MIA 91-16: X Libraries - Reference Manual

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Copyright Notice

Mjølner Informatics Report
MIA 91-16
August 1999

Legal Notice

Legal Notice Mjølner Informatics

X Libraries - Reference Manual

List of programs

Introduction

This report describes the libraries available in the Mjølner System for programming X Window System applications.

The X Window System is a hardware-independent windowing system for workstations. It was developed by MIT and DEC and has been adopted as a standard platform for graphical applications. It is especially popular on UNIX systems.

An introduction to the X Window System seen from the users point of view is outside the scope of this report. If the reader is unfamiliar with using the X Window System, he is referred to one of the many available text-books on the subject (i.e. [Jones 89], [Mansfield 89], [Nye & O'Reilly 90a], [Nye & O'Reilly 90b], [Young 90]) or the on-line UNIX manual pages (man X).

What is Xt?

For the C programmer the low-level interface used in programming X Window System applications is a library called Xlib. Xlib defines an extensive set of functions that provide complete access and control over the display, windows, and input devices. Although programmers can use Xlib to build applications, this rather low-level interface is tedious and hard to use correctly. It does little to make programming easy.

To simplify development of application programs, the development group behind the X Window System realized that the user-interface elements of a graphical application: scrollbars, commands buttons, menus, etc. should ideally be available ready-made for the programmer, and that the object-oriented style of programming is a desirable programming style when programming such applications. This resulted in a C-library called Xt on top of Xlib.

The purpose of Xt is to provide an object-oriented layer that supports user-interface abstractions called widgets. A widget is a reusable, configurable piece of C-code that operates independently of the application except through prearranged interactions.

Xt contains the basic functionality to support widgets, i.e. an architectural model for widgets that allow them to be written and used in an object-oriented fashion. Xt also contains a small core set of widgets.

One of the philosophies behind the X Window System is to be policy-free. It does not support any particular user interface style. The X Window System provides mechanisms to support many different interface styles rather than enforcing any one policy. Xt also attempts to be policy-free.

A widget set is a collection of widgets build on top of Xt that provide commonly used user-interface components tied together with a consistent appearance and user interface. Several different widget sets from various sources exists. The Athena widget set is one example. Others are Motif from Open Software Foundation (OSF) and OPEN WINDOWS from Sun and AT&T.

The BETA interface to Xt et. al.

This report documents the BETA interfaces to the X Toolkit `XtEnv`, to the Athena widget set `AwEnv`, and to OSF/Motif`MotifEnv`. The interfaces consist of several fragments:

- `xlib`: a procedural interface to the low-level library Xlib. It contains an interface to nearly all the functions in Xlib. The interface is C-like: for most C-functions in Xlib there are corresponding BETA-patterns with the same names and enter/exit parameters corresponding to the C-parameters and return values.
- `events`: Contains BETA interface to X Events. Is included by xlib
- `xtlib`: a procedural interface to Xt. This fragment includes xlib.
- `awlib`: a procedural interface to Athena. Includes xtlib.
- `motiflib`: a procedural interface to OSF/Motif. Includes xtlib.
- `xtenv`: an interface to Xt where the abstractions of Xt is modelled in BETA in an object-oriented fashion. Includes xtlib.
- `awenv`: an object-oriented interface to Aw. Includes awlib.
- Various utility patterns for awenv.
- `motifenv`: an object-oriented interface to OSF/Motif. Includes motiflib. Actually motifenv is just the environment for Motif, the interfaces to the individual widgets and gadgets are placed in separate fragmentgroups, which must be included for use. The fragment allmotif includes them all.
- `xsystemenv`: When concurrency is used by means of the SystemEnv pattern described in [MIA 90-08], in conjunction with XtEnv, the XSystemEnv pattern located in the xsystemenv fragment should be used.

xtenv, awenv, motifenv, and xsystemenv are documented in the following chapters of this report.

xlib, xtlib, awlib, and motiflib are not documented as there is a one-one correspondence between these fragments and the corresponding C libraries. There should be no need for ordinary users to call any of the low-level routines in these fragments as there are corresponding documented higher-level operations for most of them.

XtEnv, AwEnv, and MotifEnv mostly contain user interface elements like menus, windows, dialog boxes, etc. The Mjølner System also contains a library called Bifrost with more graphical oriented elements like lines, splines, and circles based on an object-oriented stencil and paint model. This library is documented in [MIA 91-13]. It Should also be noted, that a platform independent graphical user interface library - Lidskjalv - is now recommended instead of using XtEnv/AwEnv/MotifEnv directly. See [MIA 94-27] and [MIA 95-30].

Demo programs

The demo programs presented in this report are also available in a machine readable form. They are located in the directory
`$BETALIB/demo/Xt`
that also contains other demos.

Introduction

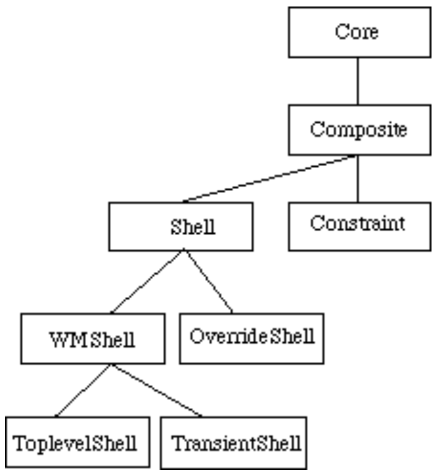
Environment Variables

The standard LD_LIBRARY_PATH environment variable is used in order to link X program correctly. This is given a default value in the Bourne Shell script `$BETALIB/configuration/env.sh`, which are used by, e.g., the compiler. Check the LD_LIBRARY_PATH, and X specific variables like MOTIFHOME and MOTIFINC in this file. If they do not correspond to the organization of the libraries at your site, please have your system administrator correct the default values in this script. Notice, however, that normally you can control the values of these variables yourself, without editing the `env.sh` script: Values that are lists are just appended to by `env.sh`, and variables, that can have just one single value are only set by `env.sh`, if not set already.

XtEnv

This chapter describes the BETA interface to the X toolkit (Xt). Xt contains the basic patterns common for many user-interface toolkits build on top of X, but does not contain any higher level user interface elements. It is typically used together with a widget set containing such user interface elements build on top of it. An example of such a widget set is the Athena Widget set. The BETA interface to the Athena Widget set is described in the next chapter.

The following figure illustrates the inheritance hierarchy among the XtEnv widgets:[1]



[1] Actually this figure is not complete: Above Core, two abstract superclasses called XtObject and RectObj exist. These classes should never be used directly by the user and will not be described in detail here. See the Interface Descriptions for details.

X Libraries - Reference Manual Mjølner Informatics

XtEnv

Using the XtEnv fragment

The basic structure of the xtenv fragment is as follows:

```
ORIGIN 'xtlib';
-- LIB: attributes --
XtEnv: XtLib
  (# <<SLOT xtenvlib: attributes>>;
  ...(* xt widget patterns and other useful patterns *)
  do ...; INNER;  ...
  #)
```

A typical BETA application using xtenv has the following outline:

```
ORIGIN '~beta/Xt/xtenv';
-- PROGRAM: descriptor --
XtEnv
  (#
  do ...
  #)
```

When xtenv starts executing it does some initializing. Afterwards it calls INNER allowing the application program to do its initialization. When the control returns to xtenv a global event handler is started. When an important user interaction event occurs, xtenv distributes the event to virtual patterns local for the user interface object in question of the application program.

Basic XtEnv widget patterns

Core

The core widget pattern is the most fundamental widget pattern, and serves as the superpattern for all other widget patterns. The core pattern defines characteristics common to all widgets, such as geometry, name, parent, sensitiveness, color, callback handling, translations and accelerators. It also defines various converters used to specify some of the other characteristics, e.g. it defines textToFont, used to specify font resources.

Core also defines an initialization method called init for the widget. This method must be called before anything can be done with the widget. It calls INNER such that specializations can add to the initialization behaviour.

init has an optional enter-part. The enter-part consists of two parameters: the father-widget and the name of the widget.

The father-widget is the widget, that shall contain this widget as a child. If the enter-part is not specified, the father-widget will be the enclosing widget of this widget according to BETA's scope-rules, see below. If this widget is not defined within the scope of the intended father-widget, the father-widget has to be specified in the enter-part.

The name of the widget is very important for the internal working of Xt as it is through this name, widgets are accessed. But the name is seldom important for the BETA-programmer. For some widgets the name is used as the default value for some of the widgets attributes. E.g. the name of a menu-item is used as the default value for the item-text presented to user when showing the menu. The name is also used if the programmer wants to change the default-values for some of attributes of some of the widgets used in the application. This can be done via a resourcefile (see [Nye & O'Reilly 90a], chapter 9). If the enter-part is not specified, the name will be the BETA-name of the descriptor for the widget.

Below some elaboration on the default father and default name of a widget is presented.

Consider the following program:

```
ORIGIN '~beta/Xt/awenv';
--- program: descriptor ---
AwEnv
(# faculty: label
     (# init:: (# do 2-> borderwidth #) #);
   University: @box
     (# Physics, Mathematics: @faculty;
        init:: (# do Physics.init; Mathematics.init #);
     #)
do University.init;
#)
```

The idea was that a window with two labels named Physics and Mathematics should appear. But executing it will give the error message

```
Xt Error: There must be only one non-shell widget which is son
of Toplevel. The widget causing the conflict is named faculty.
```

This is because the program uses the init pattern of the widgets without specifying the father and name of the widgets. To be precise, this is what happens: When the init pattern of a widget is invoked, it first checked to see if the father is NONE. This will be the case if no father is specified in the enter part of init.

If so, a search is started in the static environment of the widget pattern. If a specialization of a Core widget is found, this widget is used as the father. This search is continued until a pattern with no enclosing pattern is found. In this case the widget named TopLevel (in xtenv) is used as the father. The widget TopLevel is an instance of the pattern TopLevelShell, which among its characteristics has the constraint that it wants to have exactly one non-shell child.

Now consider the example program: The first thing that happens is that the init attribute of University is invoked. Since no father is specified, a search for one is started from the University pattern. This search finds the pattern AwEnv(#...#), which is not a Core, and which has no enclosing pattern. Thus University will get the father widget TopLevel.

The final binding of University.init then invokes Physics.init. Physics is an instance of the pattern faculty, which is declared in the same scope as University. Thus the search for a father for Physics is identical to the search for the father of University, and Physics also gets TopLevel as its father. This is when the error occurs. The reason why the name reported in the error message is faculty is explained below.

Notice that it did not matter that the instantiation of the Physics object is done within University: the default father is searched for starting from the pattern declaration of the object.

In general there are three possible solutions to the problem in the example:

1. Supply the father and name when initializing the faculty widgets:
   ```
   do ("Physics", University)->Physics.init;
      ("Mathematics", University)->Mathematics.init;
   ```
   In this case, no search for a default father is needed for the faculty widgets.
2. Make (possibly empty) specializations of faculty inside University:
   ```
   Physics: @faculty(##);
   Mathematics: @faculty(##);
   ```
   Now the search for a default father of Physics will start at the pattern faculty(##) inside University, so the University pattern will be the first found in this search, and hence the University widget will become the father of the Physics widget. Likewise for Mathematics.
3. Move the declaration of the faculty pattern inside the University pattern. This will give the same search path as in solution 2. (Conceptually, this might also be the best place to declare faculty in the first place.)

The above example was a simple one. In more complicated cases, the reason for an error of this kind can be trickier to spot. If your program uses the fragment system to move declarations of useful widgets into a library, this kind of error is likely to occur.

Remember that if an instance of an unspecialized widget is used, the widget pattern being declared in, say, the XtEnvLib attributes slot of xtenv, then the search for a default father is started at the XtEnv pattern, and therefore no father widget is found. In this case the widget will get TopLevel as father. Solutions 1 or 2 above will be appropriate in these cases.

The default name used for widgets sometimes also needs special attention: The following BETA program creates a window containing "Label"
```
ORIGIN '~beta/Xt/awenv'
--- program: descriptor ---
AwEnv
(# Hello: @Label;
do Hello.init;
#)
```

whereas the following program creates a window containing "Hello"
```
ORIGIN '~beta/Xt/awenv'
```

```
--- program: descriptor ---
AwEnv
(# Hello: @Label(##);
do Hello.init;
#)
```

Here is the reason why: The connection between the names used for widgets in BETA and the external names used in the external widgets interfaced to from BETA is that the pattern name of the BETA widget is used for the external widget name by default.

In the first example, the Hello widget is an instance of the pattern Label, and in the second example the widget is the only possible instance of the singular pattern Label(##), which is named Hello.

The appearance of the windows in this case comes from the fact that the Athena Label widget uses the external name of the widget as default label-string, if it is not specified otherwise.

A variant of this problem is the case where you specify a list of widgets using the same pattern:

```
hello1, hello2: @Label(##);
```

In this case the default name will always be the first name in the list, hello1. To avoid this behavior, use the scheme

```
hello1: @Label(##);
hello2: @Label(##);
```

or specify the name explicitly instead.

Every widget contains an instance of the pattern eventHandler, which allows the application program to do something when a particular event occurs. It is a low-level facility which is seldom needed as most widget patterns uses callbacks to notify the application-program when something interesting happens. eventhandler has local patterns keypress, keyrelease, buttonpress, ... , exposure, ... that can be used. If an instance of one of these patterns is enabled, it will be invoked, when the corresponding X-event occurs. The eventhandler has a local attribute called event, which is an instance of a virtual pattern, qualified with XAnyEvent (defined in events.bet). XAnyEvent is an abstract super-pattern for all X Events:

```
XAnyEvent: ExternalRecord (* Unspecified event *)
  (# type: @
       (* Type of THIS(XAnyEvent) *)
       long(# pos::< (# do 0->value #)#);
     serial: @
       (* number of last request processed by server *)
       long(# pos::< (# do 4->value #)#);
     send_event: @
       (* true if this came from a SendEvent request *)
       long(# pos::< (# do 8->value #)#);
     display: @
       (* Display the event was read from *)
       long(# pos::< (# do 12->value #)#);
     window: @
       (* window on which event was requested in event mask *)
       long(# pos::< (# do 16->value #)#);

     init:
       (* Used to allocate an XAnyEvent from within BETA *)
       (# do ... #);

     <<SLOT XAnyEventLib: attributes>>
  enter ptr
  exit ptr
  #);
```

The different X events are then modelled as sub-patterns of XAnyEvent, e.g.

```
XButtonEvent: XAnyEvent
  (# root: @
       (* root window that the event occured on *)
       long(# pos::<(# do 20->value#)#);
     subwindow: @
       (* child window *)
       long(# pos::<(# do 24->value#)#);
     time: @
       (* milliseconds *)
       long(# pos::<(# do 28->value#)#);
     x: @
       (* pointer x coordinate in event window *)
       long(# pos::<(# do 32->value#)#);
     y: @
       (* pointer y coordinate in event window *)
       long(# pos::<(# do 36->value#)#);
     x_root: @
       (* x coordinate relative to root *)
       long(# pos::<(# do 40->value#)#);
     y_root: @
       (* y coordinate relative to root *)
       long(# pos::<(# do 44->value#)#);
     state: @
       (* key or button mask *)
       long(# pos::<(# do 48->value#)#);
     button: @
       (* detail *)
       long(# pos::<(# do 52->value#)#);
     same_screen: @
       (* same screen flag *)
       long(# pos::<(# do 56->value#)#);

     shiftModified:
       (# exit { true iff SHIFT was held down } #);
     controlModified:
       (# exit { true iff CONTROL was held down } #);
     ...

     <<SLOT XButtonEventLib: attributes>>
  #);
```

Notice that some event types have some local utility patterns, in XButtonEvent, e.g., shiftModified, which is used to determine if the SHIFT modifier key was held down when the button-event occurred.

See the example-programs draw.bet and drawWithRefresh.bet later in this report for examples of using the eventhandler, and some of the attributes of X Events.

Composite

Composite widgets are intended to be containers for other widgets. They have the ability to manage child widgets and they are responsible for handling the geometry for them.

The Composite widget pattern is an abstract superpattern. It is never instantiated directly in an application, only subpatterns are instantiated.

Constraint

The Constraint widget pattern is a subpattern of the composite pattern, and does thus also manage the layout of children. Constraint widgets let the application provide layout information for each child. This information often takes the form of some constraints on the child's position and/or size. This is a more powerful way to arrange children because it allows you to provide different rules for how each child will be laid out.

The constraint widget is also an abstract superpattern.

Shell

Widgets negotiate their size and positions with their parent widget (i.e. the widget that directly contain them). Widgets at the top of the hierarchy do not have any parent widget. Instead they must deal with the outside world. To provide for this, each top-level widget is encapsulated in a special widget, called a Shell widget.

The Shell widget pattern is a subpattern of the composite widget pattern, but can have only one child. Shell widgets are special purpose widgets that provide an interface between other widgets and the window manager. A shell widget negotiates the geometry of the widget with the window manager, and sets the properties required by the window manager.

The Shell widget pattern is also an abstract superpattern.

WMShell

The WMShell is an abstract super pattern that contains attributes needed by the common window manager protocol, e.g. attibutes used to specify icons, preferred size and aspect ratio etc.

ToplevelShell

ToplevelShell widgets are used for normal top-level windows. xtenv creates the main toplevel widget topLevel of the applications. Some applications have multiple permanent top-level windows, and should use ToplevelShell to accomplish this.

TransientShell

A TransientShell is similar to a toplevel shell except for the way it interacts with the window manager with respect to iconifying. If an applications toplevel window (transientFor) is iconified, the window manager normally iconifies all transient shells created by the application.

OverrideShell

An OverrideShell instructs the window manager to completely ignore it. OverrideShell's completely bypass the window manager and therefore have no added window manager decorations. They are typically used for popup-menus.

VendorShell

The VendorShell is an abstract super class with implementation dependent resources defined by the different widget sets using it.

XtEnv

Other XtEnv Objects and Patterns

Toplevel

TopLevel is the default toplevel widget. It is an instance of ToplevelShell and is automatically initialized by xtenv.

Display

This pattern allows an application to have windows open on more than one workstation at the same time. After a display have been opened, the local item shell of the display can be used as a toplevel widget on that display. When children have been added to shell, the shell must be realized to show the window on screen.

Timer

This pattern allow the application to be notified when a period of time have elapsed, while being able to do other things during that time.

WorkProc

The Workproc-pattern is a means for doing background processing while the application is idle waiting for an event. After a Workproc has been started, the virtual method job is invoked by xtenv when no events are pending. It is invoked through a BETA component, such that a further binding of job may suspend itself in order not to let the user's response time suffer from time consuming background processing. Next time there are no events pending, job is resumed at the suspended place. More than one Workproc can be started at a time.

