PROVIDING (REALLY) OPEN TECHNOLOGY.

The combo box widget

March 1995 — Version 1.31

Harald Albrecht

Institut für Geometrie und Praktische Mathematik
Rheinisch-Westfälische Technische Hochschule Aachen
52056 Aachen, Bundesrepublik Deutschland
EMail: `albrecht@igpm.rwth-aachen.de`
WWW: `file://ftp.igpm.rwth-aachen.de/arc/pub/unix/html/motifcorner.html`

Contents

List of Figures

`albrecht@igpm.rwth-aachen.de`

Chapter 1
A combo box?

1.1 The true motifation

*Once upon a long time, there was a worldwide movement with the somewhat strange name "comp.windows.x.motif". In that time there were many people possessed by a space and time consuming demon called "GUI". He pretends, that people could do with his help all things easier, faster and – best of all – most intuitive. And due to the fact, that this movement was worldwide, every few days someone grouched at the same thing. "We do need a suitable thing, so our programs can stay in an intuitive touch with the user". As their magic spell they repeated again and again: "We need a combo box". With regard to this the programmers, who were not already obsessed, created their own magic. And with their computers running the fabulous U**x operating system they got a much stronger magic than all those, who only own some enchanted PC box.*

Well – and if they are not dead, they will continue their work on the fabulous combo box magic. Some people say, X11 required enhancement a long time ago and this can only lead to its salvation, if someone can figure out what it's good for and what sense it makes. But – just in this moment it will disappear and be replaced by something strange and even more incomprehensible. Other people believe, that this has already happened[1].

Even if you haven't a large knowledge of the X11 or – more worse – the Motif magic, below you'll get to know all the necessary ingredients. With these ingredients (mainly consisting of c source code and a makefile) you can obtain a ready-to-go combo box. You need an ANSI compliant c compiler (sorry Mister K&R, but your style is now really out of date) and the right motifation with the Motif library (version 1.1 or 1.2). The code should also do its duty when running with Motif 2.0.

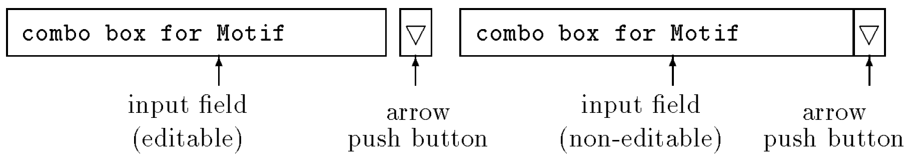

Figure 1.1: A combo box

For all those, who have never seen a combo box once before (or even thought of such a thing), I'll now provide a short description:

[1] *My apologies to Douglas Adams and – of course – the OSF*

The combo box became known due to the success of some graphical user interface, called M$ Windoze[2]. This combo box primarily represents a combination of an input field and a list (therefore the abbreviation "combo").

In its normal state, the text input field of the combo box is visible and the drop down list is hidden. Only a down pointing arrow button right beside the input field indicates, that this is not an ordinary input widget (see figure 1.1). The users then can drop down the list by clicking on the arrow button with the mouse, if they really need to browse in the list. After use, the list will be automatically hidden. Figure 1.2 shows a combo box with its list dropped down. If you don't like mice, then you may be happy to hear that the combo box widget provides full keyboard support. For example, one can drop down the list by pressing the keys META+↓.

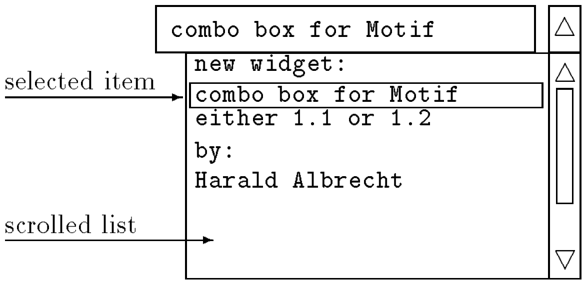

Figure 1.2: A combo box with the list dropped down

Beginning with the release 1.30 the combo box also supports a static list which is always displayed and placed just below the text input field. In this case no arrow button appears. Although such combo boxes can be configured to be either editable as well as noneditable, in general you should avoid using the noneditable variant. That's because such a widget is just an equivalent to an ordinary list widget. Nevertheless there are exceptions, where you can use a noneditable combo box with a static list: one example may be the new demo application `FontSel`.

The next figure 1.3 shows the layout of a combo box with a static list.

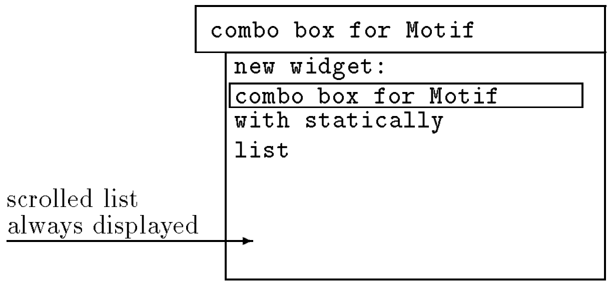

Figure 1.3: A combo box with a statically list

---

2 *All trademarks presented in this documentation are the property of their respective beholders irrespective to the name mangling (maybe I've programmed for too long in C++...)*

1.2 The combo box from the user's perspective of view

There are two variants of combo boxes available. One “flavour” contains an editable input field and the other one shows only selected items but doesn't accept any input. That is, one can't edit the text in the combo box. The combo box gives the user a visual clue of its current state. (Whether it accepts any attempts to edit the text in the input field). If the text can be changed, there is a gap between the input field and the arrow to the right side of the combo box of just half the size of the arrow button.

Not only you can use the mouse but also the keyboard because there's a whole set of operations available at your fingertips. As said before, the list can be dropped down by pressing the keys META and $\downarrow$ together. Please note that you'll hardly find a key labeled META on your computer's keyboard. The name META instead refers to an imaginary key which acts like the CTRL keys you're already familiar with. However, META adds another level of complexity. On many keyboards (PC, Silicon Graphics) this key is labeled ALT. Hewlett-Packard instead calls it Extended Char on their 700's and Sun labels it as $\diamond$. That's the flavour of true hardware independency X11 provides...

Pressing the keys META and $\downarrow$ together one more time will make the list disappear. Irrespective of the list being shown or hidden, one can browse through all the entries in the list with the cursor keys $\uparrow$, $\downarrow$, as well as Page $\uparrow$ and Page $\downarrow$. The old text in the input field is then replaced by the new item's text.

In a combo box without an editable input field, the user can also jump to the top or bottom of the list by pressing HOME or END (also known to programmers as <Key>osf-BeginLine and <Key>osfEndLine). Besides this, such a combo box variant provides a feature called "fast search". When the user presses a key, the list is searched for an item, which starts with the given letter. If the item being highlighted already starts with the same letter, the search continues with the next item in the list. Therefore one can easily browse through all items with the same first letter.

If the selection mode is set to `XmSINGLE_SELECTION` the user can deselect the current item by either clicking on it once again (in the list) or pressing the Delete key (<key>osfDelete).

The combo box is intended as a screen space saving variant of a normal list. In contrast to a list, the combo box occupies naturally only the same space as the simple input field. The user then drops down, on demand, the more convenient list. On the other hand a combo box with a static list occupies more screen space but presents to the user both the text input field and the list and provides the necessay “glue” to let these two widgets operate as one “atomic” object (well - physicians now would tell it a “quarkish” object, since atoms turned out to be divisible).

The following chapter provides an overview on using the combo box widget. The third chapter then explains in detail the internal working of this widget.

1.3 Features

These are the goods the combo box widget provides (improvements over the previous 1.22 release or new features are marked with a “$\star$”):

- Full documentation on paper as well on-line man pages.

- Concept of "mirror resources": programmers do not need to know the internal structure of the combo box widget. All resources are set and queried on the combo box widget – although they may belong to one of the combo box widget's children.

⋆ Improved handling of mirror resources. Some applications like GUI development tools or TclMotif[3] were unable to get notice of these special resources. This was caused due to their implementation through the core's setValues method and therefore they don't appeared in the resource list – there aren't any instance variables connected to them. Now the resource list (which can be queried by `XtGetResourceList()`) contains entries for all mirror resources although the initial value setting for each of this resources is always zero. As a natural consequence, the offset field of mirror resources does not contain any valid offset value. But this shouldn't matter.

