MWM and the ICCCM

The X Consortium Standard *Inter−Client Communication Conventions Manual* (ICCCM) defines standards by which X clients should communicate with each other. The Motif toolkit and MWM comply with ICCCM. Applications may define private protocols for communicating with other applications that share those protocols. If they do so, they should also conform to ICCCM standards.

Window Managers, ICCCM, and Shells

ICCCM defines protocols for communication between clients and window managers. Most of the communication takes place through properties on an application's top−level windows. The window manager can also generate events that are available to the application.

In Motif and Xt, shells handle most communication between an application and a window manager. An application seldom has to deal directly with properties or events. The application can usually specify properties by setting resources of a shell. Shells also select for and handle most events from the window manager.

This section discusses the relations between some shell resources, properties, and events concerned with communication between an application and any window manager. The following section discusses resources, properties, and events that apply to MWM in particular.

Application Startup

When a top−level window is mapped, the window manager may search the resource database for information about the window. The resource name and class come from the WM_CLASS property for the window. This property contains two consecutive strings that identify the instance and class names.

Xt sets the WM_CLASS property when a shell that is a subclass of **WMShell** is realized. The instance name is the name of the shell. For an ApplicationShell, this is generally the name of the application passed to **XtDisplayInitialize**. The class name is the application class from the highest−level widget in the hierarchy. For an ApplicationShell, this is generally the application class passed to **XtDisplayInitialize**. If the root widget is not an ApplicationShell, the class name is the widget's class name.

Most window managers display a name for a top−level window, often in a title bar. The window name comes from the WM_NAME property. This property is a string whose encoding is identified by the type of the property.

A Motif application specifies a window name using the WMShell resources **XmNtitle** and **XmNtitleEncoding**. If the shell is a TopLevelShell subclass and the **XmNiconName** resource is not NULL, the value of that resource is the default for **XmNtitle**. Otherwise, the default title is the name of the shell. For a dialog, an application can supply a title as the value of the BulletinBoard resource **XmNdialogTitle**.

XmNtitleEncoding is an atom representing the encoding of the name. The default title encoding depends on whether or not a language procedure has been set. If no language procedure has been set, the default is STRING. If a language procedure has been set, the title is assumed to be in the encoding of the locale and is passed to **XmbTextListToTextProperty** with an encoding style of **XStdICCTextStyle**. The returned property is used as the WM_NAME property. If the title is fully convertible to type STRING, the encoding is STRING; otherwise, the encoding is COMPOUND_TEXT.

Window Configuration

A window manager can assign any position and size to a window. The user and application can supply preferred positions and sizes, but the window manager is free to use or ignore these as it wishes.

The user generally specifies position and size using the *-geometry* option when invoking the command that starts the application. In Motif, the value specified for *-geometry* becomes the value of the Shell **XmNgeometry** resource. An application should never set this resource itself; it should reserve it for the user. An application specifies size and position by supplying values for the Core resources **XmNx**, **XmNy**, **XmNheight**, **XmNwidth**, and **XmNborderWidth**. When an x, y, width, or height value is specified for both **XmNgeometry** and one of the specific geometry resources, the value from **XmNgeometry** takes precedence.

The MWM *positionIsFrame* resource determines whether MWM interprets x and y values as referring to the upper left corner of the client window itself or the upper left corner of the frame that MWM puts around the client window. By default x and y values refer to the frame.

When a top−level window is mapped, MWM uses the following order of precedence in determining size and position:

- If the user specifies position and size using the *-geometry* option, MWM uses those values.
- If the MWM *interactivePlacement* resource is True, MWM waits for the user to select a position using a button press for the upper left corner of the window. If the user drags the pointer down and to the right with the mouse button pressed, the user can then determine the size of the window by releasing the mouse button. If the user does not determine a size in this way, MWM uses the window's **XmNwidth** and **XmNheight**.
- If the MWM *usePPosition* resource is True, or if *usePPosition* is *nonzero* and the window's **XmNx** or **XmNy** is nonzero, MWM uses the window's **XmNx** and **XmNy** to position the window. MWM uses the window's **XmNwidth** and **XmNheight** for the window's size. If the MWM *positionOnScreen* resource is True and if the window would be completely off the screen, MWM alters the window position so that at

least part of the window is on the screen.

