A Tentative User and Reference Manual for TclMotif 1.0

Jean-Dominique Gascuel, iMAGIS/IMAG, Grenoble, France
Jean-Dominique.Gascuel@imag.fr

Jan Newmarch, University of Canberra, Australia
jan@pandonia.canberra.edu.au

February 3, 1994

Introduction

TclMotif, alias Tm, is a binding of the *Tcl* language[1] to the *OSF/Motif* widgets. *Tcl* is an interpreted language originally intended for use as a command language for other applications. It has been used for that, but has also become useful as a language in its own right.

Tcl has been extended by a set of widgets called *Tk*. The *Tk* widgets are not based on the *X toolkit* intrinsics, but are built above *Xlib*. They allow an easy way of writing *X Window* applications.

The standard set of widgets in the X world is now the *OSF/Motif* set. This forms a large set of widgets, and these have been through a large amount of development over the last five years. Use of this set is sometimes a requirement by business, and other widget sets try to conform to them in appearance and behavior. Furthermore, you are sometimes faced with toolkits that use *X toolkit*-based widgets. In this case, you have to use a *X toolkit* compatible interface builder.

Tm allows the programmer to use the *OSF/Motif* widgets instead of the *Tk* widgets from *Tcl* programs. This increases programmer choices, and allows comparison of the features of both *Tcl* and the *Tk/OSF/Motif* style of widget programming. The binding gives the usefull subset of the *OSF/Motif* widgets, accessible through the simple interpreted *Tcl* language.

Acknowledgments

Tm is based on *Tk* for the style of widget programming. This was because it provides a good model, but it also allows the *Tcl* programmer to move relatively easily between *Tk* and *OSF/Motif* programming. An alternative style of binding to *OSF/Motif* is used in the WKSH system, which performs a similar sort of role for the Korn Shell. An intermediate style is provided by the *Wafe X toolkit*-based frontend based on *Tcl*.

As I'm trying to understand Tm in deep, I started to insert my own notes in the user manual provided by Jan Newmarch. As time is going, this notes becomes more and more important, and I decided that they may end-up in a usefull user and reference manual for Tm. They are just my own interpretation of the Scriptures.

Reading this manual

The first section, *Getting Started*, might be sufficient for programmers very familiar both with *OSF/Motif* and *Tcl*. *Tcl* beginners should start by reading the Ousterout book defining Tcl 7.

The second part, starting at section 2 *Basics*, is a description of all the basics *OSF/Motif* concepts, intented for *OSF/Motif* beginners.

The last part of this manual, starting from section 6 have been written to be a full reference manual of *Tm*, with meaningfull examples, all supported resources, default values, ...

[1] For more information on *Tcl* and *Tk*, see the very neat book written by their author, (*An Introduction To Tcl and Tk*, J. Ousterout, Addison-Wesley, 1994)

Finally, the **index** page 74 should provide an extensive and easy crossreference of all supported features.

1 Getting Started

Tcl/OSF/Motif programs may be run by the *Moat* (MOtif And *Tcl*) interpreter. When called with no arguments it reads *Tcl* commands from standard input.
When called by

```
moat file-name
```

it reads *Tcl* commands from `file-name`, executes them and then enters the *Moat* event loop. This is similar to the *Tk* 'wish' and the concept was borrowed from there.

Depending on your shell interpreter, you will probably be able to run *Tcl– OSF/Motif* programs as stand alone programs. If your *Moat* interpreter is installed in `/usr/local`, make this the first line of your executable program :

```
#!/usr/local/bin/moat
```

1.1 A simple example

The following example is in the programs directory as `progEG`. The typical structure of a *OSF/Motif* program is that the top-level object is a mainWindow. This holds a menu bar, and a container object such as a form or a rowColumn which in turn holds the rest of the application objects. So a mainWindow with a list and some buttons in a form would be created by

```
xtAppInitialize -class Example

xmMainWindow .main
xmForm .main.form
xmList .main.form.list
xmPushButton .main.form.btn1
xmPushButton .main.form.btn2
```

The `xmForm` acts as what is called the "workWindow" of the mainWindow. This resource would be set by

```
.main setValues -workWindow .main.form
```

Values would also be set into the list and buttons:

```
.main.form.list setValues \
    -itemCount 3 \
    -items "one, two, three" \
    -selectionPolicy single_select
.main.form.btn1 setValues -labelString Quit
```

```
.main.form.btn2 setValues -labelString "Do nothing"
```

Geometry would be set for the form, to put the objects in their correct relation to each other. Suppose this is the list on the left, with the two buttons one under the other on the right:

```
.main.form.list setValues \
    -topAttachment attach_form \
    -leftAttachment attach_form \
    -bottomAttachment attach_form
.main.form.btn1 setValues \
    -topAttachment attach_form \
    -leftAttachment attach_widget \
    -leftWidget .main.form.list
.main.form.btn2 setValues \
    -topAttachment attach_widget \
    -topWidget .main.form.btn1 \
    -leftAttachment attach_widget \
    -leftWidget .main.form.list
```

Once evrything has been correctly setup, we can tell *OSF/Motif* to manage all the widgets, so that they will be shown on screen :

```
.main manageChild
.main.form manageChild
.main.form.list manageChild
.main.form.btn1 manageChild
.main.form.btn2 manageChild
```

The behaviour of our application would be set by callback functions :

```
.main.form.btn1 activateCallback {exit 0}
.main.form.list singleSelectionCallback {
    puts stdout "Selected %item"
}
```

And finally, windows are created and the main event loop is entered:

```
. realizeWidget
. mainLoop
```

Once entered in the main event loop, the application is really running : widgets are created, displayed, and manipulated accordingly to user events that trigger the associated callbacks.

1.2 What next ?

Thou shall read this manual !

Tm resource names stick to usual *OSF/Motif* name with a leading `-` replacing the `XmN` prefix. The *Tm* constants are specified by their *OSF/Motif* name, without the `Xm_` prefix, either in upper or lower case.

2 Basics

OSF/Motif use a hierarchy of sub-windows to create interface elements, such as menu item, push button or text entry fields. In the *X toolkit* and *OSF/Motif* jargon, they are called "widgets"[2]. Widgets are just those visual objects that can be seen on the screen, or interacted width by the mouse or keyboard. They are organized in a hierarchy, with the application itself forming the its root.

Programming a graphic user interface mainly consists of the following steps :

- Creating all the widgets you needs, in a suitable hierarchy.
- Configuring colors, sizes, alignments, fonts, ... In *OSF/Motif*, widget get their configuration options from so called resources. These resources may be set on a per widget basis or on a per widget class basis (e.g. "all push buttons should have red background"). Furthermore, *OSF/Motif* provides inheritance between widget classes (for instance, push button have a background color resource, because they inherit its existance (but not its value) from Label, which inherits it from Primitive, which inherits it from Core).

  Usually, applications provide defaults resources for widget classes, and each user modify some of them on a per session basis (file `~/.Xdefaults`).
- Programming your interface to react to user inputs : what function should be called when the save button is pushed ?

  In *OSF/Motif* jargon, you add "callbacks" to widgets. A call back is a fragment of *Tcl* code which is executed on a dedicated event (for instance, execute `{puts stdout "Hello World"}` when the mouse button 1 is released over the "push me" button).

The following sections will detail all this concepts.

2.1 Widget Names

Tcl is a string based language (the only data type is string), and widget are organized in a hierarchical structure. To accommodate this, the naming of objects within this hierarchy is similar to the "absolute path names" of Unix files with a '`.`' replacing the '`/`' of Unix. The application itself is known as '`.`'. A Form in the application may be known as '`.form1`'. A Label in this form may be '`.form1.okLabel`', and so on.

Note that *X toolkit* requires that '`.`' can only have one child (except for dialogs, which are not mapped inside their parents). This naming convention is the same as in *Tk*.

[2] Widget stands for window objects.

2.2 Widget creation commands

Widgets belong to classes, such as Label, xmPushButton or List. For each class there is a creation command which takes the pathname of the object as first argument with optional further arguments:

`creationCommand widgetName` ?managed? `resourceList`

where :

`creationCommand`
: is the class of the widget your are creating. Basically, all the *OSF/Motif* `XmCreate-SomeWidget()` calls should be binded to a `xmSomeWidget` *Moat* command. The extensive list of currently supported creation command is given below.

`widgetName`
: the full path name of the new widget. Note that this specify both the parent widget (which should already exists), and the name of the new child.

`managed`
: Before a widget can be displayed, it must be brought under the geometry control of its parent (similar to placing a *Tk* widget). This can be done by the `manageChild` widget method, or by using the `managed` argument at creation time.

  If present, this option should be the first one. A widget might be managed but unmaped, in which case it is invisible (see `-mapedWhenManaged`, page 21). The main use of "not yet managed widget" are menus (when they are not visible), and sub-widgets which will resize to an unknown dimension at the time of creation of their parents.

`resourceList`
: An optional succession of resource name/ string_value pairs.

For instance :

```
xmForm  .form1 managed
xmLabel .form1.okLabel managed
xmPushButton .form1.cancelButton managed \
    -labelString "Get rid of me"
```

creates a Form `form1` as child of '`.`', a label `okLabel` and a push button `cancelButton` as children of `form1`. The `cancelButton` has additional arguments that set the labelString to "Get rid of me".

The set of classes generally mirrors the *OSF/Motif* set. Some widgets in *OSF/Motif* and *X toolkit* are not accessible from this binding because they are intended for use in inheritance only, such as Core, Shell and Primitive.

Gadgets, a *OSF/Motif* variation of widgets, designed to cope with early very slow X window server is not supported too, because are not needed any more.

The following basic widget will be detailed in section 7 :

`xmPushButton`	a simple button,	`xmLabel`	a fixed piece of text
`xmArrowButton`	with an arrow face,	`xmTextField`	one line text editor
`xmToggleButton`	with an on/off box,	`xmText`	a full text editor
`xmDrawnButton`	with user graphics,	`xmList`	a list selector,
`xmFrame`	a 3-D border,	`xmScale`	a slider on a scale
`xmSeparator`	a simple line,	`xmScrollBar`	horizontal or vertical

Manager widgets are used to layout several widgets together. Placing widgets inside widgets enable to create hierarchies suitable for complex user interface design. Section 8 will discuss the following manager widgets :

`xmBulletinBoard`	simple x,y layout,
`xmForm`	layout widgets with realtional constraints,
`xmRowColumn`	for regular geometry management,
`xmPanedWindow`	multiple panes separated by sashes

Section 13 present special widgets to build menus. They may contain any flavor of button, separator, or other widgets, in addition to the following :

`xmMenuBar`	a row-Column used to create an horizontal menu.
`xmPulldownMenu`	a row-Column used to create a vertical menu.
`xmPopupMenu`	a menu on its own (transient) window.
`xmCascadeButton`	a special pushbutton to call a sub-menu.

OSF/Motif provides the following more complicated widgets. They are composed of several graphic entity, but nearly alway appear as a unique widget. Their *Moat* binding will be detailed in section 11

`xmScrolledWindow`	for displaying a clip view over another widget,
`xmScrolledList`	a partial view of a list,
`xmScrolledText`	a partial view of a text,
`xmMainWindow`	contains the main application windows, a menu bar, ...
`xmCommand`	a command entry area with a history list,
`xmMessageBox`	message display area on its own window,
`xmSelectionBox`	A list to select from.
`xmFileSelectionBox`	selection of a file from a list.

OSF/Motif also has convenience functions to create dialogs. These don't create ordinary widgets, but *OSF/Motif* pretends that they do. They appear in their own (transient) window, and have push buttons at the bottom line (Ok/Cancel/...). *Moat* follows this, and the following dialogs will be described in section 14.

`xmBulletinBoardDialog`	a dialog with arbitrary contents.
`xmFormDialog`	a dialog based on form
`xmMessageDialog`	a dialog showing a message
`xmInformationDialog`	a dialog displaying information
`xmPromptDialog`	a dialog with a prompt area
`xmQuestionDialog`	a dialog asking a question
`xmWarningDialog`	a dialog showing a warning
`xmWorkingDialog`	a dialog showing a busy working message
`xmSelectionBoxDialog`	a dialog based on xmSelectionBox
`xmFileSelectionDialog`	a dialog based on xmFileSelectionBox

When you have to destroy such widgets, you must destroy the real dialog widget, that is to say the parent of the usually manipulated widget :

```
xmQuestionDialog .askMe managed
[.askMe parent] destroyWidget
```

2.3 Widget methods

Creating a widget actually creates a *Tcl* command known by its path name. This command should be executed with at least one parameter to either change the behavior of the object or the value of its components, or to get information about the object. The first parameter acts like a “method” to the object, and specifies an action that it should perform.

The general syntax is :

```
targetWidgetName   widgetCommand   ?options?
```

as in the following examples :

```
.root.label manageChild
.root setValues -title "Hello world"
```

OSF/Motif uses the concept of inheritance for resources (see section 3) and translations (see section 5). *Moat* extend this to methods, which call *OSF/Motif* function on the target widget.