TranslationTable

Can be used for the 'advanced' programmer to change the translations used by Xt. See [Nye & O'Reilly 90a], chapter 7 for more details.

AcceleratorTable

Can be used for the 'advanced' programmer to change the accelerators used by Xt. See [Nye & O'Reilly 90a], chapter 7 for more details.

RegisteredAction

Used to register so-called actions in Xt. These actions can be used in connection with translations and accelerators. See [Nye & O'Reilly 90a], chapter 7 for more details.

StringArray

This is an array of externally allocated character strings. It is used for various purposes, e.g. to specify the contents of an Athena ListWidget.

FallbackResources

Used to specify "default" resources for an application. These are set before the resource manager of Xt reads user-specified resources, and is thus used to "fall back" to, if no resources are specified by the user, but will still allow such user specified resources to be supplied

Options

Used to specify additional resources to be readable from the command line of the application.

ErrorHandler

An instance of ErrorHandler may be used to catch X errors as normal BETA runtime errors are caught. That is, a dump file is produced, specifying the dynamic call stack, when the error occurred, see [MIA 90-02]. ErrorHandler contains two virtual exceptions xterror and xliberror that can be further bound as any other exception. That is, it is also possible to continue after an X error.

WarningHandler

Like ErrorHandler, but for catching Xt warnings only. Contains the virtual xtwarning.

X Libraries - Reference Manual Mjølner Informatics

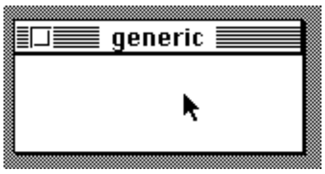
XtEnv

Examples using Xt widgets

The following program illustrates the basic structure of an XtEnv application. It does nothing but popping up a window which is 50 pixels high and 150 pixels wide.

generic.bet
```
ORIGIN '~beta/Xt/xtenv'
--- program : descriptor ---
XtEnv
(# widget: @Core
do widget.init;
   50  -> widget.height;
   150 -> widget.width
#)
```

In a specialization of XtEnv, a Core widget called widget is initialized. This will be a child of toplevel. Values are assigned to some of widget's attributes, whereafter the control returns to XtEnv. In the prefix-part toplevel is realized and XtEnv goes into the basic event loop. The realization of toplevel results in the popping up of the following window:



The decoration of the window - a titlebar with the title and a closebox - depends on the window manager running. The above dump of the window have been made on a Macintosh using the MacX X Window System. MacX contains its own window manager. Other window managers may make other decorations of the window like another type of titlebar or putting a resize-box on.

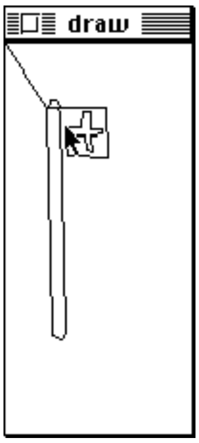
The name used in the titlebar is the name of the application.

Not very many interesting programs can be written using 'pure' XtEnv. The following program illustrates how the low-level Xlib calls can be used together with XtEnv, and it also illustrates how the low-level eventhandler facilities can be used. The demo program is thus for the advanced programmer. The demo program is an interactive drawing program. Every time the user presses the mouse button, a line is drawn.

draw.bet
```
ORIGIN '~beta/Xt/xtenv'
--- PROGRAM: descriptor ---
XtEnv
(# widget: @Core
     (# px,py: @integer (* Used to hold the previous position *);
        gc: @integer
          (* The "Graphics Context" used in the drawings *);
        eventHandler::<
          (* Furtherbind to draw a line to the button press
           * position each time the mouse is pressed.
           *)
          (# bp: @ButtonPress
               (* Called each time the mouse is pressed *)
               (#
               do (display, window, gc, px, py,
                   event.x, event.y) -> XDrawLine;
```

```
                  (event.x,event.y) -> (px,py);
               #);
             init::< (# do bp.enable #);
          #);
        init::< (# do screenResource->XDefaultGCOfScreen->gc #);
     #);
do widget.init;
   200 -> widget.height;
   100 -> widget.width
#)
```

The bp object is called every time the user presses the mouse button. The Xlib functions XDrawLine and XDefaultGCOfScreen are used in the drawing of the line. The program may be used to make drawings like this:



The above program does not handle expose events. If another window is put on top of the above window and afterward moved, the contents of the 'draw' window is erased. The following demo-program is another version of the above program that handles expose-events. It is only necessary to worry about expose-events when drawing by means of low-level Xlib calls.

drawWithRefresh.bet

```
ORIGIN '~beta/Xt/xtenv';
INCLUDE '~beta/containers/list';
--- PROGRAM: descriptor ---
XtEnv
(# widget: @Core
     (# px,py: @integer; (* Previous position *)
        gc: @integer; (* Graphics Context *)

        Line:
          (* Definition of a simple "graphical object",
           * remembering its own state and with the ability
           * to redraw itself.
           *)
          (# x1,y1,x2,y2: @integer; (* End positions *)

             draw:
               (# do (display,window,gc,x1,y1,x2,y2)->XDrawLine #);
             init:
               (# enter ((x1,y1),(x2,y2))
               do THIS(line)[] -> lineList.append
               #);
          #);
        lineList: @List
          (* List of graphical objects to redraw in response to
           * expose events
           *)
          (# element::< line;
             redraw: scan(# do current.draw #)
          #);

        eventHandler::<
          (* Furtherbind to create a new line object each time
           * the mouse is pressed, and to redraw the entire list
           * of lines in response to each expose event.
           *)
          (# bp: @ButtonPress
               (# l: ^Line
               do &Line[] -> l[];
                  ((px,py),(event.x,event.y)) -> l.init;
                  l.draw;
                  (event.x,event.y) -> (px,py);
               #);
             ex: @Exposure(# do lineList.redraw #);
             init::< (# do bp.enable; ex.enable #);
          #);

        init::<
          (#
          do 200 -> height;
             100 -> width;
             screenResource -> XDefaultGCOfScreen -> gc;
             lineList.init
          #);
     #);
do widget.init;
#)
```

X Libraries - Reference Manual Mjølner Informatics

X Libraries - Reference Manual

AwEnv

This chapter describes the BETA interface to the Athena widget set. The Athena widget set is build on top of Xt and it contains user interface elements like scrollbars, commands buttons, menus, etc.

What is the Athena Widget set?

The Athena Widget Set (aw) is developed by MIT's Project Athena. It is included in the standard X Window System distribution and is thus universally available. It contains user-interface components like menus, scrollbars, command buttons and a variety of composite widgets.

The Athena widget set was not intended to be complete - it was built mainly for testing and demonstration of Xt. But it has matured into something quite useful, and it is at the time being the only non-commercial universally available widget set.

Using the AwEnv fragment

The basic structure of the awenv fragment is as follows:

```
ORIGIN 'xtenv';
INCLUDE 'athena/awlib';
-- XtEnvLib: attributes --
... (* aw widget patterns and other useful patterns *)
AwEnv: XtEnv
  (# <<SLOT awenvlib: attributes>>;
  do INNER
  #)
```

A typical BETA application using awenv has the following outline:

```
ORIGIN '~beta/Xt/awenv'
-- program: descriptor --
AwEnv
  (#
  do ...
  #)
```

When xtenv (the prefix of awenv) starts executing, it does some initializing. Afterwards INNER is called allowing the application program to do its initialization. When the control returns to xtenv, a global event handler is started. When an important user interaction event occurs, xtenv distributes the event to virtual patterns local for the user interface object in question of the application program.

Overview of AwEnv Patterns

AwEnv contains the following patterns:

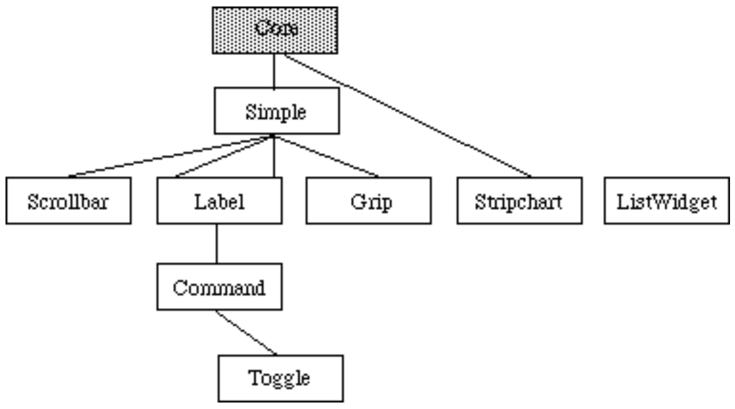
- `Simple`: The base pattern for most of the simple widgets. Provides a rectangular area with a set-able mouse cursor and special border.
- `Label`: A rectangle that may contain one or more lines of text or a bitmap image.
- `Command`: A push button that, when selected, may cause a specific action to take place. This widget can display a multi-line string or a bitmap image.
- `ListWidget`: A list of text strings presented in row column format that may be individually selected. When an element is selected an action may take place.
- `Stripchart`: A real time data graph that will automatically update and scroll.
- `Toggle`: A push button (see Command) that contains state information. Toggles may also be used as radio buttons to implement a 'one of many' group of buttons.
- `Grip`: A rectangle that, when selected, will cause an action to take place.
- `Scrollbar`: A rectangular area containing a thumb that when slided along one dimension may cause a specific action to take place. The Scrollbar may be oriented horizontally or vertically.
- `SimpleMenu`: Provides support for menus.
- `Sme`: (simple menu entry) The base pattern of all menu entries. It may be used as a menu entry itself to provide a blank entry in a menu.
- `SmeBSB`: (simple menu entry with bitmap, string, bitmap). This menu entry provides a selectable entry containing a text string. Support is also provided that allows a bitmap to be placed in the left and right margins.
- `SmeLine`: This menu entry provides an unselectable entry containing a separator line.
- `SmeCascade`: This menu entry provides cascading menu entries.
- `MenuButton`: A push button (see Command) that pops up a SimpleMenu when a button is pressed.
- `Box`: This widget will pack its children as tightly as possible in non-overlapping rows.
- `Dialog`: An implementation of a commonly used interaction semantic to prompt for auxiliary input from the user, such as a filename.
- `Form`: A more sophisticated layout widget that allows the children to specify their positions relative to the other children, or to the edges of the Form.
- `Paned`: Allows children to be tiled vertically or horizontally. Controls are also provided to allow the user to dynamically resize the individual panes.

- `ViewPort`: Consists of a frame, one or two scrollbars, and an inner window. The inner window can contain all the data that is to be displayed. This inner window will be clipped by the frame with the scrollbars controlling which section of the inner window is currently visible.
- `AsciiText`: The AsciiText widget provides a window that will allow an application to display and edit one or more lines of text. Options are provided to allow the user to add Scrollbars to its window, search for a specific string, and modify the text in the buffer.

AwEnv

Simple Athena Patterns

The following figure illustrates the inheritance hierarchy among the simple Athena widgets. Widgets from XtEnv are shaded gray:



Simple

The Simple widget is not very useful by itself, as it has no semantics of its own. Its main purpose is to be used as a common superpattern for the other simple Athena widgets. Provides a rectangular area with a setable mouse cursor and special border.

Label

A Label widget is a text string or bitmap displayed within a rectangular region of the screen. The label may contain multiple lines of characters. The Label widget will allow its string to be left, right, or center justified. Normally, this widget can be neither selected nor directly edited by the user. It is intended for use as an output device only.

Command

The Command widget is an area, often rectangular, that contains a text label or bitmap image. Those selectable areas are often referred to as 'buttons'. When the pointer cursor is on a button, it becomes highlighted by drawing a rectangle around its perimeter. This highlighting indicates that the button is ready for selection. When pointer button 1 is pressed, the Command widget indicates that it has been selected by reversing its foreground and background colours. When the button is released, the Command widget's virtual pattern callback will be invoked. If the pointer is moved out of the widget before the button is released, the widget reverts to its normal foreground and background colours, and releasing the button has no effect. This behaviour allows the user to cancel an action.

ListWidget

The ListWidget contains a list of strings formatted into rows and columns. When one of the strings is selected, it is highlighted, and the ListWidget's virtual pattern callback is invoked. Only one string may be selected at a time.

StripChart

The StripChart widget is used to provide a real time graphical chart of a single value. This widget is used by the xload program to provide the load graph. It will read data from an application, and update the chart at the update interval specified.

Toggle

The Toggle widget is an area, often rectangular, containing a text label or bitmap image. This widget maintains a Boolean state (e.g. True/False or On/Off) and changes state whenever it is selected. When the pointer is on the button, the button may become highlighted by drawing a rectangle around its perimeter. This highlighting indicates that the button is ready for selection. When pointer button 1 is pressed and released, the Toggle widget indicates that it has changed state by reversing its foreground and background colors, and its virtual method callback is invoked. If the pointer is moved out of the widget before the button is released, the widget reverts to its normal foreground and background colors, and releasing the button has no effect. This behavior allows the user to cancel an action.

Toggle buttons may also be part of a radio group. A radio group is a list of at least two Toggle buttons in which no more than one Toggle may be set at any time. A radio group is identified by any one of its members.

Grip

The Grip widget provides a small rectangular region in which user input events (such as ButtonPress or ButtonRelease) may be handled. The most common use for the Grip widget is as an attachment point for visually repositioning an object, such as the pane border in a Paned widget.

Scrollbar

The Scrollbar widget is a rectangular area containing a slide region and a thumb (also known as a slide bar). A Scrollbar can be used alone, as a value generator, or it can be used within a composite widget (for example, a Viewport). A Scrollbar can be oriented either vertically or horizontally.

When a Scrollbar is created, it is drawn with the thumb in a contrasting color. The thumb is normally used to scroll client data and to give visual feedback on the percentage of the client data that is visible.

Each pointer button invokes a specific action. That is, given either a vertical or horizontal orientation, the pointer button actions will scroll or return data as appropriate for that orientation. Pointer buttons 1 and 3 do not move the thumb automatically. Instead, they return the pixel position of the cursor on the scroll region. When pointer button 2 is clicked, the thumb moves to the current pointer position. When pointer button 2 is held down and the pointer is moved, the thumb follows the pointer.

The pointer cursor in the scroll region changes depending on the current action. When no pointer button is pressed, the cursor appears as a double-headed arrow that points in the direction that scrolling can occur. When pointer button 1 or 3 is pressed, the cursor appears as a single-headed arrow that points in the logical direction that the client will move the data. When pointer button 2 is pressed, the cursor appears as an arrow that points to the thumb.

While scrolling is in progress, the application receives notification through callback virtual patterns. For both discrete scrolling actions, the callback gives the pixel position of the pointer when the button was released. For continuous scrolling, the callback routine gives the current relative position of the thumb. When the thumb is moved using pointer button 2, the callback virtual is invoked continuously. When either button 1 or 3 is pressed, the callback virtual is invoked only when the button is released and the further binding of the callback virtual is responsible for moving the thumb.

AwEnv

Examples using simple Athena Widgets

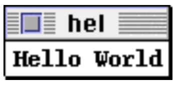
Using Label

The following program illustrates how the 'standard' demo program which writes out the string 'Hello World' can be written using AwEnv. A Label widget is used to show the string to the user.

hello.bet

```
ORIGIN '~beta/Xt/awenv'
--- PROGRAM: descriptor ---
AwEnv
(# hello: @Label;
do hello.init;
   'Hello World' -> hello.label
#)
```

When the program is run the following window is popped up:



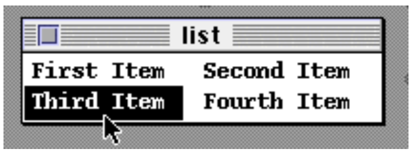
Using ListWidget

The following program illustrates how the ListWidget may be used. It is used to let the user select between four different strings.

list.bet

```
ORIGIN '~beta/Xt/awenv'
--- PROGRAM: descriptor ---
AwEnv
(# alist: @ListWidget
     (# callback::<
          (#
          do 'Item number ' -> screen.putText;
             current.listIndex -> screen.putInt;
             ' have been selected: ''' -> screen.putText;
             current.string -> screen.puttext;
             '''' -> screen.putline;
          #);
        strings::<
          (# init::<
               (#
               do 'First Item '  -> addText;
                  'Second Item ' -> addText;
                  'Third Item '  -> addText;
                  'Fourth Item ' -> addText;
               #)
          #);
     #)
do aList.init;
#)
```

When the program is run, the following window is popped up. When the user selects one of the list items, the callback-pattern in aList is invoked:

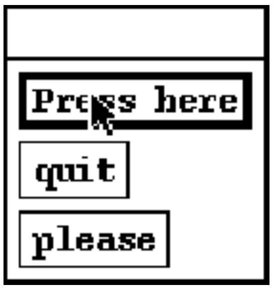


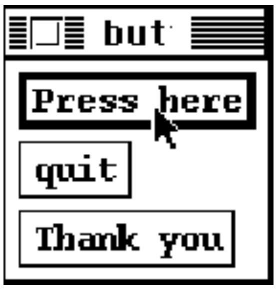
Using Command

The following program illustrates the usage of Command. The program also uses a Box widget.
button.bet

```
ORIGIN '~beta/Xt/awenv'
--- PROGRAM: descriptor ---
AwEnv
(# window: @Box
     (# commandButton: @Command
          (# callback::< (# do 'Thank you' -> please.label #)#);
        quit: @Command(# callback::< (# do stop #)#);
        please: @Label(##);

        init::<
          (#
          do commandButton.init;
             'Press here' -> commandButton.label;
             quit.init;
             please.init;
          #);
     #);
do window.init;
#)
```

The Box widget window contains three local widgets. When the program is run, the window shown to the left is popped up, and when the user clicks on the Press Here button, the window looks as shown to the right:





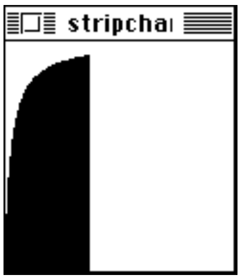
Using StripChart

The following program illustrates the usage of StripChart.

stripchart.bet

```
ORIGIN '~beta/Xt/awenv'
--- PROGRAM: descriptor ---
AwEnv
(# approxOne: @Stripchart
     (# n: @integer;
        getValue::< (# do n+1 -> n; (n,n+3) -> setQuotient #);
        init::< (# do 2 -> update #);
     #);
do approxOne.init;
#)
```

