMenu RowColumn must have a MenuShell parent. The parent of the MenuShell is the widget with which the PopupMenu is associated. Because the MenuShell is a popup child of its parent, the parent can be any widget (but not a gadget); it does not have to be a subclass of **Composite**. The Motif convenience routines that create PopupMenus automatically create a MenuShell as the parent of the PopupMenu RowColumn.

Several different widgets may share a given Popup menu. One widget must serve as the parent of the MenuShell, but other widgets may be nominated with the **XmAddToPostFromList**(3) function.

The PopupMenu's **XmNmenuPost** resource specifies the button event that posts the menu. This event will make the menu visible in any widget on the menu's list of eligible widgets. The event must be a button press, possibly with modifiers. However, you should not set **XmNmenuPost** to *BTransfer Press* because many button widgets use *BTransfer Press* as a default binding to initiate a Drag operation.

When a user creates a PopupMenu, Motif installs an event handler on the menu's widget parent. Called *PopupMenuEventHandler*, this routine performs most of the necessary setup for a PopupMenu, and manages the RowColumn widget as well.

On receipt of an event that should post a PopupMenu, this event handler searches the widget hierarchy and the popup list for the appropriate menu to post. The appropriate menu will have its **XmNmenuPost** resource match the triggering event, will have its **XmNpopupEnabled** resource set to **XmPOPUP_AUTOMATIC** or **XmPOPUP_AUTOMATIC_RECURSIVE**, and will have been created before any other PopupMenu that satisfies the same criteria. If the popup menu is found in a parent of the target widget, the **XmNpopupEnabled** resource must be set to **XmPOPUP_AUTOMATIC_RECURSIVE**.

> **Note:** In older versions of Motif, developers were required to write their own event handlers in order to implement PopupMenus. In the upgrade to version 2.0, substantial changes were made to the popup menu system to simplify the creation and management of PopupMenus. However, applications using this approach will still work correctly under the current version of Motif.

Once a menu selection has been made, control shifts to the *popupHandlerCallback* procedure in the menu's parent widget, if there is one. The callback allows a more specific menu to be posted, if necessary, or can even perform the selected menu function, if desired.

A pointer to the following structure is passed to each callback for **XmNpopupHandlerCallback**:

```
typedef struct
{
        int     reason ;
        XEvent  * xevent ;
        Widget  menuToPost  ;
        Boolean postIt ;
        Widget  target ;
} XmPopupHandlerCallbackStruct;
```

reason

> Indicates why the callback was invoked. **XmCR_POST** implies that this is a regular posting request, while **XmCR_REPLAY** indicates that the menu was just unposted, and that this callback was invoked on a replay.

xevent

> Points to the **XEvent** that triggered the handler.

menuToPost

> Specifies the PopupMenu that the menu system is to post. The application may modify this field.

postIt

Indicates whether the posting process should continue. The application may modify this field.

target

The target widget or gadget.

Posting a PopupMenu through the keyboard is controlled by the PopupMenu's **XmNmenuAccelerator** and **XmNpopupEnabled** resources. **XmNmenuAccelerator** specifies a key event that may post the menu. **XmNpopupEnabled** specifies whether or not this event actually posts the menu. It also determines whether or not accelerators and mnemonics in the PopupMenu and its submenus are enabled.

An application can have only one active PopupMenu at a time for a particular widget. If the widget has more than one PopupMenu, the application should set **XmNpopupEnabled** to True for the active menu and set **XmNpopupEnabled** to False for all inactive menus.

PulldownMenu

A PulldownMenu is always associated with another RowColumn. It becomes visible when the user activates a CascadeButton in the associated RowColumn. It becomes invisible when the user traverses upward or laterally in the menu hierarchy, activates a button in the hierarchy (other than a CascadeButton in the menu or a descendant), presses `osfCancel`, or clicks or releases a mouse button outside a menu item.

A PulldownMenu must have the following relations with other widgets:

- It must be the value of the **XmNsubMenuId** resource of the CascadeButton that is to post the menu.
- It must have a MenuShell as its parent. The Motif convenience routines that create PulldownMenus create MenuShell parents automatically. As with the PopupMenu, several different widgets may share a given Pulldown menu. One widget must serve as the parent of the MenuShell, but other widgets may be nominated with the **XmAddToPostFromList**(3) function.
- The MenuShell must have the proper parent, depending on the kind of RowColumn with which the PulldownMenu is associated. The MenuShell is a popup child of its own parent. Following are the required parents of the MenuShell:
  - If the PulldownMenu is to be pulled down from a MenuBar, the parent must be the MenuBar.
  - If the PulldownMenu is to be pulled down from a PopupMenu or another PulldownMenu, the parent must be that PopupMenu or PulldownMenu. Actually, the parent is the other menu's MenuShell; but the *parent* parameter to the Motif convenience routines that create PopupMenus must be the other menu itself (the RowColumn), not its MenuShell parent.
  - If the PulldownMenu is to be pulled down from an OptionMenu, the parent must be the parent of the OptionMenu.

