APPENDIX I

Athena Widget Resources

As suggested on the reference pages for various clients, you can set not only resources defined by the application itself, but also resources that apply to any of the widgets that make up the application. The reference page for the application sometimes lists the most important of these resources, but for fuller customization, you need to know more about each widget.

Unfortunately, the design of the X Toolkit is such that to really do the right thing, you probably need to know a bit more about Toolkit programming than the average user might like. See Appendix G, *Introduction to Xt Widget Resources*, if you are not familiar with the gory details of "the Widget hierarchy."

In this appendix, we provide some reference information about each class of widgets in the Athena Widget set used by the standard MIT clients..

I.1 Box

The Box widget provides geometry management of arbitrary widgets in a box of a specified dimension. Box moves but does not resize its children. The children are rearranged when the Box is resized, when its children are resized, or when children are managed or unmanaged. The Box widget always attempts to pack its children as closely as possible within the geometry allowed by its parent.

Box widgets are commonly used to manage a related set of Command widgets and are frequently called ButtonBox widgets, but the children are not limited to buttons.

The children are arranged on a background that has its own specified dimensions and color.

I.1.1 Resources

The following new resources are associated with the Box widget:

`hSpace` (class `HSpace`) Number of pixels to the left or to the right of each child. Default is 4.

`orientation` (class `Orientation`) Specifies whether the preferred shape of the box is tall and narrow (`vertical`, the default) or short and wide (`horizontal`).

`vSpace` (class `VSpace`) Number of pixels above or below each child. Default is 4.

I.2 Command

The Command widget is an area, often rectangular, that contains a text or pixmap label and calls an application function when "pressed" with a pointer button. This selectable area is sometimes referred to as a "button." When the pointer cursor is on the button, the button border is highlighted to indicate that the button is ready for selection. When a pointer button is pressed, the command widget indicates that it has been selected by reversing its foreground and background colors.

I.2.1 Resources

The following new resources are associated with the Command widget:

`highlightThickness` (class `Thickness`) The thickness of the line drawn by the `highlight` action.

`shapeStyle` (class `ShapeStyle`) Nonrectangular buttons may be created using this resource. Nonrectangular buttons are supported only on a server that supports the Shape Extension. If nonrectangular buttons are specified for a server lacking this extension, the shape is ignored and the widgets will be rectangular. The following shape names are currently supported: `Rectangle`, `Oval`, `Ellipse`, and `roundedRectangle`, in any case.

`cornerRoundPercent` (class `CornerRoundPercent`) When a `ShapeStyle` of `roundedRectangle` is used, this resource controls the radius of the rounded corner. The radius of the rounded corners is specified as a percentage of the length of the shortest side of the widget.

I.2.2 Translations and Actions

The following are the default translation bindings that are used by the Command widget:

```
<EnterWindow>:highlight()
<LeaveWindow>:reset()
<Btn1Down>:set()
<Btn1Up>:notify() unset()
```

With these bindings, the user can cancel the action before releasing the button by moving the pointer out of the Command widget.

The full list of actions supported by Command is as follows:

`highlight()` Displays the internal highlight border in the color (`foreground` or `background`) that contrasts with the interior color of the Command widget. This is normally bound to <EnterWindow> events, so the widget border is highlighted when the pointer enters the window.

`unhighlight()` Displays the internal highlight border in the color (`foreground` or `background`) that matches the interior color of the Command widget. This action is called internally by reset(), and does not need an explicit translation.

`set()` Enters the `set` state, in which `notify` is possible and displays the interior of the button, including the highlight border, in the foreground color. The label is displayed in the background color. This usually happens when a pointer button (button 1 by default) is pressed in the widget.

`unset()` Cancels the `set` state and displays the interior of the button, including the highlight border, in the background color. The label is displayed in the foreground color. This action is called internally by reset(), and does not need an explicit translation.

`reset()` Cancels any `set` or `highlight` and displays the interior of the button in the background color, with the label displayed in the foreground color. This is normally bound to <Btu1Up> (along with `notify`()) to reset the button appearance to its normal state, once the button's callback function has been executed.

`notify()` Executes the callback list specified by `callback`, if executed in the `set` state. This is the action that actually calls the application function to be invoked.

I.3 Dialog

The Dialog widget prompts you for additional input. 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 you type input. The third line can contain buttons that let you confirm or cancel the Dialog input.

Dialog is not really a widget, but an interface to a widget. It might also be thought of as a compound widget. It includes a label widget, a command widget, and a text widget as components. These could theoretically appear as subwidgets in a resource specification.

I.3.1 Resources

The following new resources are associated with the Dialog widget:

`icon` (class `Icon`) The name of a pixmap to be displayed immediately to the left of the Dialog widget's label.

`label` (class `Label`) A string to be displayed at the top of the Dialog widget.

`value` (class `Value`) An initial value for the string field into which you will enter text. By default, no text entry field is available. Specifying an initial value for `value` activates the text entry field. If string input is desired but no initial value is to be specified, then set this resource to `""` (empty string).

I.4 Form

The Form widget can contain an arbitrary number of children of any class. The Form provides geometry management for its children, including individual control of the position of each child. 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.

I.4.1 Resources

The following new resource is associated with the Form widget:

`defaultDistance` Specifies the default value for `horizDistance` and `vertDistance`. This value is four pixels, by default. 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.

Form is a subclass of Constraint, which means it has special kind of resources called constraint resources, which means that they apply to—and are specified as if they belong to—the child of the Form rather than to the Form itself. For example, *xcalc* uses a Form widget to organize its buttons. The resources below apply to the buttons, rather than to the Forms (e.g., xcalc.ti.button11.horizDistance : 4)

`bottom` (class `Edge`)

`top` (class `Edge`)

`left` (class `Edge`)

`right` (class `Edge`) Specify how to reposition the bottom, top, left, and right, respectively, of a child widget when the Form is resized. These resources can take one of five values. The values `ChainTop`, `ChainBottom`, `ChainLeft`, and `ChainRight` maintain a constant distance from an edge of the child to the top, bottom, left, and right edges, respectively, of the Form. The value `Rubber` (default) maintains a proportional distance from the edge of the child to the left or top edge of the Form. The proportion is determined from the initial position of the child and the initial size of the Form.