⋆ Support of both selection modes `XmSINGLE_SELECT` and `XmBROWSE_SELECT`.

⋆ The list of entries can be either dynamic (drop down list) or static.

⋆ Built-in sorting of entries. The sorting order can be controlled with a callback **XmNsortingCallback**. This way, the programmer has full control over the sorting order, if he wishes to do so.

⋆ Many new callbacks available.

- Source code with all details available.

For a detailed history of changes and new features please see the file `ComboBox.doc`. It documents all changes since the first official release of my combo box widget in December 1993.

1.4 Demo Applications

The combo box package contains two examples which focuses on using the combo box widget. To build these examples first edit the `Makefile` accordingly to your plattform's settings (ANSI C compiler, compiler flags, and other things). Then do a simple `make`. This will compile first the combo box widget and then the two demo applications `ComboBoxDemo` and `FontSel`.

The first example – `ComboBoxDemo` – shall give you an impression how the different 'flavours' of combo box widgets looks like in reality and how to interact with them as a user (from the user's point of view as well as from the programmer's view).

The second example is a somewhat more useful tool and is called `FontSel`. It uses combo boxes to implement a font selector dialog. It allows users to easily choose any font X11 font based on the font's family name, typeface and point size. Depending on the currently selected font family (together with the foundry name and the encoding) the other combo boxes show only such typefaces and pointsizes which are applicable to that particular font. With scalable fonts the user may also enter any font size in points. The currently choosen font is shown in a preview window within the dialog window to give the user the impression how the font looks like.

[3] *This Tcl binding for Motif is a really nice development tool for rapid prototyping. Since it's an interpreter (BASIC, come back!) it allows one to change most of the widget's resources on-line.*

One use of this example may be in shell scripts. If the user presses the 'Okay' button the font selector prints the selected font's name (in XLFD format) to stdout and exits. If the user instead presses 'Cancel' the program simply exits without printing anything to stdout.

1.5 Where to go from here

As you may have already noticed from the number of sheets, the combo box widget is a somewhat "bigger" widget. To clear the path through the jungle of information, here are some tips for the impatient reader. More patient programmers (hey, are there any out there?!) should pour a cup of coffee – or better tea – and read this paper carefully from top to bottom.

- For creating a combo box widget see page 26.
- More information about all available convenience functions can be found on page 23.
- For available callbacks and how to use the sorting feature, see page 16.
- Finally, browse through the massive list of resources, beginning on page 9.

Chapter 2

Using the combo box

2.1 Configuration by using resources

One of the most important aspects of widget are its resources. They are the key to configuration and asking for up-to-date state information. Below is a description of all available resources the combo box provides:

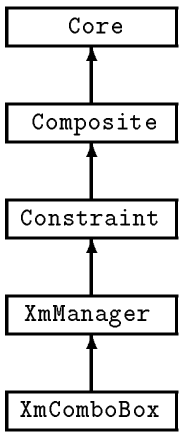

Figure 2.1: The family tree of a combo box

This documentation follows the guide lines introduced by the standard literature on the X window system (attention: singular!) and especially Motif. Below you'll find a table, which contains information about all the widget resources of the combo box: their names and classes as well as the default values. In addition, the table shows whether a resource can be set at the widget's creation time (**C**) or get and set during the program's run time (**G** and **S**). The description of each resource then follows.

A widget inherits all of its resources from its ancestors, starting with its parent and continuing until the root of the family (class) tree, the core widget class. Since the combo box contains other child widgets, one of its ancestors must be the widget class Composite. The children are, for example, an arrow button widget, a list widget and some more widgets.

To provide the accustomed keyboard interface with which the Motif user is already familiar, the combo box widget must be an ancestor of the Motif class XmManager. Only then the keys <Key>osfCancel (known as ESC) and <Key>osfActivate (labeled Return or Enter) will work as expected and can be used to confirm or cancel Motif dialogs. Other features of Constraint or XmManager, like the geometry constraints on its children, aren't needed anymore.

All those resources which the combo box inherits from its ancestors are not shown in this document. Instead, please refer to the Motif documentation in the case of XmManager or to the X toolkit documentation for the classes Core, Composite and Constraint. Nevertheless the combo box contains 50 new resources. Thus it follows the old golden Motif rule "the more one can configure the less one will actually make use of it".

XmComboBox Resource Set

Name Class	Default Type	Access
XmNactivateCallback XmCCallback	NULL XtCallbackList	C
XmNalignment XmCAlignment	XmALIGNMENT_CENTER unsigned char	CSG
XmNarrowSpacingOn XmCArrowSpacingOn	True Boolean	CSG
XmNautomaticSelection XmCAutomaticSelection	False Boolean	CSG
XmNblinkRate XmCBlinkRate	500 int	CSG
XmNcolumns XmCColumns	dynamic short	CSG
XmNcursorPosition XmCCursorPosition	0 XmTextPosition	CSG
XmNcursorPositionVisible XmCCursorPositionVisible	True Boolean	CSG
XmNdefaultActionCallback XmCCallback	NULL XtCallbackList	CSG
XmNdropDownCallback XmCDropDownCallback	NULL XtCallback	CSG
XmNdropDownCursor XmNDropDownCursor	center_ptr Cursor	CSG
XmNdropDownOffset XmCDropDownOffset	dynamic Position	CSG
XmNeditable XmCEditable	False Boolean	CG
XmNfontList XmCFontList	dynamic XmFontList	CSG
XmNitemCount XmCItemCount	0 int	CSG
XmNitems XmCItems	NULL XmStringTable	CSG
XmNlabelFontList XmCLabelFontList	dynamic XmFontList	CSG
XmNlabelInsensitivePixmap XmCLabelInsensitivePixmap	XmUNSPECIFIED_PIXMAP Pixmap	CSG
XmNlabelMarginBottom XmCLabelMarginBottom	0 Dimension	CSG
XmNlabelMarginHeight XmCLabelMarginHeight	2 Dimension	CSG
XmNlabelMarginLeft XmCLabelMarginLeft	0 Dimension	CSG
XmNlabelMarginRight XmCLabelMarginRight	0 Dimension	CSG

XmComboBox Resource Set

Name Class	Default Type	Access
XmNlabelMarginTop XmCLabelMarginTop	0 Dimension	CSG
XmNlabelMarginWidth XmCLabelMarginWidth	2 Dimension	CSG
XmNlabelPixmap XmCLabelPixmap	XmUNSPECIFIED_PIXMAP Pixmap	CSG
XmNlabelString XmCXmString	dynamic XmString	CSG
XmNlabelType XmCLabelType	XmSTRING unsigned char	CSG
XmNlistMarginHeight XmCListMarginHeight	0 Dimension	CSG
XmNlistMarginWidth XmCListMarginWidth	0 Dimension	CSG
XmNlistSizePolicy XmCListSizePolicy	XmVARIABLE unsigned char	CG
XmNlistSpacing XmCListSpacing	0 Dimension	CSG
XmNmarginHeight XmCMarginHeight	5 Dimension	CSG
XmNmarginWidth XmCMarginWidth	5 Dimension	CSG
XmNmaxLength XmCMaxLength	largest integer int	CSG
XmNmodifyVerifyCallback XmCCallback	NULL XtCallbackList	CSG
XmNmotionVerifyCallback XmCCallback	NULL XtCallbackList	CSG
XmNpersistentDropDown XmCPersistentDropDown	False Boolean	CSG
XmNscrollBarDisplayPolicy XmCScrollBarDisplayPolicy	XmAS_NEEDED unsigned char	CSG
XmNselectionCallback XmCCallback	NULL XtCallbackList	CSG
XmNselectionPolicy XmCSelectionPolicy	XmBROWSE_SELECT unsigned char	CG
XmNselectThreshold XmCSelectThreshold	5 int	CSG
XmNshowLabel XmCShowLabel	False Boolean	CG
XmNsorted XmCSorted	False Boolean	CSG
XmNsortingCallback XmCSortingCallback	NULL XtCallbackList	CSG

XmComboBox Resource Set		
Name	**Default**	**Access**
Class	**Type**	
XmNsquareArrow	False	CSG
XmCSquareArrow	Boolean	
XmNstaticList	False	CG
XmCStaticList	Boolean	
XmNstringDirection	dynamic	CSG
XmCStringDirection	XmStringDirection	
XmNtwmHandlingOn	False	CSG
XmCTwmHandlingOn	Boolean	
XmNtopItemPostion	1	G
XmCTopItemPostion	int	
XmNunselectionCallback	NULL	CSG
XmCCallback	XtCallbackList	
XmNvalue	""	CSG
XmCValue	String	
XmNvalueChangedCallback	NULL	CSG
XmCCallback	XtCallbackList	
XmNvisibleItemCount	8	CSG
XmCVisibleItemCount	Int	

Core Resource Set		
Name	**Default**	**Access**
Class	**Type**	
XmNborderWidth	0	CSG
XmCBorderWidth	Dimension	

XmNactivateCallback

Specifies a list of callbacks to be called when the user presses either the key Enter or Activate. The type of the structure passed to this callback is `XmAnyCallbackStruct` and the reason is `XmCR_ACTIVATE`.