OptionMenu

An OptionMenu lets the user choose among a set of usually mutually exclusive options. The OptionMenu is always visible. It consists of a label (a LabelGadget), a selection area (a CascadeButtonGadget), and an associated PulldownMenu. The label of the CascadeButtonGadget displays the currently selected option, one of the items in the PulldownMenu. When the user activates the CascadeButtonGadget, the PulldownMenu becomes visible with the currently selected item directly above the selection area. When the user activates an item in the PulldownMenu, the PulldownMenu is unposted and the item the user chose becomes the currently selected option.

The PulldownMenu normally contains only PushButtons. It must not contain any ToggleButtons, and Motif does not support CascadeButtons.

RowColumn has a number of resources for use specifically with an OptionMenu:

XmNlabelString

The text of the label. Setting this resource also sets the **XmNlabelString** of the LabelGadget.

XmNmnemonic

A keysym that, when pressed along with the *MAlt* modifier, posts the PulldownMenu. Motif underlines the first character in the label string that matches the mnemonic and that is in a segment whose font list element tag matches **XmNmnemonicCharSet**. Setting this resource also sets the **XmNmnemonic** of the LabelGadget.

XmNmnemonicCharSet

The font list element tag used for underlining the mnemonic. Setting this resource also sets the **XmNmnemonicCharSet** of the LabelGadget.

XmNsubMenuId

The widget ID of the PulldownMenu. Setting this resource also sets the **XmNsubMenuId** of the CascadeButtonGadget.

If the application needs to get or set any of these four resources for the LabelGadget or CascadeButtonGadget, it should always get or set it in the OptionMenu RowColumn, not the gadget itself. To get or set other resources for the gadgets, the application should use **XmOptionLabelGadget** or **XmOptionButtonGadget** and then call **XtGetValues** or **XtSetValues** on the returned widget ID. A user or application can also specify resource values in resource files by using the names of the gadgets, "OptionLabel" and "OptionButton".

Setting the **XmNmenuHistory** resource also has a special effect in OptionMenus. Setting **XmNmenuHistory** to an item in the PulldownMenu makes that item the currently selected option. It updates the label of the CascadeButtonGadget and causes the PulldownMenu to appear, when posted, with the selected item over the CascadeButtonGadget.

XmCreateOptionMenu creates an OptionMenu RowColumn and its LabelGadget and CascadeButtonGadget children. It does not create the associated PulldownMenu.

The following example creates a simple OptionMenu with three options:

```
Widget          parent, pulldown, option, pb1, pb2, pb3;
Arg             args[10];
Cardinal        n;
...
n = 0;
pulldown = XmCreatePulldownMenu(parent, "option_pd",
                                args, n);
pb1 = XmCreatePushButtonGadget(pulldown, "option_pb1",
                                args, n);
pb2 = XmCreatePushButtonGadget(pulldown, "option_pb2",
                                args, n);
pb3 = XmCreatePushButtonGadget(pulldown, "option_pb3",
                                args, n);
XtSetArg(args[n], XmNsubMenuId, pulldown);       n++;
XtSetArg(args[n], XmNmenuHistory, pb2);          n++;
option = XmCreateOptionMenu(parent, "option_rc", args, n);
...
```

The following application−class defaults file provides labels and mnemonics for an English−language locale:

```
*option_pb1.labelString:   Option 1
*option_pb2.labelString:   Option 2
*option_pb3.labelString:   Option 3
*option_rc.labelString:   Options
*option_rc.mnemonic:   O
```

RadioBox and CheckBox

RadioBoxes and CheckBoxes are collections of ToggleButtons. The primary difference is that in a RadioBox only one ToggleButton at a time can be set; in a CheckBox more than one ToggleButton can be set. In a RadioBox, a ToggleButton cannot be in an indeterminate state; in a CheckBox, a ToggleButton can be in an indeterminate state.

RadioBoxes and CheckBoxes are usually implemented as WorkAreas, though it is possible to implement them as menus. Usually the application intends for the box to remain visible after the user sets a ToggleButton, particularly in a CheckBox. The application can implement a transient RadioBox or CheckBox by placing a WorkArea inside a dialog.

The following RowColumn resources specifically control the behavior of a RadioBox or CheckBox:

XmNradioBehavior

> When True, the RowColumn ensures that at most one ToggleButton is set at a time. Setting this resource to True also causes the ToggleButton resource **XmNindicatorType** to default to **XmONE_OF_MANY** and **XmNvisibleWhenOff** to default to True.

XmNradioAlwaysOne

> When both this resource and **XmNradioBehavior** are True, RowColumn ensures that one ToggleButton is always set. The user is not allowed to unset a ToggleButton when no other ToggleButton is set.

For a RadioBox implemented as a WorkArea, the default value for **XmNisHomogeneous** is True, and by default RowColumn allows only ToggleButton and ToggleButtonGadget children.

Note that the application can foil the RowColumn's enforcement of **XmNradioBehavior** and **XmNradioAlwaysOne**, even when these resources are True. The application can use **XtSetValues** to set the state of the ToggleButtons, and it can manage and unmanage ToggleButtons regardless of their state. The behavior of a RadioBox is undefined if the application takes actions that contradict **XmNradioBehavior** or **XmNradioAlwaysOne**.

XmCreateRadioBox creates a WorkArea RadioBox and initializes **XmNradioBehavior** to True.

A CheckBox is most often a collection of ToggleButtons in a WorkArea with **XmNradioBehavior** set to False. By default, the ToggleButton **XmNindicatorType** is **XmN_OF_MANY** and **XmNvisibleWhenOff** is True.