`fromHoriz` (class `Widget`)

`horizDistance` (class `Thickness`) Specify a child widget's horizontal position relative to another widget within the Form. `fromHoriz` is the name of the widget relative to which the child widget is placed, and `horizDistance` is the number of pixels separating the two widgets. For example, if `horizDistance` is 10, the child widget will be placed 10 pixels to the right of the widget defined in `fromHoriz`. If `fromHoriz` is not defined, then `horizDistance` is measured from the left edge of the Form.

`fromVert` (class `Widget`)

`vertDistance` (class `Thickness`) Similar to previous resources, except that `fromVert` and `vertDistance` position a child widget by a specified number of pixels *vertically* away from a specified widget. If no widget is specified for `fromVert`, then `vertDistance` is measured from the top of the Form.

`resizable` (class `Boolean`) `TRUE` if children are allowed to resize themselves. Default is `FALSE`.

I.5 Grip

The Grip widget provides a small region that allows button presses and button releases. The Grip widget is typically used as an attachment point for visually repositioning an object (for example, the pane border in a Paned widget).

I.5.1 Resources

The following Core resources may be useful with the Grip widget:

`foreground, width, height, borderWidth`.

I.5.2 Translations and Actions

The Grip widget does not declare any default event translation bindings, but it does declare a single action routine named `GripAction` in its action table. The client specifies an arbitrary event translation table, giving parameters to the `GripAction` routine.

The following is an example of a `GripAction` translation table:

```
<Btn1Down>: GripAction(press)
<Btn1Motion>: GripAction(move)
<Btn1Up>: GripAction(release)
```

I.6 Label

A Label is an non-editable text string or pixmap that is displayed within a window. The string may contain multiple lines of characters. It can be aligned to the left, right, or center of its window. A Label can be neither selected nor directly edited by the user.

I.6.1 Resources

The following resources are used by the Label widget:

`bitmap` (class `Pixmap`) Specifies a bitmap to display in place of the text label. In a resource file, the resource should be specified as the name of a file in the bitmap utility format that is to be loaded into a pixmap. The string can be an absolute or a relative filename. If a relative filename is used, the directory specified by the resource name `bitmapFilePath` or the resource class `BitmapFilePath` is added to the beginning of the specified filename. If the `bitmapFilePath` resource is not defined, the default directory on a POSIX-based system is */usr/include/X11/bitmaps* .

`font` (class `font`) The font of the label.

`foreground` (class `Foreground`) The color of the text string or pixmap.

`internalHeight` (class `Height`) Represents the distance in pixels between the top and bottom of the label text or bitmap and the horizontal edges of the Label widget. Default is 2 pixels.

`internalWidth` (class `Width`) Represents the distance in pixels between the ends of the label text or bitmap and the vertical edges of the Label widget. Default is 4 pixels.

`justify` (class `Justify`) Specifies left, center, or right alignment of the label string within the Label widget. One of the values `left`, `center`, or `right` can be specified.

`label` (class `Label`) Specifies the text string that is to be displayed in the button if no bitmap is specified. Note that the label may be hardcoded by the application.

`resize` (class `Resize`) A Boolean value that specifies whether the Label widget should attempt to resize to its preferred dimensions whenever `XtSetValues` is called for it. Default is `True`. Not usually set by users.

`rowSpacing` (class `Spacing`) Specify the amount of space in pixels between each of the rows in the list. The default is 6 pixels.

I.7 List

The List widget is a rectangle that contains a list of text strings formatted into rows and columns. When one of the strings is selected, it is highlighted, and an application callback routine is invoked. Only one string may be selected at a time. Note that most of the List resources are for application use.

I.7.1 Resources

The following resources are used by the List widget:

`callback` All functions on this list are called whenever the `notify` action is invoked. This resource cannot be set from within a resource file, but only within an application program.

`columnSpacing` (class `Spacing`) Specify the amount of space in pixels between each of the columns in the list. The default is 6 pixels.

`Cursor` (class `Cursor`) The cursor to be displaced when the pointer is in the List widget. Default is `left-ptr`.

`defaultColumns` (class `Columns`) Specifies the default number of columns, which is used when neither the width nor the height of the List widget is specified or when `forceColumns` is `True`. The default is 2.

`forceColumns` (class `Columns`) Specifies that the default number of columns is to be used no matter what the current size of the List widget is. The default is FALSE.

`font` (class `Font`) Specifies the font to be used to display the list.

`foreground` (class `Foreground`) Specifies the color to be used to paint the text of the list elements.

`internalHeight` (class `Height`) Represents a margin, in pixels, between the top and bottom of the list and the edges of the List widget. Default is 2 pixels.

`internalWidth` (class `Width`) Represents a margin, in pixels, between the left and right edges of the list and the edges of the List widget. Default is 4 pixels.

`longest` (class `Longest`) Specifies the length, in pixels, of the longest string in the current list. If the client knows the length, it should specify it; otherwise, the List widget computes a default length by searching through the list. This value is not typically set in resource files.

numberStrings (class NumberStrings) Specifies the number of strings in the current list. If a value is not specified, the list must be NULL-terminated. This value is not typically set in resource files.

pasteBuffer (class Boolean) If this is True, then the value of the string selected will be put into X cut buffer 0. The default is FALSE. (Normally, the selected item is simply passed to the application. For example, a filename might be passed to the application's "open" routine.)

verticalList (class Boolean) If this is True, the elements in the list are arranged vertically; if False, the elements are arranged horizontally.

I.7.2 Translations and Actions

The List widget has three predefined actions: Set, Unset, and Notify. Set and Unset allow switching the foreground and background colors for the current list item. Notify allows processing application callbacks.

The following is the default translation table used by the List widget:

```
<Btn1Down>,<Btn1Up>:Set() Notify()
```

These translations should not typically be modified by users, and may be hardcoded by the application.

I.8 MenuButton

The MenuButton widget is a subclass of the Command widget that is used to pop-up a menu. It is an area, often rectangular, that contains a text or pixmap label. This selectable area is referred to as a button. When the pointer cursor is on the button, the button border is highlighted to indicate that the button is ready for selection. When pointer button 1 is pressed, the MenuButton widget pops up the menu that has been named in the menuName resource.

