# Compound Strings

A compound string is simply a way to encode text independent of the font or the color used to display it. By pairing encoded text with an entry in a table of "renditions," Motif allows the application programmer great flexibility in the display of text information. By using renditions from a resource file, for example, a Motif programmer can create one application that can be useful in many different countries without being recompiled. Motif uses compound strings to display all text, except in the Text and TextField widgets.

This chapter introduces the structure of a compound string and its components. The data types associated with compound strings are reviewed, and the functions used to create, modify, and handle these strings are covered as well. Though this chapter will serve as an introduction to these data types and functions, they are not described completely here. For the complete definitions of each function and data type, please refer to the *Motif Programmer's Reference*.

## The Structure of a Compound String

A compound string is an opaque data structure consisting of one or more pieces of encoded text, and information about how to display each piece.

- **XmString**

This information consists of the direction in which to draw the text, and the name (also called the "tag") of one or more "renditions," which specify attributes of the drawn text, such as the font, the color of the text and its background, whether the text should be underlined or struck through, and a list of tab stops. The data type for a compound string is **XmString**.

A compound string is divided into "components," which identify the various parts of the compound string. A compound string component could be the text itself, without the rendition and direction, or it could be the direction indicator alone, or the rendition tag itself.

There are several different kinds of compound string components.

- **XmStringComponentType**
- **XmStringDirection**
- **XmStringTag**

Each of them consists of a component identifier (data type **XmStringComponentType**), and the component value, usually in the form of a text string. The text string might contain the text of the compound string, but it also might contain the tag of a rendition to use, the name of a character set, or the name of the current locale. For a complete list of the possible values of **XmStringComponentType**, and the order in which they should appear, please refer to the **XmStringComponentType** page of the *Motif Programmer's Reference*.

The direction of a compound string is stored so that the data structure will be equally useful for describing text in left−to−right languages such as English, Spanish, French, and German, and for text in right−to−left languages such as Hebrew and Arabic. The direction is stored as data type **XmStringDirection**, which may have the values **XmSTRING_DIRECTION_L_TO_R** or **XmSTRING_DIRECTION_R_TO_L**. A widget may itself have a default direction, called its "layout direction." For information about the interaction between a widget's layout direction and the string direction of a compound string to be displayed in that widget, please see chapter 11 of this manual, "Internationalization."

Another important compound string component is the "separator," which contains no text. When displayed, a separator usually maps to a line break. Do not confuse these with the Separator widget, which is also called **XmSeparator**.

An **XmStringTag** indicates the name, or "tag," of a particular rendition. The rendition describes several attributes to use when rendering a compound string, such as the font to use, the color, a list of tab stops, and whether the text is to be displayed underlined or struck out. Each rendition contains an **XmStringTag** that must match the tag included in the string itself.

> **Note:** The rendition tag should not be confused with the charset tag, which exists for compatibility with previous releases of Motif, or the locale tag, which specifies the locale for internationalized applications. In a compound string, these tags are specified with the **XmSTRING_COMPONENT_TAG** component, while the rendition tags are specified in the **XmSTRING_COMPONENT_RENDITION_BEGIN** and **XmSTRING_COMPONENT_RENDITION_END** components. The charset tag was called the fontlist tag in previous releases of Motif.

Several renditions together make up a render table, and a compound string component can actually be associated with more than one tag, indicating more than one rendition. Multiple renditions will be merged when the text is displayed. (See section 9.2.3 for a more complete description of the merging process.)

A rendition may also be indicated through a locale specification. See the chapter of this manual on Internationalization for more information about locales.

The rendition table and rendition tag structure parallels the old font list and font list element tag structure of earlier releases of Motif. The rendition structure has augmented, not supplanted, the font list. Font Lists should be considered obsolete, and their use should be avoided. However, for the purposes of compatibility with existing software, the font management function calls and data structures still exist. If both a font list and a render table are specified for some widget, the render table will take precedence, and the font list will be ignored.

## Creating a Compound String

There are several different functions available to create a compound string.

- **XmStringGenerate**

- **XmStringComponentCreate**

The most basic such function is **XmStringGenerate**, which accepts as arguments the text to be encoded, and a rendition tag to use to display it. It also accepts information about the locale or charset to use in rendering the text. The function returns an encoded **XmString**.

The **XmStringComponentCreate** function is used to create individual components of a compound string, such as the text, the tag, the direction, or the separator, which can then be concatenated into a single longer string.

There are a few functions that create a compound string which are obsolete, but are included for compatibility with earlier releases of Motif.

- **XmStringCreate**
- **XmStringSegmentCreate**
- **XmStringSeparatorCreate**

**XmStringCreate** may also be used to create a compound string of charset type with a specified font tag, with no rendition specified. The **XmStringCreate** function has been superseded, but is included for backward compatibility. Its function can be easily duplicated by calling **XmStringGenerate** with the text type **XmCHARSET_TEXT**, a charset tag, and a NULL rendition tag. **XmStringSegmentCreate** functions in the same manner to **XmStringCreate**, but allows the programmer to specify the string direction, and whether a separator should be appended to the returned **XmString**.

Earlier releases of Motif also included the functions **XmStringCreateLtoR** and **XmStringCreate**. These are still available, but again only for purposes of backward compatibility with existing code. Use **XmStringGenerate** instead.

The **XmStringSeparatorCreate** function may also be used instead of **XmStringSegmentCreate** to create a separator component for a compound string. In display, the separator would typically be mapped to a line break. Note that this function will not do anything more than **XmStringSegmentCreate** would if called with an empty text string.