TearOffMenus

An application can allow the user to "tear off" a PulldownMenu or PopupMenu. When the user tears off a menu, Motif unposts that menu and any posted menu descendants. It gives the menu a TransientShell parent and then maps the parent as a top−level window. The torn−off menu has window−manager decorations, and its title can be specified with the **XmNtearOffTitle** resource of the RowColumn menu widget. If the title is not so specified, it will be the same as the label of the CascadeButton that posts the menu in the original menu system.

The user can interact with the torn−off menu just as in the menu hierarchy. When the user activates buttons in a torn−off menu, the actions take effect but the torn−off menu remains posted. When the user takes an action that unposts the torn−off menu, such as pressing `osfCancel`, the menu returns to its original position in the menu hierarchy. If the user reposts the original menu from the menu hierarchy while the torn−off menu is posted, an inactive representation of the torn−off menu remains visible, but the menu itself is unposted and then reposted within the menu hierarchy.

When a menu in a menu system can be torn off, a distinctive tear−off button appears at the beginning of the menu. The user can tear off the menu by activating the tear−off button as with any other button in the menu. The user can also tear off the menu by pressing `Btn2` in the tear−off button. The user can then drag the torn−off menu to another position on the screen and fix its position by releasing `Btn2`.

Menus cannot be torn off by default. The application must allow the user to tear off a menu by setting the RowColumn resource **XmNtearOffModel** to **XmTEAR_OFF_ENABLED**. When the user tears off a menu, the **XmNtearOffMenuActivateCallback** callbacks are invoked just before the **XmNmapCallback** callbacks. When the user unposts a torn−off menu, the **XmNtearOffMenuDeactivateCallback** callbacks are invoked just after the **XmNunmapCallback** callbacks.

ComboBox

The ComboBox widget manages both a List widget and a TextField widget. This combination of List and TextField gives users two ways to choose an item. The user can choose an item either by selecting one of the entries displayed by the List widget or by typing the selection directly into the TextField widget.

By default, the ComboBox displays both the TextField widget and the List widget. However, it is possible to hide the List widget so that it will only be displayed when the user requests it. If the List is hidden, the user can make it visible by clicking on a displayed arrow.

The TextField can either be editable or non−editable. If the TextField is non−editable, the user must pick one of the choices displayed by the List.

ComboBox Types

Motif provides three types of ComboBox widgets. The widget type is specified using the **XmNcomboBoxType** resource on ComboBox. The possible settings for this resource are as shown in Table 6−1.

Table 1 XmNcomboBoxType Resource Values

Value	*TextField*	*List Widget*
XmCOMBO_BOX	Editable	Always displayed
XmDROP_DOWN_COMBO_BOX	Editable	Hidden
XmDROP_DOWN_LIST	Non−editable	Hidden

XmDROP_DOWN_LIST is the default value.

ComboBoxes with hidden Lists are usually more desirable when screen space is limited. ComboBoxes with hidden Lists are also recommended when users are more likely to enter text into the TextField than to choose an item from the List.

Creating and Manipulating ComboBox

Motif provides the following three convenience functions for creating ComboBoxes:

- **XmCreateComboBox**
- **XmCreateDropDownComboBox**
- **XmCreateDropDownList**

Each function creates an instance of a ComboBox widget and returns its associated widget ID.

Each function requires an *arglist* parameter. The resource values specified in *arglist* will be passed not only to the ComboBox, but to the List and TextField as well. Thus, an application can specify resources for the List such as **XmNitems** and **XmNvisibleItemCount** by including them in the *arglist* parameter passed to the ComboBox.

The List part of the ComboBox expects an array of compound strings to fill the list. If an array is not passed to the List at creation time, the application must provide this array later. Each string becomes an item in the list, with the first string becoming the item in position 1, the second string becoming the item in position 2, and so on. Your application can obtain the **XmList** widget ID by specifying **List* as an argument to *XtNameToWidget( )*. Similarly, the TextField widget in the ComboBox can be accessed by passing **Text* to *XtNameToWidget( )*. You can then call a convenience function, such as **XmListAddItem**, to add more List items. You can also access the **List** and **TextField** widgets through the **XmNlist** and **XmNtextField** resources, but these resources are read−only.

If the TextField widget is editable, the user can type directly in the text field to enter a selection. If the application wishes to validate the entered text, it can do so by installing the **XmNmodifyVerifyCallback** on the TextField widget directly.