I.8.1 Resources

MenuButton widgets have no new user-settable resources.

I.8.2 Translations and Actions

The following default translation bindings are used by the MenuButton widget:

```
<EnterWindow>:highlight()
    <LeaveWindow>:reset()
    <BtnDown>:reset(|) PopupMenu()
```

With these bindings, the user can cancel the action before releasing the button by moving the pointer out of the MenuButton widget.

The actions supported by MenuButton are listed below:

`highlight(`***`condition`***`)` Displays the internal highlight border in the color (`foreground` or `background`) that contrasts with the interior color of the MenuButton widget. The conditions `WhenUnset` and `Always` are understood by this action procedure. If no argument is passed, `WhenUnset` is assumed.

`unhighlight()` Displays the internal highlight border in the color (`foreground` or `background`) that matches the interior color of the MenuButton widget.

`set()` Enters the set state, in which `notify` is possible, and displays the interior of the button, including the highlight border, in the foreground color. The label or pixmap is displayed in the background color.

`unset()` Cancels the set state and displays the interior of the button, including the highlight border, in the background color. The label or pixmap is displayed in the foreground color.

`reset()` Cancels any `set` or `highlight` action and displays the interior of the button in the background color, with the label or pixmap displayed in the foreground color.

`PopupMenu()` Pops up the menu specified by the `menuName` resource.

I.9 Paned

The Paned widget manages children in a vertically or horizontally tiled fashion. You may resize these panes by using the *grips* that appear near the right or bottom edge of the border between two panes.

When you position the pointer on a grip, pressing the pointer button will display an arrow that indicates which pane is being resized. By keeping the pointer button down, you can move the pointer up and down (or left and right). This, in turn, changes the border between the panes, causing one pane to shrink and some other pane (or panes) to grow. The size of the Paned widget will not change.

The choice of alternate pane is a function of the `min`, `max`, and `skipAdjust` constraints on the other panes. With the default bindings, button 1 resizes the pane above or to the left of the selected grip, button 3 resizes the pane below or to the right of the selected grip, and button 2 repositions the border between two panes only.

I.9.1 Resources

The following new resources are associated with the Paned widget:

`betweenCursor` (class `Cursor`) Cursor for changing the boundary between two panes.

`cursor` (class `Cursor`) Pointer cursor image that displays, whenever the pointer is in this widget but not in any of its children (children may also inherit this cursor).

`gripCursor` (class `Cursor`) Cursor for grip when not active.

`gripIndent` (class `GripIndent`) Offset of grip from margin (in pixels). Default is 10.

`gripTranslations` (class `Translations`) Button bindings for grip.

horizontalBetweenCursor (class Cursor) Cursor to use for the grip when changing the boundary between two panes. Default is sb_up_arrow.

horizontalGripCursor (class Cursor) Cursor to use for the grips when they are not active. Default is sb_h_double_arrow.

internalBorderColor (class BorderColor) Internal border color of the widget's window. Default is XtDefaultForeground.

internalBorderWidth (class BorderWidth) Amount of space (in pixels) kept between panes. Default is 1.

leftCursor (class Cursor) Cursor used when resizing the pane to the left of the grip. Default is sb_left_arrow.

lowerCursor (class Cursor) Cursor used when resizing the pane below the grip. Default is sb_down_arrow.

orientation (class Orientation) Orientation to use in stacking the panes. This value can be either Vertical (the default) or Horizontal.

refigureMode (class Boolean) A Boolean that specifies whether the Paned widget should adjust its children. Default is TRUE.

rightCursor (class Cursor) Cursor used when resizing the pane to the right of the grip. Default is sb_right_arrow.

upperCursor (class Cursor) Cursor used when resizing the pane above the grip. Default is sb_up_arrow.

verticalBetweenCursor (class Cursor) Cursor to use for the grip when changing the boundary between two panes. Default is sb_left_arrow.

verticalGripCursor (class Cursor) Cursor to use for the grips when they are not active. Default is sb_v_double_arrow.

Paned supports the following constraint resources. They can be specified to the Paned widget to indicate where a child widget should be positioned within the Paned widget.

allowResize (class Boolean) A Boolean that specifies whether to accept a child's request to resize. The default, FALSE, is to ignore such requests.

max (class max) Maximum height for pane. Default is to allow unlimited height.

min (class min) Minimum height for pane (in pixels). Default is 1.

preferredPaneSize (class PreferredPaneSize) Preferred size of pane. This default is dependent on the application.

resizeToPreferred (class Boolean) A Boolean that specifies whether to resize each pane to its preferred size when the Paned widget is resized. Default is False.

showGrip (class ShowGrip) A Boolean that specifies whether to show a grip for this pane. Default is True.

skipAdjust (class Boolean) By default, this resource is FALSE, meaning that the Paned widget will resize a pane automatically, whenever necessary. If this resource is TRUE, the Paned widget will skip the adjustment of the pane. .SH "Translations and Actions" The Paned widget has no action routines of its own, as all actions are handled through the grips. The grips are each assigned a default Translation table.

```
<Btn1Down>:GripAction(Start, UpLeftPane)
    <Btn2Down>:GripAction(Start, ThisBorderOnly)
    <Btn3Down>:GripAction(Start, LowRightPane)
    <Btn1Motion>:GripAction(Move, UpLeftPane)
    <Btn2Motion>:GripAction(Move, ThisBorderOnly)
    <Btn3Motion>:GripAction(Move, LowRightPane)
    Any<BtnUp>:GripAction(Commit)
```

The Paned widget interprets the GripAction as taking two arguments. The first argument may be any of the following:

```
StartUpLeftPane, ThisBorderOnly, LowRightPane).
```

MoveStart action that began this process. If these arguments are not passed, the behavior is undefined.

Commit

I.10 Scrollbar

The Scrollbar widget is a rectangular area that contains a slide region and a thumb (slide bar). A Scrollbar can be used alone (to provide a graduated scale) or within a composite widget (for example, a Viewport). A Scrollbar can be aligned 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.