XmNalignment

The alignment (left to right) for the label's text or pixmap. Possible values are `XmALIGNMENT_BEGINNING`, `XmALIGNMENT_CENTER`, and `XmALIGNMENT_END`.

XmNautomaticSelection

Indicates whether the **XmNselectionCallback** is invoked each time the user moves into a new item if **XmNautomaticSelection** is `True` and the selection mode is **Xm-BROWSE_SELECT**. If set to `False` the **XmNselectionCallback** is not invoked until the user releases the mouse button.

XmNarrowSpacingOn

Indicates whether a gap is displayed between the arrow button and the text input field if this resource is set to `True` and **XmNeditable** is also set to `True`. If this resource is set to `False` the combo box widget will never display a gab between the arrow button and the text area, irrespective of the setting of **XmNeditable**. Note that setting **XmNarrowSpacingOn** to `False` doesn not conform to common GUI styles. And it prevents the user from easily recognizing whether the combo box is editable or not.

XmNblinkRate

The time in milliseconds that the cursor spends being visible or invisible. A value of 0 prevents the cursor from blinking.

XmNborderWidth

That's no new resource but is already introduced in the core class. This class overrides the default value and sets it to zero. You can however override it again if you're dissatisfied with the new default setting.

XmNcolumns

Specifies the initial width of the input field measured in character spaces. The default value depends on the value of the width of the combo box.

XmNcursorPosition

Indicates the position in the text where the insert cursor is to be located. Its position is the offset of characters and it starts with offset 0 at the beginning of the text.

XmNcursorPositionVisible

Flags whether the insert cursor position is marked by a blinking cursor. If **XmNeditable** is set to `False` this resource is always set to `False`, too.

XmNdefaultActionCallback

Specifies a list of callbacks that is called every time an item is double clicked. The reason is `XmCR_DEFAULT_ACTION`. Please remember that you can't use this callback if **XmNstaticList** is `False` because the list is immediatly hidden after the user selected an item from the list. Thus the user can't double click on any item within the list.

XmNdropDownCallback

This is a list of callbacks to be called when the list is about to be shown or hidden. The reasons are `XmCR_SHOW_LIST` or `XmCR_HIDE_LIST`. The callbacks are called with the reason `XmCR_SHOW_LIST` just before the list is dropped down and they are called with the reason `XmCR_HIDE_LIST` after the list got hidden.

XmNdropDownOffset

Specifies the distance between the left border of the list and the left border of the input field. The default value is set to the width of the arrow button. Only positive values are allowed for **XmNdropDownOffset**, negative ones will be ignored. If you don't like the M$ Windoze touch[1], set this resource to zero.

XmNeditable

Specifies the type of combo box. If set to True, the combo box contains an editable input field. Therefore, the user can select entries out of the list and is also allowed to enter new text or modify it. In the other case, with XmNeditable set to False the user can solely browse in and select entries out of the list.

XmNfontList

Indicates the font list to be used for the input field and the list. If the value is NULL

[1] *Just as a question, why are you using Motif anyway...? If you have ever seen M$ WinWord 6.0 – the folks at the OSF were alright with the outer appearance (shadows etc. etc.) of Motif!*

at initialization, it is initialized by looking up the parent hierarchy of the widget for an ancestor that is a subclass of XmBulletinBoard, XmVendorShell or XmMenuShell. If such an ancestor is found, its font list will be taken by the combo box widget. The font list of the label however is not changed if this resource is set to a new value. Thus the programmer (or maybe the user) can use different fonts for the label's text and the items displayed in the list and the text input field.

XmNitemCount

Contains the total number of items in the list.

XmNitems

Points to an array of compound strings representing the items to be displayed in the list. By means of this a program can find out what items the list contains. It is also possible to set the contents of the list box in one step by setting this resource to a new array of compound strings. Please remember: always set **XmNitems** *and* **XmNitemCount** together in the same call to `XtSetValues` or `XtVaSetValues`. If you ignore this rule the combo box will ignore you and your attempt to set these resources.

XmNlabelInsensitivePixmap

Specifies a pixmap used as the button face if XmNlabelType is set to `XmPIXMAP` and the combo box widget is insensitive.

XmNlabelPixmap

Specifies a pixmap to be displayed to the left of the input field if **XmNlabelType** is `XmPIXMAP` and **XmNshowLabel** is `True` (of course).

XmNlabelString

Specifies the the compound string to be displayed if **XmNlabelType** is set to `XmSTRING` and **XmNshowLabel** is `True`.

XmNlabelType

The type of label (either display a string or a pixmap). Possible values are `XmPIXMAP` and `XmSTRING`.

XmNlabelMarginBottom

Same resource as XmLabel provides (just remove the string "label" within the resource's name). The amount of space between one side of the label and the nearest edge of the shadow. Setting this resource (as well as the other **XmNlabelMargin*** resources) is somewhat restricted because the label isn't allowed to grow higher than the text input field of the combo box. Thus, almost **XmNlabelMarginLeft** and **XmNlabelMarginRight** are the most important resources of the **XmNlabelMargin*** resource set.

XmNlabelMarginHeight

The spacing between one side of the label and the nearest edge of the list and the items in the list. See also the comments on **XmNlabelMarginBottom**.

XmNlabelMarginLeft

The amount of space between one side of the label and the nearest edge of the shadow. See also the comments on **XmNlabelMarginBottom**.

XmNlabelMarginRight

The amount of space between one side of the label and the nearest edge of the shadow. See also the comments on **XmNlabelMarginBottom**.

XmNlabelMarginTop

The amount of space between one side of the label and the nearest edge of the shadow. See also the comments on **XmNlabelMarginBottom**.

XmNlabelMarginWidth

The spacing between one side of the label and the nearest edge of the list and the items in the list. See the comments on **XmNlabelMarginBottom**.

XmNlistMarginHeight

The height of the margin between the border of the list and the items in the list.

XmNlistMarginWidth

The width of the margin between the border of the list and the items in the list.

XmNlistSizePolicy

Controls the reaction of the list when an item grows horizontally beyond the current width of the list and **XmNstaticList** is set to `True`. If set to `XmCONSTANT`, the list does not grow, and a horizontal scroll bar is shown. If this resource is set to `XmVARIABLE`, the list instead trys to grow such that it match the size of the longest item, thus no horizontal scroll bar will appear. When **XmNlistSizePolicy** is `XmRESIZE_IF_POSSIBLE` the list will again try to grow but in case the width of the longest item in the list is wider than the viewing area of the list a horizontal scroll bar appears.

XmNlistSpacing

The spacing between items in the list.

XmNmarginHeight

Specifies the distance between the top and the bottom of the widget window and the text displayed in the input field.

XmNmarginWidth

Like **XmNmarginHeight** this resource specifies the distance between the left and the right border of the widget window and the text.

XmNmaxLength

Specifies the maximum length of the text string that can be entered from the keyboard into the input field of the combo box. Strings that are put into the input field using **XmComboBoxSetString** ignore this resource.

XmNmodifyVerifyCallback

Contains a list of callbacks called before text is deleted from or inserted into the text area of the combo box widget. The callbacks are called only if **XmNeditable** is set to `True`. The reason is `XmCR_MODIFYING_TEXT_VALUE` and the type of the structure passed to this callback is `XmComboBoxTextVerifyCallbackStruct`.

XmNmotionVerifyCallback

Specifies the list of callbacks that is called before the insert cursor gets moved to another position. The reason is `XmCR_MOVING_INSERT_CURSOR` and the type of the structure whose address is passed to the callback is `XmComboBoxTextVerifyCallback-Struct`.