The following example illustrates how to create a drop−down ComboBox containing five items. It illustrates how to use the **XmCreateComboBox( )** function, and how to pass its children resources at creation time. This example uses *XtNameToWidget( )* to manipulate the List child, setting the List child's **XmNvisibleItemCount** to 5.

```
{
#define NUM_LIST_ITEMS 5
   Widget          comboBox;
   Arg             args[10];
   Cardinal        n, i;
```

```
    XmString        ListItem[NUM_LIST_ITEMS];
    static char    *ListString[] = { "kiwi",
                                     "raspberry",
                                     "carambola",
                                     "litchi",
                                     "coconut" };

    /* Create a list of XmStrings for the ComboBox List child */
      for (i=0; i < NUM_LIST_ITEMS; i++)
        ListItem[i] = XmStringCreate (ListString[i], XmSTRING_DEFAULT_CHARSET);


    /* Create a ComboBox of type XmDROP_DOWN_COMBO_BOX. */
    /* Resources passed to ComboBox are passed on to the
     * children of ComboBox.  So, in the argument list
     * below, the resources, XmNitems, and XmNitemCount
     * will be passed on to the List child of ComboBox.  */
    n=0;
    XtSetArg (args[n], XmNcomboBoxType, XmDROP_DOWN_COMBO_BOX); n++;
    XtSetArg (args[n], XmNarrowSpacing, 5); n++;
    XtSetArg (args[n], XmNitems, ListItem); n++;
    XtSetArg (args[n], XmNitemCount, NUM_LIST_ITEMS); n++;
    comboBox = XmCreateComboBox (parent, "ComboBox", args, n);
    XtManageChild (comboBox);
    XtAddCallback (comboBox, XmNselectionCallback, SelectionCB,
       (XtPointer)NULL);

    /* Example of manipulating a child widget directly to set the
     * visibleItemCount on the list.  */
    n=0;
    XtSetArg (args[n], XmNvisibleItemCount, 5); n++;
    XtSetValues (XtNameToWidget (comboBox,"*List"), args, n);
}
```

ComboBox Items

You may add and manipulate Combobox items using the functions

- **XmComboBoxAddItem**
- **XmComboBoxSelectItem**
- **XmComboBoxSetItem**
- **XmComboBoxDeletePos**

All of these functions take a widget ID that specifies the ComboBox; none returns a value.

- **XmComboBoxAddItem** takes an **XmString** specifying the new item and assigns to the item a position in the list.
- **XmComboBoxDeletePos** deletes an item by reference to its position.
- **XmComboBoxSelectItem** selects an item in the **XmList** of a ComboBox widget.
- **XmComboBoxSetItem** causes a specified item to be the first visible item in the list.

Controlling the Arrow

If a ComboBox has a hidden List, then the ComboBox displays an arrow. The arrow will always be displayed adjacent to the TextField. However, the placement of this arrow depends on the **XmNlayoutDirection** resource of the VendorShell (or subclass of) in which the ComboBox is contained, as follows:

- If the value of **XmNlayoutDirection** is **XmLEFT_TO_RIGHT**, ComboBox places the arrow to the right of the TextField field.
- If the value of **XmNlayoutDirection** is **XmRIGHT_TO_LEFT**, ComboBox places the arrow to the left of the TextField field.

Use **XmNarrowSize** to set the requested size of the arrow. Use **XmNarrowSpacing** to set the spacing between the arrow and the TextField.

Note that **XmNarrowSize** is a size request, not a size guarantee. For example, if the requested arrowSize is larger than the current size of the

ComboBox, ComboBox will have to make a geometry request to its parent. Whether the geometry request is wholly, partially, or not accepted depends on the parent. If this request is accepted, the arrow size request can be granted. If the ComboBox's request to grow is only partially granted or not granted at all, ComboBox will make the arrow size the maximum that will fit inside the ComboBox's allowed size.

ComboBox Matching Behavior

If the ComboBox does not have an editable TextField, any text that the user types will not appear in the TextField. Instead, typing text may cause a matching algorithm to be invoked that will attempt to match the entered text with an item in the list. If a match is found, the item is selected, and the item appears in the TextField field of the ComboBox. Whether a specific matching algorithm is applied or not is determined by the value of the **XmNmatchBehavior** resource on ComboBox. There are two values that the **XmNmatchBehavior** resource can accept: **XmNONE** and **XmQUICK_NAVIGATE**. A value of **XmNONE** indicates that no matching algorithm will be invoked. A value of **XmQUICK_NAVIGATE** indicates that when the List widget has focus, a one–character navigation is supported. In this algorithm, if the typed character is the initial character of some item in the **XmList**, this algorithm causes that item to be navigated to and selected. Subsequently typing the same character will cycle among the items with the same first character.

ComboBox Callbacks

When a selection occurs in ComboBox, the **XmNselectionCallback** callbacks are called. For example, the code fragment appearing earlier in this section established a selection callback procedure named *SelectionCB*. Here is the code for that callback:

```
void
SelectionCB (Widget w, XtPointer client_data, XtPointer call_data)
{
 XmComboBoxCallbackStruct *cb = (XmComboBoxCallbackStruct *)call_data;
 XmStringCharSet        charset;
 XmStringDirection      direction;
 XmStringContext        context;
 Boolean                separator;
 char                  *item;

   /* This callback procedure prints the item that was just selected. */

   /* Retrieve the selected text string from XmString. */
     item = XmStringUnparse(cb->item_or_text, NULL, XmCHARSET_TEXT,
                            XmCHARSET_TEXT, NULL, 0, XmOUTPUT_ALL);
     printf ("ComboBox SelectionCB: item = %s\n\n", item);
}
```

ComboBox allows a single item to be selected in many ways, including through a matching behavior. A list item can more directly be selected by scrolling to it with either the mouse or keyboard, and selecting the item directly. ComboBox supports the Browse Select selection method of **XmList**. If the ComboBox has an editable text field, the selection can be typed directly into the TextField entry field. Regardless of the selection mechanism used, when an item in the list is selected, the list item is highlighted by displaying it in reverse colors, and the selected item is displayed in the TextField field in the ComboBox.