I.10.1 Resources

You can set the dimensions of the Scrollbar two ways:

- By using the width and height resources, as you can for all widgets.
- By using the Scrollbar resources length and thickness, which are independent of the vertical or horizontal orientation.

The following new resources are associated with the Scrollbar widget:

foreground (class Foreground) Thumb color.

length (class Length) Specifies the height for a vertical Scrollbar and the width for a horizontal Scrollbar. Default is 1 (pixel).

minimumThumb (class MinimumThumb) Smallest size, in pixels, to which the thumb can shrink. Default is 7.

orientation (class Orientation) Orientation of scrollbar. This value can be either XtOrientVertical (the default) or XtOrientHorizontal. Not usually set in resource files.

`scrollDCursor` (class `Cursor`) Cursor for scrolling down. Default is `XC_sb_down_arrow`.

`scrollHCursor` (class `Cursor`) Idle horizontal cursor. Default is `XC_sb_h_double_arrow`.

`scrollLCursor` (class `Cursor`) Cursor for scrolling left. Default is `XC_sb_left_arrow`.

`scrollRCursor` (class `Cursor`) Cursor for scrolling right. Default is `XC_sb_right_arrow`.

`scrollUCursor` (class `Cursor`) Cursor for scrolling up. Default is `XC_sb_up_arrow`.

`scrollVCursor` (class `Cursor`) Idle vertical cursor. Default is `XC_sb_v_double_arrow`.

`shown` (class `Shown`) Percentage the thumb covers. Default is 0.0.

`thickness` (class `Thickness`) Specifies the width for a vertical Scrollbar and the height for a horizontal Scrollbar. Default is 14 (pixels).

`thumb` (class `Thumb`) Thumb pixmap. Default is `GrayPixmap`.

`topOfThumb` (class `TopOfThumb`) Position on scroll bar. Default is 0.0.

I.10.2 Translations and Actions

The actions supported by the Scrollbar widget are:

`StartScroll(`***`value`***`)` The possible values are `Forward`, `Backward`, or `Continuous`. This must be the first action to begin a new movement.

`NotifyScroll(`***`value`***`)` The possible values are `Proportional` or `FullLength`. If the argument to `StartScroll` was `Forward` or `Backward`, `NotifyScroll` executes the `XtNscrollProc` callbacks and passes either the position of the pointer if its argument is `Proportional` or the full length of the scroll bar if its argument is `FullLength`. If the argument to `StartScroll` was `Continuous`, `NotifyScroll` returns without executing any callbacks.

`EndScroll()` This must be the last action after a movement is complete.

`MoveThumb()` Repositions the scroll bar thumb to the current pointer location.

`NotifyThumb()` Calls the `XtNjumpProc` callbacks and passes the relative position of the pointer as a percentage of the scroll bar length.

The default bindings for Scrollbar are:

```
<Btn1Down>:StartScroll(Forward)
    <Btn2Down>:StartScroll(Continuous) MoveThumb() NotifyThumb()
    <Btn3Down>:StartScroll(Backward)
    <Btn2Motion>:MoveThumb() NotifyThumb()
    <BtnUp>:NotifyScroll(Proportional) EndScroll()
```

Examples of additional bindings you might wish to specify in a resource file are:

```
*Scrollbar.Translations: \
~<KeyPress>f: StartScroll(Forward) NotifyScroll(FullLength) EndScroll()
```

```
<KeyPress>b: StartScroll(Backward) NotifyScroll(FullLength) EndScroll()
```

I.11 Simple

The Simple widget defines characteristics that are inherited by non-composite widgets such as Labels, Lists, and Scrollbars. The Simple widget never appears in applications, but it does define resources that are inherited by its subclasses.

I.11.1 Resources

The following resources are associated with the Simple widget:

`cursor` (class `Cursor`) The cursor to use within the widget. Default is none.

`insensitiveBorder` (class `Insensitive`) The pixmap to use to indicate that the Simple widget cannot receive input. Default is `Gray`.

I.12 SimpleMenu

The SimpleMenu widget is a container for menu entries. It is a direct subclass of Shell. This is the only part of the menu that actually contains a window, since each menu pane is a gadget (a widget without a window). SimpleMenu "glues" the individual menu entries together into one menu.

I.12.1 Resources

The following new resources are used by the SimpleMenu widget:

`backingStore` (class `BackingStore`) Determines what type of backing store will be used for the menu. Legal values for this resource are `NotUseful`, `WhenMapped`, and `Always`. These values are the backing-store integers defined in *<X11/X.h>*. If `default` is specified (the default behavior) the server will use whatever it thinks is appropriate. This resource is typically set by the application.

`bottomMargin`, `topMargin` (class `VerticalMargins`) The amount of space between the top or bottom of the menu and the menu entry closest to that edge. Default is 0.

`cursor` (class `Cursor`) The shape of the mouse pointer whenever it is in this widget.

`popupOnEntry` (class `PopupOnEntry`) The `XawPositionSimpleMenu` action pops up the SimpleMenu with its label (or first entry) directly under the pointer, by default. To pop up the menu under another entry, the application can set this resource to the menu entry that *should* be under the pointer when the menu is popped up. This allows the application to offer the user a default menu entry that can be selected without moving the pointer. Not usually settable by the user.

`rowHeight` (class `RowHeight`) If this resource is 0 (the default), then each menu entry is given its desired height. If this resource has any other value, then all menu entries are forced to be `rowHeight` pixels high.

I.12.2 Translations and Actions

The following default translation bindings are used by the SimpleMenu widget:

```
<EnterWindow>:highlight()
    <LeaveWindow>:unhighlight()
    <BtnMotion>:highlight()
    <BtnUp>:MenuPopdown() notify() unhighlight()
```

With these bindings, the user can pop down the menu without activating any of the callback functions, by releasing the pointer button when no menu item is highlighted.

The actions supported by SimpleMenu are listed below:

`highlight()` Highlights the menu entry that is currently under the pointer. Only an item that is highlighted is notified when the `notify` action is invoked. The look of a highlighted entry is determined by the menu entry.

`unhighlight()` Unhighlights the currently highlighted menu item and returns it to its normal look.

`notify()` Notifies the currently highlighted menu entry that it has been selected. It is the responsibility of the menu entry to take the appropriate action.