2.4 Widget resources

In *OSF/Motif* jargon “resources” are variables shared between the widgets and the application. Their default value enable to easily handle common look and deal across application. They are also used to communicate information between the application and the interface.

Section 3 will describe resource concepts, default value policy and types. A set of resources, common to many widget, will be described in section 6.

2.5 Widget actions and callbacks

A user interface have to react to user inputs such as a mouse click, or a key stroke. As a particular user input may takes effects both on the interface and on the application, the reactions may be of two kinds :

Actions:
are behavior internal to *OSF/Motif* that manage the interface.

Callbacks:
are defined by the application. They are used to trigger application’s responses to user input.

Each widget class may define a set of Actions and Callbacks.

Section 5 deals with actions and translations, section 4 will present the main callback concepts, and section 6 the set of actions and callbacks common to many supported *Moat* widgets.

2.6 Translations

In *OSF/Motif*, reaction to user input are defined from a high level point of view : basic actions includes arming a button, selecting a list item, setting input focus to a particular widget. On the other hand, basic events are mouse clicks, keystroke and modifier key state, etc. when the mouse is over some widget.

OSF/Motif use "translations" table to bind the later to the former.

2.7 The root widget

In earlier versions than 0.8, a specialised interpreter was used, much like *Tk*'s "wish". To conform to the new extension methods of tcl7.0, this was changed. Part of the result of this is that the *X toolkit* world has to be explicitly brought into existence. This also allows the class and fallback resources to be set, and leaves hooks for things like setting the application icon to be added later to this binding.

The world manipulation function added is :

`xtAppInitialize`
: it may take parameters of `-class` and `-fallback_resources`. If the class option is omitted, the binding will deduce a class by capitalising the first letter of the application name, and – if it was an 'x' – also capitalising the second letter.

Subsequently, a bunch a root widget methods have been added to deals with *OSF/Motif* features related only to the main application window. These are :

. `mainLoop`
  this will start the application main loop, waitting for and managing events.

. `getAppResources` *`rsrc_list`*
  Get the application resources. *`rsrc_list`* is a *Tcl* list of quadruples { *`name class default var`* }, where *`name`* is the name of the resources, *`class`* their class. For each defined resource, it search a value in the application default, or in the resource database, and set the *Tcl* variable *`var`* accordingly. If not found, it sets *`var`* to *`default`*.

. `processEvent`
  Process a single event (blocking if none are present). This is usefull only if you want to design your own main event loop.

  . `addInput` *`fileId perm tclProc`* This will add an input handler to moat. *`fileId`* may be one of `stdin`, `stdout`, `stderr`, or a valid opened file, as returned by `open`. *`perm`* is a sinle character permision, which might be `r`, `w`, or `x` respectively for read, write or exception. *`tclProc`* is the tcl code that will be executed when i/o is ready.

For instance, the followin example add an interpreter that read and execute *Moat*commands when they are typped in while the interface is running :

```
# Define the interpret function, that handle error.
proc interpret {line} {
    set code [catch $line result]
        if {$code == 1} then {
        puts stderr "$result in :\n\t$line"
    } else {
        if { $result != "" } {puts stderr $result}
    }
        puts stderr "% " nonewline
}

# Bind it as an input handler.
. addInput stdin r {
        interpret [gets stdin]
}

# And display the first prompt
puts stderr "% " nonewline
```

. `removeInput` ***inputId*** Remove the input handler specified by the given identifier. Identifier are uniq string returned by the corresponding **addInput** call.

. `addTimer` ***interval tclProc*** Add a timer that will trigger ther execution of the given *Tcl* after the specified interval.

. `removeTimer` ***timerId*** Remove the timer specified by the given identifier. Timer identifier are uniq string returned by the corresponding call to `addTimer`.

3 Resources

Resources are inherited through the class hierarchy, they have default values, and several different types. In *OSF/Motif*, several base classes exist, from which actual widgets are derived. Those classes define a common set of resources, methods and behaviors.

3.1 Resource inheritance

Each widget belongs to a class, whose name is the widget creation command name. Each widget inherits resources from its super-class. For example, `xmLabel` is a subclass of `Primitive` which in turn is a subclass of `Core`. From `Core` it inherits resources such as `-background`, `-height` and `-width`. From `Primitive` it inherits resources such as `-foreground`. It is necessary to look at these super-classes to have the full resource list of a `xmLabel` instance. In addition, each class adds extra resources. For example, `xmLabel` has the additional resources `-labelType`, `-labelPixmap` and `-labelString`, among others.

Some special resource values are inherited through multiple level of the widget hierarchy at creation time. For instance, the `-buttonFontList` of a bulletin board might be inherited from the `-defaultFontList` of an ancestor sub-classing the abstracts classes `vendorShell` or `menuShell`. In this case, the resource value is copied and won't be modified if the original resource is modified.

For instance, in the following example, the button inherits its `-fontList` default value from bulletin board `-buttonFontList`, on the other hand, the button's background color is taken from the class defaults, not from the BulletinBoard. Pushing the button will change the BulletinBoard's `-buttonFontList` resource, which will not update the button's font list.

```
#! moat

xtAppInitialize

xmBulletinBoard .top managed \
    -background #A22 \
    -buttonFontList "-*-courier-*-o-*--20-*"
xmPushButton .top.bold managed \
    -y 10 \
    -labelString Bold
xmPushButton .top.quit  managed \
    -y 40 \
    -labelString Quit

.top.bold activateCallback {
    .top setValues \
        -buttonFontList "-*-courier-bold-o-*--20-*"
}
.top.quit  activateCallback {exit 0}

. realizeWidget
. mainLoop
```

3.2 X Defaults

The usual X Defaults mechanism is used to provide defaults to resources.

Default values are looked in files desiganted by the **XAPPDEFAULTS** environemnent variable, with eventually a locallization directory (desiganted by the **LANG** variable). **XAPPDEFAULTS** defaults to `/usr/lib/X11/app-defaults`, and **LANG** is usually not defined. In this simplest case, the looked file is `/usr/lib/X11/app-defaults/`***`ApplicationName`***, where ***`ApplicationName`*** is the class name of your application (see `xtAppInitialize`).

This might be overiden by `xrdb`(1). Usually, login scripts read a user-customized resource file, often named `.XDefaults`, or `.Xresources` using `xrdb -merge`. It is the usual way a user configure its environement.

Last, some application use special configuration files, that might also contains some resources (`mwm`(1) is a good exemple of this quite complex area, has is looks in not less than eight different resource files...)

Those resources files contain line specifying default resource value for widget or widget class resources. The syntax is :

`resourcePath   :   value`

where ***`value`*** is the string representation for the resource, and ***`resourcePath`*** a dot-separated path naming a particular resource.

Resource paths start with an optional application name. Without it, the default apply to all application. The following names in the path may refer to widget class (when starting with an upper case), to widget names (as defined by *Moat* creation command), or to application specific scoping. The `*` character may be used to match any portion of the resource path.

This following examples should clarify this :

`*Background`	for all widgets, in all sessions.
`*PushButton.Background`	for all the push button instances.
`xterm*Background`	for all widgets of the `xterm` application.
`jot.fileMenu.quit.Background`	for the Quit button in the fileMenu of jot.

3.3 Resource types

Some resource are just string values (as `-labelString`), but other one have more complicated types (as colors). As *Moat* is a string language, all values should be manipulated in string representations, and *Moat* uses either *OSF/Motif* internal either specific converters to make the necessary conversions.

This section will briefly describes the main types used by *Tm* and *Moat*.

Basic types : Integer, Boolean and String

In *Tcl*, evry variable's value is a character string. Nevertheless, some string have a meaning as an integer, or as a boolean. In *Tm*, a ***`String`*** could be any *Tcl* string or list, correctrly surounded by braces or double-quotes. An ***`Integer`*** is a particular string, with only decima digits in it. A ***`Boolean`*** is either one of the following words `true`, `yes`, `on`, `false`, `no`, `off` (in upper/lower/mixed cases), either an integer (0 means `False`).

Dimension

`Dimension` are particular `Integer` measuring distance in screen space. Their actual value depends on an `-units` resource, which might define something different horizontaly and verticaly (when based on current font metrics for instance).

For instance, the following size set a window size to 80x24 characteres :

```
$window setValues \
    -units 100th_font_units \
    -width 8000 -height 2400
```

Color resources

In *the X Window system*, colors may be specified using portable symbolic names (such as `NavyBlue`) defined in the `/usr/lib/X11/rgb.txt` file, or RGB hexadecimal triplets of the form `#RGB`, (with `R`, `G` and `B` being one to four hexa digits), such as `#081080` (a dark blue, defined with 8 bits by channel).

Depending of your visual type, *the X Window system* may alway provide you the exact color you specified, or give you an hopefully close approximation. RGB values are not portable, because they depend on the screen hardware gamma, the software contrast correction, and the graphic board linearity. The `rgb.txt` file should be tuned for each hardware/software configuration (by your vendor), which is rarely well done.

Font resources

Font names used by X11R4 are fully qualifying dash-separated strings, or aliased nick-names. The general form of the full name is :

`-maker-name-weight-slant-width-serif-11-80-100-100-m-60-encoding`

With :

`maker`: The font maker, such as `adobe`, `bitstream`, or `sgi`.

`name`: The name for this font, as defined by the maker. Adobe's fonts includes `Helvetica`, `Zapf Chancery`, ...

`weight`: One of `bold`, `medium`, `normal`, `demi`, or `light`.

`slant`: one of `o`(blic), `r`(oman).

`width`: width of the characters, one of `normal`, `narrow`, ...

`serif`: nothing, or `sans`.

sizes font size, in various units.

`encoding`: Usually `iso8859-1`.

The `*` character might be used to match any of the font specifier.

On unix machines, the files `/usr/lib/X11/fonts/*/fonts.dir` lists all existing fonts of the actual X server.

Font list

Font lists are coma separated list of fonts. The first font in the list is the default one, the other ones are used in coupound string. This is quite useless by now, because there is no consensus on how to get multi-font `XmStrings`, and none of the various proposition is currently implemented in *Tm*.

Widget default font list usually derives from one of their ancestor. Default for top-level shell are set from the `VendorShell` abstract class, of from the *X defaults* mechanism.

Pixmap resources

Pixmaps are small rectangular arrays of pixels, used to be drown as button, or to be tiled to fill areas.

On color display, pixmaps may be either bi-color, using the `-background` and `-foreground` resources, either full color. Pixmaps may also be partially transparent, when they are accompanied by a transparency mask.

Simple bi-color pixmaps are created from a bitmap, using the current foreground and background colors at the time they are first loaded. Once created, the colored pixmap will be retained in the server's memory by a caching mechanism. At least on some X servers, this coloring will then be retained until the *X Window* is restarted. The `bitmap` unix command may be used to create or modify bitmaps. See figure 1 for an example of pixmap used to fill a button label.

(screen display)

```
#! moat

xtAppInitialize

xmPushButton .face managed \
    -labelType pixmap -labelPixmap face \
    -armPixmap face_no

.face activateCallback {exit 0}

. realizeWidget
. mainLoop
```

(the corresponding *Moat* script)

Figure 1: Example of Pixmap button.

Enumerated resources

For some resources, the value is given by a symbolic name, which may be chosen only from a small set of legal values. *Tm* uses the *OSF/Motif* standard name, without the leading `XmN` prefix, in a free upper/lower case combination for `setValues`. *Tm* will always return lower case string on `getValues`.

4 Callbacks

When the user does things to a widget, it may cause the widget to take certain actions. For example, when a button is pressed it changes appearance to look pressed in. Some of these actions can have *Tcl* code attached to them, so that the *Tcl* code is evaluated when the action is performed. The *Tcl* code is said to be attached to a "callback" by a widget command. For example, a push button has an `activateCallback` that is called when the user presses and releases the left mouse button inside the widget; it has an `armCallback` that is called when the user presses the mouse button; it has a `disarmCallback` that is called when the user releases the mouse button inside the widget.

Tcl code is attached to a callback by giving it as the second argument to the appropriate widget method. For example,

```
$btn      armCallback {puts "Stop squashing me!!"}
$btn   disarmCallback {puts "Ah... that's better"}
$btn activateCallback {puts "Sorry Dave"; exit 0}
```

The names of the callbacks available for a particular widget are derived from the resource documentation for *OSF/Motif*. Each callback ends with the string "`Callback`" in its name. Drop the `XmN` from the Motif description to gain the widget command. Callbacks are treated differently to other resources because the *X toolkit* treats them differently – the resource is not meant to be handled directly by any ordinary application.

For each *Tm* class, a short table will list the callbacks names, and the action that fire them.

4.1 Callback substitution

When *OSF/Motif* execute a callback, in reaction to some event, it provides it some parameters (such as the current widget) or additional data revelant to a given class.