- If the MWM *clientAutoPlace* resource is True, MWM positions the window with its top left corner offset horizontally and vertically from the last client mapped. MWM uses the window's **XmNwidth** and **XmNheight** for the window's size.
- MWM positions the window in the upper left corner of the screen and uses the window's **XmNwidth** and **XmNheight** for the window's size.

Before a window is mapped, the application communicates additional position and size information to the window manager through the WM_NORMAL_HINTS property on the window. This property is of type WM_SIZE_HINTS and contains a number of fields derived from WMShell resources:

XmNminHeight, **XmNminWidth**

Specifies the minimum height and width that the application wants the widget's window to have. If an initial value is supplied for one of these resources but not for the other, the value of the unspecified resource is set to 1 when the widget is realized. If no value is specified for either resource, MWM uses the values from **XmNbaseHeight** and **XmNbaseWidth** if specified. Otherwise, MWM uses a minimum height and width of at least 1.

XmNmaxHeight, **XmNmaxWidth**

Specifies the maximum height and width that the application wants the widget's window to have. If an initial value is supplied for one of these resources but not for the other, the value of the unspecified resource is set to 32767 when the widget is realized. If the MWM resource *maximumClientSize* is specified, MWM uses that value to determine the maximum window size. Otherwise, MWM uses the maximum height and width from the WM_NORMAL_HINTS property, except that the window size may not exceed the height and width specified by the MWM *maximumMaximumSize* resource.

XmNbaseHeight, **XmNbaseWidth**

Specifies the base for a progression of preferred heights and widths for the window manager to use in sizing the widget. The preferred heights are **XmNbaseHeight** plus integral multiples of **XmNheightInc**, with a minimum of **XmNminHeight** and a maximum of **XmNmaxHeight**. The preferred widths are **XmNbaseWidth** plus integral multiples of **XmNwidthInc**, with a minimum of **XmNminWidth** and a maximum of **XmNmaxWidth**. If an initial value is supplied for one of these resources but not for the other, the value of the unspecified resource is set to 0 when the widget is realized. If no value is specified for either resource, MWM uses the values from **XmNminHeight** and **XmNminWidth** if specified. Otherwise, MWM uses a base height and width of at least 1.

XmNheightInc, **XmNwidthInc**

Specifies the increment for a progression of preferred heights and widths for the window manager to use in sizing the widget. The preferred heights are **XmNbaseHeight** plus integral multiples of **XmNheightInc**, with a minimum of **XmNminHeight** and a maximum of **XmNmaxHeight**. The preferred widths are **XmNbaseWidth** plus integral multiples of **XmNwidthInc**, with a minimum of **XmNminWidth** and a maximum of **XmNmaxWidth**. If an initial value is supplied for one of these resources but not for the other, the value of the unspecified resource is set to 1 when the widget is realized. If no value is specified for either resource, MWM uses an increment of 1.

XmNminAspectX, **XmNminAspectY**

Specifies the numerator and denominator of the minimum aspect ratio (X/Y) that the application wants the widget's window to have. If no value is specified for either resource, MWM imposes no minimum aspect ratio.

XmNmaxAspectX, **XmNmaxAspectY**

Specifies the numerator and denominator of the maximum aspect ratio (X/Y) that the application wants the widget's window to have. If no value is specified for either resource, MWM imposes no maximum aspect ratio.

XmNwinGravity

Specifies the window gravity for use by the window manager in positioning the widget. If no initial value is specified, the value is set when the widget is realized. If **XmNgeometry** is not NULL, **XmNwinGravity** is set to the window gravity returned by **XWMGeometry**. Otherwise, **XmNwinGravity** is set to **NorthWestGravity**.

After a window is mapped, an application can request changes to window size or position by calling **XtSetValues** for one or more of the Core geometry resources. A user can generally employ window manager facilities to move or resize a top–level window.

