Text

Motif has widgets for displaying two kinds of text: static text, as in labels and messages, and editable text. Static text usually appears in Label widgets or Label subclasses, including buttons, and in Lists. The application or user can specify initial text for Labels or Lists by using resource or UIL files, but the user cannot edit the text. The application can replace the text during the program by setting the appropriate resources. In Labels and Label subclasses and in Lists, Motif represents text as compound strings. These are opaque data types that contain the text itself and tags that the toolkit matches with tags in render tables in order to select the appropriate fonts or font sets to display the text.

For editing text, Motif provides Text and TextField widgets. The displayed text in these widgets may or may not be editable, depending on the value of the **XmNeditable** resource. When the Text is editable and the user enters a text character, that character is inserted into the text. Other translations and actions allow the user to navigate or to select, cut, copy, paste, or scroll the text. In Text and TextField widgets, Motif represents text as strings of either multibyte (*char*) or wide (**wchar_t**) characters. The Text widget uses a single font or font set from a render table to display the text.

This chapter discusses the Text and TextField widgets. Labels and their subclasses are discussed in Chapter 5; and compound strings, render tables, and localization of text are discussed in Chapter 11. It is also possible for an application to construct its own text−editing widget by using a DrawingArea. This is discussed in Chapter 15.

Text and TextField

The Text widget uses the type **String**. A TextField is essentially the same as a Text widget in single−line mode, except that its performance is optimized for single−line text operations. Although TextField has a complete set of convenience routines of its own, the widget argument to the Text convenience routines can be either a Text or a TextField widget.

The text in a Text widget can be multiline or constrained to be a single line, depending on the value of the **XmNeditMode** resource. In multiline Text, pressing `osfUp` moves the insertion cursor, the point at which new text is inserted, physically upward, and pressing `osfDown` moves the insertion cursor physically downward. Other actions move the insertion cursor forward and backward by paragraphs. Pressing *Space*, *Tab*, or *Enter* causes the corresponding character to be inserted into the text. For this reason, some action key bindings are different in Text from those in other widgets, as shown in Table 10−1.

Table 1 Text Action Key Bindings

Action	*Actual Key Events*
Activate	*Ctrl*<*Key*>*Return*
	`KeyosfActivate`
Extend	*Ctrl Shift*<*Key*>*space*
	Shift`KeyosfSelect`
NextField	*Ctrl*<*Key*>*Tab*
Select	*Ctrl*<*Key*>*space*
	`KeyosfSelect`

In a single−line widget, pressing *Space* still inserts a space into the text. However, `osfUp` and `osfDown` now move keyboard focus to the previous or next traversable widget, and *Tab* traverses to the next tab group. *Enter* invokes the **XmNactivateCallback** callbacks. The actions for moving by paragraphs have no effect. In other words, a single−line Text widget acts more as a simple control than a field control.

> **Note:** Asian languages are supported by vertical writing capabilities; for detailed information, refer to the reference pages for the following interfaces: **XmText**, **XmTextField**, **XmTextGetBaseline**, **XmTextGetCenterline**, **XmTextPosToXY**, **XmTextXYToPos**, and **XmTextScroll**.

Data Transfer in Text

Text and TextField allow the user to cut, copy, and paste text by using the clipboard, primary transfer, or secondary transfer. The user can also drag and drop text within a widget, between widgets, or from a Label or List widget to a Text or TextField widget. In all cases, the user first selects text in some widget and then inserts the selected text into a Text or TextField widget.

This section explains how selection works in Text and TextField. Understanding selection requires understanding of several concepts: *primary selection*, *secondary selection*, *clipboard selection*, the *destination* widget, the *insertion cursor*, the selection *anchor*, and *pending delete*.

Selections are the primary means of exchanging data between X clients. A selection is a piece of data. Each display may have several kinds of selections, but only one selection of each kind can exist at any time on the display. A client owns each selection, and the selection is attached to a window. Clients can acquire or give up ownership of a selection and can request that the owner convert the selection into some data type and place the results on a property of a particular window. This mechanism makes it possible to select and then cut, copy, or paste data from one client to another. Selections are discussed in detail in the X Window System *Inter−Client Communication Conventions Manual* (ICCCM).

Text and TextField support transfers by using the three kinds of selection common to all X clients:

Primary

> The primary selection is the principal selection on the display. Unless they are qualified, the terms *selecting text* and *the selection* refer to the primary selection.