Tm follows *Tk* in providing the powerful mechanism of callback substitution.

Before execution, the *Tcl*command list is scanned to look for `%` character. Each time on is found, the word that follows is extracted, analyzed, and if recognized, it is substituted with the corresponding data.

For example, `%item` in a `xmList` callback will be replaced by the item selected, and `%item_position` will be replaced by its position in the list. An example of use of callback substitution in a list is :

```
.list singleSelectionCallback {
    print_info %item %item_position
}
```

```
proc print_info {item position} {
    puts stdout "item was $item, at position $position"
}
```

The table below gives the recognized tags. Their meaning will be detailed in the context of the corresponding callbacks.

%click_count	%endPos	%newinsert	%selection_type
%closure	%item_length	%pattern_length	%set
%currInsert	%item_position	%pattern_length	%startPos
%currInsert	%item	%pattern	%type
%dir_length	%length	%pattern	%value_length
%dir	%mask_length	%ptr	%value
%doit	%mask	%reason	%w
%dragContext	%newInsert	%selected_items	

`%reason` should be implemented in the version 0.9, which is substitued by the reason why the callback was called.

The possible values, as defined in `Xm/Xm.h`, with the leading `XmCR_` stripped, are listed in the following table :

activate	apply	arm
browse_select	cancel	cascading
clipboard_data_delete	clipboard_data_request	command_changed
command_entered	create	decrement
default_action	disarm	drag
execute	expose	extended_select
focus	gain_primary	help
increment	input	lose_primary
losing_focus	map	modifying_text_value
moving_insert_cursor	multiple_select	no_match
none	obscured_traversal	ok
page_decrement	page_increment	protocols
resize	single_select	tear_off_activate
tear_off_deactivate	to_bottom	to_top
unmap	value_changed	

4.2 Callback cross references

The following table list all callbacks supported by *Tm* (the full method name to add the callback code is obtained by appending `Callback` ; they are listed in `<Xm/Xm.h>`, with a `XmN` prefix), and the class in which they are first defined :

Name	Defined by	Name	Defined by
`activate`	Text/Button	`losePrimary`	Text
`apply`	SelectionBox	`losingFocus`	Text
`arm`	Button	`map`	BulletinBoard
`browseSelection`	List	`modifyVerify`	Text
`cancel`	SelectionBox	`motionVerify`	Text
`cascading`	CascadeButton	`multipleSelection`	List
`commandChanged`	Command	`noMatch`	SelectionBox
`commandEntered`	Command		
`decrement`	ScrollBar	`ok`	SelectionBox
`defaultAction`	List	`pageDecrement`	ScrollBar
`destroy`	Core	`pageIncrement`	ScrollBar
`disarm`	Button	`popdown`	Shell
`drag`	Scale	`popup`	Shell
`entry`	RowColumn	`resize`	Draw.
`expose`	Draw.	`simple`	?
`extendedSelection`	List	`singleSelection`	List
`focus`	BulletinBoard	`toBottom`	ScrollBar
`gainPrimary`	Text	`toPosition`	(Text)
`help`	Mgr./Prim.	`toTop`	ScrollBar
`increment`	Scrollbar	`unmap`	BulletinBoard
`input`	DrawingArea	`valueChanged`	Text/Scale/ScrollBar

5 Actions and Translations

Actions and translations are *X toolkit* concepts that exists in *Tm* too. Each possible user input have a symbolic name, and they are called "events". Each reaction of the interface to some event also have a name, they are called the actions.

Widgets may have behaviours, which are table that say what action to fire when event arises. They are called the translations tables. *OSF/Motif* applications have translation tables that enable to use the keyboard to navigate between widgets, and to select them. This gives keyboard equivalent to mouse actions.

The translation tables are inherited through the class hierarchy. The list of all supported events and actions is quite long. Look in a *OSF/Motif* book to find about it...

5.1 Adding Actions and Translations

Actions may be added to a widget in a similar way to the C version. In that you define an action in a translation table which is set in the widget. In this binding, the *Tcl* code is placed as the arguments to the action in the translation table. Registering the translation using the `action` *Tm* action links a generic action handler which in turn will handle the *Tcl* code. Here is what it looks like to add translation to make an arrow turn left or right when 'l' or 'r' is pressed:

```
xmArrowButton .arrow managed
.arrow setValues -translations \
    {<Key>r: action(arrow_direction %w arrow_right)
     <Key>l: action(arrow_direction %w arrow_left)}
```

```
proc arrow_direction {arrow direction} {
    puts stdout "Changing direction to $direction"
    $arrow setValues -arrowDirection $direction
}
```

As with callbacks, they are supported substitutions. In the current versions, the only one is `%w` which is substitued with the current widget path (Other substitutions just return the `ERROR!!` magic string).

5.2 Trigering Actions

The method `callActionProc` is available for every widget. The purpose of this is to allow regresion tests to be performed. This takes an action as further parameter, using the usual *X toolkit* syntax. For example, to simulate the return key press occurring within an arrow button, call the `ArmAndActivate()` action:

```
.arrow callActionProc ArmAndActivate()
```

This sends (by default) a ClientMessage event to the widget. Most widgets ignore the event for most events, so this is sufficient. Some actions require event detail, though. For example, when a mouse button release occurs, the widget checks to see if the release occurred *inside* or *outside* the widget. It does this because if the event occurs inside, then the callbacks attached to the `Activate()` action are invoked, but otherwise they are not. To handle this, an event of type ButtonPress, ButtonRelease, KeyPress or KeyRelease can be prepared with some fields set. For example, a ButtonRelease occurring within the arrow can be sent by.

```
.arrow callActionProc Activate() \
    -type ButtonPress \
    -x 0 -y 0
```

Some of the Text manipulation actions require a KeyPress event, such as `self-insert()`, which inserts the character pressed. The character is actually encoded as a keycode, which is a hardware dependant code, too low-level for this binding. To prepare such an event, this toolkit uses *keysyms* which are abstractions for each type of key symbol. The alphanumerics have simple representations as themselves (‘`a`’, ‘`A`’, ‘`2`’, etc). Others have symbolic names (‘`space`’, ‘`Tab`’, ‘`BackSpace`’, etc). These are derived from the *X Window* Reference manual or in the file `<X11/keysymdefs.h>` by removing the prefix `XK_`.

For example, to insert the three characters ‘`A a`’ into `.text` :

```
.text callActionProc self-insert() \
    -type KeyPress \
    -keysym A
.text callActionProc self-insert() \
    -type KeyPress \
    -keysym space
```

```
.text callActionProc self-insert() \
    -type KeyPress \
    -keysym a
```

The set of actions that require this level of preparation of the X event is nowhere documented explicitly. You have to read between the lines of the Motif documentation, or guess at behaviour (or read Motif source code).

6 Base classes

All *Tm* widgets derive from a small set of classes, namely `Core`, `Primitive`, `Manager` and `Shell`. You cannot create any widget of those classes, because they are abstract base classes. They are used to define sets of resources, behaviors and methods common to all the derived widget classes which have binding in *Tm*. This section will describe this abstract classes.

6.1 The Core Class

The Core class is the ancestor of all *Tm* widget classes. Hence methods and resources defined in this section equally apply to all *Tm* objects. The Core class does not implement any behavior (neither action, translation nor callback), and do even not suppose that something should be drawn.

6.1.1 Core Methods

The Core class defines the basic set of methods common to all derived classes, described below :

w `realizeWidget`
: Create windows for the widget and its children, usually used only on the main widget, as in "`. realizeWidget`."

w `destroyWidget`
: Destroy the widget *w*, all sub-widgets and the associated *Tcl* commands. Note that destroying the main window (`. destroyWidget`) should gracefully exit the main loop, while `exit 0` should exit the *Tcl* interpreter.

w `mapWidget`
: Map the given widget onto screen, to make it visible. This is autmatically done when the widget is managed (see below).

w `unmapWidget`
: Unmap the widget from its parent's screen, making it invisible, but it stay in geometry management.

w `manageChild`
: Bring the widget (back) under geometry management and make it appear (again). This equivalent to the `managed` parameter when the widget is created. Some widget cannot be managed at creation time, for instance when its parent needs special setting in order to handle it properly. Another example is menus and dialogs : you might want to create them at the application initialisation, but it is not a good idea to display them permanently.

w `unmanageChild`
: Un-managing a widget un-map it from screen, making it invisible, and removes it from geometry management of the parent.

`w setSensitive` *Boolean*

An insensitive widget do not respond to user input. When such a widget is disabled (`w setSensitive false`), it is usually drawn dimmed (using a pattern). The main use is to disable buttons or menu items that are not allowed in the current state of the application.

`w setValues` *rsrc value* ...

This command is used to change resource values for an already existing widget. The required parameters are a list of pairs of resource name and string value. The following change the text colors of widget `.frm.text` :

```
.frm.text setValues \
    -background lightGray \
    -foreground #111
```

Each widget class define which resource may be set, their types and accepted values. Resource will be described in general in section 3, and with each widget description.

`w getValues` *rsrc variable* ...

This is the dual command : given a parameter list of pairs of *Tm* resource names and *Tcl* names, it set each variable to the current value of the corresponding resource. *OSF/Motif* reverse conversions are used for this purpose, and *Tm* does not actually provide all of them. This means that you should be able to setup all resource types, but may not be able to retrieve all of them.

```
proc flash {widget {fg black} {bg red}} {
     $widget getValues \
        -background old_bg  -foreground old_fg
     $widget setValues \
        -background $bg     -foreground $fg
     wait 0.1
     $widget setValues \
        -background $old_bg -foreground $old_fg
}
```

w resources Returns the list of all the active resources of the given widget. For each resource, a quadruple

{*name Class type value*}

is returned.

w `anyCallback` *tclProc* If the widget method name contain the substring "Callback", then *Tm* ask *OSF/Motif* to register the command list given in argument. When th especified event occures, it will be interpreted (in the global context). Section 4 will discusses callbacks in general.

w `parent` The `parent` widget command is used to get the parent widget name : if a regular widget `.a.b.c` have been created, then `set x [.a.b.c parent]` should set the string `".a.b"` to the variable `x`. The exact result is not always obvious, because some widgets use hiden parents, as in dialogs.

w `processTraversal` *direction* Change the widget that receive the keyboard input focus. *direction* may be one of :

`current`

`home`

`up`

`down`

`left`

`right`

`next`

`next_tab_group`

`previous_tab_group`

w `dragStart` *rsrc value* ...

w `dropSiteRegister` *rsrc value* ...
See the *Drad and Drop* section (page 58) for details about this mehods.

w `getGC` *rsrc value* ...
This method is used to retrieve the *Xlib* graphical context of a widget. There must be at leat one resource defined.

The allowed resources are `-background` and `-foreground`. See section on drawn widget (page 63) for information about user defined graphics in *Tm* widgets.

w `callActionProc`
Call an action procedure, usually used to test *Moat*, or your own code.

6.1.2 Core resources

`Core` resource name	default value	type or legal values
`-accelerators`	*none*	*String*
`-background`	*dynamic*	*Color*
`-backgroundPixmap`	*none*	*Pixmap*
`-borderColor`	*dynamic*	*Color*
`-borderWidth`	1	*Integer*
`-heigth`	*dynamic*	*Integer*
`-mappedWhenManaged`	True	*Boolean*
`-sensitive`	True	*Boolean*
`-translations`	*none*	*String*
`-width`	*dynamic*	*Integer*
`-x`	0	*Integer*
`-y`	0	*Integer*

The table describes resource common to all widgets. A `Core` widget (i.e. any widget) basically is some empty rectangle, with an optional border.

6.2 The Primitive class

The `Primitive` class derives from the `Core` class. This abstract class is designed to define resouces and behaviour common to any widget that may have something drawn on it. As the user sees something, `Primitive` is able to define some very general behaviour, which appear as translations, actions and callbacks.

6.2.1 Primitive resources

The table below describes the resources revelant for all widget deriving from `Primitive`.

`Primitive` resource name	default value	type or legal values
`-bottomShadowColor`	*dynamic*	*Color*
`-bottomShadowPixmap`	*none*	*Pixmap*
`-foreground`	*dynamic*	*Color*
`-highlightColor`	*none*	*Color*
`-highlightOnEnter`	False	*Boolean*
`-highlightThickness`	2	*Integer*
`-navigationType`	none	`none`
		`tab_group`
		`sticky_tab_group`
		`exclusive_tab_group`
`-shadowThickness`	2	*Integer*
`-topShadowColor`	*dynamic*	*Color*
`-topShadowPixmap`	*none*	*Pixmap*
`-traversalOn`	True	*Boolean*
`-unitType`	pixels	`pixels`
		`100th_millimeters`
		`1000th_inches`
		`100th_points`
		`100th_font_units`

Simple bi-color drawing are done using Primitive's foreground color, over the Core's background. Other colors default to mixing of this two ones, at the time the widget is created.