After the program has been running for about one minute, the window looks like this:

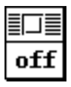


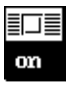
Using Toggle

The following program illustrates the usage of one Toggle.

toggle.bet
```
ORIGIN '~beta/Xt/awenv'
--- PROGRAM: descriptor ---
AwEnv
(# power: @Toggle
     (# callback::<
          (#
          do (if state then
                 'on ' -> label
             else
                 'off' -> label
             if)
          #);
     #);
do power.init;
   'off' -> power.label
#)
```

When the program is run the user get a window which may switch between the following two states:

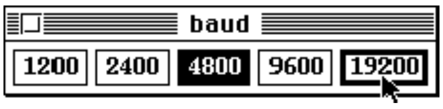


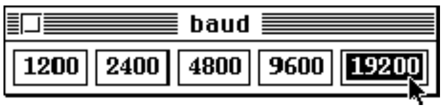


The following program illustrates how Toggles may be grouped into radio groups.

baud.bet
```
ORIGIN '~beta/Xt/awenv'
--- PROGRAM: descriptor ---
AwEnv
(# baudRate: @Box
     (# baud: Toggle
          (# callback::<
               (#
               do radiodata -> screen.putInt;
                  (if state
                   // true  then ' on' -> putLine
                   // false then ' off' -> putLine
                  if)
               #);
          #);
        b1,b2,b3,b4,b5: @baud;
        init::<
          (#
          do b1.init; '1200' -> b1.label; 1200 -> b1.radiodata;

             b2.init; '2400' -> b2.label; 2400 -> b2.radiodata;
             b1 -> b2.radioGroup; b2 -> b1.radioGroup;

             b3.init; '4800' -> b3.label; 4800 -> b3.radiodata;
             b2 -> b3.radioGroup;

             b4.init; '9600' -> b4.label; 9600 -> b4.radiodata;
             b3 -> b4.radioGroup;

             b5.init; '19200'-> b5.label; 19200 -> b5.radiodata;
             b4 -> b5.radioGroup;

             false -> vertical;
             4800 -> b1.setCurrent;
          #);
     #);
do baudRate.init;
#)
```

When the program is run the user get a window where he may select one of the five buttons:





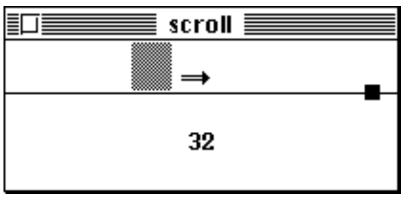
Using ScrollBar

The following program illustrates the usage of a ScrollBar:

scroll.bet

```
ORIGIN '~beta/Xt/awenv'
--- Program: descriptor ---
awenv
(# p: @paned
     (# s: @Scrollbar
          (# jumpProc::< (# do percent -> lab.putInt #);

             scrollProc::<
               (# current: @integer;
               do (topOfThumb*length-position*100) div length
                    -> current;
                  current -> screen.putInt; screen.newLine;
                  (if true
                   // (current<0) then 0 -> current
                   // (current>(100-shown)) then
                      100-shown -> current
                  if);
                  current -> topOfThumb -> lab.putInt;
               #);
          #);
        lab: @Label
          (# putInt:
               (# t: @text; i: @integer
               enter i
               do i -> t.putInt; t[] -> label
               #)
          #);
        init::<
          (#
          do s.init; false -> s.vertical; 100 -> s.length;
             10 -> s.shown; 40 -> s.thickness;
             lab.init; false -> lab.resize; 0 -> lab.putInt;
          #)
     #);
do p.init;
#)
```

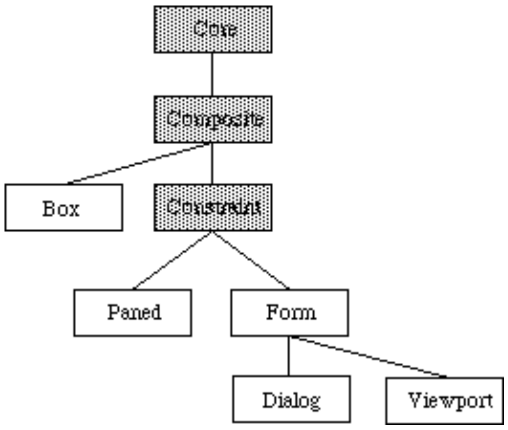
When the program is run the user get the following window where he can select an integer value by means of the scrollbar:



AwEnv

Composite Athena Widgets

The following figure illustrates the inheritance hierarchy among the composite Athena widgets. Widgets from XtEnv are shaded gray:



Box

The Box widget provides geometry management of arbitrary widgets in a box of a specified dimension. The children are rearranged when resizing events occur either on the Box or its children, or when children are managed or unmanaged. The Box widget always attempts to pack its children as tightly as possible within the geometry allowed by its parent.

Box widgets are commonly used to manage a related set of buttons and are often called ButtonBox widgets, but the children are not limited to buttons. The Box's children are arranged on a background that has its own specified dimensions and colour.

Form

The Form widget can contain an arbitrary number of children or subwidgets. The Form provides geometry management for its children, which allows individual control of the position of each child. Any combination of children can be added to a Form. The initial positions of the children may be computed relative to the positions of other children. When the Form is resized, it computes new positions and sizes for its children. This computation is based upon information provided for each child.

The default width of the Form is the minimum width needed to enclose the children after computing their initial layout, with a margin of defaultDistance at the right and bottom edges. If a width and height is assigned to the Form that is too small for the layout, the children will be clipped by the right and bottom edges of the Form.

Extra attributes for children of Form

Each child of the Form widget has a set of operations available to control the layout. These attributes allow the Form widget's children to specify individual layout requirements. Available attributes include the patterns fromHoriz, fromVert, horizDistance, vertDistance, resizable, bottom, left, right, top, see the Interface Descriptions for details.

The above mentioned patterns are actually attributes of the Core pattern. But they only function and should only be used for aggregation components of a Form.

Form's Layout semantics

The Form widget uses two different sets of layout semantics. One is used when initially laying out the buttons. The other is used when the Form is resized.

The first layout method uses the fromVert and fromHoriz attributes to place the children of the Form. A single pass is made through the Form widget's children in the order that they were created. Each child is then placed in the Form widget below or to the right of the widget specified by the fromVert and fromHoriz attributes. The distance the new child is placed from its left or upper neighbour is determined by the horizDistance and vertDistance attributes. This implies some things about how the order of creation affects the possible placement of the children.

The second layout method is used when the Form is resized. It does not matter what causes this resize, and it is possible for a resize to happen before the widget becomes visible (due to constraints imposed by the parent of the Form). This layout method uses the bottom, top, left, and right attributes. These attributes are used to determine what will happen to each edge of the child when the Form is resized. If a value of Chain<something> is specified, the edge of the child will remain a fixed distance from the chain edge of the form. For example if ChainLeft is specified for the right attribute of a child then the right edge of that child will remain a fixed distance from the left edge of the widget. If a value of Rubber is specified, that edge will grow by the same percentage that the Form grew. For instance if the Form grows by 50% the left edge of the child (if specified as Rubber will be 50% farther from the left edge of the Form). One must be very careful when specifying these attributes, for when they are specified incorrectly children may overlap or completely occlude other children when the Form widget is resized.

Dialog

The Dialog widget implements a commonly used interaction semantic to prompt for auxiliary input from a user. For example, you can use a Dialog widget when an application requires a small piece of information, such as a filename, from the user. A Dialog widget, which is simply a special case of the Form widget, provides a convenient way to create a preconfigured Form.

The typical Dialog widget contains three areas. The first line contains a description of the function of the Dialog widget, for example, the string 'Filename:'. The second line contains an area into which the user types input. The third line can contain buttons that let the user confirm or cancel the Dialog input. Any of these areas may be omitted by the application.

Extra attributes for children of Dialog:

Children of Dialog have the same extra attributes as children of Form.

Paned

The Paned widget manages children in a vertically or horizontally tiled fashion. The panes may be dynamically resized by the user by using the grips that appear near the right or bottom edge of the border between two panes.

The Paned widget may accept any widget pattern as a pane except Grip. Grip widgets have a special meaning for the Paned widget, and adding a Grip as its own pane will confuse the Paned widget.

The grips allow the panes to be resized by the user. The semantics of how these panes resize is somewhat complicated, and warrants further explanation here. When the mouse pointer is positioned on a grip and pressed, an arrow is displayed that indicates the pane that is to be to be resized. While keeping the mouse button down, the user can move the grip up and down (or left and right). This, in turn, changes the size of the pane. The size of the Paned widget will not change. Instead, it chooses another pane (or panes) to resize.

Extra attributes for children of Paned

Each child of the Paned widget has a set of operations available to control the layout. These attributes allow the Paned widget's children to specify individual layout requirements.

Available attributes include the patterns allowResize, maxSize, minSize, preferredPaneSize, resizeToPreferred, showGrip, and skipAdjust. See the Interface Descriptions for details.

The above mentioned patterns are actually attributes of the Core pattern. But they only function and should only be used for aggregation components of a Paned widget.

ViewPort

The Viewport widget consists of a frame window, one or two Scrollbars, and an inner window. The size of the frame window is determined by the viewing size of the data that is to be displayed and the dimensions to which the Viewport is created. The inner window is the full size of the data that is to be displayed and is clipped by the frame window. The Viewport widget controls the scrolling of the data directly. No application code are required for the scrolling.

When the geometry of the frame window is equal in size to the inner window, or when the data does not require scrolling, the ViewPort widget automatically removes any scrollbars. The forceBar option causes the Viewport widget to display all scrollbars permanently.

Layout Semantics

The Viewport widget manages a single child widget. When the size of the child is larger than the size of the Viewport, the user can interactively move the child within the Viewport by repositioning the scrollbars.

The default size of the Viewport before it is realized is the width and/or height of the child. After it is realized, the Viewport will allow its child to grow vertically or horizontally if allowVert or allowHoriz are set, respectively. If the corresponding vertical or horizontal scrollbar is not enabled, the Viewport will propagate the geometry request to its own parent and the child will be allowed to change size only if the Viewport's parent allows it. Regardless of whether or not scrollbars are enabled in the corresponding direction, if the child requests a new size smaller than the Viewport size, the change will be allowed only if the parent of the Viewport allows the Viewport to shrink to the appropriate dimension.

Although the Viewport is a subpattern of the Form, none of the attributes for the children of Form mentioned on above must be supplied for any of the children of the Viewport. These attributes are managed internally, and are not meant for public consumption.

AwEnv

Using composite Athena Widgets

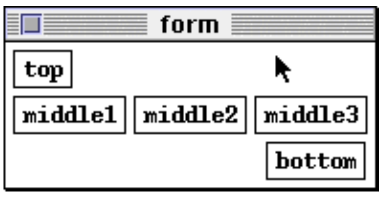
Using Form

The following program illustrates the usage of Form widgets and how layout are controlled for the children of the Form.

form.bet

```
ORIGIN '~beta/Xt/awenv'
--- PROGRAM: descriptor ---
AwEnv
(# window: @Form
     (# top: @Command(##);
        middle1: @Command(##);
        middle2: @Command(##);
        middle3: @Command(##);
        bottom:  @Command(##);

        init::<
          (#
          do top.init;
             middle1.init;
             middle2.init;
             middle3.init;
             bottom.init;

             top -> middle1.fromVert;

             top -> middle2.fromVert;
             middle1 -> middle2.fromHoriz;

             top -> middle3.fromVert;
             middle2 -> middle3.fromHoriz;

             middle1 -> bottom.fromVert;
             middle2 -> bottom.fromHoriz;
             10 -> bottom.horizDistance;
          #);
     #);
do window.init;
#)
```

Running the program results in the following window:

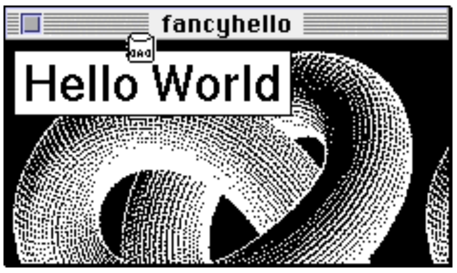


AwEnv

Cursors, Fonts and PixelMaps

The following program illustates how fonts, cursors, and pixelmaps can be used
fancyhello.bet
```
ORIGIN '~beta/Xt/awenv'
--- PROGRAM: descriptor ---
awenv
(# main: @Box
     (# lab: @Label
          (# init::<
               (#
               do '*Helvetica-Bold-R-Normal--24-*' -> textToFont
                    -> font;
                  XCcoffeemug -> symbolToCursor -> cursor;
                  'Hello World' -> label
               #)
          #);
        init::<
          (#
          do lab.init;
             '/usr/include/X11/bitmaps/escherknot'
               -> fileToPixmap -> backgroundPixmap
          #)
     #)
do main.init;
#)
```

Running the program gives the following window, notice the "Coffee mug" cursor:



X Libraries - Reference Manual Mjølner Informatics

AwEnv

Menus

The Athena widget set provides support for single paned popup and pulldown menus. Menus are implemented as a Menu container (the SimpleMenu widget) and a collection of objects that will comprise the menu entries. The SimpleMenu widget is itself a direct subpattern of the OverrideShell widget pattern, therefore no other shell is necessary when creating a menu. The children of a SimpleMenu must be subpatterns of the Sme (Simple Menu Entry) object.

The Athena widget set provides four patterns of Sme objects that may be used to build menus.

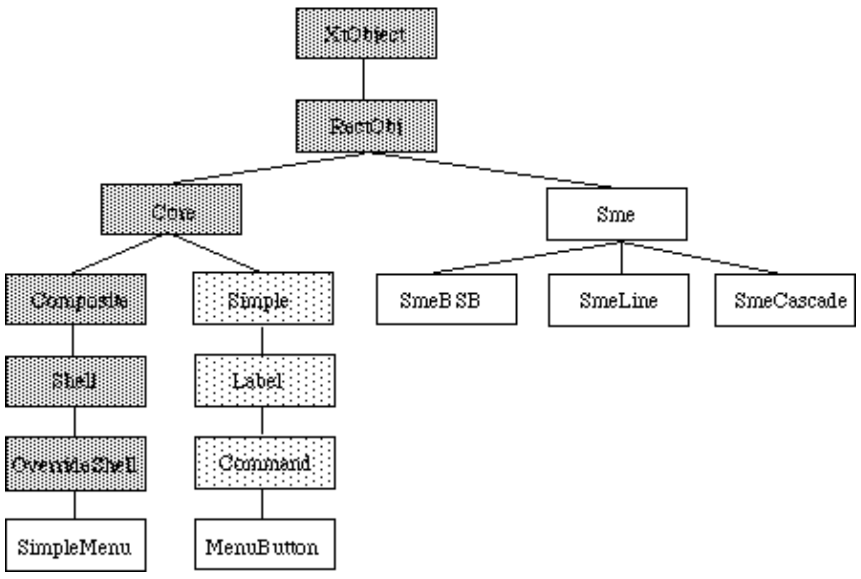
- `Sme`. The base pattern of all menu entries. It may be used as a menu entry itself to provide a blank entry in a menu.
- `SmeBSB`. This menu entry provides a selectable entry containing a text string. Support is also provided that allows a bitmap to be placed in the left and right margins.
- `SmeLine`. This menu entry provides an unselectable entry containing a separator line.
- `SmeCascade`. This menu entry provides cascading menu entries.

To allow easy creation of pulldown menus, a MenuButton widget is also provided.

The default configuration for the menus is click-move-release, and it is this interface that will be described. The menus will typically be activated by clicking a pointer button while the pointer is over a MenuButton, causing the menu to appear in a fixed location relative to that button; this is a pulldown menu. Menus may also be activated when a specific pointer and key sequence is used anywhere in the application; this is a popup menu. In this case the menu will position itself under the cursor. Typically menus will be placed so the pointer cursor is on the first menu entry, or the last entry selected by the user.

The menu will remain on the screen as long as the pointer button is held down. Moving the pointer will highlight different menu items. If the pointer leaves the menu, or moves over an entry that cannot be selected then no menu entry will highlighted. When the desired menu entry has been highlighted, release the pointer button to remove the menu, and cause the callback associated with this entry to be invoked.

The following figure illustrates the inheritance hierarchy among the menu-related Athena widgets. Widgets from xtenv are shaded gray and other Athena widgets are dotted. Notice that Sme is a specialization of the abstract superclass RectObj of Core. This means that Sme is not a widget, but what is called a "gadget", i.e. a "windowless widget" that uses its parent window to draw in. For normal use this destinction is of no concern.


XtObject
RectObj
Core
Sme
Composite
Simple
SmeBSB
SmeLine
SmeCascade
Shell
Label
OverrideShell
Command
SimpleMenu
MenuButton

SimpleMenu

The SimpleMenu widget is a container for the menu entries. It is a direct subpattern of OverrideShell. This is the only part of the menu that actually contains a window. The SimpleMenu serves as the glue to bind the individual menu entries together into a menu.

MenuButton

The MenuButton widget is an area, often rectangular, that contains a text label or bitmap image. When the pointer cursor is on the button, the button becomes highlighted by drawing a rectangle around its perimeter. This highlighting indicates that the button is ready for selection. When a pointer button is pressed, the MenuButton widget will pop up a menu.

Sme

The Sme (simple menu entry) object is the base pattern for all menu entries. While this object is mainly intended to be specialized, it may be used in a menu to add blank space between menu entries. The name Blank may be used as an alias for an Sme without specialization.

SmeBSB

The SmeBSB (simple menu entry composed of a bitmap, a string and a bitmap) object is used to create a menu entry that contains a string, and optional bitmaps in its left and right margins. Since each menu entry is an independent object, the application is able to change the font, colour, height, and other attributes of the menu entries, on an entry by entry basis. The name Item may be used as an alias for an SmeBSB without specialization.

SmeLine

The SmeLine object is used to add a horizontal line or menu separator to a menu. Since each menu entry is an independent object, the application is able to change the colour, height, and other attributes of the menu entries, on an entry by entry basis. This entry is not selectable, and will not highlight when the pointer cursor is over it. The name Line may be used as an alias for an SmeLine without specialization.

SmeCascade

The SmeCascade object is used to add a cascading submenu to another menu. When the user points inside this item, the submenu is popped of to the right of the item. The user can then select among the items of the cascaded menu. The name Cascade may be used as an alias for an SmeCascade without specialization.

Since each menu entry is an independent object, the application is able to change the colour, height, and other attributes of the menu entries, on an entry by entry basis.