Additionally, if the user performs an action that moves focus away from the ComboBox, the **XmNselectionCallbacks** are called to notify the application of the user's choice. The *item_or_text* field in the call data structure passed to the callback will contain the contents of the TextField field at the time of the callback. The application then takes whatever action is required for the specified selection.

The **XmComboBoxUpdate** function resynchronizes the internal data structures of a specified ComboBox widget. This function is useful when an application manipulates ComboBox's child widgets, possibly changing data structures. For example, you might want to use the **XmComboBoxUpdate** function after a ComboBox List child selection policy has been changed without notification.

SpinBox

A SpinBox lets a user select one choice from a set of mutually exclusive options. A SpinBox contains an increment and a decrement ArrowButton, which allow the user to change the displayed choice. You might think of a SpinBox as a "ring" of choices that operates similar to a digital clock. It displays choices consecutively, and, at the last item, wraps around to the first (or, going in the other direction, wraps from the first item to the last). As long as the user holds down an ArrowButton, the current position in the ring of choices changes continuously, with a "spinning" effect.

To control when spinning begins, set the **XmNinitialDelay** resource to specify the time, in milliseconds, that elapses before the displayed value changes. To control the rate of spinning, set the **XmNrepeatDelay** resource to specify the time, in milliseconds, that elapses between subsequent changes in the displayed value.

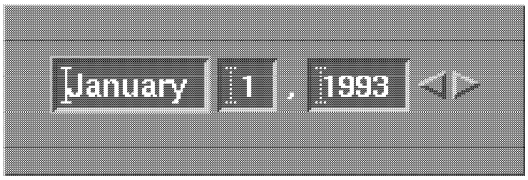
A SpinBox may include traversable text children, which display the choices, and label and separator children, which are used for informative or decorative purposes only. There are two types of traversable text children allowed in a SpinBox: numeric and string.

- For a numeric child, define the choices by passing in integers for a minimum value, a maximum value, and an incremental value. By setting the constraint resource, **XmNdecimalPoints**, you can display decimal values. The increment arrow increases the displayed SpinBox value by the given incremental value. The decrement arrow decreases the displayed value by the given incremental value.
- For a string child, provide an array of compound strings, using **XmStringTable**. The increment arrow displays the next string, moving toward the end of the array. The decrement arrow displays the previous string, moving toward the beginning of the array.

To create a SpinBox with one ring of choices, use **XmCreateSpinBox** and create a child (either numeric or string) using **XmCreateText** or **XmCreateTextField**.

The SpinBox, however, may manage multiple traversable text children, as illustrated by the SpinBox in Figure 6–1. Three traversable text children display months, days, and years. A decoration child (the comma) separates the displayed day and year. The two ArrowButtons are part of every SpinBox.

Figure 1 A SpinBox with Multiple Children



Because the text children are traversable, the user can tab from one field to the next to change focus. Only one child at a time can take focus. The ArrowButtons spin only choices in the text field with focus. It is possible to "chain" the choices so that spinning choices in one text field affects what is displayed in other text fields. For example, when a user spins the day child in Figure 6–1 up to 31, **XmNvalueChangedCallback** is called for each change in position. When the user spins from 31 to 1, the **XmNvalueChangedCallback** function can use **XtSetValues** to change the **XmNposition** resource for the month child from 1 to 2 so that February is displayed.

February has fewer days than January. Therefore, when the month changes from January to February, the **XmNmodifyVerifyCallback** procedure must change the **XmNmaximumValue** of the day child from 31 to 28 (or 29 if leap year). If the year rolls the calendar backward from February to January, then the **XmNmodifyVerifyCallback** procedure must reset the **XmNmaximumValue** of the day child back to 31.

SpinBox provides two callbacks:

XmNmodifyVerifyCallback

Whenever an ArrowButton is pressed but before the position of the SpinBox changes, **XmNmodifyVerifyCallback** is invoked. This callback lets you control the next SpinBox position or perform other actions before the SpinBox value changes. The callback function can modify two members of the callback structure: *doit* and *position*. Set these as follows:

To prevent the spinning action, set *doit* to False.

To allow normal consecutive spinning, do not change *doit* or *position*.

To spin to another value in the ring of choices, change *position* and leave *doit* set to True.

When *doit* is False, the SpinBox position and value do not change. You would set *doit* to False if you wanted the SpinBox to stop at the

first or last choice without wrapping around the ring of choices. When *doit* is True, spinning continues to the position returned in the callback structure, and **XmNvalueChangedCallback** is invoked.

XmNvalueChangedCallback

Whenever an ArrowButton is pressed and the SpinBox position has just changed, **XmNvalueChangedCallback** is invoked. This callback lets you perform any action based on the change in the callback structure members, *position* and *value*. In addition to returning a reason, this callback indicates the event that triggered the callback, which child is affected, and the SpinBox position. For **XmSTRING** children, *value* contains the displayed string.

The SpinBox widget also includes translations to arm and disarm the SpinBox, increase and decrease the SpinBox position, and go to the first and last position.