The application that calls the compound string creation functions is responsible for performing its own memory management. The creation functions all must allocate space in which to return the strings.

- **XmStringFree**

When an application is finished with a compound string, the **XmStringFree** function should be used to free the memory that had been used for that string, and to make it available for some other purpose.

The following code sample demonstrates the use of **XmStringGenerate** and **XmStringFree** to create and display a compound string in a PushButton gadget.

```
    void RendCB(Widget, XtPointer, XtPointer);
    XmString        rend_label1, rend_label2;
    widget  rend_button1, rend_button2;
    XmRenderTable RenderTable;
    ...

    /* RenderTable created with two renditions:
     * Rendition1 and Rendition2 */

    ...
    RendPullDown = (Widget)XmCreatePulldownMenu(menubar, "RendPullDown",
                                                NULL, 0);

    rend_label1=XmStringGenerate("render1", NULL, XmCHARSET_TEXT,
                                 "Rendition1");
    rend_label2=XmStringGenerate("render2", NULL, XmCHARSET_TEXT,
                                 "Rendition2");

    rend_button1 = XtVaCreateManagedWidget("rend1", xmPushButtonGadgetClass,
                                    RendPullDown,
                                    XmNrenderTable, RenderTable,
                                    XmNlabelString, rend_label1,
                                    NULL);
    rend_button2 = XtVaCreateManagedWidget("rend2", xmPushButtonGadgetClass,
                                    RendPullDown,
                                    XmNrenderTable, RenderTable,
                                    XmNlabelString, rend_label2,
```

```
                                    NULL);
    XtAddCallback(rend_button1, XmNactivateCallback, RendCB, (XtPointer) 0);
    XtAddCallback(rend_button2, XmNactivateCallback, RendCB, (XtPointer) 1);

    XmStringFree(rend_label1);
    XmStringFree(rend_label2);
```

Note that the description of *Rendition1* and *Rendition2* must be explicitly sent to the widgets by using the **XmNrenderTable** resource to specify the render table containing these renditions. See section 9.2 about renditions and render tables. Also note that *rend1* and *rend2* are the *names* of the PushButton widgets; they are not the *labels* that will appear on those widget.

### Compound Strings and Resource Files

In an internationalized program, the labels and messages used to communicate with the user should be stored independently of the binary code of the compiled application, generally in external files. Under these conditions, adapting a program to different locales becomes as simple as replacing the external files. The compound strings used in Motif button labels and menu titles can be retrieved from X resource files.

In the preceding code example, the button label could have come from a resource file instead of being coded into the program. For this example, assume that the PushButtons are children of a widget called *RendPullDown*. The following example specifies two compound strings with the same rendition.

```
*RendPullDown.rend_button1.labelString:  render1
*RendPullDown.rend_button2.labelString:  render2
```

Here, Motif's string–to–compound–string converter produces a compound string from the resource file text. If the rendition is not specified, as in this example, the converter uses the rendition corresponding to the tag **_MOTIF_DEFAULT_LOCALE**. The format for specifying a rendition tag is analogous to the format for specifying a widget. The following example will draw the *render1* label with the rendition tagged by *rendition1*, and draw the *render2* label with the *rendition2* rendition. A later section of this chapter describes how a rendition and render table can be specified in a resource file.

```
*RendPullDown.rend_button1.renderTable.rendition1:  render1
*RendPullDown.rend_button2.renderTable.rendition2:  render2
```

Note that, when a compound string is specified in this manner, the string direction is inherited from the widget **XmNlayoutDirection** resource.

## Compound String Manipulation

To make working with compound strings as simple as using regular character strings, Motif contains a number of functions analogous to the basic string manipulation functions generally available with C.

- **XmStringCompare**
- **XmStringConcat**
- **XmStringConcatAndFree**
- **XmStringCopy**
- **XmStringEmpty**
- **XmStringIsVoid**

Use **XmStringCompare** to determine whether its two input compound strings are identical. **XmStringConcat** creates a third string by concatenating its two input strings. **XmStringConcatAndFree** does the same thing, except that it also deletes the two input strings and frees the memory occupied by them. **XmStringCopy** allocates enough memory to hold a copy of its input string, and returns a copy there. The **XmStringEmpty** function returns a Boolean value of True if there is no printable text in the input compound string. The **XmStringIsVoid** function is the same as the **XmStringEmpty** function except that it will only return True if there is no text at all in the string, printable or unprintable.

The following example takes an existing compound string called *string*, and sandwiches it with a new rendition, called *rendition*. This may also be accomplished with the **XmStringPutRendition** function (see section 9.2.3.1).

```
    XmString  string;
    XmString  tmp1, tmp2;
    ...

    tmp1 = XmStringComponentCreate(
                XmSTRING_COMPONENT_RENDITION_BEGIN, "rendition");
    tmp2 = XmStringComponentCreate(
```

```
                XmSTRING_COMPONENT_RENDITION_END, "rendition");

    string = XmStringConcat(tmp1, string);
    XmStringFree(tmp1);
    string = XmStringConcat(string, tmp2);
    XmStringFree(tmp2);
```

Display of the new string will use a composite rendition formed through a merge of the pre−existing rendition with the *rendition* rendition. See section 9.2.3 on render tables for the rules of merging renditions.

# Reading a Compound String

Motif provides two different functions to read the contents of a compound string.

- **XmStringGetNextTriple**
- **XmStringPeekNextTriple**

The **XmStringGetNextTriple** function is the most basic of the compound string readers. Given an initialized "compound string context," the function returns successive of the nominated compound string on successive calls. Use **XmStringPeekNextTriple** to determine the type of the next component without reading it, and without advancing the counter. A subsequent call to **XmStringGetNextTriple** would return the value of the component just peeked at.