`MenuPopdown`(***menu***) Built-in action to pop down a menu widget.

I.13 Sme

The Sme object is the base class for all menu entries that are children of SimpleMenu. While this object is intended mainly to be subclassed, it may be used in a menu to add blank space between menu entries.

I.13.1 Resources

The Sme object defines no new resources.

I.14 SmeBSB

The SmeBSB object is used to create a menu entry that contains a string, and optional bitmaps in its left and right margins. The parent is expected to be SimpleMenu. Since each menu entry is an independent object, the application is able to change the font, color, height, and other attributes of the menu entries, on an entry-by-entry basis.

I.14.1 Resources

The following resources are used by the SmeBSB object:

`font` (class `Font`) Specifies the font used by the menu entry.

`foreground` (class `Foreground`) Specifies the foreground color of the menu entry's window. This color is also used to render all 1's in `leftBitmap` and `rightBitmap`.

`justify` (class `Justify`) Specifies how the label is to be rendered between the left and right margins when the space is wider than the actual text. When specifying the justification from a resource file, the values `left`, `center`, or `right` may be used.

`label` (class `Label`) Specifies the string to be displayed in the menu entry. The exact location of this string within the bounds of the menu entry is controlled by the resources `leftMargin`, `rightMargin`, `vertSpace`, and `justify`.

`leftBitmap` (class `LeftBitmap`), `rightBitmap` (class `RightBitmap`) Specifies a name of a bitmap to display in the left or right margin of the menu entry. All 1's in the bitmap are rendered in the foreground color of the SimpleMenu widget, and all 0's will be drawn in the background color of the SimpleMenu widget. The programmer must ensure that the menu entry is tall enough and that the appropriate margin is wide enough to accept the bitmap. If care is not taken, the bitmap might extend into either another menu entry or this entry's label. This resource is typically set by the application.

`leftMargin`, `rightMargin` (class `HorizontalMargins`) Specifies the amount of space (in pixels) to leave between the edge of the menu entry and the label string.

`vertSpace` (class `VertSpace`) Specifies the amount of vertical padding to place around the label of a menu entry. The label and bitmaps are always centered vertically within the menu. Values for this resource are expressed as a percentage of the font's height. The default value (25) increases the default height to 125% of the font's height.

I.15 SmeLine

The SmeLine object is used to add a horizontal line or menu separator to a SimpleMenu. Since each menu entry is an independent object, the application is able to change the color, height, and other attributes of the menu entries, on an entry-by-entry basis. This entry is not selectable, and does not highlight when the pointer cursor is over it.

I.15.1 Resources

The following resources are used by the SmeLine object:

`foreground` (class `Foreground`) The foreground color of the menu entry's window.

`lineWidth` (class `LineWidth`) The width of the horizontal line to be displayed.

`stipple` (class `Stipple`) If a bitmap is specified for this resource, the line will be stippled through it. This allows the menu separator to be rendered as something more exciting than just a line. For instance, if the application defines a stipple that is a chain link, then menu separators will look like chains.

I.16 StripChart

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

I.16.1 Resources

The following resources are used by the StripChart widget:

`height` (class `Height`) The height of the stripchart. Default is 120 pixels.

`highlight` (class `Foreground`) The color that will be used to draw the scale lines on the graph.

`jumpScroll` (class `JumpScroll`) When the graph reaches the right edge of the window it must be scrolled to the left. This resource specifies the number of pixels it will jump. Smooth scrolling can be achieved by setting this resource to 1.

`minScale` (class `Scale`) The minimum scale for the graph. The number of divisions on the graph will always be greater than or equal to this value. Default is 1.

`update` (class `Interval`) The number of seconds between graph updates. Each update is represented on the graph as a 1-pixel-wide line. Every `update` seconds, a new graph point will be added to the right end of the StripChart. Default is 10.

`width` (class `Width`) The width of the stripchart. Default is 120 pixels.

I.17 Text

A Text widget is a window that provides a way for an application to display one or more lines of text. The displayed text can reside in a file on disk or in a string in memory. An option also lets an application display a vertical Scrollbar in the Text window, letting the user scroll through the displayed text. Other options allow an application to let the user modify the text in the window or search for a specific string.

Three types of edit mode are available: .RS 3n

" Append-only mode lets the user enter text into the window, while read-only mode does not. Text may be entered only if the insertion point is after the last character in the window. Editable mode lets you place the cursor anywhere in the text and modify the text at that position. The text cursor position can be modified by using the keystrokes or pointer buttons defined by the event bindings. (See the section "Translations and Actions" below.)

I.17.1 Resources

The following resources are used by the Text widget:

`autoFill` (class `AutoFill`) A Boolean that specifies whether the Text widget will automatically break a line when the user attempts to type into the right margin Default is `False`.

`bottomMargin` (class `Margin`) Amount of space, in pixels, between the edge of the window and the edge of the text within the window. Default is 2.

`dataCompression` If `True` (the default), the `AsciiSrc` will compress its data to the minimum size required. This will happen either every time the text string is saved or whenever the value of the string is queried.

`displayCaret` (class `Output`) A Boolean that specifies whether to display the text caret. Default is `True`.

`displayPosition` (class `TextPosition`) Character position of first line. Default is 0.

`insertPosition` (class `TextPosition`) Character position of caret. Default is 0.

`leftMargin` (class `Margin`) Left margin in pixels. Default is 2.

`rightMargin` (class `Margin`) Amount of space, in pixels, between the edge of the window and the corresponding edge of the text within the window. Default is 2.

`scrollHorizontal`

`scrollVertical` Control the placement of scrollbars on the left and bottom edge of the text widget. Possible values are `XawtextScrollAlways`, `XawtextScrollWhenNeeded`, and `XawtextScrollNever` (the default). Not settable from a resource file.

`topMargin` (class `Margin`) Amount of space, in pixels, between the edge of the window and the corresponding edge of the text within the window. Default is 2.

`useStringInPlace` If `True`, will disable the memory management provided by the Text widget, updating the `string` resource instead. Default is `False`.