Calling **XtSetValues** for a geometry resource generates a geometry request that may propagate up the widget hierarchy to the shell. This may cause the shell to make its own geometry request, and this invokes the shell's *root_geometry_manager* procedure. This procedure uses

XConfigureWindow to ask the window manager to change the window's size or position.

If a window manager responds to a configuration request by denying it or by moving the window without resizing it, the window manager sends a synthetic **ConfigureNotify** event. If the window is resized, the window receives a real **ConfigureNotify** event.

These events may be handled by either the *root_geometry_manager* procedure or a Shell event handler. If the VendorShell resource **XmNuseAsyncGeometry** is True, the *root_geometry_manager* procedure does not wait for the window manager to respond to the configuration request, but instead returns **XtGeometryYes**. If the WMShell resource **XmNwaitForWm** is True and if the window manager grants the configuration request within the **XmNwmTimeout** interval, the *root_geometry_manager* procedure updates the widget's geometry resources and returns **XtGeometryYes**. Otherwise, the *root_geometry_manager* procedure returns **XtGeometryNo** and relies on the event handler to reconfigure the widget when it receives a subsequent **ConfigureNotify** event.

The shell's **ConfigureNotify** event handler is invoked when the user reconfigures a top−level window or when the application reconfigures a window and this reconfiguration is not handled by the *root_geometry_manager* procedure. The event handler updates the shell's core geometry fields with the values allowed by the window manager. If the size of the shell changes, the event handler calls the shell's *resize* procedure. This procedure calls **XtResizeWidget** to change the height, width, and border width of the child to be the same as those of the shell.

Icons

An application uses several properties to communicate with the window manager about icons associated with top−level windows. A Motif application can use resources of several Shell subclasses to specify values for these properties.

When a window is first mapped, it can appear in either its normal state or iconic state. An application uses a field in the WM_HINTS property to tell the window manager which initial state it prefers. A Motif application specifies the initial state by setting the WMShell resource **XmNinitialState** or the TopLevelShell resource **XmNiconic**. **XmNiconic** takes precedence over **XmNinitialState**. After a window is realized, an application can use **XtSetValues** for **XmNiconic** to either iconify or deiconify the window.

An application can supply a name, a bitmap, or a window for the window manager to use as an icon. When a top−level window is in iconic state, the window manager usually displays the icon window if one is supplied, or else the icon pixmap if one is supplied, or else the icon name. MWM uses the *iconDecoration* resource in determining what aspects of an icon to display.

The icon name comes from the WM_ICON_NAME property. Like WM_NAME, this property is a string whose encoding is identified by the type of the property.

A Motif application specifies an icon name using the TopLevelShell resources **XmNiconName** and **XmNiconNameEncoding**. The default icon name is the name of the shell. **XmNiconNameEncoding** is an atom representing the encoding of the name. The default encoding depends on whether or not a language procedure has been set. If no language procedure has been set, the default is STRING. If a language procedure has been set, the icon name is assumed to be in the encoding of the locale and is passed to **XmbTextListToTextProperty** with an encoding style of **XStdICCTextStyle**. The returned property is used as the WM_ICON_NAME property. If the icon name is fully convertible to type STRING, the encoding is STRING; otherwise, the encoding is COMPOUND_TEXT.

An application uses fields in the WM_HINTS property to supply an icon bitmap and an optional mask for displaying the bitmap in a nonrectangular shape. A Motif application specifies an icon bitmap as the value of the WMShell resource **XmNiconPixmap**, and it specifies the mask as the value of the WMShell resource **XmNiconMask**.

An application uses a field in the WM_HINTS property to supply an icon window. A Motif application specifies an icon window as the value of the WMShell resource **XmNiconWindow**. The icon window must be an InputOutput child of the root window. It must also use the root visual and the default colormap of the screen. The application must not map, unmap, or configure this window. It must, however, select for Expose events on the window and redisplay the contents when it receives these events.

The window manager may specify preferred maximum and minimum sizes and size increments for icon bitmaps and windows. To do this it puts a WM_ICON_SIZE property on the root window. MWM uses the *iconImageMaximum* and *iconImageMinimum* resources, with increments of 1, in setting this property. Before an application specifies an icon bitmap or window, it should use the Xlib routine **XGetIconSizes** to check these constraints and then supply a bitmap or window that is of one of the preferred sizes.