The following SpinBox resources control arrow geometry, sensitivity and response:

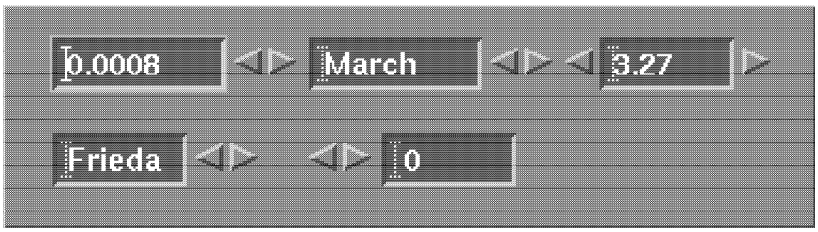
XmNdefaultArrowSensitivity

Specifies whether one or both ArrowButtons are sensitive (perform spinning) or insensitive (displayed as stippled and do not respond) by default. This allows you to turn off an arrow button; for example, you may want to stipple the increment arrow when the SpinBox reaches the last or maximum choice. The constraint resource **XmNarrowSensitivity** can override this resource setting for a particular child.

XmNarrowLayout

Sets the ArrowButton layout. Arrows can be displayed side by side, either to the left or to the right of a text child, or split at either end of the text child. They may also be displayed one on top of the other, before or after the children; or side by side, before or after the children. Arrow layout depends on **XmNlayoutDirection**. When layout direction is left to right, beginning arrows are positioned to the left. When layout direction is right to left, beginning arrows are positioned to the right. Figure 6–2 illustrates the arrow layouts when **XmNlayoutDirection** is left to right. The first two SpinBoxes (*.0008* and *March*) show layout at the end of the child. The third SpinBox (*3.27*) shows a split layout. The last SpinBox (*0*) shows layout at the beginning of the child.

Figure 2 SpinBox Arrow Layouts



XmNarrowOrientation

Sets the orientation of the ArrowButtons: vertical or horizontal.

XmNarrowSize

Sets both the width and height of the two ArrowButtons.

XmNinitialDelay

Sets the length of time the mouse button can be held down before automatic spinning begins. This resource and the **XmNrepeatDelay** resource allow you to control the rate of spinning.

XmNrepeatDelay

Sets a delay between each change in the SpinBox choice position.

The following constraints affect each SpinBox traversable child individually:

XmNarrowSensitivity

Sets the sensitivity of one or both ArrowButtons for a specific traversable child. This resource overrides the SpinBox **XmNdefaultArrowSensitivity** resource.

XmNspinBoxChildType

Specifies whether the ring of choices is of numeric or string type.

XmNposition

Specifies the current position in the range of valid numbers or in the string array. The interpretation of this resource depends on the **XmNpositionType** resource.

XmNpositionType

Specifies how the value in the **XmNposition** resource is to be interpreted: as its index or as the data value.

XmNvalues

Defines an array of strings for a SpinBox string type child.

XmNnumValues

The number of strings in the **XmNvalues** array.

XmNminimumValue

Sets the minimum value for a SpinBox numeric type child.

XmNmaximumValue

Sets the maximum value for a SpinBox numeric type child.

XmNincrementValue

Sets the value by which to raise or lower the displayed value in a SpinBox numeric type child.

XmNdecimalPoints

Specifies the number of decimal places to display for a SpinBox numeric type child.

If the Core resources **XmNwidth** and **XmNheight** are not specified, the SpinBox widget attempts to grow to accommodate large arrows or long text fields. It does not shrink if arrows or children are very small.

The following code creates a SpinBox with one string type child, as shown in Figure 6–3. The choices are values from January to December. This code invokes only the **XmNvalueChangedCallback**, which is called after the displayed value has changed.

Figure 3 A SpinBox with One Child

```
...
/*****  Create SpinBox parent  *****/
n = 0;
XtSetArg(argList[n], XmNy, nextY); n++;

spin0 = XmCreateSpinBox(parent, "spin0", argList, n);

/*****  Create XmString array of month names  *****/
setMonths();

/*****  Create TextField child  *****/
```

```
n = 0;
XtSetArg(argList[n], XmNvalues, monthValues); n++;
XtSetArg(argList[n], XmNnumValues, 12); n++;
XtSetArg(argList[n], XmNspinBoxChildType, XmSTRING); n++;
XtSetArg(argList[n], XmNselectionPolicy, XmSINGLE_SELECT); n++;
XtSetArg(argList[n], XmNeditable, False); n++;

spin0_text = XmCreateTextField(spin0, "spin0_text", argList, n);

/*****  Manage SpinBox  *****/
XtManageChild(spin0);

/*****  Call changedSpin0 AFTER displayed value has changed  *****/
XtAddCallback(spin0, XmNvalueChangedCallback, changedSpin0, (XtPointer) 0);

/*****  Manage SpinBox child  *****/
XtManageChild(spin0_text);
...
```

The *setMonths* routine creates an **XmString** array of month names; *setMonths* consists of the following code:

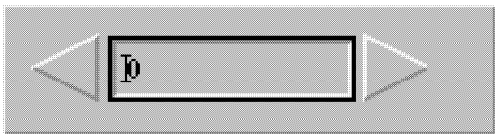
```
void
setMonths(void)
{
XmString tempString;
int      monthLoop;

    for (monthLoop = 0; monthLoop < NUM_MONTHS; monthLoop++) {
        tempString = XmStringCreate(months[monthLoop],
                                    XmFONTLIST_DEFAULT_TAG);
        monthValues[monthLoop] = tempString;
        }
}
```

The following callback code for **XmNvalueChangedCallback** modifies the variable *thisYear* whenever the boundary is crossed.

```
void
changedSpin0(Widget w, XtPointer client, XtPointer call)
{
XmSpinBoxCallbackStruct *user;
static int              thisYear = 1994;

    user = (XmSpinBoxCallbackStruct *)call;

    if (user->crossed_boundary)
        {
        if (user->reason == XmCR_SPIN_NEXT)
          thisYear++;
        else
          thisYear--;
        }
}
```

The code for the SpinBox shown in Figure 6–4 includes an **XmModifyVerifyCallback** to change resources and prevent wrapping below the minimum value.