Secondary

> The secondary selection is used to transfer data without disturbing the primary selection. Text and TextField use the secondary selection for *quick* transfer, in which the user selects and then moves or copies text by using a single series of mouse gestures.

Clipboard

> The clipboard selection usually holds data cut or copied from one client and available to be pasted into another. Text and TextField provide actions for cutting and copying text to the clipboard and for pasting text from the clipboard.

The *destination* is the widget that, at any particular time, would receive the selection if the user were to invoke a move, copy, or paste operation. A Text or TextField widget must be both sensitive and editable to become the destination. When the **XmNkeyboardFocusPolicy** of the shell is **XmEXPLICIT**, an editable widget becomes the destination when it receives keyboard focus. When the **XmNkeyboardFocusPolicy** is **XmPOINTER**, an editable widget becomes the destination when it receives any mouse button or keyboard input. If the destination widget becomes insensitive or uneditable, there is no destination widget.

The *insertion cursor* is an I−beam cursor that shows where text, including a selection, would be inserted in a Text or TextField widget. The insertion cursor appears as a solid I−beam when the widget is in *normal mode* (explained below) and when it is either the widget with keyboard focus or the destination widget. Otherwise, the insertion cursor appears as a stippled I−beam.

The *anchor* is a position in the text of a widget that marks one boundary of a selection or a potential selection. For example, the user can select a range of text by pressing, dragging, and releasing `Btn1`. The anchor is set at the point of the button press, and the selection extends to the point of the button release. When the user takes an action to extend an existing selection, Motif first adjusts the anchor by using a balance−beam method: it moves the anchor to the end of the existing selection that is farthest from the point of the button or key press that initiates the extend action.

Text and TextField have an **XmNpendingDelete** resource. This resource controls whether or not text is deleted in certain situations. Consider the following conditions:

- **XmNpendingDelete** is True (as it is by default).
- A selection exists and the insertion cursor is not disjoint from it.
- The widget is in Add Mode.

Given these conditions, an operation that inserts text (including a transfer of the secondary or clipboard selection) will delete the primary selection before inserting the text. Similarly, an operation that deletes text will delete the primary selection instead of the text that would otherwise be removed. When **XmNpendingDelete** is False, the insertion and deletion operations do not delete the selection.

Users typically use the mouse to make selections and to initiate and/or complete data transfers. (See the *Motif User's Guide* for complete details.) The **XmDisplay** resource **XmNenableBtn1Transfer** controls the purpose of `Btn1` and `Btn2` in a Text, TextField, Container, or List widget.

For example, when **XmNenableBtn1Transfer** is set to **XmOFF**, users press `Btn1` to select text and `Btn2` to trigger certain kinds of data transfer. (See **XmDisplay**(3) in the *Motif Programmer's Reference* for details.)

Users can optionally use the keyboard instead of the mouse to perform selection and transfer operations. All operations available from the mouse, except secondary selection, are available from the keyboard. Text has two keyboard selection modes, Normal Mode and Add Mode.

In Normal Mode, if text is selected, a navigation operation deselects the selected text and moves the anchor to the current position of the insertion cursor before navigating. In Add Mode, navigation operations have no effect other than navigation.

In Normal mode, when the widget contains the primary selection and the insertion cursor is disjoint from it, any operation that inserts or pastes text into the widget (except a transfer of the primary selection from the same widget) first deselects the primary selection. In Add Mode, such an operation does not deselect the primary selection.

Mouse Selection

The user makes a primary selection with `Btn1`. Pressing `Btn1` deselects any existing selection and moves the insertion cursor and the anchor to the position in the text where the button is pressed. Dragging `Btn1` selects all text between the anchor and the pointer position, deselecting any text outside that range. Releasing `Btn1` moves the insertion cursor to the position where the button is released. Clicking `Btn1` deselects any existing selection and moves the insertion cursor and the anchor to the position where `Btn1` is released.

`ShiftBtn1` extends a selection using the balance−beam method. When the user presses `ShiftBtn1`, the selection becomes anchored at the edge of the selection farthest from the pointer position. When the user releases `ShiftBtn1`, the selection extends from the anchor to the position where `ShiftBtn1` is released, and any text outside that range is deselected. The insertion cursor moves to the position where `ShiftBtn1` is released.