An application can use two fields of the WM_HINTS property to supply preferred x and y root coordinates for the icon location. A Motif application specifies these coordinates as the values of the WMShell resources **XmNiconX** and **XmNiconY**. The window manager may ignore these values. MWM uses the *useIconBox*, *iconPlacement*, and *iconPlacementMargin* resources in determining where to place icons.

Window Groups

An application can use a field of the WM_HINTS property to supply the window ID of a window to serve as the "leader" for a group of windows. The window manager may treat all windows in this group as a whole for certain purposes, such as showing a single icon when the entire group is iconified.

A Motif application specifies a window group leader as the value of the WMShell resource **XmNwindowGroup**. For VendorShell and its subclasses, if the shell has a parent, Motif sets the **XmNwindowGroup** to the parent's window at the time that the shell and its parent are both realized. Otherwise, the default value is **XtUnspecifiedWindowGroup**, which means that no window group is set.

Menus and Dialogs

A window manager may treat dialogs differently from other top−level windows, and it must not interfere with menus at all.

An application tells a window manager not to decorate or otherwise interfere with a window by setting the *override_redirect* attribute of the window to True. A Motif application does this by setting the **Shell** resource **XmNoverrideRedirect** to True, or by using an **OverrideShell**, which has a default value of True for this resource. **XmMenuShell** is a subclass of **OverrideShell**, and MenuShells are the only widgets that should have a value of True for **XmNoverrideRedirect**. An application normally does not supply a value other than the default for this resource.

An application tells a window manager to treat a window as transient or secondary by setting the window's WM_TRANSIENT_FOR property. This property contains the window ID of another top−level window, usually the window from which the transient window was popped up. A Motif application generally specifies this property by creating a DialogShell, a subclass of TransientShell, which has an **XmNtransientFor** resource. The value is a widget, and the default is set to the shell's parent at the time that both the shell and its parent are realized. The window of the **XmNtransientFor** widget is used for the WM_TRANSIENT_FOR property. For a shell that is not a subclass of TransientShell, an application can set the WMShell **XmNtransient** resource to True. The **XmNwindowGroup** is then used for the WM_TRANSIENT_FOR property. An application normally does not supply a value other than the default for **XmNtransient** or **XmNtransientFor**.

MWM treats transient windows differently from other top−level windows. By default it keeps transient windows stacked on top of their primary windows and does not allow transient windows to be iconified separately from their primary windows. The MWM *transientDecoration* and *transientFunctions* resources determine which decorations and functions apply to transient windows. An application can further specify these decorations and functions by using the VendorShell **XmNmwmDecorations** and **XmNmwmFunctions** resources, explained in Section 18.2.

Input Focus

ICCCM recognizes four models for the relationship between clients and window managers in setting input focus:

No input

The client does not expect keyboard input and does not want the window manager to set focus to any of its windows.

Passive input

The client expects keyboard input and wants the window manager to set focus to its top−level window. It does not set focus itself.

Locally active input

The client expects keyboard input and wants the window manager to set focus to its top−level window. It may also set focus to one of its subwindows when one of its windows already has the focus. It does not set focus itself when the current focus is in a window that the client does not own.

Globally active input

The client expects keyboard input but does not want the window manager to set focus to any of its windows. Instead, it sets focus itself, even when the current focus is in a window that the client does not own.

An application tells the window which model it prefers by using two properties:

- If the *input* field of the WM_HINTS property is True, the application wants the window manager to set focus to its top−level window. If this field is False, the application does not want the window manager to set focus.
- If the WM_PROTOCOLS property contains a WM_TAKE_FOCUS atom, the application sometimes sets focus itself. If the WM_PROTOCOLS property does not contain a WM_TAKE_FOCUS atom, the application does not set focus itself.

These combinations are summarized in the following table:

Table 1 Input Models

Input Model	*Input field*	**WM_TAKE_FOCUS**
No input	False	Absent
Passive	True	Absent
Locally active	True	Present
Globally active	False	Present

A window manager generally does not set input focus to a window when the WM_HINTS *input* field is False. A window with a WM_TAKE_FOCUS protocol may receive a ClientMessage when the window manager wants the window to accept keyboard focus. The window may respond by setting the input focus or by ignoring the message.