AwEnv

Examples of using Menus

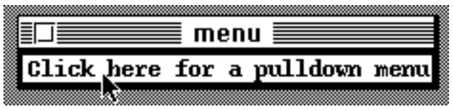
A SimpleMenu example

The following program demonstrates how the SimpleMenu and the Sme widgets can be used. The menu is activated from a MenuButton widget.

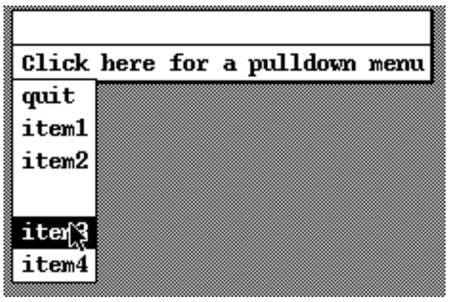
menu.bet

```
ORIGIN '~beta/Xt/awenv'
--- PROGRAM: descriptor ---
AwEnv
(# command: @MenuButton
     (# theMenu: @SimpleMenu
          (# item1,item2,item3,item4: @SmeBSB
               (# callback::<
                    (# do label -> putText; ' selected'->putLine #);
               #);

             quit: @Item(# callback::< (# do stop #)#);

             init::<
               (# bl: ^Blank;
               do quit.init;
                  item1.init;
                  'item1' -> item1.label;
                  item2.init;
                  'item2' -> item2.label;
                  &blank[] -> bl[]; bl.init;
                  20 -> bl.height;
                  item3.init;
                  'item3' -> item3.label;
                  item4.init;
                  'item4' -> item4.label;
               #);
          #);

        init::<
          (#
          do 'Click here for a pulldown menu' -> label;
             theMenu.init;
             theMenu[] -> setMenu;
          #);
     #);
do command.init;
#)
```

When the program is run, the following window is popped up:



when the users clicks with the left mouse-button, the following menu is popped up:


Click here for a pulldown menu
quit
item1
item2
item3
item4

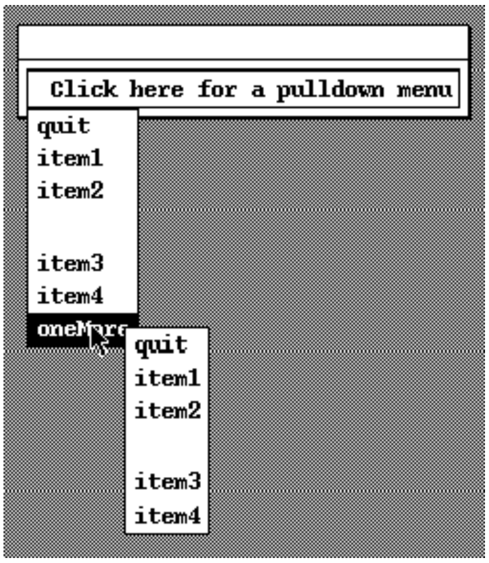
Using cascading menus

The following program is an extension of the previous, where a cascade menu item has been added.

cascademenu.bet

```
ORIGIN '~beta/Xt/awenv'
--- PROGRAM: descriptor ---
AwEnv
(# main: @Box
     (# theMenuButton: @MenuButton
          (# init::<
               (#
               do firstMenu.init;
                  firstMenu[] -> setmenu;
                  'Click here for a pulldown menu' -> label;
               #)
          #);
        Menu: SimpleMenu
          (# item1,item2,item3,item4: @SmeBSB
               (# callback::<
                    (#
                    do name -> putText; ' selected' -> putLine
                    #)
               #);
             quit: @SmeBSB(# callback::< (# do stop #)#);
             init::<
               (# bl: ^blank;
               do quit.init;
                  ('item1', THIS(Menu)) -> item1.init;
                  ('item2', THIS(Menu)) -> item2.init;
                  &blank[] -> bl[]; bl.init;
                  20 -> bl.height;
                  ('item3', THIS(Menu)) -> item3.init;
                  ('item4', THIS(Menu)) -> item4.init;
                  INNER;
               #);
          #);
        firstMenu: @Menu
          (# oneMore: @SmeCascade
               (# theMenu: @Menu;
                  init::<
                    (# do theMenu.init; theMenu -> subMenu #)
               #);
             init::< (# do oneMore.init #)
          #);
        init::<
          (# do theMenuButton.init; #);
     #);
do main.init;
#)
```

When the cascading menu entry is selected, the window looks like this:



X Libraries - Reference Manual

[Mjølner Informatics](#)

AwEnv

AsciiText - the Text Editor of Athena

The AsciiText widget provides a window that will allow an application to display and edit one or more lines of text. Options are provided to allow the user to add Scrollbars to its window, search for a specific string, and modify the text in the buffer.

The word insert point is used in this chapter to refer to the text caret. This is the caret that is displayed between two characters in the text. The insert point marks the location where any new characters will be added to the text, i.e. the i'th position in an AsciiText is after the caret at position i. To avoid confusion the pointer cursor will always be referred to as the pointer.

The AsciiText widget supports three edit modes, controlling the types of modifications a user is allowed to make:

- Append-only
- Editable
- Read-only

Read-only mode does not allow the user or the programmer to modify the text in the widget. While the programmer may reset the entire string in read-only mode, it may not modify parts of the text with Replace. Append-only and editable modes allow the text at the insert point to be modified. The only difference is that text may only be added to or removed from the end of a buffer in append-only mode.

AsciiText Widget for Users

The AsciiText widget provides many of the common keyboard editing commands. These commands allow users to move around and edit the buffer. If an illegal operation is attempted, (such as deleting characters in a read-only text widget), the terminal bell will be rung.

The default key bindings are patterned after those in the Emacs text editor:

Default Key Bindings

Ctrl-a
Beginning Of Line
Meta-b
Backward Word
Ctrl-b
Backward Character
Meta-f
Forward Word
Ctrl-d
Delete Next Character
Meta-i
Insert File
Ctrl-e
End Of Line
Meta-k
Kill To End Of Paragraph
Ctrl-f
Forward Character
Meta-q
Form Paragraph
Ctrl-g
Multiply Reset
Meta-v
Previous Page
Ctrl-h
Delete Previous Character
Meta-y
Insert Current Selection
Ctrl-j
Newline And Indent
Meta-z
Scroll One Line Down
Ctrl-k
Kill To End Of Line
Meta-d
Delete Next Word
Ctrl-l
Redraw Display
Meta-D
Kill Word
Ctrl-m
Newline
Meta-h
Delete Previous Word
Ctrl-n
Next Line
Meta-H
Backward Kill Word

Ctrl-o
Newline And Backup
Meta-<
Beginning Of File
Ctrl-p
Previous Line
Meta->
End Of File
Ctrl-r
Search/Replace Backward
Meta-]
Forward Paragraph
Ctrl-s
Search/Replace Forward
Meta-[
Backward Paragraph
Ctrl-t
Transpose Characters

Ctrl-u
Multiply by 4
Meta-Delete
Delete Previous Word
Ctrl-v
Next Page
Meta-Shift Delete
Kill Previous Word
Ctrl-w
Kill Selection
Meta-Backspace
Delete Previous Word
Ctrl-y
Unkill
Meta-Shift Backspace
Kill Previous Word
Ctrl-z
Scroll One Line Up

In addition, the pointer may be used to cut and paste text:

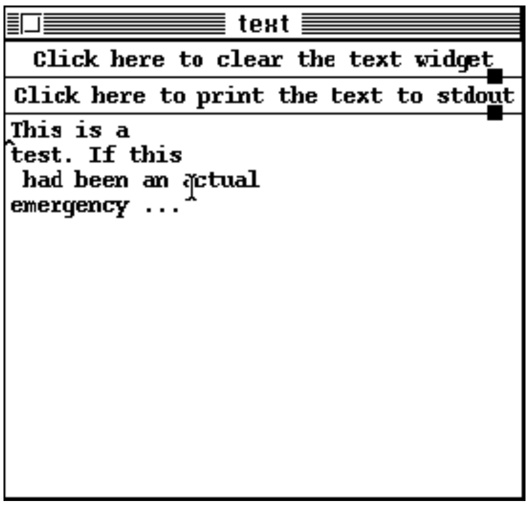
Button 1 Down
Start Selection
Button 1 Motion
Adjust Selection
Button 1 Up
End Selection (cut)

Button 2 Down
Insert Current Selection (paste)
Button 3 Down
Extend Current Selection
Button 3 Motion
Adjust Selection
Button 3 Up
End Selection (cut)

Example using AsciiText

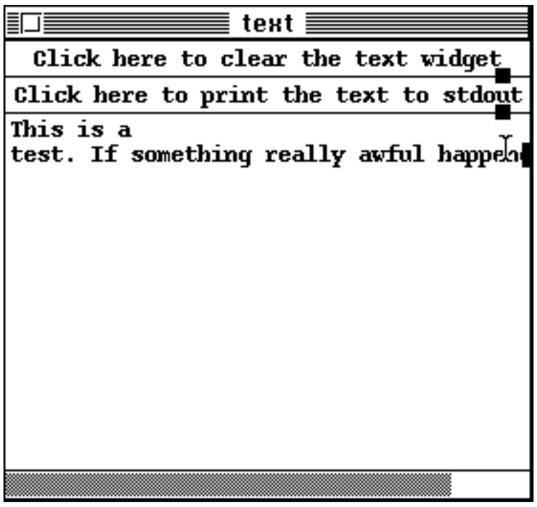
The following program illustrates the usage of AsciiText:
text.bet
```
ORIGIN '~beta/Xt/awenv'
--- PROGRAM: descriptor ---
AwEnv
(# aPane: @Paned
     (# clear: @Command
          (# callback::< (# do '' -> txt.string #)#);
        print: @Command
          (# callback::<
               (#
               do 'The text is "' -> screen.putText;
                  txt.string -> screen.putText;
                  '" ' -> screen.putLine;
               #);
          #);
        txt: @AsciiText
          (#
             init::<
               (#
               do 200 -> preferredPaneSize;
                  edit -> editType;
                  ScrollWhenNeeded -> scrollVertical;
                  ScrollWhenNeeded -> scrollHorizontal;
                  false -> autoFill;
                  'This is a \ntest. If this\n had been an
 actual\nemergency ...' -> string
          #)#);
        init::<
          (#
          do clear.init;
             'Click here to clear the text widget'
               -> clear.label;
             print.init;
             'Click here to print the text to stdout'
               -> print.label;
             txt.init;
          #);
     #);
do aPane.init;
#)
```

When the program text is run, the following window pops up on the screen:


text
Click here to clear the text widget
Click here to print the text to stdout

```
This is a
test. If this
 had been an actual
emergency ...
```

The user may edit the contents of the last window to i.e.:


text
Click here to clear the text widget
Click here to print the text to stdout

```
This is a
test. If something really awful happen
```

Prompts for Athena Widgets

The Dialog Widget provided by the Athena Widget set is not very useful in itself. It just contains an OK button, an optional Cancel button and an optional text entry field. The user of it must supply a Shell widget for father, and have to set various attributes to get even the simplest well known dialog boxes to work. Therefore three often used dialog boxes have been included in the BETA interface to the Athena Widgets. These are build up by using various other widgets from the Athena Widget set, including, of course, the Dialog widget.

These dialogs are:

- `Prompt`: A pattern which sets up a dialogbox containing a message and an OK button, which when clicked, dismisses the Prompt. Typing <RETURN> is the same as clicking on the OK button. The message and the text of the OK button may be set by the programmer. Also a small icon can be displayed in the Prompt if supplied by the programmer.

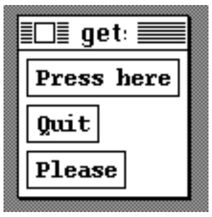
- `PromptForString`: A specialization of Prompt which sets up a dialogbox that asks the user to enter a text. It contains a message, a text entry field an OK and optionally a Cancel button. The pointer need not be inside the text entry field when typing. Typing <RETURN> is the same as clicking on OK, and typing <ESCAPE> is the same as clicking on Cancel. The message and the text of the OK and Cancel buttons may be set by the programmer.

- `PromptForBoolean`: A specialization of Prompt which sets up a dialogbox that asks the user to enter a boolean. It contains a message, two buttons for the answer an optionally a Cancel button. Typing <RETURN> is the same as clicking on the OK button, and typing <ESCAPE> is the same as clicking on Cancel. Also the first letter (in either case) of the labels of the OK and No buttons, respectively, may be typed to choose that button. The message and the text of the three buttons may be set by the programmer.

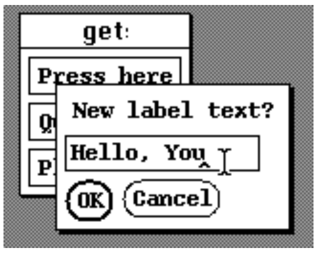
Example using Athena Prompts

The following program illustrates the use of PromptForString, PromptForBoolean and Prompt.
getstring.bet
```
ORIGIN '~beta/Xt/athena/prompts'
--PROGRAM: descriptor--

AwEnv
(# window: @Form
     (# h,v: @integer; (* Used to position the dialogs *)
        commandButton: @Command
          (# callback::<
               (#
               do (x+10,y+10) -> translatecoords -> (h,v);
                  (h, v, 0, 'New label text?',
                   'OK', 'Cancel', string.label)
                    -> PromptForString
                         (# ok::<(# do value[]->string.label #)#)
               #)
          #);
        quit: @Command
          (# callback::<
               (#
               do (x+10,y+10) -> translatecoords -> (h,v);
                  (h,v,0,'Print string before stopping?',
                   'Yes','No','Cancel')
                    -> PromptForBoolean
                         (# ok::<
                              (# do string.label->putline; stop #);
                            no::<
                              (# do stop #);
                            cancel::<
                              (#
                              do (h,v,0,'We''ll continue!','OK')->
                                    Prompt
                                      (# doubleBorder::<trueObject#)
                              #);
                            doubleBorder::< trueObject;
                         #);
               #)
          #);
        string: @Label;
        init::<
          (#
          do commandButton.init;
             'Press here' -> commandButton.label;
             quit.init; 'Quit' -> quit.label;
             (* Specify 'quit' to be below 'commandButton' *)
             commandButton -> quit.fromVert;
             string.init; 'Please' -> string.label;
             (* Specify that 'string' should be below 'quit' *)
             quit -> string.fromVert;
             (* Make 'window' accept that 'string' resizes itself *)
             true -> string.resizable;
             (* Toplevel is the shell used by 'window' by default.
              * Allow it to resize itself if its children do.
              *)
             true -> toplevel.allowShellResize;
          #);
     #);
do window.init
#)
```

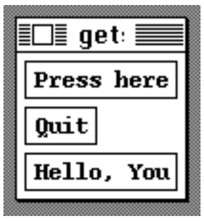
When this program is run, the following window appears:



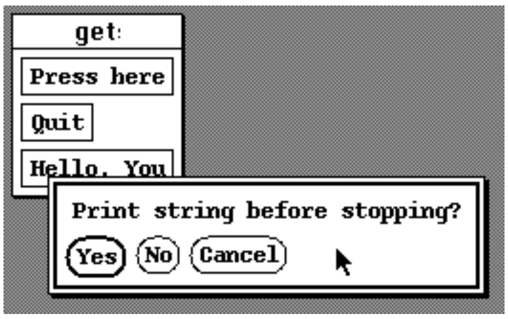
When the button with the label 'Press Here' is pressed, a PromptForString is instantiated and popped up. The PromptForString is popped up a distance (10,10) pixels inside the 'Press Here' button. This is obtained by translating these coordinates into global coordinates, using translatecoords. On the screen, it looks as follows:



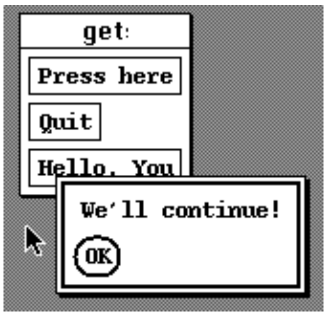
If the text 'Hello, You' is entered in the dialog, and the OK button is pressed (or <RETURN> is typed - this will invoke the OK button), the dialog disappears, and the Label named string will be changed. Then the main window will look as follows:



If then the Quit button is pressed, a PromptForBoolean is instantiated, asking if string should be printed before quiting:



If the Yes button is pressed, string.label is printed on the terminal, and the application stops. If the No button is pressed, the application just stops. But if the Cancel button is pressed, the application continues, i.e., quitting is cancelled. In this case a simple Prompt is used to inform the user of that:



MotifEnv

This chapter describes the BETA interface to OSF/Motif. OSF/Motif is build on top of Xt and it contains user interface elements like scrollbars, buttons, standard dialogs, menus, etc. It also supports multi-font strings and gadgets.

What is OSF/Motif?

The Motif widget set contains many user-interface components, including scroll bars, menus, buttons, dialogs, and a wide variety of composite widgets.

The most noticeable characteristics of the Motif Widget set is its three-dimensional (3-D) appearance. For instance buttons appear to be pushed in, when the user clicks on them. Most Motif widgets and gadgets draw a border around itself. The top and left sides of this border can be set to one color, and the right and bottom sides to another. By setting these sides to appropriate colors a 3-D shading effect can be obtained. These colors can be set directly by the programmer, but Motif will by default generate appropriate colors automatically based on the background color.

Motif has conventions about the use of its widgets and gadgets, that lead to a consistent look among all applications using Motif. Along with each Motif licence comes an OSF/Motif Style Guide with the documentation. This document contains recomandations for application design and layout.

Another notable feature of Motif is its resolution independence mechanism. All sizes and dimensions used by Motif can be specified in terms of pixels, multiples of 1/1000 of an inch, multiples of 1/100 of a point, multiples of 1/100 of a millimeter, or multiples of 1/100 of a font size. The default is to use pixels a the unit for sizes and dimensions.

OSF/Motif also supports some advanced features for internationalization such as language-independent so-called compound strings, keyboard traversal (moving around widgets and gadgets with keyboard keys rather than the mouse), and mnemonic key equivalents for invoking menu items.

OSF/Motif also supplies the mwm Motif Window Manager, which is ICCCM compliant; which decorates the windows with title bar, iconify button, help button, resize grips and which has even more features.