In order to read a compound string with **XmStringGetNextTriple**, or to peek forward with **XmStringPeekNextTriple**, a program must first establish a "compound string context" to use. This is just a state structure used to keep track of how far into a compound string the program has read.

- **XmStringInitContext**
- **XmStringFreeContext**

Motif provides the **XmStringInitContext** function to read a compound string and allocate a buffer long enough to handle any of its components. As with all of the other Motif calls that allocate memory, the calling application is responsible for doing its own memory management. Motif provides a function, **XmStringFreeContext**, to free the memory associated with the context.

Two convenient functions are also provided to do two commonly needed tasks: searching for a substring and counting the lines of text.

- **XmStringHasSubString**
- **XmStringLineCount**

The **XmStringHasSubString** function takes two compound strings, the target string and the search string, as its arguments. The function returns True if the search string (which can only have one text component) is found in the target string.

The **XmStringLineCount** function simply returns the number of separators found in the input compound string, plus one. Since the separators usually map to a line terminator, this would be the number of lines in the displayed compound string.

## Storing a Compound String in a File

The **XmString** data type is opaque to the application programmer. In addition to the text, a compound string also contains data to tell a widget how to render the string. Some of this data may be in the form of pointers to various addresses, while other data may contain values.

If a program needs to store a compound string, or pass it to some other program, it must first convert the string to some more portable format, independent of any memory references. Motif provides two functions to convert a compound string into a byte stream, and to reverse the process.

- **XmCvtXmStringToByteStream**
- **XmCvtByteStreamToXmString**
- **XmStringByteStreamLength**

The **XmCvtXmStringToByteStream** function converts the text of a compound string and all its ancillary information, including rendition tags, direction indicators, and separators, into a byte sequence in an ASN.1−compliant format. This output string can then be written to a file, or passed to some other program. The size of the returned byte stream is internally specified in the ASN.1−format header, and can be returned with the **XmStringByteStreamLength** function. Once in hand, a program can call **XmCvtByteStreamToXmString** to convert the given byte stream back into a compound string.

Note that the compound string may contain rendition tags, and these are not resolved into the renditions they indicate. The text of the rendition

tags is actually transcribed into the byte stream output of **XmCvtXmStringToByteStream**. For any other program to make sense of the given data, it may need a copy of the render table, too. Motif provides two functions to convert render tables to and from a portable format:

- **XmRenderTableCvtFromProp**
- **XmRenderTableCvtToProp**

The **XmRenderTableCvtToProp** function takes a render table and outputs a character string containing the resource functions and values for each of the render table's renditions. Use **XmRenderTableCvtFromProp** to convert such a character string table back into a Motif render table. Refer to section 9.2.3 for more information about renditions and render tables.

## Displaying a Compound String

In most cases, a programmer need not worry about displaying a compound string. If the string is submitted to a widget, that widget is responsible for figuring out how to draw its own labels and text. Therefore, the functions described in this section are not used by most Motif applications. However, if a programmer chooses to *write* a widget, these functions will be quite useful.

- **XmStringDraw**
- **XmStringDrawImage**
- **XmStringDrawUnderline**

The **XmStringDraw** function draws a compound string in a specified window, in a specified display. The **XmStringDrawImage** function is nearly identical, except that, when drawing the string, it uses both the foreground and background bits of each character, where **XmStringDraw** only uses the specified foreground color.

**XmStringDrawUnderline** is the same as **XmStringDraw**, with the addition of one more argument, also a compound string. If this compound string matches any pattern in the compound string to be drawn, that part of the drawn string will be underlined.

In order to draw a string properly, it may be necessary to predict the drawn string's dimensions.

- **XmStringExtent**
- **XmStringHeight**
- **XmStringWidth**
- **XmStringBaseLine**

The **XmStringExtent** function returns the height and width of the smallest rectangle that could enclose the given compound string. The **XmStringHeight** and **XmStringWidth** functions return only the height or width of the string. Use **XmStringBaseLine** to find the number of pixels between the tallest possible letter and the baseline of the first line of text in the input compound string.

## Compound String Tables

Motif now provides support for a table of compound strings. This is simply an array of compound strings, typically to be used to display or accept tabular information. Motif provides functions for creating and manipulating such arrays.

- **XmStringToXmStringTable**
- **XmStringTableToXmString**

The **XmStringToXmStringTable** function takes a single compound string as its input, along with a compound string component, *break_component*. Upon return, the string is broken up, wherever the specified *break_component* appears, and the function returns an array (table) of the resulting compound strings. The component marking the breaks will not appear in the output table.

Use **XmStringTableToXmString** to convert a table of compound strings back into a single compound string. This function also uses a *break_component* argument, which it inserts into the resulting compound string between each of the elements of the input table.

The following two functions convert regular text to a table of compound strings and back again. They both use the parsing capabilities of Motif, which are described in section 9.4.

- **XmStringTableParseStringArray**
- **XmStringTableUnparse**

The **XmStringTableParseStringArray** function accepts an array of simple text strings, calls **XmStringParseString** on each one with the given parse table, and puts the result into a table of compound strings. The **XmStringTableUnparse** function also uses an input parse table, to convert the elements of the input compound string table back into regular text strings. See section 9.4 for more information about parsing and parse tables.

# Renditions

A rendition (data type **XmRendition**) is an opaque data type used to specify data used in rendering compound strings.