A Motif application can set the *input* field of the WM_HINTS property by specifying a value for the WMShell resource **XmNinput**. The application can install the WM_TAKE_FOCUS atom on the WM_PROTOCOLS property by calling **XmAddWMProtocols** or **XmAddWMProtocolCallback**, explained in Section 18.3.

A Motif application normally should avoid setting input focus itself. The application can control the location of focus within its subwindows by using the VendorShell resource **XmNkeyboardFocusPolicy**, the Gadget, Primitive, and Manager resource **XmNtraversalOn**, and the **XmProcessTraversal** routine. If the application wants a widget to receive no input at all, it can use **XtSetSensitive** to make the widget insensitive. If the application needs to set focus directly, it should usually use **XtSetKeyboardFocus** and avoid using **XSetInputFocus**. For more information, see Chapter 13.

A number of MWM resources influence keyboard focus. When *keyboardFocusPolicy* is "explicit" (the default), the user must press `Btn1` on a window or its decoration to give it focus. When *keyboardFocusPolicy* is "pointer", the window that contains the pointer has the focus. With an explicit policy, other resources determine whether a window has focus when it is first mapped (*startupKeyFocus*), deiconified (*deiconifyKeyFocus*), or raised (*raiseKeyFocus*). When *autoKeyFocus* is True, and the window with focus is iconified or withdrawn, focus passes to the window that last had focus. When *enforceKeyFocus* is True, MWM sets focus to globally active windows.

Colormaps

An application can create and set colormaps for its windows, but only the window manager should install colormaps. Each window manager has a colormap focus policy that determines which top−level window has the colormap focus at a given time. When a window has colormap focus, the window manager installs one or more colormaps associated with that window.

If all windows in an application use the same colormap, the application need take no special action to tell the window manager to use that colormap. The window manager keeps track of the colormap attribute for each top−level window and installs that colormap when the window has colormap focus.

If an application uses different colormaps for some windows in its hierarchy, it must tell the window manager about those colormaps by setting a WM_COLORMAP_WINDOWS property on the top−level window. This property is a list of windows whose colormaps the window manager should install when the top−level window has colormap focus. The list should be in order of priority, with the windows whose colormaps the application would most like to have installed listed first. The application can use **XSetWMColormapWindows** to set this property.

On many servers, only one hardware colormap can be installed at a time. This may cause colors in windows that use different colormaps to be displayed incorrectly when their own colormaps are not installed. To reduce contention for colormaps, applications should use the facilities for standard colormaps described in *Xlib−−C Language X Interface*.

The MWM *colormapFocusPolicy* resource determines the colormap focus policy. When the value is "keyboard", the window with keyboard focus has the colormap focus. When the value is "pointer", the window under the pointer has the colormap focus, regardless of whether that window also has keyboard focus. When the value is "explicit", the colormap focus changes only when the user invokes the **f.focus_color** function.

When a window with colormap focus has a WM_COLORMAP_WINDOWS property, the user can install the next and previous colormaps on the list by invoking the **f.next_cmap** and **f.prev_cmap** functions.

Application Shutdown and Restart

An application may run under a session manager with facilities for saving and restoring the state of the application. An application communicates with a session manager by placing WM_COMMAND and WM_CLIENT_MACHINE properties on its top−level windows. WM_COMMAND

contains a string that would restart the client in its current state.

A Motif application should have only one non−NULL WM_COMMAND property for each logical application (that is, for each ApplicationShell hierarchy). Xt sets the WM_COMMAND property for an ApplicationShell when the shell is realized, using the command that started the application. Note that if an application is using an unrealized ApplicationShell with multiple TopLevelShell popup children, Xt will not place a WM_COMMAND property on any window, and the application must put this property on some (possibly unmapped) window in the application.

WM_CLIENT_MACHINE contains a string that represents the name of the host on which the application is running. Xt sets the WM_CLIENT_MACHINE for a WMShell or subclass when the shell is realized.