Primitive objects might be highlighted when they are "entered" (get the input focus), by drawing a border around them, of a given color.

They can also be enclosed by a beweled shadow frame, to make them appear standing in or out (so called "3D shapes").

Using the `-unitType` resource, one might choose between screen dependend units (the default), font related units, or device indepedant units. This will affect any subsequent dimensions resources for that widget only.

`-navigationType` refer to the way keyboard may be used to navigate between widgets, without using the mouse. This is used by managers to quickly navigating between input fields, for instance using the `<Tab>` key.

6.2.2 Primitive callbacks

Method name	Why
`helpCallback`	The help key is pressed.
`destroyCallback`	Widget is destroyed.

The table above give the only two callbacks defined for evry drawable widgets, for which the only supported substitution is `%w` that expand to the widget path.

`destroyCallback` may be used to automaticaly call some cleanup procedure when a widget is deleted.

When the `Help()` action arised (either through the `KHelp` key, either by a virtual binding), *OSF/Motif* looks for a callback to execute in the current widget. If none is found, it look in the parent, the parent's parent, and so one up to the main window. Hence, the `helpCallback` may be used to implement a general or a context sensitive help facility.

6.2.3 Primitive actions

As for any widgets, there is action that match each callback. This actions trigger the callbacks execution and the standard widget responses, if any.

For the `Primitive` class, they are :

`Help()`
: If there is no callback defined for this widget, this action will propagate the help action to the widget's parent. If no callback are defined upto the root widget, the action will simply be forgeted.

`Destroy()`
: The callbacks will be called before destroying a widget, to enable application specific cleanup to take place automatically when a widget is destroyed.

6.2.4 Primitive translations

The only translation defined for the `Primitive` class is :

```
<KHelp>:  Help()
```

which means that the symbolic key `KHelp` will trigger the `Help()` action. This key is defined in a `keysym` file used by the *X Window* server.

6.3 Shell classes

The *Tm* Shell classes are used to define resources and behaviours that are common to evry widgets that use theyre own window, such as top level windows, popup menues, and dialogs.

OSF/Motif describe several different base class for this purpose, some inhetited from *X toolkit*, some defined inside *OSF/Motif* :

`Shell`
: The basic shell, ancestor of all other abstract shell classes.

`TopLevelShell`
: Top level windows are responsibles of iconization.

`TransientShell`
: Transient windows are temporary windows, that should not stay visible on screen, and should be iconized along with the top level they are transient for.

`VendorShell`
: Vendor shell resources are setup in the *X* serveur, and contain meaningfull defaults for a particular implementation.

`WMShell`
: Handle protocols to assure communications between the application and the window manager.

The tables below display the resources avalaible for all those shells.

`ApplicationShell` resource name	default value	type or legal values
`-argc`	Set by `XtInitialize()`	*`Integer`*
`-argv`	Set by `XtInitialize()`	*`String Array`*

`TopLevelShell` resource name	default value	type or legal values
`-iconic`	`False`	*`Boolean`*
`-iconName`	`""`	*`String`*
`-iconNameEncoding`	`xa_string`	`compound_text`
		`xa_string`

`TransientShell` resource name	default value	type or legal values
`-transientFor`	*`none`*	*`Widget`*

`VendorShell` resource name	default value	type or legal values
-defaultFontList	*dynamic*	*font list*
-deleteResponse	destroy	do_nothing unmap destroy
-keyboardFocusPolicy	explicit	explicit pointer
-mwmDecorations	-1	*Integer*
-mwmFunctions	-1	*Integer*
-mwmInputMode	-1	*Integer*
-mwmMenu	""	*String*
-shellUnitType	pixels	pixels 100th_milimeters 1000th_inches 100th_points 100th_font_units
-useAsyncGeometry	False	*Boolean*

`WMShell` resource name	default value	type or legal values
-baseHeight	*none*	*Integer*
-baseWidth	*none*	*Integer*
-heightInc	*none*	*Integer*
-iconMask	*none*	*Pixmap*
-iconPixmap	*none*	*Pixmap*
-iconWindow	*none*	*Window*
-iconX	-1	*Integer*
-iconY	-1	*Integer*
-initialState	normalState	iconicState normalState
-input	False	*Boolean*
-maxAspectX	*none*	*Integer*
-maxAspectY	*none*	*Integer*
-maxHeight	*none*	*Integer*
-maxWidth	*none*	*Integer*
-minAspectX	*none*	*Integer*
-minAspectY	*none*	*Integer*
-minHeight	*none*	*Integer*
-minWidth	*none*	*Integer*
-title	argv[0]	*String*
-titleEncoding	xa_string	compound_text xa_string
-transient	False	*Boolean*
-waitForWm	True	*Boolean*
-widthInc	*none*	*Integer*
-windowGroup		*Window*
-winGravity	*dynamic*	*Integer*
-wmTimeout	5000ms	*Integer*

`Shell` resource name	default value	type or legal values
`-allowShellResize`	`False`	*`Boolean`*
`-geometry`	`""`	*`String`*
`-overrideRedirect`	`False`	*`Boolean`*
`-saveUnder`	`False`	*`Boolean`*
`-visual`	Inherited	*`String`*

Window resizing constraints may be set on its dimensions of the window, or on its aspect (ratio between width and height). Beside minimal and maximal dimensions, window dimension may be constrained to follow a given increment. For instance, using the following setting, the only width allowed for interactive resizing will be 150 and 250 :

```
-minWidth 100 -baseWidth 50 -widthInc 100 -maxWidth 300
```

Window aspects are set using a numerator/denominator formula :

$$\frac{minAspectX}{minAspectY} \leq \frac{\text{width}}{\text{height}} \leq \frac{minAspectX}{minAspectY} \tag{1}$$

Hence, the following setting constrains the width to stay between a third and twice the height :

```
-minAspectX 1 -manAspectY 3 -maxAspectX 2 -minAspectY 1
```

Interactive window resizing may also be ignored by setting the `-allowShellResize` resource to `False`.

Window icon resouces may be used to define the window icon type, its placement, ...

Icons may be drawn using a (possibly partially transparent) pixmap, or by using a specific alternate window (`-iconWindow`). A window may be setup to appear in iconic state at creation (`-initialState iconicState`), and its current state may be retrieved or changed using the `-iconic` resource.

7 Basic widgets

This section will detail the basic *OSF/Motif* widgets, from which all the more sofisticated one derives.

7.1 `xmLabel`

A label widget is just a small written piece of text. For instance, executing the following *Moat* script

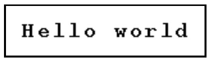
```
#! moat

xtAppInitialize

xmLabel .lbl managed -labelString "Hello world"

. realizeWidget
. mainLoop
```

would display the following window on your screen :



Note that the text will be broken into seperate lines only if you put newlines in it. It may contains non-ascii characteres (using the encoding defined in the font, usually `ISO8859-1`). See figure 2 for a more complexe example.

Resources

(screen display)

```
#! moat

xtAppInitialize

xmLabel .lbl managed
.lbl setValues -labelString {
    If you text contains newlines,
    it will be broken into separate lines.
    it may contains non-ascii characteres(àçèìñòœøù).
}

.lbl setValues \
    -stringDirection string_direction_r_to_l \
    -alignment alignment_end \
    -fontList -*-courier-bold-r-*--18-* \
    -marginLeft 10 -marginWidth 10 \
    -x 200 -y 100

. realizeWidget
. mainLoop
```

(the corresponding *Moat* script)

Figure 2: A more complex label example.

xmLabel resource name	default value	type or legal values
-accelerator	""	*String*
-acceleratorText	""	*String*
-alignment	center	alignment_center
		alignment_beginning
		alignment_end
-fontList	inherited	*fontList*
-labelInsensitivePixmap	*none*	*Pixmap*
-labelPixmap	*none*	*Pixmap*
-labelString	widget name	*String*
-labelType	string	string
		pixmap
-marginBottom	0	*Integer*
-marginHeight	0	*Integer*
-marginLeft	0	*Integer*
-marginRight	0	*Integer*
-marginTop	0	*Integer*
-marginWidth	0	*Integer*
-mnemonic	""	*String*
-mnemonicCharSet	dynamic	*String*
-recomputeSize	True	*Boolean*
-stringDirection	l_to_r	string_direction_l_to_r
		string_direction_r_to_l

The label may display the `-labelString` or `-labelPixmap` resource, depending of the `-labelType` value. Labels are always top/bottom centered (inside their margins), but may be left or right flushed or centered, depending on `-alignment`.

When a label is insensitive, the displayed text is grayed using a 50% pattern. Pixmap type labels may also be defined to display a different pixmap using `-labelInsensitive-Pixmap`.

When the displayed material changes, the label may or may not recompute its size, depending of `-recomputeSize`.

Some resources are only used in derived class.

The following resources are inherited from the Primitive (page 23), and Core classes (page 21) :

`-accelerators`	(Core)	`-backgroundPixmap`	(Core)
`-background`	(Core)	`-borderColor`	(Core)
`-borderWidth`	(Core)	`-bottomShadowColor`	(Primitive)
`-bottomShadowPixmap`	(Primitive)	`-foreground`	(Primitive)
`-heigth`	(Core)	`-highlightColor`	(Primitive)
`-highlightOnEnter`	(Primitive)	`-highlightPixmap`	(Primitive)
`-highlightThickness`	(Primitive)	`-mappedWhenManaged`	(Core)
`-navigationType`	(Primitive)	`-sensitive`	(Core)
`-shadowThickness`	(Primitive)	`-topShadowColor`	(Primitive)
`-topShadowPixmap`	(Primitive)	`-translations`	(Core)
`-traversalOn`	(Primitive)	`-unitType`	(Primitive)
`-width`	(Core)	`-x`	(Core)
`-y`	(Core)		

Callbacks

Label do not define specific callbacks, but just inherit them from the Primitive class, namely `helpCallback` and `destroyCallback`.

7.2 xmText, xmScrolledText and xmTextField

Text widgets display a text string, but also allow the user to edit it. A **xmTextField** widget display a single-line editable text, while a **xmText** widget usually span multiple lines. A **xmScrolledText** would automatically displays scroll bars if it is larger than the alloted space on screen. Those **xmScrollBar**s enable the user to change the currently viewed part of the text.

Selection of parts of text are done by keyboard or mouse interactions, as described below in the translations.

A scrolled text widget *w* is a composite widget that have the following childrens :

w.HorScrollBar *w*.VertScrollBar *w*.ClipWindow

The associated *Tcl*proc might be used to directly access them, as in the following example :

```
xmScrolledText .txt managed
set rsrc_list [.txt.ClipWindow resources]
```

Methods

In addition to the standard `Core` methods, texts widgets defined the following new ones to deal with selection and clipboard :

txt setString *the_text*
: Change the current text to *the_text*.

txt getString
: return the whole text as result.

txt getSubString *start len var*
: Set the *Tcl*variable *var* to the substring starting at position *start* for *len* characteres. If *len* if larger than some internal threshold, only the first part of the text will be set to *var*. This method returns either `succeded` , `truncated` or `failed` .

txt insert *position string*
: Insert *string* in the text, starting at position *position*. Use zero to insert at the beginning of the text.

txt replace *start stop string*
: Replace the portion of text between *start* and *stop* by the new value *string*.

txt setSelection *start stop*
: Set the current selection to the substring starting at *start*, end ending at *stop*.

txt getSelection
: Returns the primary selection of the text. If nothing is selected, just returns nothing.

`txt getSelectionPosition` *start stop*
: If there is something selected, set the *Tcl*variables *start* and *stop* accordingly and returns `true`, else returns `false`.

`txt clearSelection`
: deselect the current selection.

`txt remove`
: remove the currently selected part of the text.

`txt copy`
: copy the current selection into the clipboard.

`txt cut`
: copy the current selection into the clipboard, then remove it from the text.

`txt paste`
: replace the current selection by the clipboard contains.

`txt setAddMode` *bool*
: Set weither or not the text is in "add mode". When in add mode, text insertion won't modify the current selection.

`txt setHighlight` *start stop mode*
: Change the highlight appearance of the text between *start* and *stop*, but not the current selection. *mode* may be either `normal` , `selected` or `secondary_selected` .

`txt findString` *start stop string dir pos*
: Search the current text for *string* between the position *start* and *stop*. The direction *dir* might be either `forward` , either `backward` .

  If found, the position of the first occurence is set to the *Tcl*variable *pos*, and it returns `true`, else it returns `false`.

`txt getInsertPosition`
: Returns the position of the insert cursor. Zero is the first charactere in the text.

`txt setInsertPosition` *position*
: Set the cursor insertion point.

`txt getLastPosition`
: Returns the position of the last caractere in the text buffer, in other words, its length.

`txt scroll` *lines*
: Scroll the text widget by *lines* lines. A positive value scroll it upward, a negative value backward.