XmNpersistentDropDown

Welcome to the sad section in the life of X11 applications. This resource and the inherent problems are due to the focus policy of the window manager you're using. See the further details in the later sections.

XmNscrollBarDisplayPolicy

Controls the display of the vertical scroll bar in the list when **XmNstaticList** is set to `True`. When **XmNscrollBarDisplayPolicy** is set to `XmAS_NEEDED`, the vertical scroll bar is displayed only when the list contains more items than can be currently displayed. If set to `XmSTATIC` the vertical scroll bar is always displayed.

XmNselectionCallback

Specifies a list of callbacks that is called when an item is selected. The reason is `XmCR_SINGLE_SELECT`. The callbacks are not called when the user browses through the items in the list except when **XmNautomaticSelection** is set to `True`.

XmNselectionPolicy

This resource controls whether the user can deselect the currently selected item within a combo box. The default value is `XmBROWSE_SELECT`, so always one item gets selected if the combo box is editable. In the case of `XmSINGLE_SELECT` the user may deselect the current item either by clicking on it in the drop down list or pressing the key Delete (<Key>osfDelete). Other values for **XmNselectionPolicy** are currently not allowed. (Multiple selections will be probably availabe with the next release.)

XmNselectThreshold

Specifies the amount of pixels the mouse must move until the selection of text is recognized in click and drag mode (not: drag and drop mode).

XmNshowLabel

Specifies, whether a label containing text or a pixmap should be displayed on the left side of the combo box. This resource can only be specified during the creation of the combo box, afterwards setting it will have no effect.

XmNsorted

Specifies whether the items in the list are sorted or not[2]. In an unsorted list, the items are shown in the order in which they're put into the list. When adding an item to a sorted list, this item will be put at the right place (alphabetically sorted).

XmNsortingCallback

Contains a list of callbacks that is called whenever a new item is added to the combo box widget and **XmNsorted** is `True`. The reason is `XmCR_SORTING` and the type of the structure whose address is passed to the callback is `XmComboBoxSortingCallback-Struct`. For detailed information see the following sections.

XmNsquareArrow

Controls the width of the arrow button in respect to its height. If this resource is `True` the arrow button is always layed out square. Otherwise the arrow button's width will be approximatly $^4/_5$th of its height.

XmNstaticList

Controls whether the list is always statically displayed right below the text input area of the combo box if set to `True`. If you use a combo box with **XmNstaticList** set to `True` and **XmNeditable** set to `False` you should consider using a simple list widget instead.

[2] *I don't understand why the OSF don't provide a simple sort option. One often need to sort the items in the list alphabetically. Hmm – maybe they restrained it so people will buy Motif 2.0... and what's about owner drawn lists, and, and, and...*

XmNstringDirection

Specifies the initial direction to draw the string (from the left to the right or in the opposite direction). The whole story seems to be incomprehensible... therefore this resource is provided for completeness only[3].

XmNtwmHandlingOn

If you have trouble with the list then this resource may help you. Set it to True, if the list gets hidden at the moment the mouse cursor enters the list.

XmNtopItemPosition

This resource is intended as an informal hint. It specifies the index of the item which is displayed at the visible top of the list.

XmNunselectionCallback

Specifies a list of callbacks that is called whenever the currently selected item is deselected and no other item gets selected. This callback is activated only when **XmNselectionPolicy** is `XmSINGLE_SELECT`. The reason is `XmCR_UNSELECT`. The type of the structure whose address is passed to this callback is `XmAnyCallbackStruct`.

XmNvalue

Contains the text of the input field. This also concerns combo boxes which are not editable (**XmNeditable** set to `False`).

XmNvalueChangedCallback

Specifies a list of callbacks to be called after text is deleted from or inserted into the text area of the combo box widget. The reason is `XmCR_VALUE_CHANGED`. The type of the structure whose address is passed to this callback is `XmAnyCallbackStruct`.

XmNvisibleItemCount

Specifies the number of items that can fit in the visible space of the list. The combo box uses this value to determine the height of its list.

A shortcoming (you always know: it's not a bug – it's a feature!) of the first version of the combo box has been corrected (yet long ago...). You can specify all resources (irrespective of the fact that they may belong to a child widget) in an application's resource file with the common known syntax, for example: *myApp***mycombobox*`.columns: 20`. The combo box widget handles all the necessary mirroring of child resources transparently (that was really hard work! Just take a look at the source and you'll know why...).

2.2 Callback Information

A pointer to the following structure is passed to the **XmNdropDownCallback** callback:

```
typedef struct {
    int      reason;
    XEvent   *event;
} XmComboBoxDropDownCallbackStruct;
```

[3] *The author yet could not figure out whether the OSF supports vertical text from top to bottom or vice versa. But he tends to the assumption that the programmers at OSF lack the right* motif*ation.*

reason

Within this structure, "reason" indicates the reason why the callback was activated. This can be `XmCR_SHOW_LIST` as well as `XmCR_HIDE_LIST`. A callback is called with the reason `XmCR_SHOW_LIST` just before the list is dropped down, whereas the callback is called with `XmCR_HIDE_LIST` after the list is hidden.

event

Points to the XEvent, which triggered the callback. May be `NULL` if the callback is triggered by a call to `XmComboBoxShowList` or `XmComboBoxHideList`.

A pointer to the following structure is passed to the **XmNactivateCallback** callback:

```
typedef struct {
    int      reason;
    XEvent   *event;
} XmAnyCallbackStruct;
```

reason

Indicates why the callback was invoked. This is `XmCR_ACTIVATE` for `XmNactivate-Callback`.

event

Points to the XEvent that triggered the callback.

A pointer to the following structure is passed to the **XmNselectionCallback** and **XmN-defaultActionCallback** callbacks:

```
typedef struct {
    int      reason;
    XEvent   *event;
    XmString value;
    int      index;
} XmComboBoxSelectionCallbackStruct,
  XmComboBoxDefaultActionCallbackStruct;
```

reason

Within this structure, "reason" indicates the reason why the callback was activated. At this time, the reason can be either `XmCR_SINGLE_SELECT` or `XmCR_BROWSE_SELECT` depending on the setting of **XmNselectionPolicy**. It indicates that the user selected a (potentially new) item in the list. Whether this callback is activated every time the user moves into a new item of the list or only when the user releases the mouse button is controlled through the setting of **XmNautomaticSelection**.

event

Points to the XEvent, which triggered the callback.

value

Points to a `XmString`, which contains the text of the current selected item. This `XmString` must never be freed or changed by the called program code!

index

Index (starting from 1) of the currently selected item in the list.

A pointer to the following structure is passed to the **XmNunselectionCallback** callback:

```
typedef struct {
    int      reason;
    XEvent   *event;
} XmComboBoxUnselectionCallbackStruct;
```

reason

Indicates the reason why the callback was activated. This is `XmCR_UNSELECT`.

event

Points to the XEvent, which triggered the callback.

A pointer to the following structure is passed to the **XmNmodifyVerifyCallback** and **XmNmotionVerifyCallback** callbacks:

```
typedef struct {
    int         reason;
    XEvent      *event;
    Boolean     doit;
    Position    currInsert, newInsert;
    Position    startPos, endPos;
    XmTextBlock text;
} XmComboBoxTextVerifyCallbackStruct, *XmComboBoxTextVerifyPtr;
```

reason

Indicates the reason why the callback was activated. This can be either `XmCR_MODIFYING_TEXT_VALUE` or `XmCR_MOVING_INSERT_CURSOR`.

event

Points to the XEvent, which triggered the callback.

doit

Indicates whether the offending action should be performed. Setting this field of the callback structure to `False` denies the action.

currInsert

Specifies the current position of the insert cursor.

newInsert

Indicates the position at which the user attempts to position the cursor.

startPos

Indicates the starting position of the text to modify. If the reason is `XmCR_MOVING_INSERT_CURSOR` this value is the same as `currInsert`.

endPos

Specifies the ending position of the text to modify. If no text is replaced or deleted, then the value will be the same as `startPos`. If the reason is `XmCR_MOVING_INSERT_-`

CURSOR this value is the same as `currInsert`.

`text`

Holds the address of a structure of type `XmTextBlockRec` which holds the textual information to be inserted.

The following table describes the reasons why the individual verification callback structe fields are valid.