A session manager can inform an application when a top−level window is about to be deleted or when the application should try to save its state. An application expresses interest in these notifications by adding a WM_DELETE_WINDOW atom or a WM_SAVE_YOURSELF atom to the WM_PROTOCOLS property.

If a WM_DELETE_WINDOW protocol exists, the session manager sends a ClientMessage when it wants to delete a top−level window. The application may ask for user confirmation and may decide to comply or not comply with the request. If it decides to comply, the application can either unmap or destroy the window.

If a WM_SAVE_YOURSELF protocol exists, the session manager sends a ClientMessage when it wants the application to save its current state in such a way that it could be restored. The application should do whatever is necessary to save its internal state and then update the non−NULL WM_COMMAND property with a command that will restart the application in its current state. Finally, the application updates the WM_COMMAND property on the window that has the WM_SAVE_YOURSELF protocol if it has not already done so. This informs the session manager that the application has finished saving its state.

Motif installs a WM_DELETE_WINDOW protocol for VendorShell and its subclasses. It also installs a procedure to be called after any application−supplied WM_DELETE_WINDOW handlers are invoked. This procedure destroys the widget, unmaps the window, or does nothing, depending on the value of the VendorShell resource **XmNdeleteResponse**. If the procedure destroys an ApplicationShell, it then exits the application.

An application can add its own WM_DELETE_WINDOW and WM_SAVE_YOURSELF protocols by using **XmAddWMProtocols** or **XmAddWMProtocolCallback**.

When the user invokes the **f.kill** command, MWM sends a ClientMessage if an application has a WM_DELETE_WINDOW protocol and a separate ClientMessage if an application has a WM_SAVE_YOURSELF protocol. If the application has no WM_DELETE_WINDOW protocol, the **f.kill** command kills the client. In this case, if a WM_SAVE_YOURSELF protocol exists, MWM sends the ClientMessage and then waits for the time specified by the *quitTimeout* resource before killing the client.

MWM Properties and Resources

In addition to the properties and protocols described in ICCCM, Motif uses properties and protocols of its own. A Motif application usually specifies these properties using VendorShell and BulletinBoard resources.

Decorations

A Motif application expresses preferences for MWM window decorations by supplying a value for the VendorShell resource **XmNmwmDecorations**. The value is the bitwise inclusive OR of one or more flag bit constants, each of which indicates a preference for or against a particular decoration. If a value has been supplied for this resource, MWM displays only those decorations specified by both **XmNmwmDecorations** and the MWM *clientDecoration* resource (for primary windows) or specified by both **XmNmwmDecorations** and the MWM *transientDecoration* resource (for transient windows). If no value has been supplied for **XmNmwmDecorations**, MWM displays the decorations specified by the *clientDecoration* or *transientDecoration* resource.

Functions

A Motif application expresses preferences for MWM window functions by supplying a value for the VendorShell resource **XmNmwmFunctions**. The value is the bitwise inclusive OR of one or more flag bit constants, each of which indicates a preference for or against a particular function. If a value has been supplied for this resource, MWM displays only those functions specified by both **XmNmwmFunctions** and the MWM *clientFunctions* resource (for primary windows) or specified by both **XmNmwmFunctions** and the MWM *transientFunctions* resource (for transient windows). If no value has been supplied for **XmNmwmFunctions**, MWM displays the functions specified by the *clientFunctions* or *transientFunctions* resource.

BulletinBoard may change the initial value of **XmNmwmFunctions** if its parent is a subclass of VendorShell. The BulletinBoard resource **XmNnoResize** determines whether the decorations of the VendorShell parent include resize controls.

Input Mode

An application can inform MWM that it should impose constraints on which windows can obtain input. A Motif application does this by supplying a value for the VendorShell resource **XmNmwmInputMode**. For a BulletinBoard whose parent is a DialogShell, the application can set **XmNmwmInputMode** indirectly by specifying a value for the BulletinBoard resource **XmNdialogStyle**.

The possible modes are as follows:

Modeless

: Input goes to any window.

Primary application modal

: Input does not go to ancestors of this window or their descendants.