Using the MotifEnv Fragment

The motifenv fragment group simply defines the MotifEnv prefix for applications using the BETA interface to Motif. For each widget/gadget wanted in the application, the fragment group in the sub-directory motif, defining the BETA interface to it, must be explicitly included.

```
ORIGIN 'xtenv';
INCLUDE 'motif/basics';
-- LIB: attributes --
MotifEnv: XtEnv
  (# <<SLOT MotifEnvLib: attributes>>
  do INNER
  #)
```

An application using motifenv thus have the following outline:

```
ORIGIN  '~beta/Xt/MotifEnv'
INCLUDE '~beta/Xt/motif/rowcolumn'
INCLUDE '~beta/Xt/motif/pushbutton'
-- PROGRAM: descriptor --
MotifEnv
  (# ...
  do ...
  #)
```

In this case the program is using the RowBolumn- and PushButton widgets.

For ease of use, the fragment group allmotif includes the interface to all the widgets/gadgets, and may be used instead of motifenv fragment group at the price of a slightly bigger executable.

X Libraries - Reference Manual Mjølner Informatics

MotifEnv

Basic MotifEnv Patterns

The following are some basic patterns defined in MotifEnv, which are used in most other MotifEnv patterns. They define various Motif specific extensions to XtEnv.

- `MotifCallback`: Unlike simple XtEnv callbacks, most Motif Callbacks returns data relevant for the callback along with the callback. The data returned is described using a specialization of the pattern called XmAnyCallbackStruct. The MotifCallback pattern defines a virtual pattern qualified by XmAnyCallbackStruct, and a static instance data MotifCallback is used as prefix for all the different Motif callbacks.

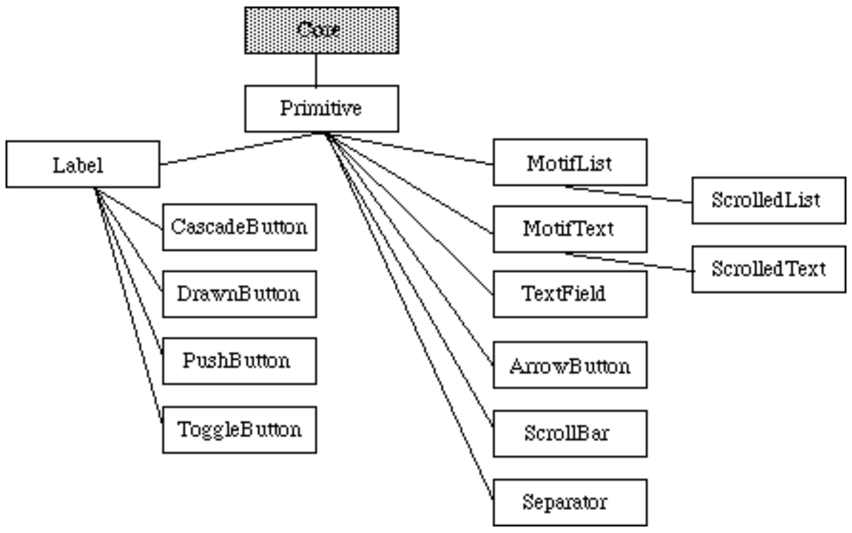
- `MotifString`: This is the BETA interface to the so-called Compound Strings of OSF/Motif. A compound string is composed of text-segments, each having a Character Set (defining the font to use), a Direction (left-to-right or right-to-left), and the String consituting the text to show.

- `MotifStringArray`: This is the BETA interface to an externally allocated array of MotifStrings, used in, e.g., MotifLists.

- `MotifFontList`: Used, e.g., to define Character Sets for MotifStrings.

Primitive Motif Widgets

The following figure shows the patterns constituting the primitive MotifEnv patterns. Patterns from XtEnv are shaded gray.



Notice that the Motif List and and Text widgets are modelled by patterns called MotifList and MotifText, respectively, to avoid confusing them with corresponding patterns of the basic BETA libraries. The ScrolledList and ScrolledText patterns do not correspond directly to existing Motif widgets, but are supplied for convenience.

The following gives a brief overview of the Primitive widgets. The text editor widgets are described later:

- `Primitive`: used as a supporting superpattern for other widget patterns. It handles border drawing and highlighting, traversal activation and deactivation, and various callbacks needed by other widgets.

- `ArrowButton`: consists of a directional arrow surrounded by a border shadow. When it is selected, the shadow changes to give the appearance that the ArrowButton has been pressed in. When the ArrowButton is unselected, the shadow reverts to give the appearance that the ArrowButton is released, or out.

- `Separator`: a primitive widget that separates items in a display. Several different line drawing styles are provided, as well as horizontal or vertical orientation.

- `ScrollBar`: consists of two arrows placed at each end of a rectangle. The rectangle is called the scroll region. A smaller rectangle, called the slider, is placed within the scroll region. The data is scrolled by clicking either arrow, selecting on the scroll region, or dragging the slider. When an arrow is selected, the slider within the scroll region is moved in the direction of the arrow by an amount supplied by the application. If the mouse button is held down, the slider continues to move at a constant rate.

- `MotifList`: allows a user to select one or more items from a group of choices. Items are selected from the list in a variety of ways, using both the pointer and the keyboard. MotifList operates on a StringArray that is defined by the application. Each string becomes an item in the MotifList, with the first string becoming the item in position 1, the second string becoming the item in position 2, and so on.

- `ScrolledList`: Utility pattern used to instantiate a MotifList within a ScrolledWindow.

- `Label`: can contain either a MotifString or a pixmap. When a Label is insensitive, its text is stippled, or the user-supplied insensitive pixmap is displayed.

- `CascadeButton`: links two MenuPanes or a MenuBar to a MenuPane. It is used in menu systems and must have a RowColumn parent with its rowColumnType resource set to XmMENU_BAR, XmMENU_POPUP or XmMENU_PULLDOWN. It is the only widget that can have a Pulldown MenuPane attached to it as a submenu. The submenu is displayed when this widget is activated within a MenuBar, a PopupMenu, or a PulldownMenu. Its visuals can include a label or pixmap and a cascading indicator when it is in a Popup or Pulldown MenuPane; or, it can include only a label or a pixmap when it is in a MenuBar.

- `DrawnButton`: consists of an empty widget window surrounded by a shadow border. It provides the application developer with a graphics area that can have PushButton input semantics.

- `PushButton`: consists of a text label or pixmap surrounded by a border shadow. When a PushButton is selected, the shadow changes to give the appearance that it has been pressed in. When a PushButton is unselected, the shadow changes to give the appearance that it is out.

- `Togglebutton`: Usually this widget consists of an indicator (square or diamond) with either text or a pixmap on one side of it. However, it can also consist of just text or a pixmap without the indicator. The toggle graphics display a 1-of-many or N-of-many selection state. When a toggle indicator is displayed, a square indicator shows an N-of-many selection state and a diamond indicator shows a 1-of-many selection state. A ToggleButton implies a selected or unselected state. In the case of a label and an indicator, an empty indicator (square or diamond shaped) indicates that ToggleButton is unselected, and a filled indicator shows that it is selected. In the case of a pixmap toggle, different pixmaps are used to display the selected/unselected states.

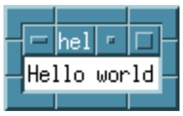
Examples using Primitive Motif Widgets

The following small program shows how to make the traditional "Hello world" program using a Motif Label widget:

hello.bet

```
ORIGIN '~beta/Xt/motifenv';
INCLUDE '~beta/Xt/motif/label';

-- PROGRAM: descriptor --
MotifEnv
(# hello: @Label;
do hello.init;
   'Hello world' -> hello.labelString;
#)
```

When the program is run, the following window appears:



The border of the window, with grips for resizing the window, and the titlebar, with, e.g., iconify button, is added by the window manager, in this case mwm, the Motif Window Manager. The actual Label widget is the one showing the "Hello world" text.[2]

Notice that the Label widget has no 3-D appearance in itself. If that is wanted, a Frame widget could be used as father of the Label.

The following program shows how to construct a list of four items, that the user can select:

list.bet

```
ORIGIN '~beta/Xt/motifenv';
INCLUDE '~beta/Xt/motif/lists';

-- PROGRAM: descriptor --
MotifEnv
(#
   lst: @MotifList
     (#
        browseSelectionCallback::<
          (* Called when an item is selected *)
          (# do (if data.item_position=itemCount then Stop if) #);
        defaultActionCallback::<
          (* Called when an item is double-clicked *)
          (# item: @MotifString;
          do 'The item ''' -> screen.putText;
             data.item -> item; item.getText -> screen.putText;
             ''', at position ' -> screen.putText;
             data.item_position -> screen.putInt;
             ' has been double-clicked' -> screen.putLine;
          #);
        init::<
          (# strings: @MotifStringArray;
          do strings.init;
             'Item 1' -> strings.addText;
             'Item 2' -> strings.addText;
             'Item 3' -> strings.addText;
```

```
               'Item 4' -> strings.addText;
               'Quit  ' -> strings.addText;
               (strings, 1) -> additems;
               itemCount -> visibleItemCount;
            #)
       #)
do lst.init;
#)
```

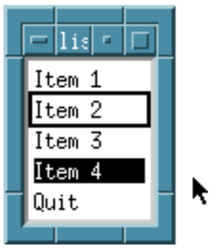
The list is constructed using the MotifString pattern, and the items are set up using the pattern MotifStringArray. MotifStringArray has almost the same attributes a the corresponding StringArray pattern of XtEnv. The difference is that a MotifStringArray is an array of so-called compound strings, that will be further explained in connection with the multi_font example below.

In the virtual init pattern of the ListWidget called lst, the MotifStringArray is initialized with five strings, and then added at position 1 in the list. The assignment of the itemCount resource (number of items) to the visibleItemCount resource, means that the window of the ListWidget will be big enough that all items are be visible.

A ListWidget have four different "modes" of operation:

1. Single Selection mode - only one item can be selected at a time;
2. Multiple selection mode - multiple items may be selected, even non-adjacent items;
3. Extended select - ranges of adjecent items may be selected; and
4. Browse select - only one item may be selected at a time, but when moving the pointer with mouse button one pressed, the item under the mouse is highlighted, and when the mouse is released, the item is selected.

Browse select is the default, and when the above program is run, the window may look like this:



Item 2 is selected, and the mouse has been moved down to item 4 with the first mouse button pressed.

In all callbacks to Motif widgets and gadgets, an item called data is present. This is an instance of a virtual XmAnyCallbackStruct pattern. In all callbacks to MotifList widgets, the pattern data is an instance of is further bound to a pattern called XmListCallbackStruct. In the defaultActionCallback virtual of lst (which is called when an item is double-clicked) in the example, some of these attributes are used the print out information about the item selected. Likewise, in the browseSelectionCallback virtual (called when an item is selected in Browse selection mode), the data-object is used to determine if the last item (Quit) has been selected, and in that case the program is stopped.

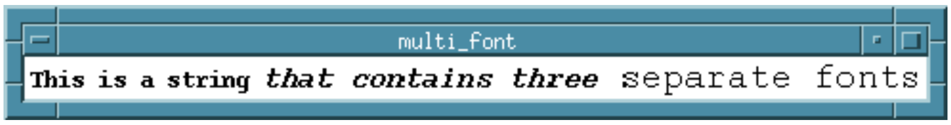
The following small program shows how to use the Compound Strings of OSF/Motif, via the BETA abstraction called MotifString. It also shows how to define a fontlist for a Label widget. To incorporate more than one font in a MotifString, you must create a font list, that specifies multiple character sets, for the widget that will be displaying the MotifString. This example is somewhat advanced, and may be skipped until multiple-font strings are needed.

multi_font.bet

```
ORIGIN '~beta/Xt/motifenv';
INCLUDE '~beta/Xt/motif/label';
-- PROGRAM: descriptor --
MotifEnv
(#
   multi: @Label
     (#
        fontlst: @MotifFontList
          (# init::<
               (#
               do ('-*-courier-*-r-*--12-*', 'charset1')
                    -> addText;
                  ('-*-courier-bold-o-*--14-*', 'charset2')
                    -> addText;
                  ('-*-courier-medium-r-*--18-*', 'charset3')
                    -> addText;
               #)
          #);
        txt: @MotifString
          (# init::<
               (#
               do ('This is a string ',   'charset1',
                   XmSTRING_DIRECTION_L_TO_R) -> setTextSegment;
                  ('eerht sniatnoc taht', 'charset2',
                   XmSTRING_DIRECTION_R_TO_L) -> appendSegment;
                  (' separate fonts',     'charset3',
                   XmSTRING_DIRECTION_L_TO_R) -> appendSegment;
               #)
          #);
        init::<
          (# string: @labelString; (* The MotifStringResource *)
          do fontlst.init;
             fontlst -> fontList;
             txt.init;
             txt -> string.set;
          #)
     #);
do multi.init;
#)
```

Within the Label, two objects are used: A MotifFontList called fontlst, and a MotifString called txt. In fontlst, three standard X Windows font names are bound to three names, that may be used within the widget, the fontlist is used for. These three internal names denote so-called Character Sets. In the initialization of the Label called multi, this font list is assigned to the fontList resource of the Label. Then MotifStrings displayed by multi may use these character sets: In the initialization of the MotifString, three segments are added, using these three character sets. Note that the character sets are identified by the names in the font list. Notice also the specification of string direction in the three segments: The second segment is appended using right-to-left direction.

When the program is run, the following window appears:



This is a string *that contains three* separate fonts

Notice, that the second segment has been reversed in direction.

---

[2] Here the Label appears black and white. Normally the default color of Motif widgets is a bright blue color, but for documentation purposes, black and white looks better. The program was therefore invoked as hello -bg white, giving a white background, and consequently a black foreground. This is the case for most of the following screen snapshots too.

X Libraries - Reference Manual Mjølner Informatics

MotifEnv

Motif Text Editor Widgets

The three remaining widgets in the figure of the primitive widget-hierarchy are:

- `MotifText`: provides a single- or multiline text editor for customizing both user and programmatic interfaces.
  It can be used for single-line string entry, forms entry with verification procedures, and full-window editing.
  It provides an application with a consistent editing system for textual data. The screen's textual data adjusts to the application writer's needs.
  MotifText provides separate callbacks to verify movement of the insert cursor, modification of the text, and changes in input focus. Each of these callbacks provides the verification function with the widget instance, the event that caused the callback, and a data structure specific to the verification type. From this information the function can verify if the application considers this to be a legitimate state change and can signal the MotifText whether to continue with the action.
  A MotifText allows the user to select regions of text. Selection is based on the Interclient Communication Conventions (ICCC) selection model. A MotifText supports both primary and secondary selection.

- `ScrolledText`: Utility pattern for instantiating a MotifText within a ScrolledWindow.

- `TextField`: Like MotifText, but contains just one line of text. Used primarily for text entry fields in dialogs.

Examples using Motif Text Editor Widgets

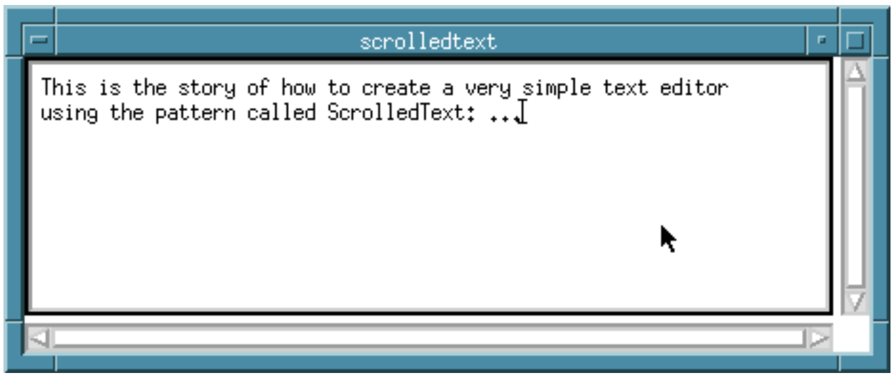
This is an example showing how to create a MotifText within a ScrolledWindow using the pattern ScrolledText.

scrolledtext.bet

```
ORIGIN '~beta/Xt/motifenv';
INCLUDE '~beta/Xt/motif/texts';
-- PROGRAM: descriptor --
MotifEnv
(# txt: @ScrolledText
     (# init::<
          (#
          do 10 -> rows;
             80 -> columns;
             false -> resizeWidth;
             false -> resizeHeight;
          #)
     #);
do txt.init;
#)
```

This program creates a MotifText within a composite ScrolledWindow widget, which provides scroll bars for its child. The ScrolledText widget is conceptually a specialization of MotifText, that has some ScrolledWindow attributes too.

When the above program is run, a text editor with 10 lines, each of 80 characters appears:



This is the story of how to create a very simple text editor
using the pattern called ScrolledText: ...

The MotifText widget and the corresponding TextField widget have several callbacks installed: Virtual callbacks are called just before, and right after text is inserted into the buffer, when the widget gains and looses keyboard focus, when the widget gains or looses ownership of the X Window selection, when the insert cursor is moved within the buffer, etc.

The following example shows how to use the modifyVerifyCallback, which is called just before text is inserted into the buffer.The example shows how to implement the often used "enter password" field, where the characters typed is not visible on screen. To do this, within the callback, the character to be inserted into the buffer is changed to a '*'. This is done by manipulating the callback-data, which contains pointers to externally allocated stuctures containing, among other things, the character(s) to be inserted to the text buffer, after the callback.

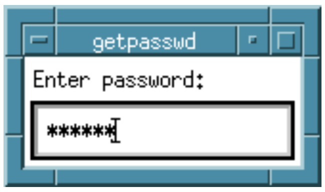
The example is somewhat complex, and may be skipped at first reading.

getpasswd.bet

```
ORIGIN '~beta/Xt/motifenv';
INCLUDE '~beta/Xt/motif/texts';
INCLUDE '~beta/Xt/motif/rowcolumn';
INCLUDE '~beta/Xt/motif/label';

-- PROGRAM: descriptor--
MotifEnv
(# rowcol: @RowColumn
     (# prompt: @Label
          (# init::< (# do 'Enter password:' -> labelString #)#);
        getpasswd: @MotifText
          (# passwd: @text; (* Used to save the password typed *)
             init::<
               (# do XmSINGLE_LINE_EDIT -> editMode #);
             modifyVerifyCallback::<
               (* Called before text is deleted or inserted *)
               (# typedtext: @XmTextBlockRec;
                  string: @CString;
                  inx: @integer;
                  txt: @text;
               do data.text -> typedtext;
                  (if typedtext.ptr=0 then
                      (* BackSpace/Delete was typed:
                       * Delete char in copy of typed password
                       *)
                      data.currInsert -> inx;
                      (if inx>0 then
                          (inx, inx) -> passwd.delete
                      if);
                   else
                      (if typedtext.length = 1 then
                          (* Allow simple access to the external
                           * C string in the XmTextBlockRec struct
                           *)
                          typedtext.ptr -> string;
                          (* Save the character typed *)
                          0-> string.inxget -> txt.put;
                          (txt[], data.currInsert+1)
                            -> passwd.insert;
                          (* Replace character with '*' *)
                          (0, '*') -> string.inxput;
                       else
                          (* Disable multi-character insertions
                           * (e.g. pasting)
                           *)
                          false -> data.doit;
                      if);
                  if)
               #);
             activateCallback::<
               (# (* The <Return> key was pressed *)
               do 'You have entered ''' -> screen.puttext;
                  passwd[] -> screen.puttext;
                  '''.' -> screen.putline;
                  Stop;
               #);
          #);
        init::< (# do prompt.init; getpasswd.init; #);
     #);
do rowcol.init;
#)
```

To access the external string, an instance of a pattern called XmTextBlockRec (declared in MotifLib) is used.