`txt showPosition` *position*
: Scroll the text such that *position* become visible.

txt `getTopCharacter`
Returns the position of the first visible charactere of the text in the widget.

txt `setTopCharacter` *position*
Scroll the text so that *position* will be the first visible charactere in the widget.

txt `disableRedisplay`
The text will not being redisplayed.

txt `enableRedisplay`
The text will redisplay automatically when it changes.

txt `getEditable`
Returns `true` if the text is editable (that is, the user can edit it), `false` if not.

txt `setEditable` *bool*
Set the edit permition flag of the text widget.

txt `setSource` *ref top ins*
Set the text edited/displayed by this widget to the one that is also edited/displayed by the text widget *ref*. The text will be scrolled such that the top charactere will be *top*, and the insertion cursor positionned at *ins*.

Resources

`xmText` resource name	default value	type or legal values
`-autoShowCursorPosition`	`True`	*Boolean*
`-cursorPosition`	`0`	*Integer*
`-editable`	`True`	*Boolean*
`-editMode`	`single_line_edit`	`multiple_line_edit` , `single_line_edit` .
`-marginHeight`	`5`	*Integer*
`-marginWidth`	`5`	*Integer*
`-maxLength`	*maxint*	*Integer*
`-source`	new source	*Text Source*
`-topCharacter`	`0`	*Integer*
`-value`	`""`	*String*
`-verifyBell`	`True`	*Boolean*

`xmTextInput` resource name	default value	type or legal values
`-pendingDelete`	`True`	*Boolean*
`-selectionArray`	not supported	
`-selectionArrayCount`	not supported	
`-selectThreshold`	`5`	*Integer*

`xmTextOutput` resource name	default value	type or legal values
`-blinkRate`	500ms	*Integer*
`-columns`	computed from `-width`	*Integer*
`-cursorPositionVisible`	`True`	*Boolean*
`-fontList`	Inherited	*Font list*
`-resizeHeight`	`False`	*Boolean*
`-resizeWidth`	`False`	*Boolean*
`-rows`	computed from `-height`	*Integer*
`-wordWrap`	`False`	*Boolean*

The `xmText` widget inherits resources from two abstract classes, `xmTextInput` and `xmTextOutput` . `xmTextField` use the resource subset that correspond to single-line text (e.g. it does not have a `-editMode` resource).

The text source resource might be used to open multiple windows editing a single text, as in the exemple below :

It needs a convertion to/fr
Pointer !

```
xmPanedWindow .top managed
xmScrolledText .top.a managed \
    -editMode multi_line_edit \
    -value {Que j'aime a faire apprendre un nombre utile aux sages,
            Immortel Archimede, artiste, ingenieur,
            qui de ton jugement peut priser la valeur,
            pour moi il eu de serieux avantages.}
xmScrolledText .top.b managed \
    -editMode multi_line_edit
.top.b setSource .top.a 0 0
```

The `xmTextInput` and `xmTextOutput` abstract classes are just used to group resources dedicated to text editing or displaying. Large text should be displayed or edited with the `xmScrolledText` widget, which automatically provides scroll bars when needed.

Furthermore, text widgets inherit any resources defined in the Core (page 21), Primitive (page 23), and `xmLabel` (page 28) classes.

`-accelerators`	(Core)	`-alignment`	(Label)
`-backgroundPixmap`	(Core)	`-background`	(Core)
`-borderColor`	(Core)	`-borderWidth`	(Core)
`-bottomShadowColor`	(Primitive)	`-bottomShadowPixmap`	(Primitive)
`-fontList`	(Label)	`-foreground`	(Primitive)
`-heigth`	(Core)	`-highlightColor`	(Primitive)
`-highlightOnEnter`	(Primitive)	`-highlightPixmap`	(Primitive)
`-highlightThickness`	(Primitive)	`-labelPixmap`	(Label)
`-labelString`	(Label)	`-labelType`	(Label)
`-mappedWhenManaged`	(Core)	`-marginBottom`	(Label)
`-marginLeft`	(Label)	`-marginRight`	(Label)
`-marginTop`	(Label)	`-navigationType`	(Primitive)
`-recomputeSize`	(Label)	`-sensitive`	(Core)
`-shadowThickness`	(Primitive)	`-stringDirection`	(Label)
`-topShadowColor`	(Primitive)	`-topShadowPixmap`	(Primitive)
`-translations`	(Core)	`-traversalOn`	(Primitive)
`-unitType`	(Primitive)	`-width`	(Core)
`-x`	(Core)	`-y`	(Core)

Text verify callbacks

The text widgets allows special processing by the application of text entered. After a character has been typed, or text pasted in, initial processing by the Text widget determines what the user is entering. This text is then passed to special callback functions. These functions can make copies of the text, can alter it, or can set a flag to say do not display it. Simple uses for this are a password entry widget that reads the text but does not display it (or echoes '`*`' instead), or text formatting widgets.

The callback mechanism for this is basically the same as for other callbacks, and similar sorts of substitutions are allowed. For example, the term `%currInsert` is replaced by the current insertion position. Other substitutions do not give a value, but rather give the name of a *Tcl* variable. This allows the application to change the value as required. For example, to turn off echoing of characters, the following should be done :

```
.text modifyVerifyCallback {
       set %doit false
}
```

An alternate style would have been to call a separate procedure to handle the work to be done. The *Tcl* variable is in the context of the callback caller, so `upvar` should be used :

```
.text modifyVerifyCallback {no_echo %doit}
proc no_echo {doit} {
       upvar 1 $doit do_insert
       set do_insert false
}
```

Actually, the Tcl variable here is the global variable _Tm_Text_Doit. For this reason, variables beginning with _Tm_ are reserved for use by the Tm library.

Callbacks

The supported callbacks are :

Method name	Why
`helpCallback`	The help key is pressed.
`destroyCallback`	Widget is destroyed.
`activateCallback`	Some event trigger the Activate action.
`gainPrimaryCallback`	Ownership of the primary selection is gained.
`losePrimaryCallback`	Ownership of the primary selection is loosed.
`losingFocusCallback`	Before losing input focus.
`modifyVerifyCallback`	Before deletion or insertion.
`motionVerifyCallback`	Before moving the insertion point.
`valueChangedCallback`	Some text was deleted or inserted.

The following callbacks substitutions are defined for the text specific callbacks :

`%doit`
: In a verify callback, the variable name of the flag to know if we should do it.

`%currInsert` , `%newInsert`
: In a `motionVerifyCallback`, the insertion point before and after the projected motion.

`%startPos` , `%endPos`
: Define a substring in the widget's text string.

`%ptr` , `%length`
: Define the string which is to be modifyed, in a `modifyVerify` callback. For instance, the following example may be used to chande inputs to uppercase :

```
proc allcaps {ptr length} {
   upvar 1 $ptr p
   upvar 1 $length l

   if {$l == 0} return
   set upper [string toupper $p]
   set p $upper
}

.text modifyVerifyCallback {allcaps %ptr %length}
```

In addition, text widgets inherit callbacks from the Primitive class, namely help and destroy callbacks.

7.3 Buttons

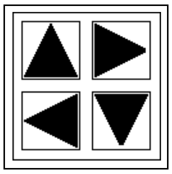
OSF/Motif use several flavors of the button to be pushed, namely :

`xmPushButton`
: The regular button, displaying a text or pixmap label, surrounded by a beveled shadow. When focus is gained, the button appear brighter, if it is sensitive.

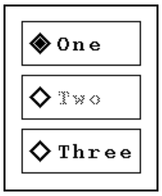
Pressing the mouse change the shadow to make the impression that the button ase been pushed in, when mouse is released, the button appear normal.

The default push buttons of a dialog may be specified by `-showAsDefault true`, in which case an additional border is drawn using the margin resources.

`xmArrowButton`
: A button showing an arrow, whose direction is given by the `-arrowDirection` resource.



`xmToggleButton`
: A button, which displays a state in an on/off indicator. Usually, a toggle button consist of a square or diamond indicator with an associated label.



An empty or filled indicator, or a different pixmap may be used to indicates the selected/unselected state of the button.

A set of "radio buttons" might be grouped into a manager (see section 8), with the `-radioBehavior` set to `True`, to ensure that only one of them will be selected at a given time. If the manager's `-radioAlwaysOne` resource is also set, then there will alway be exactly one toggle button set.

Resources

Button resources are :

`xmPushButton` resource name	default value	type or legal values
`-armColor`	computed	*`Color`*
`-armPixmap`	*`none`*	*`Pixmap`*
`-defaultButtonShadowThickness`	`0`	*`Dimension`*
`-fillOnArm`	`True`	*`Boolean`*
`-multiClick`		`multiclick_discard`
		`multiclick_keep`
`-showAsDefault`	`0`	*`Dimension`*

`xmArrowButton` resource name	default value	type or legal values
`-arrowDirection`	`arrow_up`	`arrow_up`
		`arrow_down`
		`arrow_left`
		`arrow_right`

`xmToggleButton` resource name	default value	type or legal values
`-fillOnSelect`	`True`	*`Boolean`*
`-indicatorOn`	`True`	*`Boolean`*
`-indicatorSize`	*`none`*	*`Dimension`*
`-indicatorType`	`n_of_many`	`n_of_many`
		`one_of_many`
`-selectColor`	computed	*`Color`*
`-selectInsensitivePixmap`	*`none`*	*`Pixmap`*
`-selectPixmap`	*`none`*	*`Pixmap`*
`-set`	*`False`*	*`Boolean`*
`-spacing`	`4`	*`Dimension`*
`-visibleWhenOff`	computed	*`Boolean`*

Furthermore, text widgets inherit any resources defined in the Core (page 21), Primitive (page 23), and Label (page 28) classes.

`-accelerators`	(Core)	`-allignment`	(Label)
`-backgroundPixmap`	(Core)	`-background`	(Core)
`-borderColor`	(Core)	`-borderWidth`	(Core)
`-bottomShadowColor`	(Primitive)	`-bottomShadowPixmap`	(Primitive)
`-fontList`	(Label)	`-foreground`	(Primitive)
`-heigth`	(Core)	`-highlightColor`	(Primitive)
`-highlightOnEnter`	(Primitive)	`-highlightPixmap`	(Primitive)
`-highlightThickness`	(Primitive)	`-labelPixmap`	(Label)
`-labelString`	(Label)	`-labelType`	(Label)
`-mappedWhenManaged`	(Core)	`-marginBottom`	(Label)
`-marginHeight`	(Label)	`-marginLeft`	(Label)
`-marginRight`	(Label)	`-marginRight`	(Label)
`-marginTop`	(Label)	`-navigationType`	(Primitive)
`-recomputeSize`	(Label)	`-sensitive`	(Core)
`-shadowThickness`	(Primitive)	`-stringDirection`	(Label)
`-topShadowColor`	(Primitive)	`-topShadowPixmap`	(Primitive)
`-translations`	(Core)	`-traversalOn`	(Primitive)
`-unitType`	(Primitive)	`-width`	(Core)
`-x`	(Core)	`-y`	(Core)

Callbacks

In addition to the usual `helpCallback` and `destroyCallback`, button widgets define the following new ones :

Method name	Why
`armCallback`	Button pressed.
`disarmCallback`	Button released, when the pointer still on it.
`activateCallback`	Some event triger the Activate function.

The toggle button also define the `%set` callback substitution, which is replaced by the boolean state of the button.

7.4 Decorativ widgets

Simple decorativ widgets include `xmFrame` and `xmSeparator` . The former is simply a container widget that display a frame around its child, using in/out shadowing or etching. The later is a primitive widget that look like a flat or beveled line ; it is used t separates items in a display.

This two widget classes do not interact with user input, hence they do not have actions, callbacks or translations.

New decoration resources are :

`xmFrame` resource name	default value	type or legal values
`-marginWidth`	`0`	*`Dimension`*
`-marginHeight`	`0`	*`Dimension`*
`-shadowType`	dynamic	`shadow_in`
		`shadow_out`
		`shadow_etched_in`
		`shadow_etched_out`

`xmSeparator` resource name	default value	type or legal values
`-margin`	`0`	*`Dimension`*
`-orientation`	`horizontal`	`horizontal`
		`vertical`
`-separatorType`	`shadow_etched_in`	`shadow_etched_in`
		`shadow_etched_out`
		`no_line`
		`single_line`
		`double_line`
		`single_dashed_line`
		`double_dashed_line`

In addition, they inherit the following resources from the Primitive (page 23), and Core classes (page 21) :

`-backgroundPixmap`	(Core)		
`-background`	(Core)	`-borderColor`	(Core)
`-borderWidth`	(Core)	`-bottomShadowColor`	(Primitive)
`-bottomShadowPixmap`	(Primitive)	`-foreground`	(Primitive)
`-heigth`	(Core)	`-mappedWhenManaged`	(Core)
`-shadowThickness`	(Primitive)	`-topShadowColor`	(Primitive)
`-topShadowPixmap`	(Primitive)	`-unitType`	(Primitive)
`-width`	(Core)	`-x`	(Core)
`-y`	(Core)		

7.5 xmList

A list is used to display an ordered set of strings. Mouse or keyboard intercation permit to select item(s).