Clicking `CtrlBtn1` moves the insertion cursor to the position where `CtrlBtn1` is released without affecting the selection.

Clicking `Btn2` moves the insertion cursor to the position where `Btn2` is released. Then, unless the insertion cursor is in the midst of the selection, it copies the primary selection to the insertion cursor and moves the insertion cursor to the end of the copied text. The original selection remains selected. Clicking `ShiftBtn2` has the same effect except that it moves the primary selection to the insertion cursor, deleting the original selection if possible.

Dragging `AltBtn2` outside of the primary selection starts a secondary selection consisting of all text between the position of the pointer and the position where `AltBtn2` was pressed. Releasing `AltBtn2` copies the secondary selection to the insertion cursor in the destination widget. Before copying the secondary selection, if the destination contains the primary selection and the insertion cursor is not disjoint from it, releasing `AltBtn2` deletes the primary selection. Dragging `AltShiftBtn2` also makes a secondary selection, and releasing `AltShiftBtn2` moves the secondary selection to the destination widget.

Dragging `Btn2` with the insertion cursor positioned within a primary selection initiates a drag operation. The user may press a modifier key to indicate whether the drag is a copy, move, or link operation. Releasing `Btn2` either in the same Text widget or a different widget moves the insertion cursor to the position where `Btn2` is released, drops the selected text at that point, and moves the insertion cursor to the end of the dropped text.

Pressing `osfCancel` during the operation aborts the operation and no data exchange occurs. If the user presses `osfHelp` over a drop site, the user has the option to continue or to cancel the drop operation in response to the help information that the application provides.

Keyboard Selection

Selection operations available with the mouse, except secondary selection, are also available from the keyboard. Text has two keyboard selection modes, Normal Mode and Add Mode. In Normal Mode, if text is selected, a navigation operation deselects the selected text and moves the anchor to the current position of the insertion cursor before navigating. In Add Mode, navigation operations have no effect other than navigation. In both modes, pressing `osfSelect` has the same effect as pressing `Btn1` at that position.

In Normal mode, when the widget contains the primary selection and the insertion cursor is disjoint from it, any operation that inserts or pastes text into the widget (except a transfer of the primary selection from the same widget) first deselects the primary selection. In Add Mode, such an operation does not deselect the primary selection.

Pressing `osfExtend` extends the current selection to the insertion cursor using the balance−beam method. The current selection becomes anchored at the edge of the selection farthest from the insertion cursor. The selection then extends from the anchor to the insertion cursor, and any text outside that range is deselected.

Shifted navigation operations also extend a selection. In Normal Mode, if no text is selected, a shifted navigation operation moves the anchor to the insertion cursor, navigates, selects the navigated text, and deselects any text outside that range. In the remaining cases−−Normal Mode and Add Mode with any selection−−a shifted navigation operation extends the selection using the balance−beam method. Before navigation, the current selection becomes anchored at the edge of the selection farthest from the insertion cursor. After navigation, the selection extends from the anchor to the insertion cursor, and any text outside that range is deselected.

`osfPrimaryCopy` copies the primary selection to the insertion cursor. `osfPrimaryCut` cuts the primary selection to the insertion cursor.

`osfCopy` copies the current selection in the Text widget to the clipboard; `osfCut` cuts the selection; and `osfPaste` inserts the contents of the clipboard at the insertion cursor.

Text Editing and Callbacks

Text has a number of callback lists for communication with the application. Text invokes callbacks whenever the widget gains or loses focus, when it gains or loses the primary selection, before the insertion cursor is moved or text is modified, and when the text string changes or the **activate()** action is invoked.

Text passes these callbacks a pointer to either an **XmAnyCallbackStruct** or an **XmTextVerifyCallbackStruct** (or **XmTextVerifyCallbackStructWcs**) structure. The two verification structures contain the current and new positions of the insertion cursor, the starting and ending positions of the text to be modified, a pointer to an **XmTextBlockRec** (or **XmTextBlockRecWcs**) structure with information about the text to be modified, and a Boolean in/out *doit* member that the callback procedure can set to tell the widget whether or not to go ahead with the modification.

Following is a summary of the callbacks:

XmNmotionVerifyCallback

> Text invokes this list, passing a pointer to an **XmTextVerifyCallbackStruct** as the widget data, before moving the insertion cursor. The application can prevent the action by setting the *doit* member of the callback struct to False.