When the program is run, and after six characters has been typed, the window looks like this:



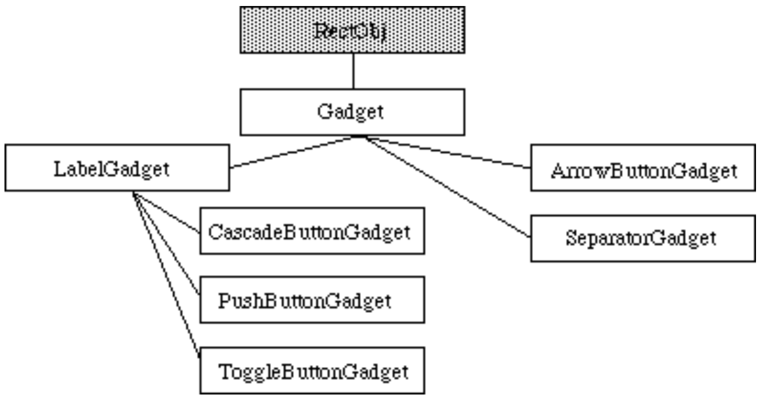
When the <RETURN> key is typed, the activateCallback is called. Within this, the uncrypted password is printed on the screen.

X Libraries - Reference Manual   Mjølner Informatics

MotifEnv

Motif Gadgets

The following figure shows the patterns constituting the MotifEnv gadget patterns. Patterns from XtEnv are shaded gray.



The following is a brief description of the Motif gadgets:

- `Gadget`: a pattern used as a supporting superpattern for other gadget patterns. It handles shadow-border drawing and highlighting, traversal activation and deactivation, and various callbacks needed by gadgets. The difference between a Primitive and a Gadget is, that a gadget i "window-less", i.e., it uses the window of its parent to draw into. The color and pixmap resources defined by Manager are directly used by gadgets. If one of these resources for is changed for a Manager widget, all of the gadget children within the Manager also change. Gadget is a specialization of RectObj; in contrast Primitive is a specialization of XtObject.

- `LabelGadget`: Like Label, but uses its parent window for drawing. Almost the same interface as Label. Requires less memory than Label.

- `CascadeButtonGadget`: Like CascadeButton, but uses its parent window for drawing. Almost the same interface as CascadeButton. Requires less memory than CascadeButton.

- `PushButtonGadget`: Like PushButton, but uses its parent window for drawing. Almost the same interface as PushButton. Requires less memory than PushButton.

- `ToggleButtonGadget`: Like ToggleButton, but uses its parent window for drawing. Almost the same interface as ToggleButton. Requires less memory than ToggleButton.

- `ArrowButtonGadget`: Like ArrowButton, but uses its parent window for drawing. Almost the same interface as ArrowButton. Requires less memory than ArrowButton.

- `SeparatorGadget`: Like Separator, but uses its parent window for drawing. Almost the same interface as Separator. Requires less memory than Separator.

Examples using Motif Gadgets

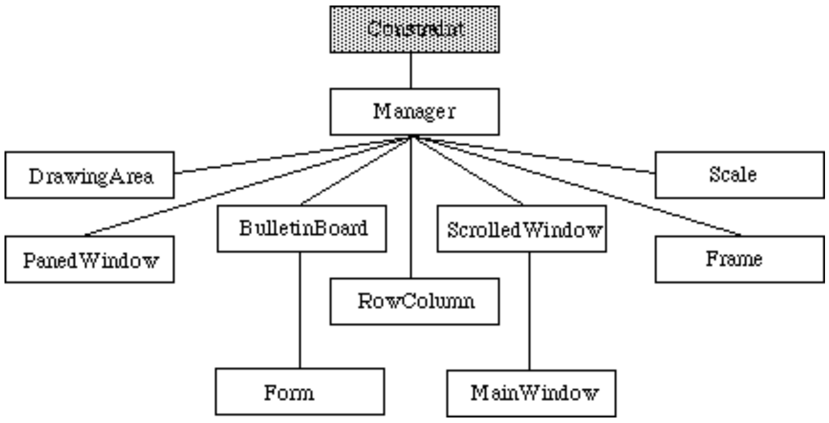
With very few exceptions, a gadget can be used whereever, the corresponding widget would otherwise be used. The exceptions are: Gadgets do not support eventhandlers, translations or popup children. In several of the examples on the following pages, gadgets are used instead of widgets. Gadgets takes some of the load of the X Window server, since they do not create their own window, but instead draw in their parent window. The parents, however, must be prepared to support the gadgets in this sharing of the window. In general all Motif composites support gadgets.

X Libraries - Reference Manual Mjølner Informatics

MotifEnv

Motif Manager Widgets

The following figure shows the patterns constituting the MotifEnv manager patterns. Patterns from XtEnv are shaded gray.



The following is a brief description of most of the Motif Manager patterns. The Menu related patterns (special RowColumns, not shown in the figure) are described later:

- `Manager`: a pattern used as a supporting superpattern for other composite patterns. It supports the visual resources, graphics contexts, and traversal resources necessary for the graphics and traversal mechanisms.

- `BulletinBoard`: a composite widget that provides simple geometry management for child widgets. It does not force positioning on its children, but can be set to reject geometry requests that result in overlapping children. BulletinBoard is the base widget for most dialog widgets and is also used as a general container widget. If its parent is a DialogShell, BulletinBoard passes title and input mode (based on dialog style) information to the parent, which is responsible for appropriate communication with the window manager.

- `Form`: a BulletinBoard with no input semantics of its own. Constraints are placed on children of the Form to define attachments for each of the child's four sides. These attachments can be to the Form, to another child widget or gadget, to a relative position within the Form, or to the initial position of the child. The attachments determine the layout behavior of the Form when resizing occurs.

- `DrawingArea`: an empty widget that is easily adaptable to a variety of purposes. It does no drawing and defines no behavior except for invoking callbacks. Callbacks notify the application when graphics need to be drawn (exposure events or widget resize) and when the widget receives input from the keyboard or mouse. Applications are responsible for defining appearance and behavior as needed in response to DrawingArea callbacks.

- `Frame`: a very simple manager used to enclose a single child in a border drawn by the Frame. It uses the Manager class resources for border drawing and performs geometry management so that its size always matches its child's size plus the margins defined for it. Frame is most often used to enclose other managers when the application developer desires the manager to have the same border appearance as the primitive widgets.

- `RowColumn`: a general purpose row/column manager capable of containing any widget type as a child. In general, it requires no special knowledge about how its children function and provides nothing beyond support for several different layout styles. However, it can be configured as a menu, in which case, it expects only certain children, and it configures to a particular layout. The menus supported are: MenuBar, Pulldown or Popup MenuPanes, and OptionMenu. The type of layout performed is controlled by how the application has set the various layout resources. Menus are described in detail later in this document. It can be configured to lay out its children in either rows or columns. In addition, the application can specify how the children are laid out, as follows:

    1. the children are packed tightly together into either rows or columns,
    2. each child is placed in an identically sized box (producing a symmetrical look), or
    3. a specific layout (the current x and y positions of the children control their location).

  In addition, the application has control over both the spacing that occurs between each row and column and the margin spacing present between the edges of the RowColumn widget and any children that are placed against it. By default the RowColumn widget has no 3-D visuals associated with it; if an application wishes to have a 3-D shadow placed around this widget, it can create the RowColumn as a child of a Frame widget.

- `Radiobox`: Utility pattern to instantiate a RowColumn widget of type XmWORK_AREA. Typically, this is a composite widget that contains multiple ToggleButtonGadgets. The RadioBox arbitrates and ensures that at most one ToggleButtonGadget is on at any time. This pattern provides initial values for several RowColumn resources. It initializes packing to XmPACK_COLUMN, radioBehavior to True, isHomogeneous to True, and entryClass to xmToggleButtonGadgetClass. In a RadioBox the ToggleButton or ToggleButtonGadget resource indicatorType defaults to XmONE_OF_MANY, and the ToggleButton or ToggleButtonGadget resource visibleWhenOff defaults to True.

- `ScrolledWindow`: combines one or two ScrollBar widgets and a viewing area to implement a visible window onto some other (usually larger) data display. The visible part of the window can be scrolled through the larger display by the use of ScrollBars.

- `MainWindow`: provides a standard layout for the primary window of an application. This layout includes a MenuBar, a CommandWindow, a work region, a MessageWindow, and ScrollBars. Any or all of these areas are optional. The work region and ScrollBars in the MainWindow behave identically to the work region and ScrollBars in the ScrolledWindow widget. The user can think of the MainWindow as an extended ScrolledWindow with an optional MenuBar and optional CommandWindow and MessageWindow. In a fully-loaded MainWindow, the MenuBar spans the top of the window horizontally. The CommandWindow spans the MainWindow horizontally just below the MenuBar, and the work region lies below the CommandWindow. The MessageWindow is below the work region. A MainWindow can also create three Separator widgets that provide a visual separation of MainWindow's four components.

- `Scale`: used by an application to indicate a value from within a range of values, and it allows the user to input or modify a value from the same range. A Scale has an elongated rectangular region similar to a ScrollBar. A slider inside this region indicates the current value along the Scale. The user can also modify the Scale's value by moving the slider within the rectangular region of the Scale. A Scale can also include a label set located outside the Scale region. These can indicate the relative value at various positions along the scale. A Scale can be either input/output or output only. An input/output Scale's value can be set by the application and also modified by the user with the slider. An output-only Scale is used strictly as an indicator of the current value of something and cannot be modified interactively by the user.

- `PanedWindow`: a Composite that lays out children in a vertically tiled format. Children appear in top-to-bottom fashion, with the first child inserted appearing at the top of the PanedWindow and the last child inserted appearing at the bottom. The user can adjust the size of the panes. To facilitate this adjustment, a pane control sash is created for most children. The sash appears as a square box positioned on the bottom of the pane that it controls. The user can adjust the size of a pane by using the mouse or keyboard. The PanedWindow is also a Constraint, which means that it creates and manages a set of constraints for each child.

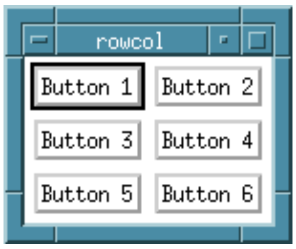
Examples using Motif Manager Widgets

The following example shows one usage of the very useful RowColumn widget. Here six PushButtonGadgets are placed in three horizontal rows.

rowcol.bet

```
ORIGIN '~beta/Xt/motifenv';
INCLUDE '~beta/Xt/motif/rowcolumn';
INCLUDE '~beta/Xt/motif/pushbuttongadget';

-- PROGRAM: descriptor --
MotifEnv
(# rowCol: @RowColumn
     (# buttons: [6]^PushButtonGadget;
        init::<
          (# t: @text;
          do (for i: 6 repeat
                  &PushButtonGadget[] -> buttons[i][];
                  buttons[i].init;
                  t.clear; 'Button ' -> t.putText;
                  i -> t.putInt;
                  t[] -> buttons[i].labelString;
             for);
             XmHORIZONTAL -> orientation;
             XmPACK_COLUMN -> packing;
             3 -> numColumns;
          #);
     #);
do rowCol.init;
#)
```

Notice that when the orientation is horizontal, the numColumns resource specifies the number of rows. When the program is run, the following window appears:

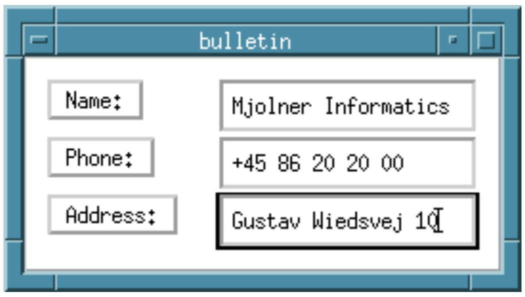


The following example shows how the BulletinBoard can be used: Six children are placed at absolute (x,y) positions.

bulletin.bet

```
ORIGIN  '~beta/Xt/motifenv';
INCLUDE '~beta/Xt/motif/bulletinboard';
INCLUDE '~beta/Xt/motif/texts';
INCLUDE '~beta/Xt/motif/pushbutton';

-- PROGRAM: descriptor --
MotifEnv
(# board: @BulletinBoard
     (# button: PushButton
          (# activateCallback::<
               (#
               do (for i: buttons.range repeat
                       (if buttons[i][]= this(button)[] then
                           editors[i].value -> screen.putLine;
                       if)
                  for)
               #);
          #);
        buttons: [3] ^button;
        editors: [3] ^MotifText;
        init::<
          (#
          do (for i: buttons.range repeat
                  &button[] -> buttons[i][];
                  buttons[i].init;
                  10 -> buttons[i].x;
                  10 + (30*(i-1)) -> buttons[i].y;
             for);
             (for i: editors.range repeat
                  &MotifText[] -> editors[i][];
                  editors[i].init;
                  100 -> editors[i].x;
                  10 + (30*(i-1)) -> editors[i].y;
             for);
             ' Name: ' -> buttons[1].labelString;
             ' Phone: ' -> buttons[2].labelString;
             ' Address: ' -> buttons[3].LabelString;
          #);
     #);
do board.init;
#)
```

When the program is run, the following window appears:



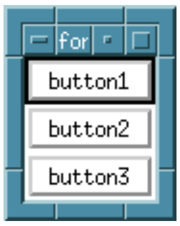
The three text entry fields have been filled out.

The next example shows one way to use the Form widget: Three PushButtons are aligned above each other using the Form constraint resources of the children. Notice that this could be done much easier using a RowColumn widget.

form.bet

```
ORIGIN '~beta/Xt/motifenv';
INCLUDE '~beta/Xt/motif/form';
INCLUDE '~beta/Xt/motif/pushbutton';

-- PROGRAM: descriptor --
MotifEnv
(# window: @Form
     (# button1: @PushButton(# #);
        button2: @PushButton(# #);
        button3: @PushButton(# #);
        init::<
          (#
          do button1.init;
             button2.init;
             button3.init;
             XmATTACH_FORM -> button1.topAttachment;
             XmATTACH_FORM -> button1.leftAttachment;
             XmATTACH_FORM -> button1.rightAttachment;
             XmATTACH_WIDGET -> button2.topAttachment;
             button1 -> button2.topWidget;
             XmATTACH_FORM -> button2.leftAttachment;
             XmATTACH_FORM -> button2.rightAttachment;
             XmATTACH_WIDGET -> button3.topAttachment;
             button2 -> button3.topWidget;
             XmATTACH_FORM -> button3.leftAttachment;
             XmATTACH_FORM -> button3.rightAttachment;
             XmATTACH_FORM -> button3.bottomAttachment;
          #);
     #);
do window.init
#)
```

In general specific constraints may be put on each edge of the children. Here one PushButton - button1 - is attached to the Form itself on three sides. This means that button1 will attempt to resize these three edges if the Form is resized. A second PushButton - button2 - is then attached vertically to button1: Its topAttachment resource is specified as the constant XmATTACH_WIDGET, and button1 is then specified as the topWidget of button2. Likewise a third button - button3 - is attached vertically to button2. When the application is run, the following window appears:



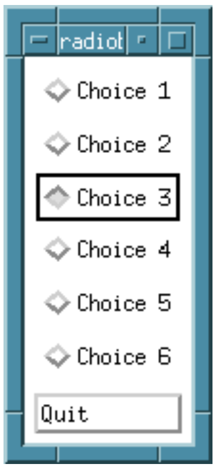
Several special variants of the RowColumn widget are supplied by MotifEnv. This includes various menus described later, and it also includes the pattern RadioBox, used to create a traditional "radio panel" of buttons. The following example demonstrates the RadioBox pattern:

radiobox.bet

```
ORIGIN '~beta/Xt/motifenv';
INCLUDE '~beta/Xt/motif/rowcolumn';
INCLUDE '~beta/Xt/motif/togglebuttongadget';
INCLUDE '~beta/Xt/motif/pushbuttongadget';

-- PROGRAM: descriptor --
MotifEnv
(# main: @RowColumn
     (# panel: @RadioBox
          (# r1, r2, r3, r4, r5, r6: @ToggleButtonGadget
               (# valueChangedCallback::<
                    (#
                    do name -> screen.puttext;
                       (if data.set then ' set' -> putLine
                        else ' unset' -> putLine
                       if)
```

```
                  #);
              #);
            init::<
              (#
              do ('Choice 1', panel) -> r1.init;
                 ('Choice 2', panel) -> r2.init;
                 ('Choice 3', panel) -> r3.init;
                 ('Choice 4', panel) -> r4.init;
                 ('Choice 5', panel) -> r5.init;
                 ('Choice 6', panel) -> r6.init;
              #);
         #);
       quit: @PushButtonGadget
         (# init::< (# do 'Quit' -> labelString #);
            activateCallback::< (# do stop #)
         #);
       init::< (# do panel.init; quit.init #);
    #)
do main.init
#)
```

When this program is run, the window shown below with six ToggleButtonGadgets and one PushButtonGadget (Quit) appears. The window is shown in a state, where the third item is selected. Notice, that the button appears to be pushed inwards.