- **XmRendition**

A rendition has two parts: the set of data used to render a compound string (such as fonts, colors, and tabs), and a name, or "tag" (data type **XmStringTag**), by which to identify it.

# Creating a Rendition

A rendition is neither a widget nor a gadget. However, the style used to specify resources for widgets and gadgets is a simple and familiar way to set data values, and it has been implemented for renditions well. To change the foreground color for a rendition, the application programmer must set a "resource" called **XmNrenditionForeground** for that rendition, in a manner identical to that of a programmer changing a color resource for some widget. For a complete description of the XmRendition resource set, see the XmRendition page in the *Motif Programmer's Reference*.

- **XmRenditionCreate**

**XmRenditionCreate** takes an argument list (use **XtSetArg** to create the list) of resource name and value pairs as input, and returns a rendition, which can then be entered into the render table.

The following code fragment establishes a rendition that would display a compound string in the 8x16 font, in blue, and underlined with a single line.

```
int n;
XmRendition Rendition;
XmStringTag RenditionTag;

   XtVaGetValues(parent, XmNcolormap, &cmap, NULL);
   if ( XAllocNamedColor(XtDisplay(parent), cmap, "blue", &color, &unused))
    {
      pixel_color = color.pixel;
    } else {
      pixel_color = XmUNSPECIFIED_PIXEL;
    }

   n = 0;
   XtSetArg( args[n], XmNrenditionForeground, pixel_color); n++;
   XtSetArg( args[n], XmNfontName, "8x16" ); n++;
   XtSetArg( args[n], XmNfontType, XmFONT_IS_FONT ); n++;
   XtSetArg( args[n], XmNunderlineType, XmSINGLE_LINE );  n++;
   RenditionTag = (XmStringTag) "Rendition1";
   Rendition = XmRenditionCreate( parent, RenditionTag, args, n );
```

# Editing a Rendition

These functions are used to read and modify a rendition.

- **XmRenditionRetrieve**
- **XmRenditionUpdate**

The **XmRenditionUpdate** function works in a similar way to **XmRenditionCreate**, merging the given argument list into the definition of the rendition. Resources in the input ArgList that were **XmAS_IS** will be set to the given values, and resources in the input ArgList that were already set will be reset to the new value. Resources not in the input ArgList will remain unchanged. The **XmRenditionRetrieve** function also uses an ArgList, but the user must specify the address (instead of the value) of each resource value to be returned.

The following code retrieves a rendition. If the rendition was underlined, the underline is removed. If the rendition was not underlined, the underline is inserted.

```
int n;
unsigned char underline;

n = 0;
```

```
XtSetArg( args[n], XmNunderlineType, &underline ); n++;
XmRenditionRetrieve( Rendition, args, n );

if (( underline == XmAS_IS) || ( underline == XmNO_LINE ))
     underline = XmSINGLE_LINE;

else if ( underline == XmSINGLE_LINE)
      underline = XmNO_LINE;

n = 0;
XtSetArg( args[n], XmNunderlineType, underline ); n++;
XmRenditionUpdate( Rendition, args, n );
```

The modified rendition must still be added to the render table, and the modified render table sent back to the widget, before it is accessible to a widget needing to display a compound string.

- **XmRenditionFree**

The **XmRenditionCreate** function allocates memory for the created rendition structure. The application calling this function is responsible for maintaining its own memory management scheme. Use **XmRenditionFree** to recover memory space allocated to a rendition when the application is done with it. The application may dispose of the rendition as soon as it has been added to a render table.

# Render Tables

A collection of renditions, each identified by a rendition "tag" (type **XmStringTag**), constitutes a render table (type **XmRenderTable**). Each rendition contains specifications describing how to display text, including the font, the color, the tab stops, and other information.

- **XmRenderTable**

A compound string can specify its rendition in more than one way. The typical way is simply for each compound string text component to be associated with a rendition tag, which is matched with the entry in the render table with the same tag. However, an application may use the default rendition for the current locale (tag=**_MOTIF_DEFAULT_LOCALE**). Alternatively, older applications may specify a charset tag, which will identify a font to use to display some text. The charset and the locale options may not both be specified, but both may exist with one or more rendition tags.

Renditions are accumulated as a compound string is read. If there have been three rendition begin tags and only one corresponding rendition end tag preceding a particular segment of text, then there are two renditions associated with that text. The *last* rendition specified that is still in effect is the primary rendition. If a segment of text is associated with more than one rendition, and if there are unspecified values in the primary rendition, the widget must create an effective rendition for that segment. This is formed by using the previous (active) rendition to fill in the unspecified values of the primary rendition. If this effective rendition still has unspecified values, then the next active rendition back is used, and so on. Finally, if the resulting rendition still has resources with unspecified values and the segment has a locale or charset tag (these are optional and mutually exclusive) this tag is matched with a rendition in the render table, and the missing rendition values are filled in from that entry.

If no matching rendition is found for a particular tag, then the **XmNoRenditionCallback** of the **XmDisplay** object is called and the render table is searched again for that tag.

If the resulting rendition does not specify a font or fontset, then for segments whose text type is **XmCHARSET_TEXT**, the render table will be searched for a rendition tagged with **XmFONTLIST_DEFAULT_TAG**, and if a matching rendition is found, it will be merged into the current rendition. If the resulting rendition contains no font or fontset, the **XmNnoFontCallback** will be called with the default rendition and "" as the font name. If no rendition matches or no font was found after the callback, then the first rendition in the render table will be merged into the current rendition. If this rendition still has no font, then the segment will not be rendered and a warning will be issued.