Reason	Valid Fields
`XmCR_MODIFYING_TEXT_VALUE`	*reason, event, doit, currInsert, newInsert startPos, endPos, text*
`XmCR_MOVING_INSERT_CURSOR`	*reason, event, doit, currInsert, newInsert*

A pointer to the following structure is passed to the **XmNsortingCallback** callback:

```
typedef struct {
    int       reason;
    XEvent    *event;
    int       operation;
    void      *item_internal;
    XmString  item;
    int       result;
} XmComboBoxSortingCallbackStruct, *XmComboBoxSortingPtr;
```

reason

Indicates the reason why the callback was activated. This is `XmCR_SORTING`.

event

Contains always `NULL` and is just there for compatibility reasons.

operation

Indicates the operation to carry out and can be either `XmOP_INIT`, `XmOP_DONE` or `XmOP_COMPARE`.

item_internal

Must be set during the `XmOP_INIT` operation to a suitable representation of the new item which is to be added to the list. In all subsequent calls to this callback with `XmOP_COMPARE` or `XmOP_DONE` this field will contain whatever you've put into it during `XmOP_INIT`. For detailed information refer to the following section.

item

During `XmOP_INIT` this field points to the item to be added to the list. When sorting the list (`XmOP_COMPARE`) the field contains the address of an item in the list which should be compared against the new item (as remembered in `item_internal`).

result

When comparing an item (out of the list) in `item` with the new item (as remembered in `item_internal`) the result has to be stored into this structure's field.

The following table describes which individual callback structure fields are valid during which operations.

Operation	Valid Fields
XmOP_INIT	*reason, event, operation, item, item_internal*
XmOP_DONE	*reason, event, operation, item_internal*
XmOP_COMPARE	*reason, event, operation, item, item_internal, result*

Enough about callbacks in general for now. The combo box widget now provides many callbacks to play around with. With some of them you should already be familiar with (the text callbacks for example) whereas some others are new or adapted in a special manner to the combo box. Since the **XmNsortingCallback** is totaly different from all other callbacks you may have ever encountered with Motif, I'll now explain it in more detail.

2.3 Controlling the sorting order

Sometimes the sorting order (when **XmNsorted** is `True`) of the combo box isn't the way you need it to be. In this case you can get full control over the sorting order by installing a **XmNsortingCallback**.

The callback has to carry out three different operations depending on the value of the variable `operation` within the structure passed to the callback.

`XmOP_INIT`
: The address of the new item which is about to be added to the list of items is given in the field `item` of the structure passed to the callback. The program then should convert this item from `XmSTRING` format to a more convenient format – a `char *` for example. The pointer to this data should be placed into the field `item_internal` for later reference during the operations `XmOP_DONE` and `XmOP_COMPARE`.

`XmOP_DONE`
: This tells the program that the sorting operation for a new item has been completed. The program then should clean up any data allocated during the `XmOP_INIT` operation – for example: freeing the data pointed to by `item_internal`.

`XmOP_COMPARE`
: During sorting, the combo box widget calls the callback with this operation code in order to find out the sorting order of the new item (field `item_internal`) in respect to another item already in the list (pointed to by `item`). The order then has to be placed in the field `result` of the structure whose address has been passed to the callback. A result of $<$ `0` means that the new item comes after the item whose address is pointed to by `item`. If `result` contains a positive value, the new item comes before the item addressed by `item`.

To enlighten these explanations somewhat you'll find below an code example. This example sorts the items such that shorter items (measured in terms of characters) always come first and only if two items are of the same length, they are lexically sorted. The code is an excerpt from `FontSel.c`.

```
void SortCB(Widget w, XtPointer pClientData, XtPointer pCallData)
{
    XmComboBoxSortingPtr pData = (XmComboBoxSortingPtr) pCallData;
    char                 *pOctets;
    int                  len1, len2;
```

```
    switch ( pData->operation ) {
        case XmOP_INIT:
            XmStringGetLtoR(pData->item, XmFONTLIST_DEFAULT_TAG,
                            (char **) &(pData->item_internal));
            break;
        case XmOP_DONE:
            XtFree((char *) pData->item_internal);
            break;
        case XmOP_COMPARE:
            XmStringGetLtoR(pData->item, XmFONTLIST_DEFAULT_TAG,
                            &pOctets);
            len1 = strlen(pOctets);
            len2 = strlen((char *) pData->item_internal);
            if ( len2 < len1 )
                pData->result = -1;
            else if ( len2 == len1 )
                pData->result = strcmp((char *) pData->item_internal,
                                       pOctets);
            else
                pData->result = 1;
            XtFree((char *) pOctets);
            break;
    }
} /* SortCB */

...

XtAddCallback(combobox, XmNsortingCallback,
              (XtCallbackProc) SortCB, NULL);
```

2.4 Translations

XmComboBox contains all the translations of XmManager. In addition to the already defined translations, the combo box adds (to be more precise: overrides) more translations to its input field, depending on `XmNeditable`.

<Key>osfDown:	**ComboBox-Manager(down)**
<Key>osfUp:	**ComboBox-Manager(up)**
Alt<Key>osfDown:	**ComboBox-Manager(show-hide-list)**
Meta<Key>osfDown:	**ComboBox-Manager(show-hide-list)**
Alt<Key>osfUp:	**ComboBox-Manager(hide-list)**
Meta<Key>osfUp:	**ComboBox-Manager(hide-list)**
<Key>osfPageUp:	**ComboBox-Manager(page-up)**
<Key>osfPageDown:	**ComboBox-Manager(page-down)**
<Key>osfCancel:	**ComboBox-Manager(cancel)**
<Key>Return:	**ComboBox-Manager(activate) activate()**

A combo box with **XmNeditable** set to `False` adds even more translations to the input field.

<Key>osfDelete:	**ComboBox-Manager(wipe-out)**
<Key>osfBeginLine:	**ComboBox-Manager(top)**
<Key>osfEndLine:	**ComboBox-Manager(bottom)**

The list widget gets no new translations – it already possesses more than enough translations. And *this* is the trick: only the input widget ever gets keyboard input, but never the list widget. The list solely processes mouse events and never keypresses, because it's simply "stuffed" into an vendorShell[4].

In case of a static list no shell widget is needed and the scrolled widget will be an immediate child of the combo box widget (for further details see the next chapter). If such a combo box is editable, then both the text widget and the list accept the keyboard focus. To provide a consistent user interface the list then gets some new translations:

<Key>osfPageUp:	**ComboBox-Manager(page-up)**
<Key>osfPageDown:	**ComboBox-Manager(page-down)**

Shall it be necessary to add your own translations to either the input or the list widget, you can access these widgets by means of the functions `XmComboBoxGetEditWidget()` as well as `XmComboBoxGetListWidget()` and `XmComboBoxGetLabelWidget()`. However, the use of this functions is discouraged since it requires the knowledge of the widget hierarchy within the combo box.

Nothing is perfect... Sun at least. You may encounter problems with Motif's drag'n'drop mechanism. If this is the case you should disable drag'n'drop on combo boxes. Therefore recompile the combo box widget's source with `-DNODRAGNDROP`. As a result the list widget gets a *no operation* translation which simply disables the initiation of drag and drop.

<Btn2Down>:	**ComboBox-Manager(no-operation)**

The version 1.21 had problems with drag'n'drop because the drop down list was put into a wmShellWidget. This leads to core dumps whenever a user tried to start a drag operation in the drop down list. To get rid of this problem one must use a vendorShellWidget instead. Sorry that I didn't realize this trick earlier.

2.5 Action Routines

XmComboBox provides the following action routines, as described below. (Well - not really much action here...).

ComboBox-Manager(*operation*)

The parameter *operation* specifies what operation the combo box should carry out. "up" moves the current selection in the list one item up, "down" moves it in the opposite direction. "show-hide-list" instructs the combo box to show or hide the list. Another operation called "activate" signals that the user pressed the Return (or Enter) key to take over the current selection. The remaining operations "top" and "bottom" move the current selection to the top or bottom of the list (you've already guessed that). Last but not least, "cancel" indicates to the combo box to hide the drop down list. If the user presses the ESC key, and the list is already hidden, the key is further

[4] *ups – to be honest: with some window managers the list do get the keyboard focus whenever the mouse cursor enters the shell window.*

propagated to the combo box's parent. "wipe-out" simply wipes out any selected item (or text) in the text input fiedl. The really last operation in this list is the self descriptive "no-operation".