Many standard applications consists of one main window, with a menu bar with menus and a working area that can be scrolled. For this purpose, a MainWindow widget with support for this setup is provided. The following is an excerpt of an application using the MainWindow widget:

mainwindow.bet

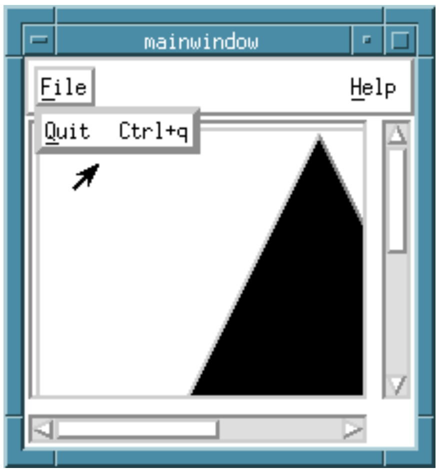
```
ORIGIN '~beta/Xt/allmotif'
--PROGRAM: descriptor--
MotifEnv
(# main: @MainWindow
     (# work: @ArrowButtonGadget;
        mbar: @MenuBar
          (# specification of menu bar not shown ... #);
        init::<
          (#
          do mbar.init;     work.init;
             200 -> height; 300 -> work.height;
             200 -> width;  300 -> work.width;
          #);
     #);
```

```
   fallbackResources::<
     (# init::<
          (#
          do '*XmMainWindow.scrollingPolicy: XmAUTOMATIC'
               -> addtext
          #);
     #);
do main.init;
#)
```

For work area, an ArrowButtonGadget is used. This could be any other widget. In the initialization, the ArrowButtonGadget is made bigger than the working area of the MainWindow. This will force the scroll bars to appear.

The scrollingPolicy resource of the MainWindow cannot be changed on the fly, but only at creation time. Thus the easiets way to do it, is to specify it in a standard X Toolkit resource file, but the fallbackResources virtual of XtEnv can also be used as is done here. Setting the scrollingPolicy to XmAUTOMATIC will make scrollbars appear when needed.

When the program is run the following window appears:



The window is shown in a situation, where the File menu has been posted. It will later be shown how to program the menu bar with this File menu.

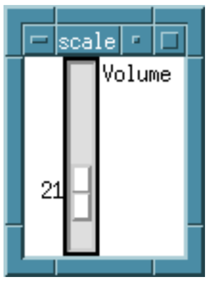
The next example shows how to use the special purpose Scale widget, useful for adding, e.g., a potentiometer-like control to a panel of controls.

scale.bet

```
ORIGIN '~beta/Xt/motifenv';
INCLUDE '~beta/Xt/motif/scale';
-- PROGRAM: descriptor --
MotifEnv
(# volume: @Scale
     (# init::<
          (#
          do 0 -> minimum;
             100 -> maximum;
             XmVERTICAL -> orientation;
             'Volume' -> titleString;
             true -> showValue;
          #);
        valueChangedCallback::<
```

```
          (#
          do 'New volume: ' -> screen.puttext;
             data.value -> screen.putint;
             screen.newline;
          #);
     #);
do volume.init;
#)
```

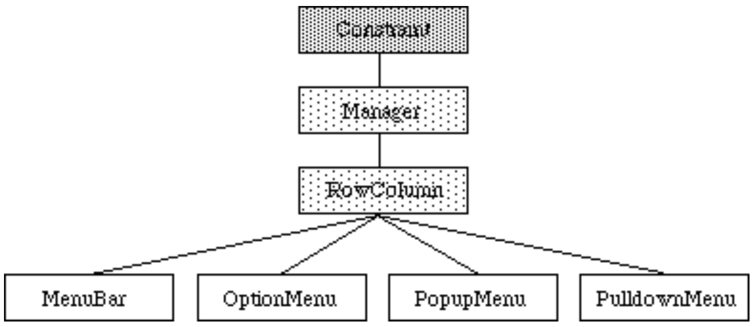
The following window will appear when the program is run:



The valueChangedCallback is called after the scale has been changed. In this case the callback data is furtherbound to XmScaleCallbackStruct, that has an attribute called value, that contains the current value of the Scale.

Motif Menus

The following figure shows the menu-related patterns of MotifEnv. Patterns from XtEnv are shaded gray, and other MotifEnv patterns are shaded light-gray:



The following is a brief presentation of the menu-related patterns:

- `PopupMenu`: Utility pattern used to instantiate a RowColumn widget of type XmMENU_POPUP. When the Popup MenuPane is created, a MenuShell widget is automatically created as the parent of the MenuPane. The PopupMenu is used as the first MenuPane within a PopupMenu system; all other MenuPanes are of the Pulldown type. A Popup MenuPane displays a 3-D shadow, unless the feature is disabled by the application. The shadow appears around the edge of the MenuPane. A PopupMenu is "popped up" using manageChild, and "popped down" using unmanageChild. The menu can be positioned before being managed using the local position pattern.

- `PullDownMenu`: Utility pattern to create a RowColumn of type XmMENU_PULLDOWN. When creating a PulldownMenu, a MenuShell is automatically created as the parent of PulldownMenu. If the parent specified is a PopupMenu or a PulldownMenu, the MenuShell is created as a child of the parent's MenuShell; otherwise, it is created as a child of the specified parent widget. A PulldownMenu displays a 3-D shadow, unless the feature is disabled by the application. The shadow appears around the edge of the PulldownMenu. A PulldownMenu is used when creating submenus that are to be attached to a CascadeButton or a CascadeButtonGadget. This is the case for all MenuPanes that are part of a PulldownMenu system (a MenuBar), the MenuPane associated with an OptionMenu, and any MenuPanes that cascade from a Popup MenuPane.

- `PulldownMenus` that are to be associated with an OptionMenu must be created before the OptionMenu is created.
- The PulldownMenu must be attached to a CascadeButton or CascadeButtonGadget that resides in a MenuBar, a Popup MenuPane, a Pulldown MenuPane, or an OptionMenu. This is done by using the button resource subMenuId.
- To function correctly when incorporated into a menu, the PulldownMenu's hierarchy must be considered; this hierarchy depends on the type of menu system that is being built as follows:

    - ♦ If the PulldownMenu is to be pulled down from a MenuBar, its parent must be the MenuBar.

- ♦ If the PulldownMenu is to be pulled down from a PopupMenu or another PulldownMenu, its parent must be that PopupMenu or PulldownMenu.
- ♦ If the PulldownMenu is to be pulled down from an OptionMenu, its parent must be the same as the OptionMenu parent.

- `OptionMenu`: Utility pattern for instantiating a RowColumn widget of type XmMENU_OPTION. An OptionMenu widget is a specialized RowColumn manager composed of a label, a selection area, and a single Pulldown MenuPane. When an application creates an OptionMenu widget, it supplies the label string and the PulldownMenu. In order to succeed, there must be a valid subMenuId resource set. When the OptionMenu is created, the PulldownMenu must have been created as a child of the OptionMenu's parent and must be specified. The LabelGadget and the selection area (a CascadeButtonGadget) are created by the OptionMenu. An OptionMenu is laid out with the label displayed on one side of the widget and the selection area on the other side. The selection area has a dual purpose; it displays the label of the last item selected from the associated PulldownMenu, and it provides the means for posting the PulldownMenu. The PulldownMenu is posted by moving the mouse pointer over the selection area and pressing a mouse button defined by OptionMenu's RowColumn parent. The PulldownMenu is posted and positioned so that the last selected item is directly over the selection area. The mouse is then used to arm the desired menu item. When the mouse button is released, the armed menu item is selected and the label within the selection area is changed to match that of the selected item. The OptionMenu also operates by using the keyboard interface mechanism. If the application has established a mnemonic with the OptionMenu, typing Alt with the mnemonic causes the PulldownMenu to be posted with traversal enabled. The standard traversal keys can then be used to move within the Menu. Selection can occur as the result of pressing the Return key or typing a mnemonic or accelerator for one of the menu items. An application may use the menuHistory resource to indicate which item in the PulldownMenu should be treated as the current choice and have its label displayed in the selection area. By default, the first item in the PulldownMenu is used.

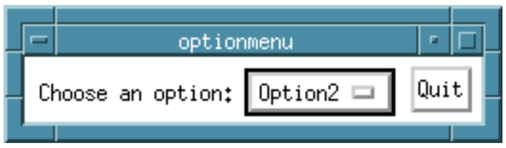
- `MenuBar`: Utility pattern for instantiating a RowColumn widget of type XmMENU_BAR. A MenuBar is generally used for building a Pulldown menu system. Typically, a MenuBar is created and placed along the top of the application window, and several CascadeButtons are inserted as the children. Each of the CascadeButtons has a PulldownMenu associated with it. These PulldownMenus must have been created as children of the MenuBar. The user interacts with the MenuBar by using either the mouse or the keyboard. A MenuBar displays a 3-D shadow along its border. The application controls the shadow attributes using the visual-related resources supported by Manager. A MenuBar widget is homogeneous in that it accepts only children that are specializations of CascadeButton or CascadeButtonGadget. Attempting to insert a child of a different class results in a warning message. If a MenuBar does not have enough room to fit all of its subwidgets on a single line, the MenuBar attempts to wrap the remaining entries onto additional lines if allowed by the geometry manager of the parent widget.

Examples using Motif Menus

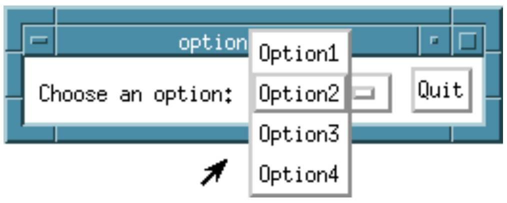
The following is an example of using an option menu.
optionmenu.bet
```
ORIGIN '~beta/Xt/motifenv';
INCLUDE '~beta/Xt/motif/rowcolumn';
INCLUDE '~beta/Xt/motif/pushbuttongadget';

-- PROGRAM: descriptor --
MotifEnv
(# main: @RowColumn
     (# menuPane: @PulldownMenu
          (# item1,item2,item3,item4: @PushButtonGadget
               (# activateCallback::<
                    (# do name->puttext; ' selected'->putLine #);
               #);
             init::<
               (#
               do ('Option1', menuPane) ->item1.init;
                  ('Option2', menuPane) ->item2.init;
                  ('Option3', menuPane) ->item3.init;
                  ('Option4', menuPane) ->item4.init;
               #);
          #);
        optMenu: @OptionMenu
          (# init::<
               (# subMenu::< (# do menuPane -> value #);
               do 'Choose an option:' -> labelString;
                  menuPane.item2 -> menuHistory;
               #);
          #);
        quit: @PushButtonGadget
          (# init::< (# do 'Quit' -> labelString #);
             activateCallback::< (# do stop #)
          #);
        init::<
          (#
          do XmHORIZONTAL -> orientation;
             menuPane.init;
             optMenu.init;
             quit.init;
          #);
     #);
do main.init;
#)
```

First a menu pane to use in the option menu is created: This is a PullDownMenu with four PushButtonGadgets as items. Then the option menu widget is created. This will glue the menu pane, a label and a button together. Notice, that specification of the menu pane for the option menu is done using the subMenu virtual local to the init virtual. This is because the OptionMenu widget needs to know what menu pane to use already when the OptionMenu is created.

When the program is run, the following window appears:



As can be seen, the OptionMenu starts out with Option2 selected. This was specified in the program using the menuHistory resource. When the OptionMenu button is clicked, the menu pops up, and a new option can be selected:



The following example shows how to create and use a PopupMenu:

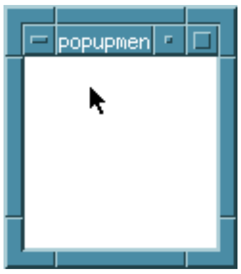
popupmenu.bet

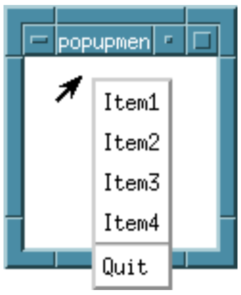
```
ORIGIN '~beta/Xt/motifenv';
INCLUDE '~beta/Xt/motif/primitive';
INCLUDE '~beta/Xt/motif/rowcolumn';
INCLUDE '~beta/Xt/motif/separatorgadget';
INCLUDE '~beta/Xt/motif/pushbuttongadget';

-- PROGRAM: descriptor --
MotifEnv
(# theMenu: @PopupMenu
     (# item1,item2,item3,item4: @PushButtonGadget
          (# activateCallback::<
               (# do name -> puttext; ' selected' -> putLine #);
          #);
        quit: @PushButtonGadget
          (# activateCallback::< (# do stop #)#);
        init::<
          (# sep: ^SeparatorGadget;
          do ('Item1', theMenu) ->item1.init;
             ('Item2', theMenu) ->item2.init;
             ('Item3', theMenu) ->item3.init;
             ('Item4', theMenu) ->item4.init;
             &SeparatorGadget[] -> sep[]; sep.init;
             ('Quit', theMenu) -> quit.init;
          #);
     #);
   main: @Primitive
     (# eventHandler::<
          (# bp: @ButtonPress
               (#
               do event -> theMenu.position;
                  theMenu.manageChild
               #);
             init::< (# do bp.enable #);
          #);
        init::< (# do 100 -> width; 100 -> height;#);
     #);
do main.init;
   ('theMenu', main) -> theMenu.init;
#)
```

The PopupMenu is created having five PushButtonGadgets and one SeparatorGadget as children. In the example, this menu is to be popped up from a Primitive. This is done, by further binding the eventhandler of the Primitive: a ButtonPress eventprocessor is instantiated and enabled. When a button is pressed within the Primitive, the do-part of this eventprocessor is invoked. This will do two things:

1. The event causing the invocation is handled to the local position pattern of the PopupMenu. This will place the menu in a way so that the first menu item is just where the mouse was pressed. The position pattern uses informations in the event object for this.
2. The menu is popped up. This is done by using the manageChild atttribute.

When the program is executed, the window shown left below appears. When a mouse button is then pressed, the menu pops up as shown in the right window below.





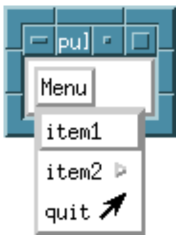
The next example shows how to create a pulldown menu system using a MenuBar, and also it shows how to create hierarchical menus using CascadeButtonGadgets.

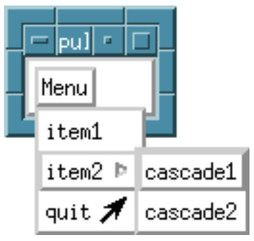
pulldownmenu.bet

```
ORIGIN '~beta/Xt/motifenv';
INCLUDE '~beta/Xt/motif/rowcolumn';
INCLUDE '~beta/Xt/motif/cascadebutton';
INCLUDE '~beta/Xt/motif/pushbuttongadget';
INCLUDE '~beta/Xt/motif/cascadebuttongadget';

-- PROGRAM: descriptor --
MotifEnv
(# menus: @MenuBar
     (# menubutton: @CascadeButton
          (* The button that activates the menu *)
          (# init::< (# do 'Menu' -> labelString #)#);
        menu: @PulldownMenu
          (# item1: @PushButtonGadget
               (# activateCallback::<
                    (# do 'Item1 selected' -> putLine #);
               #);
             item2: @CascadeButtonGadget
               (# submenu: @PullDownMenu
                    (# cascade1: @PushButtonGadget
                         (# activateCallback::<
                              (# do 'Cascade 1 selected'->putLine #);
                         #);
                       cascade2: @PushButtonGadget
                         (# activateCallback::<
                              (# do 'Cascade 2 selected'->putLine #);
                         #);
                       init::<
                         (# do cascade1.init; cascade2.init #);
                    #);
                  init::< (# do submenu.init; submenu->subMenuId #);
               #);
             quit: @PushButtonGadget
               (# activateCallback::< (# do stop #)#);
             init::< (# do item1.init; item2.init; quit.init #);
          #);
        init::<
          (#
          do menubutton.init;
             menu.init;
             menu -> menubutton.subMenuId;
     #)#);
do menus.init;
#)
```

A MenuBar can contain CascadeButtons, each having a PullDownMenu attached. In the example only one CascadeButton is created, thus the menu bar will only contain one menu. The CascadeButton - menubutton - is initialized first. Its label string is set to "Menu". Then the PulldownMenu is initialized. It contains three items, the first and last one being normal PushButtonGadgets. The second item, however, is a CascadeButtonGadget. This is used to bind a sub menu to the main menu. The submenu is a normal PullDownMenu, which is initialized, and then attached to the CascadeButtonGadget using the subMenuId resource of the CascadeButtonGadget. Finally the main menu is attached to the CascadeButton in the MenuBar, by using the submenuId resource of the CascadeButton in the MenuBar.

When the program is executed, a window containing a menu bar with just one button appears. Below in the window shown left, the menu button has been activated, and the menu has been posted. Notice the arrow in the second menu item, indicating that the item has a sub menu. If the second item is activated, the submenu will pop up, as shown in the right window below.

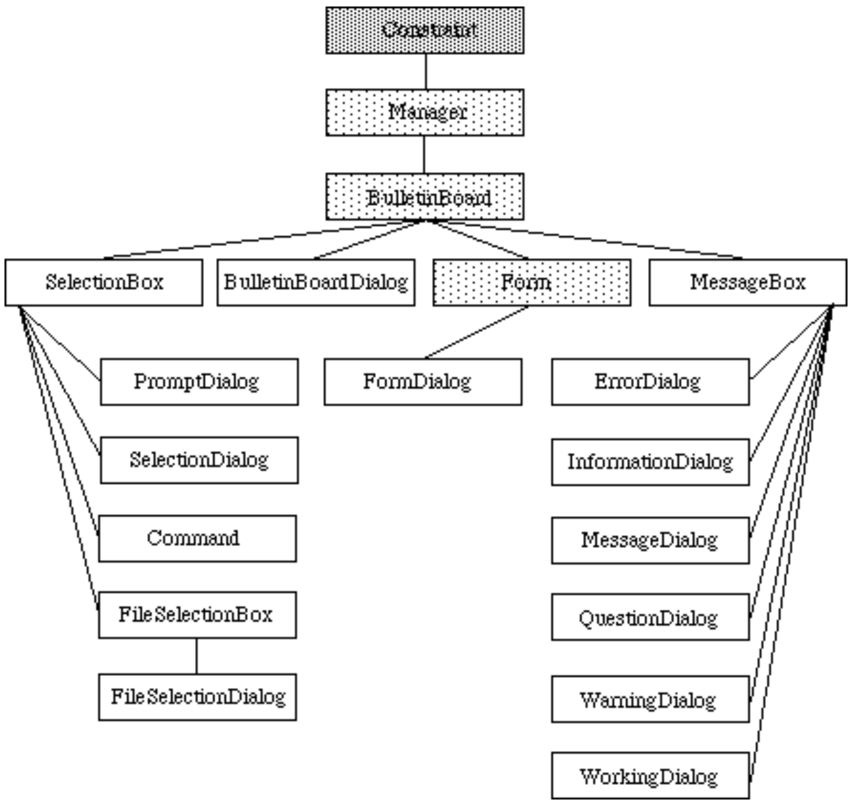




MotifEnv

Motif Dialog Patterns

The following figure shows the patterns constituting the MotifEnv dialog patterns. Patterns from XtEnv are shaded gray, and other MotifEnv patterns are shaded light-gray.