I.17.2 Translations and Actions

Many standard keyboard editing facilities are supported by the event bindings. The following actions are supported: l| l. Cursor MovementDelete

–

.TH Forward-characterDelete-next-character Backward-characterDelete-previous-character Forward-wordDelete-next-word Backward-wordDelete-previous-word Forward-paragraphDelete-selection Backward-paragraph Beginning-of-lineSelection End-of-line Insert-selection Next-lineSelect-word Previous-lineSelect-all Next-pageSelect-start Previous-pageSelect-adjust Beginning-of-fileSelect-end End-of-fileExtend-start Scroll-one-line-upExtend-adjust Scroll-one-line-downExtend-end New LineMiscellaneous Newline-and-indentRedraw-display Newline-and-backupInsert-file NewlineInsert-char KillDisplay-caret Kill-wordFocus-in Backward-kill-wordFocus-out Kill-selectionSearch Kill-to-end-of-lineMultiply Kill-paragraphForm-paragraph Kill-to-end-of-paragraphTranspose-characters No-op

–

$delete action deletes a text item. The `kill` action deletes a text item and puts the item in the kill buffer (X cut buffer 1).

$insert-selection action retrieves the value of a specified X selection or cut buffer, with fallback to alternative selections or cut buffers.

I.17.2.1 Cursor Movement Actions

`forward-character()`

`backward-character()` These actions move the insert point forward or backward one character in the buffer. If the insert point is at the end (or beginning) of a line, this action moves the insert point to the next (or previous) line.

`forward-word()`

`backward-word()` These actions move the insert point to the next or previous word boundary. A word boundary is defined as a space, a tab, or a carriage return.

`forward-paragraph()`

`backward-paragraph()` These actions move the insert point to the next or previous paragraph boundary. A paragraph boundary is defined as two carriage returns in a row with only spaces or tabs between them.

`beginning-of-line()`

`end-of-line()` These actions move to the beginning or end of the current line. If the insert point is already at the end or beginning of the line, no action is taken.

`next-line()`

`previous-line()` These actions move the insert point up or down one line. If the insert point is currently *n* characters from the beginning of the line then it will be *n* characters from the beginning of the next or previous line. If *n* is past the end of the line, the insert point is placed at the end of the line.

`next-page()`

`previous-page()` These actions move the insert point up or down one page in the file. One page is defined as the current height of the text widget. These actions always place the insert point at the first character of the top line.

`beginning-of-file()`

`end-of-file()` These actions place the insert point at the beginning or end of the current text buffer. The text widget is then scrolled the minimum amount necessary to make the new insert point location visible.

`scroll-one-line-up()`

`scroll-one-line-down()` These actions scroll the current text field up or down by one line. They do not move the insert point. Other than the scrollbars, this is the only way that the insert point may be moved off of the visible text area. The widget will be scrolled so that the insert point is back on the screen as soon as some other action is executed.

I.17.2.2 Delete Actions

`delete-next-character()`

`delete-previous-character()` These actions remove the character immediately after or before the insert point. If a carriage return is removed, the next line is appended to the end of the current line.

`delete-next-word()`

`delete-previous-word()` These actions remove all characters between the insert point location and the next word boundary. A word boundary is defined as a space, a tab or a carriage return.

`delete-selection()` This action removes all characters in the current selection. The selection can be set with the selection actions.

I.17.2.3 Selection Actions

`select-word()` This action selects the word in which the insert point is currently located. If the insert point is between words, it will select the previous word.

`select-all()` This action selects the entire text buffer.

`select-start()` This action sets the insert point to the current pointer location, where a selection then begins. If many of these selection actions occur quickly in succession then the selection count mechanism will be invoked.

`select-adjust()` This action allows a selection started with the `select-start` action to be modified, as described above.

`select-end(`***name***[,***name***,...]) This action ends a text selection that began with the `select-start` action, and asserts ownership of the selection or selections specified. A ***name*** can be a selection (e.g., PRIMARY) or a cut buffer (e.g., CUT_BUFFER0). Note that case is important. If no ***names*** are specified, PRIMARY is asserted.

`extend-start()` This action finds the nearest end of the current selection, and moves it to the current pointer location.

`extend-adjust()` This action allows a selection started with an `extend-start` action to be modified.

`extend-end(`***name***[,***name***,...]) This action ends a text selection that began with the `extend-start` action, and asserts ownership of the selection or selections specified. A ***name*** can be a selection (e.g., PRIMARY) or a cut buffer (e.g., CUT_BUFFER0). Note that case is important. If no ***name*** is given, PRIMARY is asserted.

`insert-selection(`***name***[,***name***,...]) This action retrieves the value of the first (leftmost) named selection that exists or the cut buffer that is not empty. This action then inserts it into the Text widget at the current insert point location. A ***name*** can be a selection (e.g., PRIMARY) or a cut buffer (e.g., CUT_BUFFER0). Note that case is important.

I.17.2.4 New Line Actions

`newline-and-indent()` This action inserts a newline into the text and adds spaces to that line to indent it to match the previous line. (Note: this action still has a few bugs.)

`newline-and-backup()` This action inserts a newline into the text *after* the insert point.

`newline()` This action inserts a newline into the text *before* the insert point.

I.17.2.5 Kill Actions

`kill-word()`

`backward-kill-word()` These actions act exactly like the `delete-next-word` and `delete-previous-word` actions, but they store the word that was killed into the kill buffer (CUT_BUFFER_1).

`kill-selection()` This action deletes the current selection and stores the deleted text into the kill buffer (CUT_BUFFER_1).

`kill-to-end-of-line()` This action deletes the entire line to the right of the insert point, and stores the deleted text into the kill buffer (CUT_BUFFER_1).

`kill-paragraph()` This action deletes the current paragraph. If the insert point is between paragraphs, it deletes the paragraph above the insert point, and stores the deleted text into the kill buffer (CUT_BUFFER_1).

`kill-to-end-of-paragraph()` This action deletes everything between the current insert point and the next paragraph boundary, and puts the deleted text into the kill buffer (CUT_BUFFER_1).

I.17.2.6 Miscellaneous Actions

`redraw-display()` This action recomputes the location of all the text lines on the display, scrolls the text to center vertically the line containing the insert point on the screen, clears the entire screen, and then redisplays it.