2.6 Interface Functions

Having pointed out the configuration by resources, this section presents a survey on the interface functions. These functions provide an easy to use interface to the combo box for most applications for querying and setting data.

Most interface functions provide the same functionality as the ones used to control XmTextField and XmList widgets. But now the syllable "XmTextField" or "XmList" is replaced by "XmComboBox". You must never access resources of the combo box' child widgets directly but always through the combo box widget itself. Many functions do some internal processing, which must be carried out in order to get the combo box widget to work successfully, so never bypass them.

XmComboBox function	**refer to:**
`void XmComboBoxAddItem(Widget w, XmString item, int pos)` Add a new item to the list's contents. If `XmNsorted` is set to `True` the new item is inserted alphabetically. If the list is unsorted the same assumptions can be applied as with `XmListAddItems`.	`XmListAddItem`
`void XmComboBoxAddItems(Widget w, XmString *items, int item_count, int pos)`	`XmListAddItems`
`void XmComboBoxAddItemUnselected( Widget w, XmString item, int pos)`	`XmListAddItemUnselected`
`void XmComboBoxDeleteItem(Widget w, XmString item)`	`XmListDeleteItem`
`void XmComboBoxDeleteItems(Widget w, XmString *items, int item_count)`	`XmListDeleteItems`
`void XmComboBoxDeletePos(Widget w, int pos)`	`XmListDeletePos`
`void XmComboBoxDeleteItemsPos(Widget w, int item_count, int pos)`	`XmListDeleteItemsPos`
`void XmComboBoxDeleteAllItems(Widget w)`	`XmListDeleteAllItems`
`void XmComboBoxReplaceItems( Widget w, XmString *old_items, int item_count, XmString *new_items)`	`XmListReplaceItems`
`void XmComboBoxReplaceItemsPos( Widget w, XmString *new_items, int item_count, int position)`	`XmListReplaceItemsPos`
`Boolean XmComboBoxItemExists(Widget w, XmString item)`	`XmListItemExists`

XmComboBox function	**refer to:**
`int XmComboBoxItemPos(Widget w, XmString item)`	`XmListItemPos`
`Boolean XmComboBoxGetMatchPos(Widget w, XmString item, int **pos_list, int *pos_count)`	`XmListGetMatchPos`
`void XmComboBoxSelectPos(Widget w, int pos, Boolean notify)`	`XmListSelectPos`
`void XmComboBoxSelectItem(Widget w, XmString item, Boolean notify)`	`XmListSelectItem`
`int XmComboBoxGetSelectedPos(Widget w)` This function differs from the well known list widget function `XmListGetSelectedPos`. The list in a combo box can contain only one selected item. Therefore this function returns the index of the selected item in the list or zero, if none is currently selected.	—
`void XmComboBoxClearItemSelection(Widget w)` If the resource **XmNselectionPolicy** is `XmSINGLE_SELECT` and an item is currently selected, this functions deselects it and clears the text input area. Then it calls the **XmNdeselectionCallback**.	—
`void XmComboBoxClearSelection(Widget w, Time time)`	`XmTextFieldClearSelection`
`Boolean XmComboBoxCopy(Widget w, Time time)`	`XmTextFieldCopy`
`Boolean XmComboBoxCut(Widget w, Time time)`	`XmTextFieldCut`
`XmTextPosition XmComboBoxGetInsertionPosition(Widget w)`	`XmTextFieldGetInsertionPosition`
`XmTextPosition XmComboBoxGetLastPosition(Widget w)`	`XmTextFieldGetLastPosition`
`int XmComboBoxGetMaxLength(Widget w)`	`XmTextFieldGetMaxLength`
`char *XmComboBoxGetSelection(Widget w)`	`XmTextFieldGetSelection`
`Boolean XmComboBoxGetSelectionPosition(Widget w, XmTextPosition *left, XmTextPosition *right)`	`XmTextFieldGetSelectionPosition`
`char *XmComboBoxGetString(Widget w)`	`XmTextFieldGetString`
`void XmComboBoxInsert(Widget w, XmTextPosition position, char *value)`	`XmTextFieldInsert`
`Boolean XmComboBoxPaste(Widget w)`	`XmTextFieldPaste`
`Boolean XmComboBoxRemove(Widget w)`	`XmTextFieldRemove`
`void XmComboBoxReplace(Widget w, XmTextPosition from_pos, XmTextPosition to_pos, char *value)`	`XmTextFieldReplace`

XmComboBox function	**refer to:**
`void XmComboBoxSetAddMode(Widget w, Boolean state)`	`XmTextFieldSetAddMode`
`void XmComboBoxSetHighlight(Widget w, XmTextPosition left, XmTextPosition right, XmHighlightMode mode)`	`XmTextFieldSetHighlight`
`void XmComboBoxSetInsertionPosition( Widget w, XmTextPosition position)`	`XmTextFieldSetInsertion-Position`
`void XmComboBoxSetMaxLength(Widget w, int max_length)`	`XmTextFieldSetMaxLength`
`void XmComboBoxSetSelection(Widget w, XmTextPosition first, XmTextPosition last, Time time)`	`XmTextFieldSetSelection`
`void XmComboBoxSetString(Widget w, char *value)` At least Motif 1.1 has a very anoying bug. If you try to set the contents of an XmTextField to an empty string (""), you'll get garbage, when asking the XmTextField for its contents afterwards. This interface function provides a work around.	`XmTextFieldSetString`
`void XmComboBoxShowPosition(Widget w, XmTextPosition position)`	`XmTextFieldShowPosition`

2.7 The disliked keyboard focus

Now I'll annotate a somewhat tricky subject. If the resource **XmNpersistentDropDown** defaults to its initial value of `False`, one can observe the following effect (this may be annoying).

When the user drops down the list and then moves the mouse cursor out of the window where the combo box resides in, the list will be hidden. This is because the widget lost the keyboard focus. Some users are irritated because they didn't expect that behavior when they just moved the mouse to get it out of the way. If your window manager's focus policy is "pointer following" and if you move the pointer into another window, that window will automatically get the keyboard focus. On the other hand, if your window manager's focus policy is "click to type", you have to move the pointer into another window and click a mouse button to give that window the keyboard focus. In the "click to type" mode, you will hardly see the list's annoying behavior but this strongly depends on the window manager. Even the same window manager (in this case olwm) behaves in a different way if the user activates a window either by clicking somewhere into it or clicking on the title bar. With some window managers (most notably olwm and twm) the list is also suddenly hidden when the user moves the mouse into the list in order to select there an item.

To bypass this behavior a really dirty hack was used[5]. The user can activate that hack, if he sets **XmNpersistentDropDown** to `True` in the application's resources. Sorry, on

[5] *In fact the M$ Windoze programmers must also to hide additional functionality in the Windoze code in order to put their combo box to work...*

some window managers (twm for example) this will not fix the annoying behavior. Then try setting **XmNtwmHandlingOn** to `True` instead (new with the combo box version 1.2). With olwm however, for example, you may not need to set one of those resources.

Try it if you like – but don't complain at me if it doesn't work as expected. But at least it worked on the machine the author commonly uses: a cute little SGI Indigo running 4dwm as the window manager. Also tested on a Sun (...roughest development system I've ever seen. Even programming a ZX81 was much more fun!) running olwm or twm as the window manager. On my LinuxBox at home the source has been tested to my satisfaction with fvwm as the window manager.

2.8 Creating a combo box

Using the combo box in your own programs is easy. Just create the widget with one of the innumerable Xt toolkit functions (like `XtCreateManagedWidget` or `XtVaCreateManagedWidget`).

Specify as the widget class `xmComboBoxWidgetClass` – that's all. There are no so called convenience functions, since there isn't any need for them.

Somewhere in your program you'll need something like the following code fragment:

```
Widget ComboBox;

ComboBox = XtVaCreateManagedWidget(
    "combobox", xmComboBoxWidgetClass, Parent,
    XmNeditable, True or False,
    other Resources...
    NULL);
```

Setting the resource `XmNeditable` is not absolutely necessary, but it's better to do so.

After creation, items can be added to the combo box's list by means of interface functions like `XmComboBoxAddItem`. Which item the user selected can be determined by calling `XmComboBoxGetString`.