Figure 4 A SpinBox that Does Not Wrap



```
...
```

```
/*****  Create SpinBox parent  *****/
n = 0;
XtSetArg(argList[n], XmNarrowSize, 35); n++;
XtSetArg(argList[n], XmNrepeatDelay, 250); n++;
XtSetArg(argList[n], XmNinitialDelay, 500); n++;
XtSetArg(argList[n], XmNarrowLayout, XmARROWS_SPLIT); n++;

spin4 = XmCreateSpinBox(parent, "spin4", argList, n);

n = 0;
XtSetArg(argList[n], XmNmaximumValue, 4); n++;
XtSetArg(argList[n], XmNspinBoxChildType, XmNUMERIC); n++;

spin4_text = XmCreateTextField(spin4, "spin4_text",
                               argList, n);


/*****  Manage SpinBox  *****/
XtManageChild(spin4);

/*****  Call modifySpin4 BEFORE displayed value is changed  *****/
XtAddCallback(spin4, XmNmodifyVerifyCallback, modifySpin4,
              (XtPointer) 0);

/*****  Manage SpinBox child  *****/
XtManageChild(spin4_text);
...
```

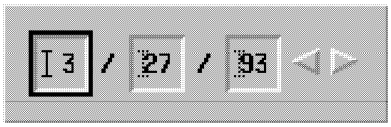
The callback code for the previous call to **XmNmodifyVerifyCallback** uses *doit* to stop the SpinBox from wrapping at the minimum value:

```
void
modifySpin4(Widget w, XtPointer client, XtPointer call)
{
XmSpinBoxCallbackStruct *user;
int                     newHigh;
Cardinal                n;
Arg                     argList[5];

    user = (XmSpinBoxCallbackStruct *)call;

    if (user->crossed_boundary)
        {
        if (user->reason == XmCR_SPIN_NEXT)
            {
            n = 0;
            XtSetArg(argList[n], XmNmaximumValue, &newHigh); n++;
            XtGetValues(user->widget, argList, n);

            newHigh++;

            n = 0;
            XtSetArg(argList[n], XmNmaximumValue, newHigh); n++;
            XtSetValues(user->widget, argList, n);
            user->position = 0;
            }
        else if (user->reason == XmCR_SPIN_PRIOR)
                user->doit = False;
        }
}
...
```

The SpinBox shown in Figure 6–5 has multiple children, including two decoration children. In addition, this SpinBox includes 'chaining', the process where a change in one child causes values to change in the child and on or more other children. Chaining is performed by the valueChanged callback.

The code for this SpinBox is as follows.

Figure 5 A SpinBox with Multiple, Chained Children




```
...
/*****  Create SpinBox parent  *****/
n = 0;
XtSetArg(argList[n], XmNy, nextY); n++;
XtSetArg(argList[n], XmNinitialDelay, 0); n++;
XtSetArg(argList[n], XmNrepeatDelay, 150); n++;

spin6 = XmCreateSpinBox( parent, "spin6", argList, n );


/*****  Increment Y position for next SpinBox  *****/
nextY +=  Y_OFFSET;

/*****  Create SpinBox child  *****/
n = 0;
XtSetArg(argList[n], XmNwidth, 30); n++;
XtSetArg(argList[n], XmNposition, thisMM - 1); n++;
XtSetArg(argList[n], XmNminimumValue, 1); n++;
XtSetArg(argList[n], XmNmaximumValue, 12); n++;
XtSetArg(argList[n], XmNspinBoxChildType, XmNUMERIC); n++;

spin6_text1 = XmCreateTextField( spin6, "spin6_text1",
                                 argList, n );


/*****  Create SpinBox decoration child  *****/
n = 0;
decoString = XmStringCreateLtoR("/", XmSTRING_DEFAULT_CHARSET);
XtSetArg(argList[n], XmNlabelString, decoString); n++;

spin6_deco1 = XmCreateLabel(spin6, "spin6_deco1", argList, n);

/*****  Create SpinBox child  *****/
n = 0;
XtSetArg(argList[n], XmNwidth, 30); n++;
XtSetArg(argList[n], XmNposition, thisDD - 1); n++;
XtSetArg(argList[n], XmNminimumValue, 1); n++;
XtSetArg(argList[n], XmNmaximumValue, 31); n++;
XtSetArg(argList[n], XmNspinBoxChildType, XmNUMERIC); n++;

spin6_text2 = XmCreateTextField( spin6, "spin6_text2",
                                 argList, n );

/*****  Create SpinBox decoration child  *****/
n = 0;
decoString = XmStringCreateLtoR("/", XmSTRING_DEFAULT_CHARSET);
XtSetArg(argList[n], XmNlabelString, decoString); n++;
spin6_deco2 = XmCreateLabel( spin6, "spin6_deco2", argList, n );


XmStringFree(decoString);

/*****  Create SpinBox child  *****/
n = 0;
XtSetArg(argList[n], XmNwidth, 30); n++;
```

```
XtSetArg(argList[n], XmNposition, thisYY); n++;
XtSetArg(argList[n], XmNmaximumValue, 99); n++;
XtSetArg(argList[n], XmNspinBoxChildType, XmNUMERIC); n++;


spin6_text3 = XmCreateTextField(spin6, "spin6_text3", argList, n);

/*****  Manage SpinBox  *****/
XtManageChild(spin6);

/*****  Call changedSpin6 AFTER displayed value has changed  *****/
XtAddCallback(spin6, XmNvalueChangedCallback, changedSpin6,
              (XtPointer) 0);

/*****  Manage SpinBox children  *****/
XtManageChild(spin6_text1);
XtManageChild(spin6_deco1);
XtManageChild(spin6_text2);
XtManageChild(spin6_deco2);
XtManageChild(spin6_text3);
...
```