`insert-file([`***`filename`***) This action activates the insert file popup. The ***`filename`*** option specifies the default filename to put in the filename buffer of the popup. If no ***`filename`*** is specified the buffer is empty at startup.

`insert-char()` This action may be attached only to a key event. It calls `XLookupString` to translate the event into a (rebindable) Latin-1 character (sequence) and inserts that sequence into the text at the insert point position.

`insert-string(`***`string`***[,***`string`***,...]) This action inserts each ***`string`*** into the text at the insert point location. Any ***`string`*** beginning with the characters "0x" and containing only valid hexadecimal digits in the remainder is interpreted as a hexadecimal constant and the corresponding single character is inserted instead.

`display-caret(`***`state`***,***`when`***) This action allows the insert point to be turned on and off. The ***`state`*** argument specifies the desired state of the insert point. This value may be any of the string values accepted for Boolean resources (e.g., `on`, `True`, `off`, `False`, etc.). If no arguments are specified, the default value is `True`. The ***`when`*** argument specifies, for `EnterNotify` or `LeaveNotify` events, whether or not the focus field in the event is to be examined. If the second argument is not specified, or specified as something other than `always`, then if the action is bound to an `EnterNotify` or `LeaveNotify` event, the action will be taken only if the focus field is `True`. An augmented binding that might be useful is:

```
*Text.Translations: #override \
<FocusIn>:display-caret(on) \n\
<FocusOut>:display-caret(off)
```

`focus-in()`

`focus-out()` These actions do not currently do anything.

`search(`***direction***,[***string***) This action activates the search popup. The ***direction*** must be specified as either `forward` or `backward`. The string is optional and is used as an initial value for the "Search for:" string.

`multiply(`***value***) The multiply action allows the user to multiply the effects of many of the text actions. Thus the following action sequence:

```
multiply(10) delete-next-word()
```

will delete 10 words. It does not matter whether these actions take place in one event or many events. Using the default translations the key sequence Control-u, Control-d will delete 4 characters. Multiply actions can be chained; thus,

```
multiply(5) multiply(5)
```

is the same as:

```
multiply(25)
```

If the string `reset` is passed to the multiply action the effects of all previous multiplies are removed and a beep is sent to the display.

`form-paragraph()` This action removes all the carriage returns from the current paragraph and reinserts them so that each line is as long as possible, while still fitting on the current screen. Lines are broken at word boundaries if at all possible. This action currently works only on Text widgets that use ASCII text.

`transpose-characters()` This action will switch the positions of the character to the left of the insert point and the character to the right of the insert point. The insert point will then be advanced one character.

`no-op([`***action***) The no-op action makes no change to the text widget, and is used mainly to override translations. This action takes one optional argument. If this argument is `RingBell` then a beep is sent to the display.

I.17.2.7 Event Bindings

The default event bindings for the Text widget are:

```
char defaultTextTranslations[] = "\ Ctrl<Key>F:forward-character() \n\
             Ctrl<Key>B:backward-character() \n\
             Ctrl<Key>D:delete-next-character() \n\
             Ctrl<Key>A:beginning-of-line() \n\
             Ctrl<Key>E:end-of-line() \n\
             Ctrl<Key>H:delete-previous-character() \n\
             Ctrl<Key>J:newline-and-indent() \n\
             Ctrl<Key>K:kill-to-end-of-line() \n\
             Ctrl<Key>L:redraw-display() \n\
             Ctrl<Key>M:newline() \n\
             Ctrl<Key>N:next-line() \n\
             Ctrl<Key>O:newline-and-backup() \n\
             Ctrl<Key>P:previous-line() \n\
             Ctrl<Key>V:next-page() \n\
             Ctrl<Key>W:kill-selection() \n\
             Ctrl<Key>Y:unkill() \n\
             Ctrl<Key>Z:scroll-one-line-up() \n\
             Meta<Key>F:forward-word() \n\
             Meta<Key>B:backward-word() \n\
             Meta<Key>I:insert-file() \n\
             Meta<Key>K:kill-to-end-of-paragraph() \n\
             Meta<Key>V:previous-page() \n\
             Meta<Key>Y:stuff() \n\
             Meta<Key>Z:scroll-one-line-down() \n\
             :Meta<Key>d:delete-next-word() \n\
             :Meta<Key>D:kill-word() \n\
             :Meta<Key>h:delete-previous-word() \n\
             :Meta<Key>H:backward-kill-word() \n\
             :Meta<Key>\<:beginning-of-file() \n\
             :Meta<Key>\>:end-of-file() \n\
             :Meta<Key>]:forward-paragraph() \n\
             :Meta<Key>[:backward-paragraph() \n\
             ~Shift Meta<Key>Delete:delete-previous-word() \n\
              Shift Meta<Key>Delete:backward-kill-word(|) \n\
             ~Shift Meta<Key>Backspace:delete-previous-word() \n\
              Shift Meta<Key>Backspace:backward-kill-word(|) \n\
             <Key>Right:forward-character() \n\
             <Key>Left:backward-character() \n\
             <Key>Down:next-line() \n\
             <Key>Up:previous-line() \n\
             <Key>Delete:delete-previous-character() \n\
             <Key>BackSpace:delete-previous-character() \n\
             <Key>Linefeed:newline-and-indent() \n\
             <Key>Return:newline() \n\ <Key>:insert-char() \n\
             <FocusIn>:focus-in() \n\ <FocusOut>:focus-out() \n\
             <Btn1Down>:select-start() \n\
             <Btn1Motion>:extend-adjust() \n\
             <Btn1Up>:extend-end(PRIMARY, CUT_BUFFER0) \n\
             <Btn2Down>:insert-selection(PRIMARY, CUT_BUFFER0) \n\
             <Btn3Down>:extend-start() \n\
             <Btn3Motion>:extend-adjust() \n\
             <Btn3Up>:extend-end(PRIMARY, CUT_BUFFER0)";
```

A user-supplied resource entry can use application-specific bindings, a subset of the supplied default bindings, or both. The following is an example of a user-supplied resource entry that uses a subset of the default bindings:

```
Xmh*Text.Translations: \
    <Key>Right:forward-character() \n\
    <Key>Left:backward-character() \n\
     Meta<Key>F:forward-word() \n\
     Meta<Key>B:backward-word() \n\
```

```
        :Meta<Key>]:forward-paragraph() \n\
        :Meta<Key>[:backward-paragraph() \n\
        <Key>:  insert-char()
```

An augmented binding that is useful with the *xclipboard* utility is:

```
    *Text.Translations: #override \
        Button1 <Btn2Down>:extend-end(CLIPBOARD)
```

The Text widget fully supports the X selection and cut buffer mechanisms. The following actions can be used to specify button bindings that will cause Text to assert ownership of one or more selections, to store the selected text into a cut buffer, and to retrieve the value of a selection or cut buffer and insert it into the text value.