If you want to pre-select a certain item use the following call on a combo box which has the **XmNeditable** resource set to `False`.

```
XmComboBoxSelectPos(ComboBox, list item number, False);
```

or:

```
XmComboBoxSelectItem(ComboBox, itemString, False);
```

In every case the third paramater to both function calls above prevents the combo box calling the **XmNselectionCallback** callbacks. With **XmComboBoxSelectPos** the second parameter specifies the item number within the list of the item to copy to the text input field. In contrast **XmComboBoxSelectItem** expects a `XmString`. If this string is found in the list it'll then be transfered to the input area.

On combo boxes with **XmNeditable** set to `True` instead use another convenience function if you want also to set a text which isn't already in the list.

```
XmComboBoxSetString(ComboBox, "My Text");
```

Sometimes you may want to clear any selection in a combo box with a noneditable text field. First ensure that **XmNselectionPolicy** is set to `XmSINGLE_SELECT` because this is the only mode which allows the user to deselect an item. Then simply call:

```
XmComboBoxClearItemSelection(ComboBox);
```

This will clear any selection and the text input area and even call the **XmNunselection-Callback** if there was an previous selected item. You can use this convenience function also when **XmNeditable** is `True`, but its most common use should be in connection with noneditable combo boxes.

Because the documentation already got that huge please refer for additional information about the convenience functions to the sources as well as to the demo program `ComboBoxDemo.c`. Especially the new demo `FontSel.c` shows many common programming techniques using the combo box widget.

2.9 Where it is running successfully

Below a list of all configurations, where the combo box widget is known to work.

DEC	DECstation 5000/240: ULTRIX 4.3 with Motif 1.1
	VAX: VMS5.5-2 with Motif 1.1
IBM	RS6000:AIX 3.2 with Motif 1.1, 1.2
Hewlett-Packard	700's running HPUX 9.01 with Motif 1.2
PC	SCO Unix with Motif 1.1, OpenDesktop 3.0
	Linux (of course!) with Motif 1.2
	FreeBSD 386 with Motif 1.2
Silicon Graphics	IRIS Indigo: IRIX 4.0.5 with Motif 1.1
Sun	SparcStation: SunOS 4.1.3 with Motif 1.1 or Motif 1.2
	Solaris 2.4 with Motif 1.2

At the moment the list is not very long – but this should change as time goes by (no, I don't look in your eyes, X11...). Since the source code of the combo box widget is available under the terms of the "GNU Public License"[6] there's no hindrance porting it to other platforms.

Please bear in mind the slightly difference between the so-called "open software" the OSF provides and the "free" software: the latter one is really open because everybody can get the source as well as the "know how" for free or a little fee. With the OSF however you have to pay a somewhat larger fee to get some insight into Motif. Maybe I've misunderstood the word "open" – it seems not to mean something like frankness and sincerity...

[6] *See the file* `COPYING` *for more details. You should have received it together with the source of the combo box.*

Chapter 3

Behind the scenes

After all the theory I'll now provide some insight into the bits and bytes of a Motif widget: real programming with Motif. Sorry to say, but the Motif documentation in particular hides many details (how to inherit a XmManager widget, which translations and action routines a widget *really* provides, and, and, and...).

But now – below you'll find internals and hints, which give a combo box its life.

3.1 Recycling is it!

In times, where people have to live carefully in their environment, waste reduction and recycling have become famous. This source code considers this convention and reuses almost six Motif widgets (please take a look at figure 3.1). This prevented me from reinventing not one, but many wheels. The picture shows the hierarchy within a combo box widget when **XmNstaticList** is set to `False`.

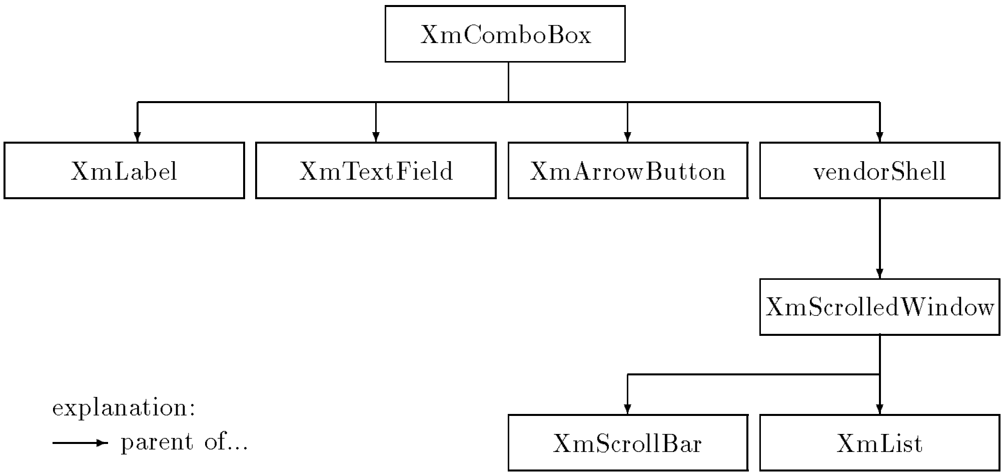

Figure 3.1: Widget hierachy within a XmComboBox

If **XmNstaticList** is set to `True`, then no shell widget is needed and the hierarchy looks like the one shown in figure 3.2.

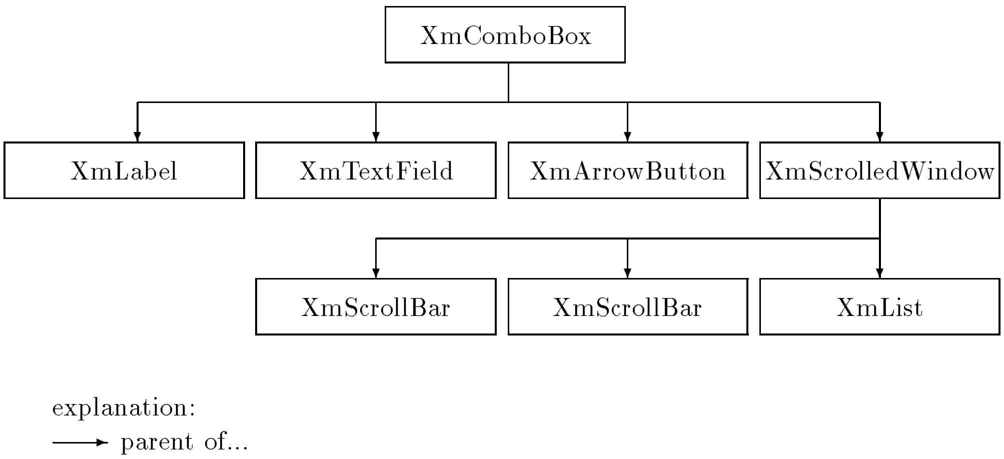

Figure 3.2: Widget hierachy within a XmComboBox with a static list

The combo box widget appears to the programmer, as well as to the user, to be a uniform widget. The resources are always queried and set on the combo box widget – never on one of its children.

3.2 The internal functions in detail

Below is a list of the more important functions, which make up the whole thing (or mess, depending on your point of view). The interface functions are not discussed here, because of their simplicity. The interested reader may refer to the source code instead.

```
void DefaultGeometry(XmComboBoxWidget w, Dimension *TotalWidth,
Dimension *TotalHeight, Dimension *EditCtrlWidth)
```

This function calculates the space needed by the whole combo box widget on the screen. The widget needs room for a text input field, an arrow button and probably a label widget. If the edit mode of the combo box is `True` this function also considers the extra space between the the text widget and the arrow button. `DefaultGeometry()` is a helper function that is called from the widget's method `QueryGeometry()` as well as during the initialization of a combo box widget. This way `Initialize()` can determine a suitable default size if another size wasn't explicitly requested.

```
void DoLayout(XmComboBoxWidget w)
```

It seems to be like an unwritten law that the function that lays out the position and size of a widget's children is always called `DoLayout()`. As one can see also the author conforms to this tradition.

This function – as well as other ones – are so-called "methods". This is due to the object orientated approach (well, an "approach" at least) the Xt toolkit imposes on widget writers. This especially method resizes the text field widget accordingly to the combo box widget's actual size and leaves just enough room for an arrow button. If the combo box shall display a label `DoLayout()` places it to the left side of the text widget.