A **xmScrolledList** should be used when the number of item may be too large to display in the alloted space in the interface : the interface is automatically changed to display a **xmScrollBar** (see below) to move the visible part of the list.

A scrolled list widget *w* is a composite widget that have the following childrens :

w.HorScrollBar *w*.VertScrollBar *w*.ClipWindow

The associated names might be used to directly access them, as in the following example :

```
xmScrolledList .list managed
.list.VertScrollBar setValues -troughColor red
```

Different selection mode exist :

single_select
: Only one item may be selected at any time. A button click in the list deselect any previous selection, and select that item. Each time a selection is made, **singleSelectionCallback** is called.

multiple_select
: Shift-clicks may be used to make multiple selections. **multipleSelectionCallback** is called each time an item is selected or unselected.

extended_select
: Any single mouse click deselect anything, and select the current item. Any shift-click extend the current selection up to the item underbeneath the mouse.

  extendedSelectionCallback is called for each item selection or deselection.

browse_select
: Mouse dragging may be used to select a range of items. Using shift-clicks or shift-drags, more than one range may be selected at a given time.

  browseSelectionCallback is called for each newly selected item, once mouse button is released.

  This is the default mode.

In all mode, the **defaultActionCallback** is called when the user double-click on an item. The following methods are provided to manage the selection list :

list addItem *item position*
: Add the specified ***item*** (any *Tcl*string value) to the existing list, at the given ***position***. If ***position*** is 1 or greater, the new item will be the first one, second one, ... If ***position*** is 0, the insertion is made at the end.

list `addItemUnselected` *item position*
: Normally, if you an item already selected, the second instance will also be selected. this method ensure that the newly inserted item will not be selected.

list `deletePosition` *position*
: Delete the item specified by ***position***. If ***position*** is 0, the last item is deleted.

list `deleteItem` *item*
: Delete the first occurence of ***item*** in the list. A warning will occur if the item does not exist.

list `deleteAllItems`
: Delete all items in the list.

list `selectPosition` *position notify*
: Select the item at the given ***position*** in the list. If ***notify*** is `true`, the corresponding callback is called.

list `selectItem` *item notify*
: Select the first given item in the list. If ***notify*** is `true`, the corresponding callback is called.

list `deselectItem` *item*
: Deselect the first given item in the list. If the item is at multiple position in the list, only the first occurence is deselected (even if it's not the selected one !).

list `deselectPosition` *position*
: Deselect the item at the given position in the list.

list `itemExists` *item*
: Reply `true` if the ***item*** is in the list, `false` if not.

list `itemPosition` *item*
: Return the position in th elist of the given ***item***, or `0` if it does not exists.

list `positionSelected` *position*
: Reply `true` if the ***position*** is currently selected, `false` if not.

list `setItem` *item*
: Scroll the list so that the first occurence of ***item*** will be at the top of the currently displayed part of the list.

list `setPosition` *position*
: Scroll the list so that the ***position***'th item will be at the top of the currently displayed part of the list.

list `setBottomItem` *item*
: Scroll the list so that the first occurence of ***item*** will be at the bottom of the currently displayed part of the list.

list `setBottomPosition` *position*
: Scroll the list so that the ***position***'th item will be at the bottom of the currently displayed part of the list.

Resources

List specific resources are :

xmList resource name	default value	type or legal values
-automaticSelection	False	*Boolean*
-doubleClickInterval	Inherited	*Integer*
-fontList	Inherited	*Font List*
-itemCount	computed	*Integer*
-items	*none*	*String array*
-listMarginHeight	0	*Integer*
-listMarginWidth	0	*Integer*
-listSizePolicy CO	variable	constant resize_if_possible variable
-listSpacing	0	*Integer*
-scrollBarDisplayPolicy	as_needed	as_needed static
-selectedItemCount	0	*Integer*
-selectedItems	*none*	*String array*
-selectionPolicy	browse_select	browse_select extended_select multiple_select single_select
-stringDirection	Inherited	string_direction_l_to_r string_direction_r_to_l
-topItemPosition	1	*Integer*
-visibleItemCount	1	*Integer*

Other resources are derived from the Core (page 21), Primitive (page 23), and Label (page 28) classes.

Callbacks

List specific supported callbacks are :

Method name	Why
defaultActionCallback	An item was double-clicked.
singleSelectionCallback	An single item was selected.
multipleSelectionCallback browseSelectionCallback extendedSelectionCallback	An item was selected, when in the corresponding selection mode.

The following substitutions are defined for this callbacks :

`%item`
: The currently selected item string.

`%item_length`
: The string length of the currently selected item.

`%item_position`
: The current item position, `1` indicating the first one.

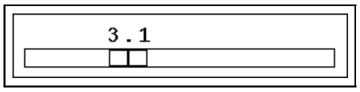
`%selected_items`
: Valid only in multiple, browse or extended callbacks, this substitution returns a comma-separated list of all currently selected items.

Care should be take to enclose `%item` and `%selected_items` between braces, to avoid parsing error when item string contain spaces.

In addition, text widgets inherit the standard callbacks from the Primitive class, namely `helpCallback` and `destroyCallback`.

7.6 xmScale

A scale widget display a cursor that can be moved between a minimal and a maximal value.



Resources

The scale widget class define the new resources given below.

`xmScale` resource name	default value	type or legal values
`-decimalPoints`	0	*Integer*
`-fontList`	Inherited	*Font List*
`-highlightOnEnter`	`False`	*Boolean*
`-highlightThickness`	2	*Dimension*
`-maximum`	100	*Integer*
`-minimum`	0	*Integer*
`-orientation`	`vertical`	`horizontal` `vertival`
`-processsingDirection`	computed	`max_on_bottom` `max_on_left` `max_on_right` `max_on_top`
`-scaleHeight`	0	*Dimension*
`-scaleWidth`	0	*Dimension*
`-scaleMultiple`	$(max - min)/10$	*Integer*
`-showValue`	`False`	*Boolean*
`-titleString`	""	*String*
`-value`	0	*Integer*

The slider may be moved between the integer `-minimum` and `-maximum`. Fractional values are obtained using the `-decimalPoints` resource, to display a decimal point. The slider size may be set by `-scaleHeight` and `-scaleWidth`. `-showValue` tells to display a textual readout of the current value. `-scaleMultiple` is used for large slider move (Control-arrow on the keyboard).

Callbacks

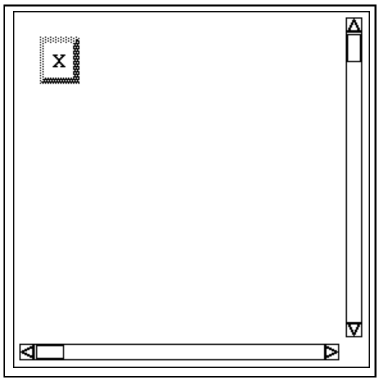
Method name	Why
`valueChangedCallback`	The scale value had changed.
`dragCallback`	The slider is being dragged.

In this callbacks, the `%value` substitution may be used to retrieve the current scale position.

In addition, `xmScale` inherits the usual `helpCallback` from the Primitive abstract class.

7.7 xmScrollBar

The **xmScrollBar** widget is made to allow moving the current vue of a widget too large to be displayed at once. Usually, scroll bars will be part of a **xmScrolledWidget**, **xmScrolledText** or **xmScrolledList** widget.



An **xmScrollBar** may be horizontal or vertical (depending on **-orientation**. It is composed of the two arrows, a larger rectangle called the scroll region, and a smaller one: the slider. The data is scrolled by clicking either arrow, clicking inside the scroll region, or dragging the slider. When the mouse is held down in the scroll region or in either arrow, the data continues to move at a constant speed.

The following example use two scrollbars to move a target button :

```
#! moat

xtAppInitialize
xmBulletinBoard .top managed

xmScrollBar .top.h managed \
    -orientation horizontal -width 250 \
    -y 260 -minimum 10 -maximum 240

xmScrollBar .top.v managed \
    -orientation vertical -height 250 \
    -x 260 -minimum 10 -maximum 240

xmPushButton .top.target managed \
    -labelString "X"

proc track_it {} {
    .top.h getValues -value x
    .top.v getValues -value y
    .top.target setValues -x [expr 8+$x] -y [expr 8+$y]
}
.top.h dragCallback track_it
```

```
.top.v dragCallback track_it
.top.h valueChangedCallback track_it
.top.v valueChangedCallback track_it

track_it

. realizeWidget
. mainLoop
```

Resources

The scroll bar widget class define the new resources given below.

`xmScrollBar` resource name	default value	type or legal values
`-increment`	`1`	*Integer*
`-initialDelay`	`250` ms	*Integer*
`-maximum`	`100`	*Integer*
`-minimum`	`0`	*Integer*
`-orientation`	`vertical`	`horizontal` `vertical`
`-pageIncrement`	`10`	*Integer*

`-processsingDirection`	computed	`max_on_bottom`
		`max_on_left`
		`max_on_right`
		`max_on_top`

`-repeatDelay`	`50` ms	*Integer*
`-showArrows`	`True`	*Boolean*
`-sliderSize`	computed	*Integer*
`-troughColor`	computed	*Color*
`-value`	`0`	*Integer*

The `-value` resource contains the current position of the slider's begin, between `-minimum` and $maximum - sliderSize$.

The slider move between `-minimum` and `-maximum`, by `-increment` steps (clipped at the ends). Clicking either arrow move by `-pageIncrement`. `-sliderSize` may be used to reflect the portion of the widget which is currently in the view. `-troughColor` is the slider fill color.

Constant speed moving is parametrized by `-repeatDeleay` and `-initialDelay`. If `-showArrows` is set to `False`, the scroll bar won't have arrows on both sides.

Callbacks

Method name	Why
`decrementCallback`	value was decremented.
`dragCallback`	The slider is being dragged.
`incrementCallback`	value was incremented.
`pageDecrementCallback`	value was decremented by pageIncrement.
`pageIncrementCallback`	value was incremented by pageIncrement.
`toTopCallback`	value was reset to minimum.
`toBottomCallback`	value was reset to maximum.
`valueChangedCallback`	The value had changed.

In the corresponding callbacks, the `%value` substitution will return the current scroll bar position.

8 Manager widgets

Manager widgets are used to layout several widgets together, enabling to construct complex interfaces from simpler widgets.

Theyre main purpose is to find a suitable geometry that enclose all managed children ; at creation time, when the user manually resize the window, or when widgets dynamically change itself.

Normaly, manager do not interact with events, they just forward them to to appropriate child. The notable exception is navigation : use of keyboard to change the currently selected children widget.

8.1 The `xmManager` abstract class

This class is not a subclass of Primitive, but have some graphical representation, so we have a subset of Primitive's resources and behavior here.

Resources

The *OSF/Motif* Manager abstract widget class is used to define the common resource set described below.

`xmManager` resource name	default value	type or legal values
`-bottomShadowColor`		*`Color`*
`-bottomShadowPixmap`	*`none`*	*`Pixmap`*
`-foreground`	computed	*`Color`*
`-highlightColor`	computed	*`Color`*
`-highlightPixmap`	*`none`*	*`Pixmap`*
`-navigationType`	`tab_group`	`none`
		`tab_group`
		`sticky_tab_group`
		`exclusive_tab_group`
`-shadowThickness`	`0`	*`Dimension`*
`-stringDirection`	Inherited	`string_direction_l_to_r`
		`string_direction_r_to_l`
`-topShadowColor`	computed	*`Color`*
`-topShadowPixmap`	*`none`*	*`Pixmap`*
`-traversalOn`	`True`	*`Boolean`*
`-unitType`	Inherited or `pixels`	`pixels`
		`100th_millimeters`
		`1000th_inches`
		`100th_points`
		`100th_font_units`

Callbacks

The Manager abstract class also defines callbacks for all manager subclass, descibed in the table below.

Method name	Why
`focusCallback`	The widget will receive input focus.
`helpCallback`	The usual Help callback.
`mapCallback`	The widget is mapped on screen.
`unmapCallback`	The widget is unmapped from screen.

There is no special substitution associated with this callbacks.

8.2 xmBulletinBoard

The **xmBulletinBoard** manager is the simplest one. Children widgets are positionned using their **-x** and **-y** resources. There is no particular management when the widget is resized.