For segments whose text type is **XmMULTIBYTE_TEXT** or **XmWIDECHAR_TEXT**, the render table will be searched for a rendition tagged with **_MOTIF_DEFAULT_LOCALE**, and if a matching rendition is found, it will be merged into the current rendition. If the resulting rendition contains no font, the **XmNnoFontCallback** will be called with the default rendition and "" as the font name. If no rendition matches or no font was found after the callback, then the segment will not be rendered and a warning will be issued.

For example, imagine a render table containing three renditions. One belongs to the default locale, and has the tag **_MOTIF_DEFAULT_LOCALE**, while the other two renditions are called *fred* and *susan*. Further suppose the default locale rendition specifies a font, but the foreground and background color resources are **XmUNSPECIFIED_PIXEL**. Rendition *fred* specifies the foreground color as red, but the background field is **XmUNSPECIFIED_PIXEL**, and rendition *susan* gives blue for the foreground and purple for the background. Following is a schematic representation of such a render table:

```
_MOTIF_DEFAULT_LOCALE:
```

```
        XmNfontName:   variable
        XmNfontType:   XmFONT_IS_FONT
        XmNrenditionForeground: XmUNSPECIFIED_PIXEL
        XmNrenditionBackground: XmUNSPECIFIED_PIXEL
        ...
fred:
        XmNfontName:   XmAS_IS
        XmNfontType:   XmAS_IS
        XmNrenditionForeground: red
        XmNrenditionBackground: XmUNSPECIFIED_PIXEL
        ...
susan:
        XmNfontName:   XmAS_IS
        XmNfontType:   XmAS_IS
        XmNrenditionForeground: blue
        XmNrenditionBackground: purple
        ...
```

A text component associated with the *fred* and *susan* tags (in that order) will have red type on a purple background. But a component using *susan* and *fred* will have blue type on a purple background. Both components will use the font corresponding to the locale rendition, *variable* in this case. Note that the order of the renditions in the merge sequence is the opposite of the order in which the corresponding **XmSTRING_COMPONENT_RENDITION_BEGIN** components appear in the compound string. That is, the primary rendition for some piece of text corresponds to the *last* rendition component read.

If the search of the render table results in no font or fontset, then if there is a rendition in the render table with a tag of **_MOTIF_DEFAULT_LOCALE**, and if that rendition specifies a font, then that font will be used. If no font or fontset is specified at this point, the text component will not be rendered and a warning message will be displayed.

### Creating a Render Table

Before creating a render table, an application program must first have created at least one of the renditions that will be part of the table.

- **XmRenderTableAddRenditions**

As the name implies, this function is also used to augment a render table with new renditions. To create a new render table, call the **XmRenderTableAddRenditions** function with a NULL argument in place of an existing render table. The following code creates a render table from three renditions.

```
n = 0;
XtSetArg( args[n], XmNtopAttachment, XmATTACH_FORM ); n++;
XtSetArg( args[n], XmNwidth, LIST_WIDTH ); n++;
XtSetArg( args[n], XmNvisibleItemCount, NUM_ITEMS ); n++;
List = XmCreateList( Manager, "List", args, n );
XtManageChild(List);

n = 0;
XtSetArg( args[n], XmNfontName, "fixed" ); n++;
XtSetArg( args[n], XmNfontType, XmFONT_IS_FONT ); n++;
XtSetArg( args[n], XmNunderlineType, XmNO_LINE );  n++;
Renditions[0] = XmRenditionCreate( List, (XmStringTag)tags[0],
                  args, n );

n = 0;
XtSetArg( args[n], XmNfontName, XmAS_IS ); n++;
XtSetArg( args[n], XmNfontType, XmFONT_IS_FONT ); n++;
XtSetArg( args[n], XmNunderlineType, XmSINGLE_LINE );  n++;
Renditions[1] = XmRenditionCreate( List, (XmStringTag)tags[1],
                  args, n );

n = 0;
XtSetArg( args[n], XmNfontName, "variable" ); n++;
XtSetArg( args[n], XmNfontType, XmFONT_IS_FONT ); n++;
XtSetArg( args[n], XmNunderlineType, XmDOUBLE_LINE );  n++;
Renditions[2] = XmRenditionCreate( List, (XmStringTag)tags[2],
```

```
                    args, n );

    RenderTable =
        XmRenderTableAddRenditions( NULL, Renditions, 3,
                                XmREPLACE );

    n = 0;
    XtSetArg( args[n], XmNrenderTable, RenderTable ); n++;
    XtSetValues( List, args, n );
```

Note that the list widget must specify the resulting render table in its **XmNrenderTable** resource in order for that widget to have access to the render table's rendition data.

- **XmStringPutRendition**

To create a new compound string with a new rendition out of an old compound string, use the **XmStringPutRendition** function. Given an **XmString** and the tag of a rendition from a render table, this function places **XmSTRING_COMPONENT_RENDITION_BEGIN** and **XmSTRING_COMPONENT_RENDITION_END** components containing the input rendition tag around a copy of the old string. This function does not strip old rendition data out of the string, so the resulting string will merge the existing rendition with the new one when it is rendered. See the example in section 9.1.2.

## Editing a Render Table

The **XmRenderTableAddRenditions** may also be used to update an existing render table by adding, replacing, or merging new renditions into the table. The following functions also provide important editing functionality.