Full application modal

: Input goes to this window or its descendants and to other applications but not to other windows in this application.

System modal

: Input goes only to this window or its descendants.

Window Menu

An application can supply items for MWM to add to the end of the window menu for a window by specifying a value for the _MOTIF_WM_MENU property. A Motif application does this by supplying a value for the VendorShell resource **XmNmwmMenu**. The window menu itself is the value of the MWM *windowMenu* resource.

MWM Messages

An application can specify a message for MWM to send the application when the user invokes the **f.send_msg** function. The application places a _MOTIF_WM_MESSAGES atom on the WM_PROTOCOLS property for the window. The application also places an atom on the _MOTIF_WM_MESSAGES property. When the **f.send_msg** function is invoked with this atom as the argument, MWM sends the application a ClientMessage. The application can use **XmAddWMProtocols** to place a _MOTIF_WM_MESSAGES atom on the WM_PROTOCOLS property, and it can use **XmAddProtocolCallback** to place an atom on the _MOTIF_WM_MESSAGES property and associate it with a routine to be called when MWM sends the ClientMessage.

MWM Information

MWM maintains a _MOTIF_WM_INFO property on the root window of each screen it manages. This property is available for applications to inspect but not to change. The **XmIsMotifWMRunning** routine examines this property when determining whether or not MWM is running.

Atom and Protocol Management

Xlib provides two atom utility routines. **XInternAtom** returns an existing atom or (if the third argument is False) creates and returns an atom that matches the given string. **XGetAtomName** returns the string that matches the given atom.

Motif has a number of routines to help an application install protocol atoms and handle ClientMessages sent when the protocols are invoked. These routines maintain an internal registry of properties, protocol atoms associated with the properties, and callback routines associated with the protocol atoms. The application can use these routines with shells that are subclasses of VendorShell.

XmAddProtocols associates one or more protocol atoms with a property for a given shell. If the shell is realized, it adds those protocols to the property for the shell's window. If the shell is not realized, it arranges for the protocols to be added to the property and for a ClientMessage event handler to be added at the time the shell is realized. **XmAddWMProtocols** is a specialized version that adds protocols for the WM_PROTOCOLS property.

XmAddProtocolCallback adds a callback routine to a callback list associated with a protocol. It calls **XmAddProtocols** if the protocol has not yet been registered. When the protocol manager's ClientMessage event handler receives a ClientMessage for the protocol, it invokes the procedures on the associated callback list. The first argument to each callback procedure is the shell associated with the protocol. The second argument is the client data, if any, specified in the call to **XmAddProtocolCallback**. The third argument is a pointer to an **XmAnyCallbackStruct** structure whose *reason* member is **XmCR_PROTOCOLS** and whose *event* member is a pointer to the ClientMessage event. In the ClientMessage event, the *message_type* member is the property that contains the protocol, the *format* member is 32, and the data.l[0] member is the protocol atom. **XmAddWMProtocolCallback** is a specialized version of **XmAddProtocolCallback** that adds a callback for a protocol on the WM_PROTOCOLS property.

An application can also use **XmSetProtocolHooks** to specify a routine to be called before or after a callback list is invoked for a protocol. **XmSetWMProtocolHooks** is a specialized version that adds prehooks and posthooks for a protocol on the WM_PROTOCOLS property.

Once an application has registered a protocol and optional callback routines, it can make the protocol active or inactive. A protocol is active if it has been added to the associated property for the window. A protocol is inactive if it has been removed from the associated property. **XmActivateProtocol** makes a registered protocol active, and **XmDeactivateProtocol** makes a protocol inactive. **XmActivateWMProtocol** and **XmDeactivateWMProtocol** are specialized versions that activate or inactivate a protocol on the WM_PROTOCOLS property.

XmRemoveProtocolCallback removes a callback routine from the callback list associated with the protocol. **XmRemoveProtocols** removes one or more protocols and all callbacks associated with those protocols from the internal registry. If the shell is realized, it removes those protocols from the associated property. **XmRemoveWMProtocolCallback** and **XmRemoveWMProtocols** are specialized versions that remove callbacks or protocols for the WM_PROTOCOLS property.