Resources

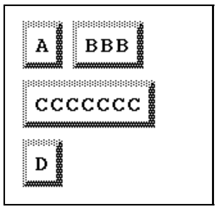
xmBulletinBoard resource name	default value	type or legal values
-allowOverlap	True	*Boolean*
-autoUnmanage [co]	True	*Boolean*
-buttonFontList	Inherited	*Font List*
-cancelbutton	*none*	*Widget*
-defaultbutton	*none*	*Widget*
-defaultPosition	True	*Boolean*
-dialogStyle	computed	dialog_system_modal
		dialog_primary_application_modal
		dialog_application_modal
		dialog_full_application_modal
		dialog_modless
		dialog_work_area
-dialogTitle	*none*	*String*
-labelFontList	Inherited	*Font List*
-marginHeight	10	*Dimension*
-marginWidth	10	*Dimension*
-noResize	False	*Boolean*
-resizePolicy	any	resize_any
		resize_grow
		resize_none
-shadowType	shadow_out	shadow_in
		shadow_out
		shadow_etched_in
		shadow_etched_out
-textFontList	Inherited	*Font List*
-textTranslations [cro]	""	*String*

When **-allowOverlap** is set to **False**, any placement of children that would result in an overlap will be rejected.

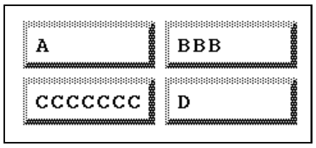
Setting **-noResize** to **True** will disable any resize of the widget, while **-resizePolicy** may be used to control more what kind of resize should be allowed.

8.3 `xmRowColumn`

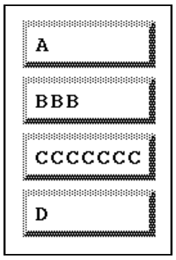
The `xmBulletinBoard` manager place its children in one or more columns (or rows). Different packing styles, main direction and size options permit to have aligned or unaligned rows (or columns), as in the following examples :



`-packing pack_tight`
`-orientation horizontal`



`-packing pack_column`
`-numColumns 2`



`-orientation horizontal`

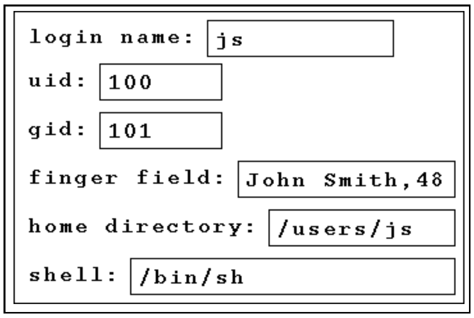
Resources

xmRowColumn resource name	default value	type or legal values
-adjustLast	True	*Boolean*
-adjustMargin	True	*Boolean*
-entryAlignment	alignment_center	alignment_center alignment_beginning alignment_end
-entryBorder	0	*Integer*
-entryClass	dynamic	*Widget Class*
-isAligned	True	*Boolean*
-isHomogeneous	True	*Boolean*
-labelString [co]	""	*String*
-marginHeight	Inherited	*Dimension*
-marginWidth	Inherited	*Dimension*
-menuAccelerator	?	*String*
-menuHelpWidget	*none*	*Widget*
-menuHistory	*none*	*Widget*
-menuPost	""	*String*
-mnemonic	*none*	*KeySym*
-mnemonicCharSet	*dynamic*	*String*
-numColumns	1	*Integer*
-orientation	computed	horizontal vertical
-packing	computed	pack_column pack_none pack_tight
-popupEnabled	True	*Boolean*
-radioAlwaysOne	True	*Boolean*
-radioBehavior	False	*Boolean*
-resizeHeight	True	*Boolean*
-resizeWidth	True	*Boolean*
-rowColumnType [co]	work_area	menu_bar menu_option menu_popup menu_pulldown work_area
-spacing	3 or 0	*Dimension*
-subMenuId	*none*	*Widget*
-whichButton	computed	*Integer*

8.4 xmForm

A form is a manager widget, created to layout widgets using neighbourhood relationship, such as "this widget should be positionned at the left of this one". This is quit general, and enable to define widgets that may resize gracefully.

The following exemple gives illustrate this :



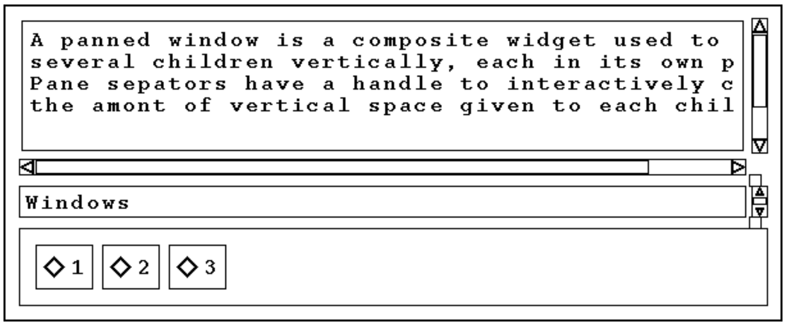
This constraints are defined in terms of attachment of each side of children widgets to the form border, to another widget, to a relative position in the form, or to the initial position of the child. When a resizing occurs, children are adjusted according to this constraints.

Resources

`xmForm` resource name	default value	type or legal values
`-fractionBase`	`100`	*Integer*
`-horizontalSpacing`	`0`	*Dimension*
`-rubberPositioning`	`False`	*Boolean*
`-verticalSpacing`	`0`	*Dimension*
`-`*side*`Attachment`	`attach_none`	`attach_form`
		`attach_none`
		`attach_opposite_form`
		`attach_opposite_widget`
		`attach_position`
		`attach_self`
		`attach_widget`
`-`*side*`Offset`	`0`	*Integer*
`-`*side*`Position`	`0`	*Integer*
`-`*side*`Widget`	*none*	*Widget*

8.5 xmPanedWindow

A panned window is a composite widget used to layout several children vertically, each in its own pane. Pane sepators have a handle to interactively change the amont of vertical space given to each children.



Resources

`xmPanedWindow` resource name	default value	type or legal values
`-marginHeight`	3	*Dimension*
`-marginWidth`	3	*Dimension*
`-refigureMode`	True	*Boolean*
`-sahsHeight`	10	*Dimension*
`-sashIndent`	-10	*Dimension*
`-sashShadowThikness`	*dynamic*	*Dimension*
`-sahsWidth`	10	*Dimension*
`-separatorOn`	True	*Boolean*
`-spacing`	8	*Dimension*

`-refigureMode`: The children should be reseted to their appropriate positions when the paned window is resized.

`xmPanedWindow` Constraint resource name	default value	type or legal values
`-allowResize`	True	*Boolean*
`-paneMaximum`	1000	*Dimension*
`-paneMimimum`	1	*Dimension*
`-skipAdjust`	False	*Boolean*

`-skipAdjust` : The paned window should not automatically resize this pane.

9 Drag and Drop

Drag and drop was introduced into *OSF/Motif* 1.2. It is complicated. We shall first look at the drop side. A widget has to first register itself as a drop site, so that when an attempt is made to drop something on it, it will try to handle it. This registration is done by the widget method `dropSiteRegister` . This registration must include *Tcl* code to be executed when a drop is attempted, and this is done using the resource `-dropProc` . The first part of what makes D&D hard is that you have potentially two different applications attempting to communicate, one dropping and the other accepting the drop. A protocol is needed between these, so that they share a common language. This is done in registration by saying what types of protocol are used, and how many there are. This is done using X atoms, and the major ones are `COMPOUND_TEXT`, `TEXT` and `STRING`. Thus registration is done, for example, by

```
.l dropSiteRegister \
    -dropProc {startDrop %dragContext} \
    -numImportTargets 1 \
    -importTargets COMPOUND_TEXT
```

This allows `.l` to be used as a drop site, accepting `COMPOUND_TEXT` only. Multiple types are allowed, using the Motif list structure of elements separated by commas as in `"COMPOUND_TEXT, TEXT, STRING"`. When a drop occurs, the procedure `startDrop` is called, with one substituted parameter. This parameter is a `dragContext`, which is a widget created to by *OSF/Motif* to handle the drag part of all this. You must include this parameter, or the next stage doesn't get off the ground.

When a drag actually occurs, *OSF/Motif* creates a `dragContext` widget. A drag is started by holding down the middle button in a drag source, which is discussed later. The `dragContext` widget contains information about the drag source, which is to be matched up against where the drop occurs. When the drop occurs, by releasing the middle button, the *Tcl* code registered as `dropProc` is executed. This should have the `dragContext` widget as parameter. This code may try to determine if the drop should go ahead, but more normally will just act as a channel through to the actual information transfer. Still here ? Good ! The `dragProc` doesn't actually do the information transfer, it just determines whether or not it is possible, and if it is, what protocols should be used, and how.

The drop receiver may decide that it wants something encoded as `TEXT`, followed by something encoded as `COMPOUND_TEXT`, and then by something in `STRING` format (beats me why, though...). it signals this by a (*Tcl*) list of dropTransfer pairs, consisting of the protocol (as an X atom name) and the widget that is being dropped on. Huh? Why the widget that is being dropped on? Because when a drop on a widget takes place, this is actually dealt with by the dragContext widget, and this is about to hand the transfer over to a transferWidget. Yes, I know you are using *Tcl* because you couldn't handle triple indirections (or rather, don't want too!), but they occur anyway... So here is a simple dragProc:

```
proc startDrop {dragContext} {
    $dragContext dropTransferStart \
        -dropTransfers {{COMPOUND_TEXT .l}} \
        -numDropTransfers 1 \
        -transferProc {doTransfer %closure {%value}}
}
```

The dragContext widget uses the command `dropTransferStart` to signal the beginning of the information transfer (it could also signal that the drop is to terminate, with no information transfer). It will accept one chunk of information in the `COMPOUND_TEXT` format, and pass this on to the `.l` widget. The information transfer is actually carried on by the Tcl procedure in the transferProc resource. The only formats currently accepted (because they are hard-coded into *Tm*) are `COMPOUND_TEXT`, `TEXT` and `STRING`.

The `transferProc` resource is a function that is called when the drop receiver actually gets the information dropped on it. This should take at least two parameters. The `%value` is substituted for the actual information dropped on it, and `%closure` is the second element in the dropTransfer list which should be the widget the drop is happening on. (Why not let *Tm* determine this? I dunno. Consistency with *OSF/Motif* doco? Brain damage late at night?) Then the dropped on widget can take suitable action. This function resets the label to the text dropped on it:

```
proc doTransfer {destination value} {
     $destination setValues -labelString $value
}
```

where destination is substituted by `%closure` and value by `%value`.

10 Send

Tk has a primitive called `send`. In this, each interpreter has a name, and you can send *Tcl* commands from one interpreter to another. When an interpreter receives a sent command it executes it, and returns any result back to the original interpreter. This mechanism is also available to *Tm* so that Motif applications can set commands to other Motif applications, and also to and from *Tk* ones.

If a *Tm* application succeeds in registering its name, from then on, it can send to another. For example,

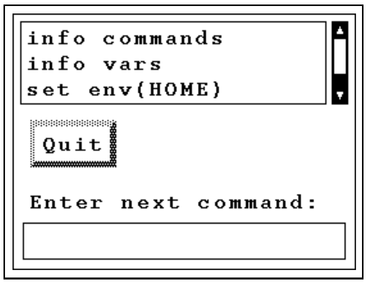
```
send interp2 {puts stdout "hello there"}
```

instructs "interp2" to display a message.

11 More Widgets

11.1 `xmCommand`

A command widget is composed of an history area (a `xmScrolledList`), a label to display the prompt, and a text field to edit the current command. The command widget is a subclass of `xmSelectionBox`. You are able to add an extra child, called the work area. In the example below, this was used to add a button bar :



Methods

The command widget recognize a few new methods :

`cmd` `appendValue` *`command`*
: Append *`cmd`* to the string already in the text field. The string will be truncated before the first
`n` encontered.

`cmd` `error` *`error_message`*
: Temporalily display the *`error_message`* at the bottom of the history area. It will automatically disapear once the user entered another command.

`cmd` `setValue` *`command`*
: Replace the string in the text field by *`command`*. The old command is not entered in the history.

Resources

`xmCommand` resource name	default value	type or legal values
`-command`	`""`	*String*
`-historyItems`	`""`	*String Table*
`-historyItemCount`	`0`	*Integer*
`-historyMaxItems`	`100`	*Integer*
`-historyVisibleItemCount`	`8`	*Integer*
`-promptString`	`">"`	*String*

Other ones are inherited from `xmSelectionBox` and its ancestors.

Callbacks

Method name	Why
`commandChangedCallback`	The current command changed (you type a key in)
`commandEnteredCallback`	The command was entered (return key)

Both of this callbacks suport the `%value` and `%length` substitution, which are replaced by the string (or string length) that fired the callback.

11.2 `xmDrawingArea` and `xmDrawnButton`

Tm (version 1.0) have a very limited support for *Xlib* drawable area or buttons : you can only draw string on them.