XmNmodifyVerifyCallback or **XmNmodifyVerifyCallbackWcs**

> Text invokes this list, passing a pointer to an **XmTextVerifyCallbackStruct** structure or an **XmTextVerifyCallbackStructWcs** structure as the widget data, before deleting or inserting any text. The application can prevent the action by setting the *doit* member of the callback struct to False.

XmNvalueChangedCallback

> Text invokes this list, passing a pointer to an **XmAnyCallbackStruct** as the widget data, after text is inserted or deleted.

XmNfocusCallback

> Text invokes this list, passing a pointer to an **XmAnyCallbackStruct** as the widget data, when the widget gains input focus.

XmNlosingFocusCallback

> Text invokes this list, passing a pointer to an **XmTextVerifyCallbackStruct** as the widget data, before the widget loses input focus. The application can prevent the action by setting the *doit* member of the callback struct to False.

XmNgainPrimaryCallback

> Text invokes this list, passing a pointer to an **XmAnyCallbackStruct** as the widget data, when the widget gains ownership of the primary selection.

XmNlosePrimaryCallback

> Text invokes this list, passing a pointer to an **XmAnyCallbackStruct** as the widget data, when the widget loses ownership of the primary selection.

XmNactivateCallback

> Text invokes this list, passing a pointer to an **XmAnyCallbackStruct** as the widget data, when the **activate()** action is invoked. By default no translations are bound to this action, but in a single–line Text widget or a TextField widget, pressing `osfActivate` invokes the**XmNactivateCallback** callbacks.

These callbacks provide a great deal of flexibility for an application to alter the behavior of the Text widget. For example, an application can prevent text from being inserted, as when the user types a password, by using the **XmNmodifyVerifyCallback** or **XmNmodifyVerifyCallbackWcs** callbacks. The application can prevent any text from appearing by setting the *doit* member of the **XmTextVerifyCallbackStruct** (or **XmTextVerifyCallbackStructWcs**) to False. The application can also alter the text that will appear by creating a new text string and setting the *ptr* member of the **XmTextBlockRec** structure (or the *wcsptr* member of the **XmTextBlockRecWcs** structure) to the new string.

Following is an example of an **XmNmodifyVerifyCallback** that substitutes a string of characters for any text a user enters. Because the **XmNmodifyVerifyCallback** procedures are most commonly invoked after the user enters a character, this routine usually substitutes the replacement string for each character the user types. This example could be used with a single−line Text widget as part of a simple password −entry program. In this case, the **XmNmodifyVerifyCallback** procedure would need additional code to save the characters the user types, and the program would need an **XmNactivateCallback** procedure to check whether the saved characters match the password.

```
/* XmNmodifyVerifyCallback procedure that replaces text the user enters
 * with a replacement string passed in as application data. */
void ModifyVerifyCB(Widget w, XtPointer app_data,
                    XtPointer widget_data)
{
  char *replace_string = (char *) app_data;
  XmTextVerifyCallbackStruct *widget_info =
    (XmTextVerifyCallbackStruct *) widget_data;
  if (widget_info->text->length > 0) {
    widget_info->text->length = strlen(replace_string);
    widget_info->text->ptr = replace_string;
/* The widget will automatically free this ptr.  Therefore,
   your app should make a copy of it with strcpy. */
  }
}
```

Text and TextField differ from most other Motif widgets in that calling some convenience routines and setting some resources causes the widget to invoke callback procedures. In general

- Setting resources or calling convenience routines that change the contents of the text invokes the **XmNmodifyVerifyCallback** and **XmNmodifyVerifyCallbackWcs** callbacks. If these procedures allow the text to be modified, the **XmNvalueChangedCallback** callbacks are invoked.
- Setting resources or calling convenience routines that change the position of the insertion cursor invokes the **XmNmotionVerifyCallback** callbacks.
- Setting resources or calling convenience routines that cause the widget to gain the primary selection invokes the **XmNgainPrimaryCallback** callbacks.
- Setting resources or calling convenience routines that cause the widget to lose the primary selection invokes the **XmNlosePrimaryCallback** callbacks.

If the application needs to distinguish between callbacks invoked as a result of user action and callbacks invoked as a result of application action (such as setting a resource or calling a convenience routine), it needs to set a flag before taking the application action and clear the flag afterward.