- **XmRenderTableGetRenditions**
- **XmRenderTableRemoveRenditions**
- **XmRenderTableCopy**
- **XmRenderTableGetTags**

These functions are used to read and modify render tables. Use the **XmRenderTableGetRenditions** function to call up one or more renditions for a table. A program might do this in order to modify some of the renditions in the table. The **XmRenderTableRemoveRenditions** function will delete any unneeded renditions from a given table. A related function is **XmRenderTableCopy**, which will copy only renditions matching the input list of rendition tags into a new render table. Use **XmRenderTableGetTags** to retrieve a list of tags from a render table.

The following code uses the render table created in the last section, and modifies the *variable* rendition to include a single line "strike through." (This produces a horizontal line through the drawn text, as if the text had been crossed out.)

```
    int n;
    XmRendition Rendition;

        Rendition = XmRenderTableGetRenditions(RenderTable,
                        (XmStringTag) "variable", 1);

        n = 0;
        XtSetArg( args[n], XmNstrikethruType, XmSINGLE_LINE ); n++;
        XmRenditionUpdate( Rendition, args, n );

        XmRenderTableAddRenditions(RenderTable, Rendition, 1,
                        XmMERGE_NEW);

        n = 0;
        XtSetArg( args[n], XmNrenderTable, RenderTable ); n++;
        XtSetValues( List, args, n );
```

Note that the modified render table must be sent again to the widgets in which it will be used before becoming effective.

- **XmRenderTableFree**

Although the **XmRenderTableAddRenditions** and **XmRenderTableCopy** functions allocate memory for a render table, the application program is responsible for its own memory management. Use the **XmRenderTableFree** function to free memory when a render table is no longer needed. Note, however, that the **XmRenderTableAddRenditions** function automatically frees the memory occupied by its input render table.

### Creating a Render Table in a Resource File

Renditions and render tables may be specified in resource files. For a properly internationalized application, in fact, this is the preferred method. When the render tables are specified in a file, the program binaries are made independent of the particular needs of a given locale, and may be easily customized to local needs. Render tables are specified in resource files with the following syntax:

resource_spec : [ tag [, tag ]* ]

where *tag* is some string suitable for the **XmNtag** resource of a rendition. This line creates an initial render table containing one more renditions than the number of tags specified. The renditions are attached to the specified tags, with the untagged rendition going with the tag **_MOTIF_DEFAULT_LOCALE**. If no tags are specified, then a render table will be created that contains only a rendition with a tag of **_MOTIF_DEFAULT_LOCALE**.

Specific values for specific rendition resources are specified using the following syntax:

resource_spec [*|.] rendition [*|.] resource_name :value

where *resource_spec* specifies the render table, *rendition* is either the class Rendition or a tag, *resource_name* is either the call or name of a particular resource, and *value* is the specification of the value to be set.

Any resource line that consists of just a resource name or class component with no rendition component or loose binding will be assumed to specify resource values for a rendition with a tag of **_MOTIF_DEFAULT_LOCALE**.

For example, the following:

```
*List.renderTable: green, variable
*List.renderTable.green.renditionForeground: Green
*List.renderTable.green.fontName: AS_IS
*List.renderTable.variable.underlineType: SINGLE_LINE
*List.renderTable.variable.renditionForeground: Red
*List.renderTable.variable.fontName: variable
*List.renderTable.variable.fontType: FONT_IS_FONT
*List.renderTable.renditionForegound: black
*List.renderTable.fontName: fixed
*List.renderTable.fontType: FONT_IS_FONT
*List.renderTable.variable.underlineType: NO_LINE
*List.renderTable*tabList: 1in, +1.5in, +3in
```

would set the *renderTable* resource of *List* to a render table consisting of three renditions tagged with **_MOTIF_DEFAULT_LOCALE**, *green*, and *variable* with values for resources set as described in the resource specifications. Note that the *tabList* resource will be shared by all three renditions.

### Widget Resources and Render Tables

Several widget resource sets have been modified in Motif version 2.0 to accommodate the use of renditions and render tables. Several widgets have added the **XmNrenderTable** resource to their set. Several others have had more complex changes. The following list outlines the new resources for the affected widget classes. For complete information about the resource set of any widget, please refer to the appropriate section of the *Motif Programmer's Reference*.

**XmList**, **XmLabel**, **XmLabelGadget**, **XmScale**, **XmText**, **XmTextField**

These widget classes use the **XmNrenderTable** resource to nominate their render table. The resource may be set at any time.

**XmMenuShell**

The **XmMenuShell** widget allows different render tables to be specified for different descendants. For button descendants, use the **XmNbuttonRenderTable** resource; and for label descendants, use the **XmNlabelRenderTable** resource. These resources may be set and reset at any time.

**VendorShell**, **XmBulletinBoard**

These widgets also allow different render tables for different kinds of descendants. The **XmNbuttonRenderTable** resource controls the render table for button descendants, the **XmNlabelRenderTable** controls any label descendants, and the **XmNtextRenderTable** resource covers the text descendants. These resources may be set and reset at any time.

**XmDisplay**

The **XmDisplay** widget has two resources that provide callback branches in case a descendant widget finds itself without enough information to render any given text. The **XmNnoFontCallback** resource nominates a callback routine to be invoked if a widget cannot find a font to use for any given compound string, and the **XmNnoRenditionCallback** resource nominates a callback in case a valid rendition cannot be found. These resources are of type **XtCallbackList**, and may only be set at the creation of the widget.

Upon execution of the callback routine nominated by either of these resources, a pointer to the following structure is passed to the **XmNnoFontCallback** and **XmNnoRenditionCallback** callbacks:

```
typedef struct
{
        int             reason ;
        XEvent          *event ;
        XmRendition     rendition ;
        char            *font_name ;
        XmRenderTable   render_table ;
        XmStringTag     tag ;
}XmDisplayCallbackStruct;
```

*reason*

Indicates why the callback was invoked. The possible values are **XmCR_NO_FONT** and **XmCR_NO_RENDITION**.