The callback code for the previous call to **XmNvalueChangedCallback** uses **XtSetValues** calls to chain the day and the month children, and the month and the year children. It also changes **XmNmaximumValue** for the day child whenever the month child changes, as follows:

```
...
void
changedSpin6(Widget w, XtPointer client, XtPointer call)
{
XmSpinBoxCallbackStruct *user;
Cardinal                n;
Arg                     argList[5];
int                     saveMM;
int                     saveDD;
int                     saveYY;

    user = (XmSpinBoxCallbackStruct *)call;

    if (user->widget == spin6_text1)
        {
        thisMM = user->position + 1;

        if (thisDD <=3)
            setMaxDay(spin6_text2, thisMM -1);
        else if (thisDD > 27)
            setMaxDay(spin6_text2, thisMM);

        saveYY = thisYY;

        if (user->crossed_boundary)
            if (user->reason == XmCR_SPIN_NEXT)
                thisYY++;
            else
                thisYY--;

        if (thisYY != saveYY)
            {
            if (thisYY < 0)
                thisYY = 99;
            else if (thisYY > 99)
                thisYY %= 100;

            n = 0;
            XtSetArg(argList[n], XmNposition, thisYY);
            n++;
```

```
            XtSetValues(spin6_text3, argList, n);
            }
        }
    else if (user->widget == spin6_text2)
        {
        thisDD = user->position + 1;

        if (thisDD <=3 && user->reason == XmCR_SPIN_PRIOR)
            setMaxDay(spin6_text2, thisMM -1);
        else if (thisDD > 27 && user->reason == XmCR_SPIN_NEXT)
            setMaxDay(spin6_text2, thisMM);

        saveMM = thisMM;
        saveYY = thisYY;

        if (user->crossed_boundary)
            if (user->reason == XmCR_SPIN_NEXT)
                thisMM++;
            else
                thisMM--;

        if (thisMM != saveMM)
            {
            if (thisMM < 1)
                {
                thisMM = 12;
                thisYY--;
                }
            else if (thisMM > 12)
                {
                thisMM = 1;
                thisYY++;
                }

            n = 0;
            XtSetArg(argList[n], XmNposition, thisMM - 1);
            n++;

            XtSetValues(spin6_text1, argList, n);

            if (thisYY != saveYY)
                {
                n = 0;
                XtSetArg(argList[n], XmNposition, thisYY);
                n++;

                XtSetValues(spin6_text3, argList, n);
                }
            }
        }
    else if (user->widget == spin6_text3)
        {
        thisYY = user->position;

        if (user->reason == XmCR_OK)
            {
            if (thisDD <= 3)
                setMaxDay(spin6_text2, thisMM - 1);
            else if (thisDD > 27)
                setMaxDay(spin6_text2, thisMM);
            }
        }
}
```

The **XmSpinBoxValidatePosition** function is available for use by applications that need to implement a policy for tracking user modifications to editable SpinBox children of type **XmNUMERIC**. It is up to the application to specify when and how the user modifications take effect.

SimpleSpinBox

The **XmSimpleSpinBox** widget, a subclass of **XmSpinBox**, is a user interface control to increment and decrement an arbitrary TextField. For example, it can be used to cycle through the months of the year or days of the month.

The XmSimpleSpinBox widget redefines some of its XmSpinBox superclass constraint resources as normal resources for the class: **XmNdecimalPoints**, **XmNincrementValue**, **XmNmaximumValue**, **XmNminimumValue**, **XmNnumValues**, **XmNposition**, **XmNpositionType**, **XmNspinBoxChildType**, **XmNvalues**. Widget subclassing is not supported for the XmSimpleSpinBox widget class.

SimpleSpinBox provides the following functions:

- **XmCreateSimpleSpinBox** −−creates an instance of SimpleSpinBox.
- **XmSimpleSpinBoxAddItem** −−takes an **XmString** specifying the new item and assigns to the item a position in the list.
- **XmSimpleSpinBoxDeletePos** −−deletes an item by reference to its position.
- **XmSimpleSpinBoxSetItem** −−causes a specified item to be the first visible item in the list.