If the list is currently visible this function also adjusts the drop down list's size and position on the screen. This is achieved by calling `DoDropDownLayout()`. This is applied only when the list isn't statically displayed. Otherwise the list is layed out such that it uses the space below the text input widget.

```
void DoDropDownLayout(XmComboBoxWidget w)
```

Remember that the drop down list is placed into a shell widget configured like an overrideShell. Thus the list isn't clipped to the space on the screen the parent widget of the combo box occupies. In consequence the position of the shell's upper left corner is measured in absolute rather than the ordinary relative coordinates. (To be more precise: the "absolute" coordinates are relative to the root window.)

Therefore `DoDropDownLayout()` calculates the absolute position of the drop down list's upper left corner and repositions the shell widget accordingly. Thus the list always lines up with the text field widget within the combo box. Please remember that this shell widget is not automaticly iconified together with the combo box widget. This has to be done "by hand" with an approbiate callback hooked to the nearest shell the combo box lives in.

```
XtGeometryResult QueryGeometry(XmComboBoxWidget w,
XtWidgetGeometry *Request, XtWidgetGeometry *Reply)
```

This is another method the Xt toolkit calls whenever the widget's parent wants to know the combo box's preferred geometry. But not without reason the Xt documentation states: "'*The parent may call this function when it wants to know...*"'. Pay great attention to "may" since the parent can decide to completely ignore its child's opinion concerning its geometry. In fact XmForm never calls `QueryGeometry()` on anyone of its children. It merely expects them to set the preferred geometry values during their initialization.

The conclusion: you always have to implement this method although it may never get called. Maybe someone should put a `printf()` statement in this function to see if it is ever called.

```
XtGeometryResult GeometryManager(XmComboBoxWidget w,
XtWidgetGeometry *Request, XtWidgetGeometry *Reply)
```

This is the counterpart of `QueryGeometry`. If one of the combo box's children wants to change its size and/or position it has to query first the combo box widget. Lesser educated children can also bypass the Xt etiquette by calling `XtConfigureWidget`. The combo box now accepts size requests from the label widget. If **XmNstaticList** is `True` it'll also requests from the list widget.

```
void Resize(XmComboBoxWidget w)
```

The `Resize` method is at least a really "handy" function. Just a call to `DoLayout` is sufficient to adjust the position as well as the size of all children. That's it.

```
void ShellCallback(Widget w, XtPointer cbw, XEvent *event,
Boolean *ContDispatch)
```

This callback is necessary due to the focus problem – one of the bizarre mysteries motif programming offers. As you might have already read, the list widget is put into a wmShell and configured in a way that the window manager ignores the shell. But this is not always true. Some window managers send a LeaveNotify message to the nearest shell the combo box resides in when the mouse cursor enters the list's shell. This then results in a FocusOut being misinterpreted by the combo box widget and the list is almost immediately hidden.

As another problem, some users may be irritated that the list is also hidden when they move the mouse out of the dialog or application window where the combo box lives. To circumvent this, there's a resource called `XmNpersistentDropDown` which enables a special handling of FocusOut events. If set, every FocusOut sent to the combo box will be ignored if preceeded by a FocusOut sent to the shell.

But this callback is also necessary in detection of the iconification of that window where the combo box resides. To prevent a possibly shown list from floating around somewhere on the screen `ShellCallback` hides the list whenever the shell is iconified. The callback is installed by `Initialize()` during the set-up of a combo box. But there's also one more reason for this callback. The dialog or application window may be moved while the drop down list is shown. Whenever this happens the list has to be moved too, in order to align it with the text field widget again. This is achieved by a simple call to `DoDropDownLayout()`.

```
Boolean SetValues(XmComboBoxWidget current, XmComboBoxWidget req,
XmComboBoxWidget new, ArgList args, Cardinal *NumArgs)
```

The interface between the program and the resources of every instance of a combo box is a somewhat difficult task. That's because combo boxes provide many resources belonging to their children. Despite this, all these resources appear to the programmer, as well as to the user, as ordinary resources of the combo box widget itself. To provide this service, there's a list of every mirrored resource in `ComboBox.c`. The list is checked by `SetValues` to find out to which child the queried resource belongs to. Some of the resources are read only, so please refer to the previous chapter. Unfortunately this mechanism can't prevent a ill-behaving programmer from directly accessing the children's resources. Thus be honest and set or query resources only on behalf of the combo box widget.

```
void GetValuesAlmost(XmComboBoxWidget w, ArgList args, Cardinal *NumArgs)
```

Once more this function constitutes the interface between the application and the widget's resources. This time the function handles the querying of especially the mirrored resources.

```
void EditFocusCallback(Widget w, XtPointer pClientData,
XmAnyCallbackStruct *info)
```

Whenever the combo box loses the keyboard focus, it hides the list. But in some cases, FocusOut events are sent which should better be ignored. The difficult task is to distinguish the different sorts of FocusOut events. The reasons for a FocusOut are numerous: the user moved the keyboard focus to another widget within the same window. Or the user moved the mouse cursor to another window when the window manager is in the "pointer following" mode. Some window managers generate such events the same time the mouse cursor enters the drop down list of the combo box.

As a result of this, there's a complex interaction between this function and some other functions, most notably `ShellCallback`. Since this isn't easy to explain, you'd be better off reading the source if you're interested in this area.

```
void CBoxManager(Widget w, XEvent *Event, String *params, Cardinal
*num_params)
```

This is the one and only action procedure provided and used by the combo box widget. Please remember that this action procedure is registered application wide since not only the combo box widget itself but also some of its children trigger it. `CBoxManager` mainly handles keyboard input like the cursor keys, ESC or Enter.

```
void ListSelectionCallback(Widget w, XtPointer pClientData,
XmAnyCallbackStruct *info)
```

Whenever the user selects an item in the drop down list this function transfers this item to the text field widget. In addition it also calls the registered callbacks. To ease the coding, `ListSelectionCallback()` relies heavily on `CBoxManager`. That function serves most of the duty to be done on interaction with the user.

```
void EditVerifyCallback(Widget w, XtPointer pClientData,
XmTextVerifyCallbackStruct *info)
```

This function plays an important role when `XmNeditable` is set to `False`. In this case it implements the fast search within the list. Every time the user presses a key (letter or number as well as some other symbols), `EditVerifyCallback()` is called by the text field widget. The widget simply wants to know whether it can accept the user's input. This function always denies this question but uses the user's input to search an appropiate item in the list. If such an item is found it is transfered to the text widget and in addition the item in the list becomes highlighted.

Why this way and why not adding the simple translation "<Key>" to the text field widget? Well – in this case *all* keyboard input would be redirected to the same action procedure. But this is not the intention since this function is interested only such keys (like "a" or "Z") that have no special meaning (in contrast to "osfHome" or "osfActivate"). This problem is due to the translation manager. When adding such a "wildcard" translation, it will be prefered by the the translation manager[1].

```
void EditChangedCallback(Widget w, XtPointer pClientDate,
XmAnyCallbackStruct *info)
```

Whereas the previous function is only needed on combo boxes with an not editable text input field this function serves its duty when `XmNeditable` is set to `True`. After the user has changed the text in the input area this callback gets called by the Xm-

[1] *This was one of that "nice" problems you'll always stomble over when you least expect it. But I solved it with a quite nice hack...*

TextField widget within the combo box. Then this function checks whether the text exists in the list and highlights it.

```
void Initialize(Widget request, XmComboBoxWidget new, ArgList wargs,
Cardinal *ArgCount)
```

In many other widgets, this method is comparable easy and handy but this time `Initialize()` has do do a lot of work during initialization of a combo box widget. As one main task `Initialize()` creates the necessary children: first, a XmTextField constituting the text input field of the combo box. Furthermore, the arrow button (a XmArrowButton widget) for showing and hiding the drop down list. The list is built up from a XmScrolledWindow enclosed by a wmShell. This shell is configured like an overrideShell, thus it will be ignored by the window manager. This is accomplished by setting `XmNoverrideRedirect` to `True`. But why use a wmShell instead of an overrideShell? That's due to the problems with Motif 1.2. On some platforms, one cannot create a simple shell or overrideShell as a child of a motif widget. All you get is a lovely core dump with a almost meaningless stack trace.

Another task is to layout the children appropriately according to the widget's actual geometry. This step is necessary because XmForm widgets never even dreamed about querying one of its childen for its preferred size. By calling `DoLayout()` the text input widget, as well as the arrow button and probably the label, are positioned to their adequate places.