*event*

Points to the **XEvent** that triggered the callback. It can be NULL.

*rendition*

Specifies the rendition with the missing font.

*font_name*

Specifies the name of the font or font set which could not be loaded.

*render_table*

Specifies the render table with the missing rendition.

*tag*

Specifies the tag of the missing rendition.

### Porting Render Tables

The render table is an opaque data type. If a programmer wishes to save a render table for some future use, or to pass the table to some other program, the application using the render table must first translate the render table into a portable format. Motif provides two functions to convert render tables to and from such a format:

- **XmRenderTableCvtFromProp**
- **XmRenderTableCvtToProp**

The **XmRenderTableCvtToProp** function takes a render table and outputs a character string containing the resource meanings and values for each of the render table's renditions. Use **XmRenderTableCvtFromProp** to convert such a character string table back into a Motif render table.

## Font Lists

The font list paradigm of Motif 1 has been supplanted by renditions and render tables. The font list management routines are provided only for backward compatibility with older applications written. They should not be used for new applications, which can use renditions and render tables for the same effect. Following is a list of the remaining font list management functions:

- **XmFontListAppendEntry**

- **XmFontListCopy**
- **XmFontListEntryCreate**
- **XmFontListEntryFree**
- **XmFontListEntryGetFont**
- **XmFontListEntryGetTag**
- **XmFontListEntryLoad**
- **XmFontListFree**
- **XmFontListFreeContext**
- **XmFontListGetNextFont**
- **XmFontListInitFontContext**
- **XmFontListNextEntry**
- **XmFontListRemoveEntry**
- **XmFontListAdd** (superseded by **XmFontListAppendEntry**)
- **XmFontListCreate** (superseded by **XmFontListAppendEntry**)

## Horizontal Tabs

To control the placement of text, a compound string can contain <tab> characters. To interpret those characters on display, a widget will refer to the rendition in effect for that compound string, where it will find a list of tab stops.

- **XmTab**
- **XmTabList**

A tab stop is encoded in an **XmTab** data type, and a **XmTabList** is an array of one or more **XmTab**s. A tab stop contains a tab value, the units of the value (such as inches, centimeters, pixels), the offset model, and the alignment model. The offset model dictates whether the tab measurement is absolute, measured from the left margin of the compound string display (or the right margin if the layout direction is right–to–left), or relative, measured from the previous tab stop. The tab stop information also contains information about the alignment of the text and the tab stop. A user could choose to align the tab stop with the beginning, center, or end of the text appearing at the stop. However, the alignment model is included only for future use, and the only alignment model supported by Motif will align the beginning of the following text with the tab stop.

Note that only the tab list associated with the first text component on any displayed line is used. If a rendition contains a NULL tab list, the default tab list is used. This consists of ten tabs, each of eight font units (**XmFONT_UNIT**).

## Creating a Tab Stop

The following functions are used to create a single tab stop.

- **XmTabCreate**
- **XmTabGetValues**
- **XmTabSetValue**

Use the **XmTabCreate** convenience function to create a tab stop from a tab value, the tab units, and the offset model. To specify the offset model, there are the defined constants **XmABSOLUTE** and **XmRELATIVE**. Note that the tab value must be a positive number, and that, if a tab stop would cause a text component to overwrite the previous component, the x position of the component will be moved (the tab is ignored) to follow the first component. There is also an input argument to **XmTabCreate** to define the alignment of the text relative to the tab stop. Presently, **XmALIGNMENT_BEGINNING** is the only valid value.

Tab stops may be measured in any of the following units:

- **XmPIXELS**
- **XmINCHES**
- **XmCENTIMETERS**
- **XmMILLIMETERS**
- **XmPOINTS**
- **XmFONT_UNITS**

Font units are calculated from the dimensions of the font used. Horizontal font units (used in horizontal tabs) are calculated as follows:

- If the font has an **AVERAGE_WIDTH** property, the horizontal font unit is the **AVERAGE_WIDTH** property divided by 10.
- If the font has no **AVERAGE_WIDTH** property but has a **QUAD_WIDTH** property, the horizontal font unit is the **QUAD_WIDTH** property.
- If the font has no **AVERAGE_WIDTH** or **QUAD_WIDTH** property, the horizontal font unit is the sum of the font structure's *min_bounds.width* and *max_bounds.width* divided by 2.3.

The **XmTabGetValues** function separates a tab stop into its constituent parts, as outlined above, and the **XmTabSetValue** function is used to change only the *value* field, or the actual measurement of an existing tab stop. **XmTabSetValue** has no effect on the units, the offset model, or the alignment of a tab stop.

## Creating a Tab List

Motif provides convenience functions to create tab lists in an application.

- **XmTabListInsertTabs**
- **XmStringTableProposeTablist**

The **XmTabListInsertTabs** function adds one or more **XmTab**s to an already existing **XmTabList**. Use it with an empty input tab list to create a tab list.

The **XmStringTableProposeTablist** function takes an **XmStringTable** structure containing tabbed compound strings and, using its rendition information and input about the desired padding between columns, returns a proposed tab list. This tab list is defined so that, if used to render the strings in the table, it would cause the strings to line up in columns with no overlap and with the specified amount of padding between the widest item in each column and the start of the next column. Each tab in the tab list would have the same unit type, offset model, and alignment type, specified on input.