Text Resources and Geometry

In addition to the resources discussed in the previous section, Text has many others, including the following:

- The text itself, **XmNvalue** or **XmNvalueWcs**. For Text and TextField, the text is represented to the application as an array of either *char* elements (for **XmNvalue**) or **wchar_t** elements (for **XmNvalueWcs**). The application can set or get either resource.
- The render table (**XmNrenderTable**) that the widget uses to select a font or font set and other attributes to display the text. Text and TextField use only the font–related rendition resources of the render table.
- Resources representing the insertion cursor position and blink rate, the position of text at the top of the window, and whether the insertion cursor is always visible. A text position (of type **XmTextPosition**) is an integer representing the number of characters from the beginning of the buffer.
- A resource (**XmNmaxLength**) representing the maximum length of the text string that the user can enter.
- A resource (**XmNwordWrap**) that specifies whether lines are broken at word boundaries when the text would be wider than the widget. Breaking a line at a word boundary does not insert a newline into the text.

In addition, Text and TextField have several resources that determine the geometry of the widget:

- Two resources, **XmNmarginHeight** and **XmNmarginWidth**, that determine the margins between the text and the shadow, if present. Text and TextField also use the Primitive resources that determine shadow and highlight appearance.
- Resources that specify the number of rows of text (**XmNrows**) and the number of horizontal character positions (**XmNcolumns**). Single–line Text and TextField always have one row.
- Resources that determine whether or not the widget grows vertically (**XmNresizeHeight**) or horizontally (**XmNresizeWidth**) to accommodate all its text. **XmNresizeHeight** does not apply to single–line Text or TextField.
- Resources that apply only when the widget is inside a ScrolledWindow whose **XmNvisualPolicy** is **XmVARIABLE**. **XmNscrollHorizontal** determines whether or not the widget should have a horizontal ScrollBar and should scroll horizontally instead of growing when the text expands beyond the width allocated for it. **XmNscrollVertical** determines whether or not the widget should have a vertical ScrollBar and should scroll vertically instead of growing when the text expands beyond the height allocated for it. **XmNscrollLeftSide** and **XmNscrollTopSide** determine which side of the widget receives the corresponding ScrollBar. These resources do not apply to TextField, and **XmNscrollVertical** and **XmNscrollLeftSide** do not apply to single–line Text.

XmNresizeWidth is initialized to False when **XmNscrollHorizontal** is True or **XmNwordWrap** is True. **XmNresizeHeight** is initialized to False when **XmNscrollVertical** is True. Word wrap will not take effect if **XmNscrollHorizontal** is True.

If the user or application initializes or sets a specific height (**XmNheight**) or width (**XmNwidth**), that value is used as the corresponding dimension of the widget. In addition, if a height is specified, **XmNrows** is recalculated based on that height and, if a width is specified, **XmNcolumns** is recalculated based on that width.

If the user or application initializes or sets **XmNrows** but not **XmNheight**, the geometry calculation depends on the value of **XmNresizeHeight**. If **XmNresizeHeight** is False, as it is by default, the height of the widget is simply the height needed to display **XmNrows** of text. The same relations hold for **XmNcolumns**, **XmNwidth**, and **XmNresizeWidth**.

If the user or application does not initialize either **XmNrows** or **XmNheight**, the geometry calculation depends on the value of **XmNresizeHeight**. If **XmNresizeHeight** is True, the height of the widget is the height needed to display all the text. If **XmNresizeHeight** is False, the height of the widget is the height needed to display the default for **XmNrows**, which is one row of text. The same relations hold for **XmNcolumns**, **XmNwidth**, and **XmNresizeWidth**, except that the default number of columns is 20.

If the contents of the text (**XmNvalue** or **XmNvalueWcs**) change, as a result of user editing or an action by the application, the geometry calculation depends on the value of **XmNresizeHeight**. If **XmNresizeHeight** is True, the height of the widget is the height needed to display all the text. If **XmNresizeHeight** is False, the height of the widget does not change. The same relations hold for **XmNvalue**, **XmNvalueWcs**, **XmNresizeWidth**, and the width of the widget.