Drawing methods

To manipulate such a widget, the currently defined methods are :

w `drawImageString` *gc x y string*

Use the given graphical context *gc* to draw the string *string* starting at position *x*,*y*. The 0,0 coordinate is at the upper-left of the widget.

For instance, the following code produce an hello widget :

```
xmDrawingArea .top managed
.top exposeCallback {
        set gc [.top getGC -foreground black]
        .top drawImageString $gc 10 10 "Hello World"
}
```

Note that it is necessary to use an `exposeCallback` to get the message redisplayed when needed.

Resources

`xmDrawningArea` resource name	default value	type or legal values
`-marginHeight`	10	*Dimension*
`-marginWidth`	10	*Dimension*
`-resizePolicy`	`resize_any`	`resize_any`
		`resize_grow`
		`resize_none`

`xmDrawnButton` resource name	default value	type or legal values
`-multiClick`	Inherited from display	`multiclick_discard`
		`multiclick_keep`
`-pushButtonEnabled`	`False`	*Boolean*
`-shadowType`	`shadow_out`	`shadow_in`
		`shadow_out`
		`shadow_etched_in`
		`shadow_etched_out`

Callbacks

Method name	Why
`exposeCallback`	The area/button should be redrawn
`inputCallback`	A keyboard oy mouse event arrived for the area.
`resizeCallback`	The area/button is resized
`acticateCallback`	The button was activated
`armCallback`	The button is squashed
`disarmCallback`	The button is released

11.3 xmMainWindow

This composit widget is to be used for the main window of an application. As you add child to it (a **xmMenuBar**, a **xmCommandWindow**, a **xmMessageBox**, a work area, **xmScrollBar**(s), ...) it manage them, as you could do manually with a **xmForm**.

The management of the work area is not imediat : the main window should know which of is son is the work area widget before you can manage this child. The following example will produce a prototype interface of a standard application :

```
#! moat

xtAppInitialize

xmMainWindow .top \
    -showSeparator True \
    -commandWindowLocation command_below_workspace

xmMenuBar .top.bar managed
xmCascadeButton .top.bar.File managed
xmCascadeButton .top.bar.Help managed

xmDrawingArea .top.work \
    -width 500 -height 400 \
    -background black
.top setValues -workWindow .top.work
.top.work manageChild

xmCommand .top.com managed \
    -historyVisibleItemCount 0 \
    -textFontList  -*-courier-medium-r-*--12-*-*-*-*-*-*
.top.com commandEnteredCallback {%value}

.top setValues -width 600 -height 500
.top manageChild

. realizeWidget
. mainLoop
```

Resources

This widget defines the following resources (renaming resources of its parents) :

`xmMainWindow` resource name	default value	type or legal values
`-commandWindow`	*none*	*Widget*
`-commandWindowLocation`	above	`command_above_workspace`
		`command_below_workspace`
`-mainWindowMarginHeight`	0	*Dimension*
`-mainWindowMarginWidth`	0	*Dimension*
`-menuBar`	*none*	*Widget*
`-messageWindow`	*none*	*Widget*
`-showSeparator`	False	*Boolean*

Callbacks

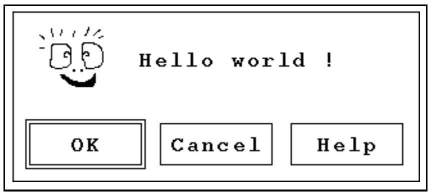
Method name	Why
`commandChangedCallback`	You type a key in, recall an history item, ...
`commandEnteredCallback`	`<key>Enter`, double-click, ...
`focusCallback`	The window get focus.
`mapCallback`	The window was mapped on screen.
`unmapCallback`	The window was unmapped.

12 Boxes

Boxes are complex widgets with a work area, and a line of buttons. They are designed to handle common layout of several more basic widgets. Boxes might be used as is, or as building blocks of more complex interfaces. They are also often used inside *dialog* (standalone windows), see section 14.

12.1 xmMessageBox

Message box are used to display simple messages. **xmMessageBox** may also display a symbol (pixmap) to show warnings, error conditions, ... This may be done by setting the **-dialogType** resource, or by specifying a pixmap (**-symbolPixmap**), as in the following example :



A message box is a composite widget, whose component children might be managed or unmanaged. This is done using the usual *Tm* commands **manageChild** and **unmanageChild** applyed on the automatically derived children objects. If the message box is named *w*, the known childrens are :

w.Cancel	*w*.Help	*w*.Message
w.OK	*w*.Separator	*w*.Symbol

The next example start a **xmMessageBox**, then drops unwanted features (buttons and the separator line), then add an icon, and finally manage it.

```
xmMessageBox .message \
    -messageString "Some simple message"
foreach child {OK Cancel Help Separator} {
    .message.$child unmanageChild
}
.message.Symbol setValues -labelPixmap face
.message manageChild
```

Resources

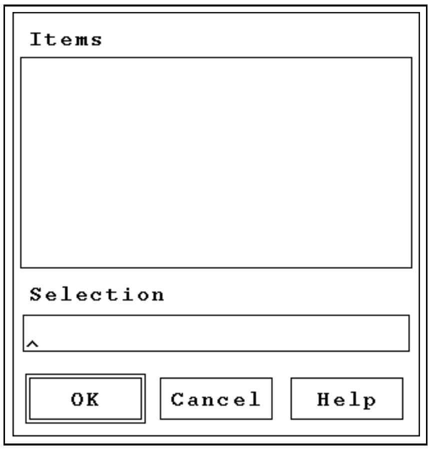
xmMessageBox resource name	default value	type or legal values
-cancelLabelString	"Cancel"	*String*
-defaultButtonType	dialog_ok_button	dialog_cancel_button dialog_help_button dialog_ok_button
-dialogType	dialog_message	dialog_error dialog_information dialog_message dialog_question dialog_warning dialog_working
-helpLabelString	"Help"	*String*
-messageAlignment	alignment_beginning	alignment_center alignment_beginning alignment_end
-messageString	""	*String*
-minimizeButtons	False	*Boolean*
-okLabelString	"Ok"	*String*
-symbolPixmap	depend of -dialogType	*Pixmap*

Callbacks

Method name	Why
cancelCallback	The cancel button was activated
helpCallback	The help button was activated, or an Help action arise.
okCallback	The ok button was activated
focusCallback	The window get focus.
mapCallback	The window was mapped on screen.
unmapCallback	The window was unmapped.

Furthermore, xmMessageBox also inherits destroyCallback from Core.

12.2 xmSelectionBox



A selection box is a composit box designed to ease creation of interfaces that enable the user to choose one (or several) items from a list. A selection box has a number of component children, which may be managed or unmanaged by the application.

These children widgets are often managed or unmanaged to add or remove elements from a dialog. *OSF/Motif* gives no information about types of these widgets, so managing and unmanaging are really the only two operations that you should perform on these widgets.

The corresponding *Tcl* commands are automatically created when the master command is created.

If the SelectionBox is named `w`, they are :

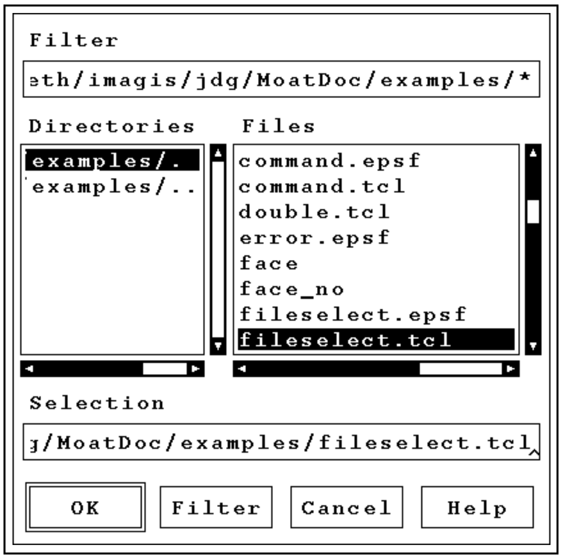
`w.Apply`	`w.OK`
`w.Cancel`	`w.Selection`
`w.Help`	`w.Separator`
`w.ItemsList`	`w.Text`
`w.Items`	

Method name	Why
`applyCallback`	The Apply button is released.
`cancelCallback`	The Cancel button is released.
`okCallback`	The Ok button is released.
`noMatchCallback`	Nothing match the selection expression.

The selection box widget also inherits all the callbacks defined in **`xmList`**, and in **`xmText`**.

`%length` **`%value`**

12.3 xmFileSelectionBox



The file selection box is designed to let the user interactively specify a directory and a file. A filter may be used to display only certain files, based on a regular expression matching their name.

w.Apply	*w*.FilterLabel	*w*.Items
w.Cancel	*w*.FilterText	*w*.OK
w.DirList	*w*.Help	*w*.Selection
w.Dir	*w*.ItemsList	*w*.Separator
w.Text		

Resources

`xmFileSelectionBox` resource name	default value	type or legal values

Callbacks

Method name	Why

`%value` `%value_length` `%mask` `%mask_length` `%dir` `%dir_length` `%pattern` `%pattern_length`

13 Menus

Menus are ... In *OSF/Motif*, this is done by using separate widgets for all the actors :

A **menu bar :**
that might be used to group (horizontally by default) several menu buttons. This will be described in the `xmMenuBar` section.

menu buttons :
A special subtype of `xmPushButton` that automatically popup a pulldown menu. When this widget is created as a child of another popup menu, (hence in a cascading submenu), a small arrows is added at the right of the label. This will be described in the `xmCascadeButton` section.

the **pulldown menu :** This a special king of `xmRowColumn` widget, intented to hold several buttons (and separators) vertically. This will be described in the next section.

13.1 xmPulldownMenu

A pulldown menu is a special kind of vertical `xmRowColumn`. It is managed only when it should be displayed. Pulldown or cascading menu are managed when the user clik on some `xmCascadeButton`. Popup menu are managed by a more general event, typicaly through a defined translation of the main window.

Menu items are child widgets (`xmLabel`, buttons, `xmSeparator`, or `xmCascadeButton`). The order of definition gives the item order.

Method name	Why
`popupCallback`	The menu is managed and mapped.
`popdownCallback`	The menu is un-mapped.

13.2 xmCascadeButton

The cascade button is a subclass of the usual push button (`xmPushButton`, page 39) that force management of a pulldown menu.

`xmCascadeButton` resource name	default value	type or legal values
`-windowId`	*none*	*Widget*

13.3 xmMenuBar

A menu bar is an "ever displayed horizontal pulldown menu", that may only contain cascade buttons. It is used to permanently display the buttons that trigger the pulldown menus of an application (for instance at the top of a `xmMainWindow`).

13.4 Exotic menus

Examples for a left menu bar, that is always managed

A pulldown menu in a dialog, that start to be displayed at the current setting.

A menu that display icons. (A suitable bushes of push buttons) ?

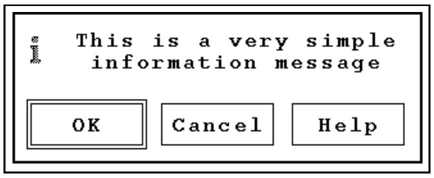
14 Dialogs

Dialogs are widgets that appear in their own window on the screen, when they are managed. Usually, they are *modeless* : interactions continue with all visible widgets, while they are visible.

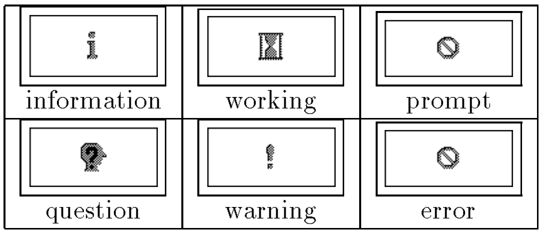
*Moat*does support the *modal* mode through the `dialogStyle` resource, when set to e.g. `dialog_full_application_modal` . The modal interaction is exited when the dialog disappear, typically when the user have activated some push button.

14.1 Simple informational dialogs

, , , , , , .



The simplest dialogs are message boxes in a dialog, with an optional icon. The predefined icons are :



Tm defines the following *Tcl* commands to create this dialogs : **`xmMessageDialog`**, **`xmInformationDialog`**, **`xmWorkingDialog`**, **`xmPromptDialog`**, **`xmQuestionDialog`**, **`xmWarningDialog`**, **`xmErrorDialog`**.

As for the corresponding message boxes, a particular child is accessible with the specific *Tcl* command.

14.2 General manager dialogs

The more general dialogs use the two multi-purpose managers inside. *Moat* defines the following `xmFormDialog` and `xmBulletinBoardDialog` commands to create them.

14.3 `xmSelectionDialog`

This is the standard *OSF/Motif* dialog used to select an item. See `xmSelectionBox` (page 69) for the corresponding box.

14.4 `xmFileSelectionDialog`

This is the standard *OSF/Motif* dialog used to select a directory and a file name. See `xmFileSelectionBox` (page 70) for the corresponding box.

Index

Contents