`insert-selection(`***name***`[,`***name***`,...])` Retrieves the value of the first (left-most) named selection that exists or the cut buffer that is not empty and inserts it into the input stream. The specified name can be that of any selection (for example, PRIMARY or SECONDARY) or a cut buffer (i.e., CUT_BUFFER0 through CUT_BUFFER7). Note that case matters.

`select-start()` Unselects any previously selected text and begins selecting new text.

`select-adjust()`

`extend-adjust()` Continues selecting text from the previous start position.

`start-extend()` Begins extending the selection from the farthest (left or right) edge.

`select-end(`***name***`[,`***name***`,...])`

`extend-end(`***name***`[,`***name***`,...])` Ends the text selection, asserts ownership of the specified selection(s), and stores the text in the specified cut buffer(s). The specified name can be that of a selection (for example, PRIMARY or SECONDARY) or a cut buffer (i.e., CUT_BUFFER0 through CUT_BUFFER7). Note that case is significant. If CUT_BUFFER0 is listed, the cut buffers are rotated before storing into buffer 0.

I.18 Toggle

The Toggle widget is an area, often rectangular, containing a text or pixmap label. 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 border is highlighted to indicate 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 `notify` action 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 Toggle buttons in which no more than one Toggle may be set at any time. A radio group is identified by the widget ID of any one of its members.

The difference between a Command widget and a Toggle widget is that a Command widget typically invokes an application function when it is invoked. A Toggle widget simply changes its state (and presumably the state of some application data.) Toggles are thus useful for mailing configuration settings, which can then be applied by an associated Command widget.

I.18.1 Resources

The following new resources are associated with the Toggle widget:

`radioGroup` (class `Widget`) Specifies another Toggle widget that is in the radio group to which this Toggle widget should be added. A radio group is a group of Toggle widgets, only one of which may be set at a time. If this value is `NULL` (the default), then the Toggle is not part of any radio group and can change state without affecting any other Toggle widgets. If the widget specified in this resource is not already in a radio group, then a new radio group is created containing these two Toggle widgets. No Toggle widget can be in multiple radio groups.

`state` (class `State`) Specifies whether the Toggle widget is set (`True`) or unset (`False`). The default is `False`.

I.18.2 Translations and Actions

The following default translation bindings are used by the Toggle widget:

```
<EnterWindow>:highlight(Always)
    <LeaveWindow>:unhighlight()
    <Btn1Down>,<Btn1Up>:toggle() notify()
```

The actions supported by Toggle are listed below:

`highlight(`***`condition`***`)` Displays the internal highlight border in the color (`foreground` or `background`) that contrasts with the interior color of the Toggle widget. The conditions `WhenUnset` and `Always` are understood by this action procedure. If no argument is passed, `WhenUnset` is assumed.

`unhighlight()` Displays the internal highlight border in the color (`foreground` or `background`) that matches the interior color of the Toggle widget.

`set()` Enters the set state, in which `notify` is possible, and displays the interior of the button in the foreground color. The label or pixmap is displayed in the background color.

`unset()` Cancels the set state and displays the interior of the button, including the highlight border, in the background color. The label or pixmap is displayed in the foreground color.

`toggle()` Changes the current state of the Toggle widget, setting the widget if it was previously unset, and unsetting it if it was previously set. If the widget is to be set and is in a radio group, then this action procedure may unset another Toggle widget, causing all routines on its callback list to be invoked. The callback routines for the Toggle to be unset are called before those for the Toggle to be set.

`reset()` Cancels any `set` or `highlight` action and displays the interior of the button in the background color, with the label displayed in the foreground color.

I.18.3 Radio Groups

Two types of radio groups are typically desired by applications. In the first type, the default translations for the Toggle widget implement a "zero, or one of many" radio group. This means that no more than one button can be active, but no buttons need to be active.

The other type of radio group is "one of many" and has the more restricted policy that exactly one radio button will always be active. Toggle widgets can be used to provide this interface by modifying the translation table of each Toggle in the group:

```
<EnterWindow>: highlight(Always)
    <LeaveWindow>: unhighlight()
    <Btn1Down>,<Btn1Up>: set() notify()
```

This translation table does not allow any Toggle to be unset unless another Toggle has been set.

I.19 Viewport

The Viewport widget consists of a frame window, one or two Scrollbars, and an inner window (usually containing a child widget). 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.

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 scroll bars. The `forceBars` resource causes the Viewport widget to display any scroll bar permanently.

I.19.1 Resources

The following new resources are associated with the Viewport widget:

`allowHoriz` (class `Boolean`) Flag to allow horizontal scroll bars. Default value is `FALSE`. Setting this resource to `TRUE` allows a Viewport child to increase in size horizontally.

`allowVert` (class `Boolean`) Flag to allow vertical scroll bars. Default value is `FALSE`. Setting this resource to `TRUE` allows a Viewport child to increase in size vertically.

`forceBars` (class `Boolean`) Flag to force display of scroll bars. Default value is `FALSE`. Normally, when the geometry of the frame window is equal in size to the inner window, or when the data does not require scrolling, Viewport automatically removes any scroll bars. Setting `forceBars` to `TRUE` causes the Viewport widget to display any scroll bar permanently.

`useBottom` (class `Boolean`) Flag to indicate bottom/top bars. Default is `FALSE`, meaning to put scrollbars on top.

`useRight` (class `Boolean`) Flag to indicate right/left bars. Default is `FALSE`, meaning to put scrollbars on the left.