If the application sets another resource that affects the height needed by the widget, such as **XmNmarginHeight** or **XmNrenderTable**, the geometry calculation depends on the value of **XmNresizeHeight**. If **XmNresizeHeight** is True, the height of the widget is the height needed to display all the text with the new resource values. If **XmNresizeHeight** is False, the height of the widget is the height needed to display **XmNrows** of text by using the new resources. The same relations hold for these resources, **XmNresizeWidth**, **XmNcolumns**, and the width of the widget.

> **Note:** Most of the resources described in this section have other meanings when **XmNlayoutDirection** is **XmTOP_TO_BOTTOM** (for vertical writing). For example, if word–wrapping is turned on, text is broken at the end of a column and is continued at the next column.

Convenience Routines

Text convenience routines permit the application to perform many functions, including these:

- Insert and replace text.
- Cut, copy, and paste by using the clipboard.
- Get and set the editable state, the insertion cursor position, the maximum length of text, the primary selection and its position, the source, the text string, and the position of the first character displayed. For Text and TextField, all routines that have parameters or return values that are strings have both **char *** and **wchar_t *** versions.
- Convert between a text position and x and y coordinates.
- Display text at a given position and scroll the text.

ScrolledText

ScrolledText is a Text widget inside a ScrolledWindow. The application can use **XmCreateScrolledText** to create a Text widget inside a ScrolledWindow. This routine creates Text and ScrolledWindow widgets, and forces the following initial values for ScrolledWindow resources:

- **XmNscrollingPolicy** is set to **XmAPPLICATION_DEFINED**.
- **XmNvisualPolicy** is set to **XmVARIABLE**.
- **XmNscrollBarDisplayPolicy** is set to **XmSTATIC**.
- **XmNshadowThickness** is set to 0.

Note that the **XmCreateScrolledText** function returns the widget ID of the created Text widget. An application can then use **XtParent** to get the widget ID of the ScrolledWindow.

Storing Text in a File

A common requirement of many text editors is the ability to read text from a file, allow the user to edit the text, and then store the text in a file. An application usually obtains pathnames from the user by means of a FileSelectionBox, often invoked as a dialog from a MenuBar File Menu. Following are very simple routines that use ANSI C input/output facilities to read text from a file into a Text widget and save text from a Text widget into a file:

```
void ReadTextFromFile(Widget w, char *filename)
{
  FILE           *file;
  char            buffer[MAXSIZE];
  char           *ptr, *end;
  XmTextPosition  last_pos;
  int             val;
  if (file = fopen(filename, "r")) {
    XmTextSetString(w, "");
    ptr = buffer;
    end = buffer + MAXSIZE - 1;
    while((val = getc(file)) != EOF) {
      if (ptr < end) {
        *ptr++ = (char) val;
      } else {
        *ptr = '\0';
        last_pos = XmTextGetLastPosition(w);
        XmTextReplace(w, last_pos, last_pos, buffer);
        ptr = buffer;
      }
    }
    if (ptr > buffer) {
      *ptr = '\0';
      last_pos = XmTextGetLastPosition(w);
      XmTextReplace(w, last_pos, last_pos, buffer);
    }
    (void) fclose(file);
  }
}

void SaveTextToFile(Widget w, char *filename)
{
  FILE     *file;
  char     *text;
  if (file = fopen(filename, "w")) {
    text = XmTextGetString(w);
    (void) fputs(text, file);
    (void) fclose(file);
    XtFree(text);
  }
}
```

Sharing Text Sources

Each Text widget has a data structure of type **XmTextSource** that functions as the source and sink of text for the widget. The source is the value of the **XmNsource** resource.

Two or more Text widgets can share the same source. In this case, editing of Text in one widget changes the text of the source and therefore the text of all widgets that share that source. For example, an application might use a PanedWindow with multiple text widgets, each functioning as a "window" onto a single text source. Editing changes in one pane are reflected in all Text panes that share the same source.

An application creates a Text source by creating a Text widget. The program uses **XmTextGetSource** for Text or **XtGetValues** for the **XmNsource** resource to obtain that widget's source. The application then creates another Text widget, supplying the source obtained from the first widget by using **XmTextSetSource**, the initialization argument list, or **XtSetValues** of the **XmNsource** resource.

Setting a Text source destroys the existing source of the widget if no other widgets are sharing that source. To replace a Text source but keep it for later use, the application can create an unmanaged Text widget and set its source to the Text source the program wants to keep.

If the application does not supply a source, Text creates a default string source.