The following is a brief presentation of the different patterns:

- `SelectionBox`: a general dialog widget that allows the user to select one item from a list. A SelectionBox includes the following:

    1. a scrolling list of alternatives,
    2. an editable text field for the selected alternative,
    3. labels for the list and text field, and
    4. three or four buttons. The default button labels are OK, Cancel, and Help. By default an Apply button is also created; if the parent of the SelectionBox is a DialogShell it is managed, and otherwise it is unmanaged.

  One additional WorkArea child may be added to the SelectionBox after creation. The user can select an item in two ways: by scrolling through the list and selecting the desired item or by entering the item name directly into the text edit area. Selecting an item from the list causes that item name to appear in the selection text edit area. The user may select a new item as many times as desired. The item is not actually selected until the user presses the OK PushButton.

- `PromptDialog`: Utility pattern to instantiate a DialogShell and an unmanaged SelectionBox child of the DialogShell. A PromptDialog prompts the user for text input. It includes a message, a text input region, and three managed buttons. The default button labels are OK, Cancel, and Help. An additional button, with Apply as the default label, is created unmanaged; it may be explicitly managed if needed.

- `SelectionDialog`: Utility pattern to instantiate a DialogShell and an unmanaged SelectionBox child of the DialogShell. A SelectionDialog offers the user a choice from a list of alternatives and gets a selection. It includes the following:

    1. a scrolling list of alternatives,
    2. an editable text field for the selected alternative,
    3. labels for the text field, and
    4. four buttons. The default button labels are OK, Cancel, Apply, and Help.

- `FileSelectionBox`: used to traverse through directories, view the files and subdirectories in them, and then select files. A FileSelectionBox has five main areas:

    1. a text input field for displaying and editing a directory mask used to select the files to be displayed,
    2. a scrollable list of filenames,
    3. a scrollable list of subdirectories,
    4. a text input field for displaying and editing a filename, and
    5. a group of PushButtons, labeled OK, Filter, Cancel, and Help.

  The list of filenames, the list of subdirectories, or both can be removed from the FileSelectionBox after creation by using the patterns unmanageFileNames and unmanageSubdirectories.

- `FileSelectionDialog`: utility pattern to instantiate a DialogShell and an unmanaged FileSelectionBox child of the DialogShell.

- `Command`: a special-purpose composite widget for command entry that provides a built-in command-history mechanism. Command includes a command-line text-input field, a command-line prompt, and a command history list region. Whenever a command is entered, it is automatically added to the end of the command-history list and made visible. This does not change the selected item in the list, if there is one.

- `MessageBox`: MessageBox is a dialog pattern used for creating simple message dialogs. Specializations based on MessageBox are provided for several common interaction tasks, which include giving information, asking questions, and reporting errors. A MessageBox dialog is typically transient in nature, displayed for the duration of a single interaction. A MessageBox can contain a message symbol, a message, and up to three standard default PushButtons: OK, Cancel, and Help. It is laid out with the symbol and message on top and the PushButtons on the bottom. The Help button is positioned to the side of the other push buttons.

- `ErrorDialog`: Utility pattern to instantiate a DialogShell and an unmanaged MessageBox child of the DialogShell. An ErrorDialog warns the user of an invalid or potentially dangerous condition. It includes a symbol, a message, and three buttons. The default symbol is an octagon with a diagonal slash. The default button labels are OK, Cancel, and Help.

- `InformationDialog`: Utility pattern to instantiate a DialogShell and an unmanaged

MessageBox child of the DialogShell. An InformationDialog presents a short information to the user. It includes a symbol, a message, and three buttons. The default symbol is lower case i. The default button labels are OK, Cancel, and Help.

- `MessageDialog`: Utility pattern to instantiate a DialogShell and an unmanaged MessageBox child of the DialogShell. A MessageDialog gives the user some message. It includes a symbol, a message, and three buttons. By default there is no symbol. The default button labels are OK, Cancel, and Help.

- `QuestionDialog`: Utility pattern to instantiate a DialogShell and an unmanaged MessageBox child of the DialogShell. A QuestionDialog asks the user a question. It includes a symbol, a message, and three buttons. The default symbol is a question mark. The default button labels are OK, Cancel, and Help.

- `WarningDialog`: Utility pattern to instantiate a DialogShell and an unmanaged MessageBox child of the DialogShell. A WarningDialog warns the user of an invalid or potentially dangerous condition. It includes a symbol, a message, and three buttons. The default symbol is an exclamation point. The default button labels are OK, Cancel, and Help.

- `WorkingDialog`: Utility pattern to instantiate a DialogShell and an unmanaged MessageBox child of the DialogShell. A WorkingDialog informs the user, that some lengthy computations is going on, for instance. It includes a symbol, a message, and three buttons. The default symbol is an hourglass. The default button labels are OK, Cancel, and Help.

- `FormDialog`: Utility pattern to instantiate a DialogShell and an unmanaged Form child of the DialogShell. A FormDialog is used for interactions not supported by the standard dialog set. It does not include any labels, buttons, or other dialog components. Such components should be added to the FormDialog by the application.

- `BulletinBoardDialog`: Utility pattern to instantiate a DialogShell and an unmanaged BulletinBoard child of the DialogShell. A BulletinBoardDialog is used for interactions not supported by the standard dialog set. It does not include any labels, buttons, or other dialog components. Such components should be added to the BulletinBoardDialog by the application.

Examples using Motif Dialog Patterns

The following example completes the MainWindow example shown earlier, by adding the menu bar, and creating a MessageDialog to pop up when the help button in the menu bar is activated.
mainwindow.bet, continued

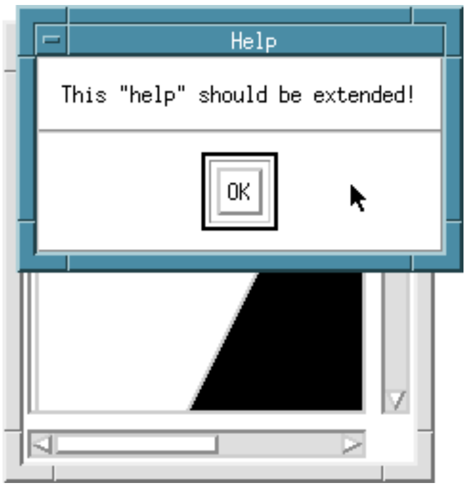
```
mbar: @MenuBar
  (# fileButton: @CascadeButton
       (# init::<
            (# do 'File' -> labelString; 'F' -> mnemonic #)
       #);
     fileMenu: @PullDownMenu
       (# quit: @PushButtonGadget
            (# init::<
                 (#
                 do 'Quit' -> labelString;
                    'Q' -> mnemonic;
                    'Ctrl<Key>q' -> accelerator;
                    'Ctrl+q' -> acceleratorText;
                 #);
               activateCallBack::< (# do stop #);
            #);
          init::< (# do quit.init #);
       #);
     helpButton: @CascadeButton
       (* The button in the MenuBar that activates the HelpBox *)
       (# helpBox: @MessageDialog
            (* The dialog that displays the help message *)
            (# init::<
                 (#
                 do 'This "help" should be extended!'
                       -> messageString;
                    'Help' -> dialogTitle;
                    unManageCancel; unManageHelp;
                 #);
            #);
          activateCallback::< (# do helpBox.manageChild #);
          init::<
            (#
            do 'Help' -> labelString; 'H' -> mnemonic;
               helpBox.init
            #);
       #);
     init::<
       (#
       do fileButton.init;  fileMenu.init;
          fileMenu -> fileButton.subMenuId;
          helpButton.init;
          helpButton -> menuHelpWidget;
       #);
  #);
```

The example shows how to create the MenuBar and File menu for the MainWindow. Except for the use of mnemonics and accellerators, this has been explained in the section on Menus earlier in the manual. A mnemonic is a "keyboard equivalent", which can be used to invoke menus and menu items, when the menus are posted. Accelerators can be used to invoke menu items even when the menu containing the item is not posted.

The help dialog is invoked from a CascadeButton called helpButton in the MenuBar. Two special things happen in the init virtual of the MessageDialog:

1. The dialogTitle resource is set to the text "Help". This is because some window managers decorate the dialogs with title bar etc. The dialogTitle resource determines what will appear in the title bar in that case.
2. A MessageDialog by default contains three buttons: an OK button, a Cancel button, and a Help button. The Cancel and Help buttons do not make much sence within a help dialog, so these are removed using the unmanageCancel and unmanageHelp patterns. This is the easiest way to make a dialog with just an OK button, since, unfortunately, Motif does not include such a dialog.

The windows below show how the help dialog looks, when popped up from the mainwindow application.

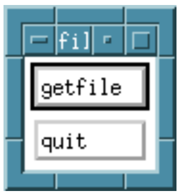


The following example shows how to use the FileSelectionBox pattern to invoke a standard file specification dialog:

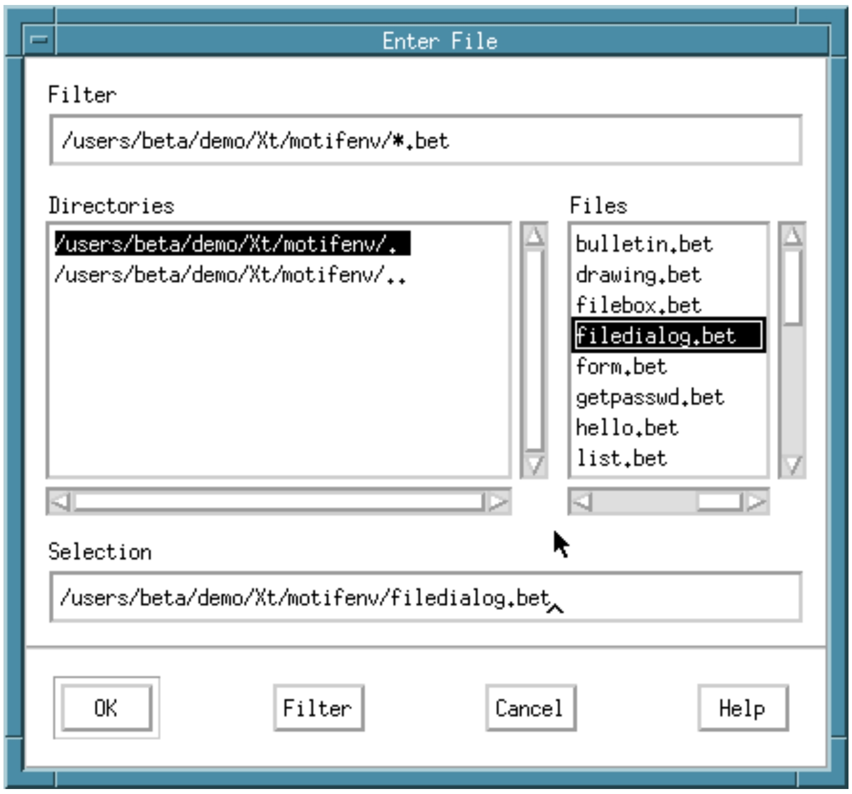
filedialog.bet

```
ORIGIN '~beta/Xt/motifenv';
INCLUDE '~beta/Xt/motif/fileselectionbox';
INCLUDE '~beta/Xt/motif/rowcolumn';
INCLUDE '~beta/Xt/motif/pushbutton';

-- PROGRAM: descriptor --
MotifEnv
(#
   buttons: @RowColumn
     (# fs: @FileSelectionDialog
          (# init::< (# do 'Enter File' -> dialogTitle #);
             OkCallback::<
               (# do dirSpec->putline; fs.unManageChild #);
             CancelCallback::<
               (# do 'Cancel'->putline; fs.unManageChild #);
             HelpCallback::<
               (# do 'Sorry, no help available'->putline #);
          #);
        getfile: @PushButton
          (# activateCallBack::< (# do fs.manageChild #)#);
        quit: @PushButton
          (# activateCallBack::< (# do stop #)#);
        init::<
          (#
          do fs.init; getfile.init; quit.init;
          #);
     #);
do buttons.init;
#)
```

The main RowColumn contains two PushButtons:



The FileSelectionBox is popped up using the manageChild attribute. The dialog looks like this:



Three callbacks are further bound in the FileSelectionDialog: When OK is pushed, the file name selected is printed out. This is obtained using the dirSpec resource. Then the dialog is popped down using the unmanageChild attributes. If Cancel is pushed, "Cancel" is printed, and the dialog is popped down. If Help is pushed, a small disclaimer is printed out. This could easily be changed to pop up a real help dialog as in the previous example. The Filter button is used to apply the regular expression in the topmost text entry field to the list of files displayed in the dialog.

MotifEnv

Other MotifEnv Patterns

The user may in some situations want to create a shell for a special menu or dialog type. This can be done using the MenuShell and DialogShell respectively. However, because of the wide variety of utility patterns to create menus and dialogs, normally the user will not need to use these two patterns directly.

- `MenuShell`: a custom OverrideShell widget. An OverrideShell widget bypasses the window manager when displaying itself. It is designed specifically to contain Popup or Pulldown MenuPanes. Most application writers never encounter this widget if they use the patterns PopupMenu or PulldownMenu, which automatically create a MenuShell widget as the parent of the MenuPane. However, if these patterns are not used, the application programmer must create the required MenuShell. In this case, it is important to note that the type of parent of the MenuShell depends on the type of menu system being built.
    - ♦ If the MenuShell is for the top-level Popup MenuPane, the MenuShell must be created as a child of the widget from which the Popup MenuPane is popped up.
    - ♦ If the MenuShell is for a MenuPane that is pulled down from a Popup or another Pulldown MenuPane, the MenuShell must be created as a child of the Popup or Pulldown MenuPane.
    - ♦ If the MenuShell is for a MenuPane that is pulled down from a MenuBar, the MenuShell must be created as a child of the MenuBar.
    - ♦ If the MenuShell is for a Pulldown MenuPane in an OptionMenu, the MenuShell's parent must be the OptionMenu.

- `DialogShell`: Modal and modeless dialogs use DialogShell as the Shell parent. DialogShell widgets cannot be iconified. Instead, all secondary DialogShell widgets associated with an ApplicationShell widget are iconified and de-iconified as a group with the primary widget. A client indirectly manipulates a DialogShell via the different dialog patterns, and it can directly manipulate its BulletinBoard-derived child. Much of the functionality of DialogShell assumes that its child is a BulletinBoard, although it can potentially stand alone.

XSystemEnv

When concurrency is used by means of the SystemEnv pattern described in [MIA 90-8], in conjunction with XtEnv, the XSystemEnv pattern located in the xsystemenv fragment should be used.

A program using xsystemenv should look something like:

```
ORIGIN 'xsystemenv';
-- program:descriptor --
systemEnv
  (# setWindowEnv::< (# do myWindowEnv[] -> theWindowEnv[] #);
     myWindowEnv: @MotifEnv (# ... #);
     ...
  #)
```

The setWindowEnv virtual and theWindowEnv reference are declared in basicsystemenv. The theWindowEnv reference does not have to be to a motifenv instance as long as it is at least an xtenv instance (e.g. awenv, motifenv, or xtenv).

The xtenv instance assigned to theWindowEnv is used for scheduling purposes to allow BETA coroutines to cooperate with the X event driven user interface.

For concurrency details, see BasicSystemEnv in [MIA 90-8].

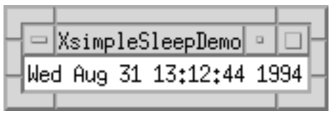
Examples of using XSystemEnv

The following simple program uses a Motif Label widget for showing the current time and date in a window. It uses a System coroutine to update the clock once per second, by sleeping between each update of the clock.

XsimpleSleepDemo.bet:

```
ORIGIN  '~beta/Xt/xsystemenv';
INCLUDE '~beta/Xt/motifenv';
INCLUDE '~beta/Xt/motif/label';
INCLUDE '~beta/sysutils/time.bet';

--- program:descriptor ---
systemEnv
(# setWindowEnv::<
     (# do mymotif[] -> theWindowEnv[] #);
   updateClock: @|System
     (#
     do cycle
        (#
        do 1 -> sleep;
           systemTime -> formatTime -> mymotif.clock.labelString;
        #);
     #);
   mymotif: @motifEnv
     (# clock: @label
          (# init::< (# do systemTime->formatTime->labelString #)#);
     do clock.init;
     #);
do updateClock[] -> fork;
#)
```

When this program is run, the following window appears, in which the time is updated every second.



Wed Aug 31 13:12:44 1994

The same effect could be obtained by using a WorkProc.

Miscellaneous

In the two directories `misc` and `demo/misc`, a few X related extras are placed.

Xterm Interface

You can set the title and icon name of the xterm you are executing in, and you can start another UNIX process in a separate xterm window. The demo in

```
$BETALIB/demo/Xt/misc/xtermdemo.bet
```

shows examples of both.

EditRes Interface

The `editres` program is a standard program distributed with the X Window System. It allows graphical browsing and changing of a program's X resources.

To support the `editres` program, the program must support a special "EditRes Protocol". To enable this protocol, the `enableEditRes` method in

```
$BETALIB/Xt/misc/editres.bet
```

should be invoked. After this, the program's resources can be manipulated by `editres`.

Bibliography

[Nye & O'Reilly 90a]

Adrian Nye & Tim O'Reilly:
X Toolkit Intrinsics Programming Manual,
Volume Four of *The X Window System Series*,
O'Reilly & Associates Inc, 1990, ISBN 0-937175-34-X

[Nye & O'Reilly 90b]

Adrian Nye & Tim O'Reilly:
X Toolkit Intrinsics Programming Manual, OSF/Motif 1.1 Edition,
Volume Four (Motif) of *The X Window System Series*,
O'Reilly & Associates Inc, 1990, ISBN 0-937175-62-5

[Young 90]

Douglas A. Young:
The X Window System. Programming and Applications with Xt, OSF/Motif Edition,
Prentice Hall, 1990, ISBN 0-13-497074-8

[Jones 89]

Oliver Jones:
Introduction to the X Window System,
Prentice Hall, 1989, ISBN 0-13-499997-5

[Mansfield 89]

Niall Mansfield:
An X Window System User's Guide,
Addison-Wesley, Amsterdam, 1989, ISBN 0-201-51341-2.

[X WWW]

Kenton Lee:
Technical X Window System and Motif WWW Sites,
`http://www.rahul.net/kenton/xsites.html`.

[Motif WWW]

MW3:
Motif on the World Wide Web,
`http://www.cen.com/mw3`.