Tab stops and tab lists may also be specified in a resource file. To create a tab stop in a resource file, use the following syntax:

```
resource_spec : tab [, tab ]*
```

The resource value string consists of one or more tabs separated by commas. Each *tab* identifies the value of the tab, the unit type, and whether the offset is relative or absolute. The tab stop specification uses the following syntax:

```
tab = [ + ] float [ units ]
```

The tab contains a decimal number and an indication of the units to be used. If no units are specified, the default unit is pixels. The presence or absence of a sign indicates, respectively, a relative offset or an absolute offset model. A relative tab stop is measured from the previous tab stop in the list, while an absolute tab stop is measured from the left margin (or the right margin if the layout direction is right–to–left).

For example, the following line of a resource file sets a tab list for a rendition (*rend1*) in a render table (*render_table*) used by a widget called *List*. It specifies a tab list consisting of a 1–inch absolute tab followed by a 1–inch relative tab (equivalent to a 2–inch absolute tab, unless and until a third tab is inserted between the two):

```
*List.render_table.rend1.tabList: 1in, +1in
```

The recognized units and their abbreviations include:

*pixels*

*pix*, *pixel*, *pixels*

*inches*

*in*, *inch*, *inches*

*centimeter*

*cm*, *centimeter*, *centimeters*

*millimeters*

*mm*, *millimeter*, *millimeters*

*points*

*pt*, *point*, *points*

*font units*

*fu*, *font_unit*, *font_units*

See section 9.3.1 for more information about font units.

# Editing a Tab List

These are the functions provided to maintain and modify a tab list:

- **XmTabListRemoveTabs**
- **XmTabListCopy**
- **XmTabListReplacePositions**
- **XmTabListTabCount**
- **XmTabListGetTab**

To remove tabs from a list, a program may use either **XmTabListRemoveTabs**, **XmTabListCopy**, or **XmTabListReplacePositions**. The first of these actually deletes the indicated **XmTab** entries from the list. The second and third functions create a new **XmTabList** using a subset of the original list. For **XmTabListCopy**, the subset must be in order in the original list, while **XmTabListReplacePositions** can pick an assortment of tab stops out of the old list to put in the new one.

The **XmTabListGetTab** function returns a copy of the tab stop at the indicated position in the list, and **XmTabListTabCount** simply returns the number of tab stops in a given tab list.

- **XmTabFree**
- **XmTabListFree**

The functions that create tabs and tab lists allocate memory in which to put them. When an application is finished with the tabs, it should use **XmTabFree** and **XmTabListFree** to free that memory for other uses.

# Parse Tables

A *parse mapping* (data type **XmParseMapping**) is a fragment of multibyte text paired with a compound string component. When translating text into a compound string, using the **XmParse** functions, any occurrences of the parse mapping text will be translated into the compound string component with which it is paired in a parse mapping. Likewise, when converting a compound string back into regular text, each compound string component that matches a parse mapping will be translated into the corresponding text string. A *parse table* (data type **XmParseTable**) is nothing more than an ordered array of **XmParseMapping**s.

# Structure of a Parse Mapping

Though the parse mapping is at heart nothing more than the text/compound string pairs, Motif stores each parse mapping with *resources* that indicate how the parsing is to proceed. Note that these are not widget resources, but the widget resource model provides a convenient way to specify and recall parse mapping instructions.

- **XmParseMappingCreate**
- **XmParseMappingGetValues**
- **XmParseMappingSetValues**

Following this model, an application uses the **XtSetArg** function to build a resource−style argument list to submit to the **XmParseMappingCreate**, **XmParseMappingGetValues**, or **XmParseMappingSetValues** in the same manner that would be used to create or change the resource values of some widget.

The parse mapping "resources" are outlined briefly below. For more information about the **XmParseMapping** data type and its resources, please see the *Motif Programmer's Reference*.

**XmNpattern**

Specifies a pattern to be matched in the text being parsed. At present, this is a maximum of one character.

**XmNpatternType**

Specifies the type of the pattern that is the value of **XmNpattern**. Following are the possible values:

**XmCHARSET_TEXT** (default)
**XmMULTIBYTE_TEXT**
**XmWIDECHAR_TEXT**

**XmNsubstitute**

Specifies the compound string to be included in the compound string being constructed when **XmNincludeStatus** is **XmINSERT** or **XmTERMINATE**.

**XmNincludeStatus**

Specifies how the result of the mapping is to be included in the compound string being constructed. Unless the value is **XmINVOKE**, the result of the mapping is the value of **XmNsubstitute**. Following are the possible values for **XmNincludeStatus**:

**XmINSERT**

Concatenate the result to the compound string being constructed and continue parsing. This is the default value.

**XmINVOKE**

Invoke the **XmNinvokeParseProc** on the text being parsed and use the returned compound string instead of **XmNsubstitute** as the result to be inserted into the compound string being constructed. The include status returned by the parse procedure (**XmINSERT** or **XmTERMINATE**) determines how the returned compound string is included.

**XmTERMINATE**

Concatenate the result to the compound string being constructed and terminate parsing.

**XmNinvokeParseProc**

Specifies the parse procedure to be invoked when **XmNincludeStatus** is **XmINVOKE**. This defaults to NULL. See the following section about the parse procedure.

**XmNclientData**

This is a **XtPointer** used to indicate data to be used by the parse procedure. The default value is NULL.

## Parse Procedures

The Motif parsing mechanism allows for the execution of an arbitrarily complex parsing algorithm. If the simple substitution of compound string components for text strings is not powerful enough for a particular application, a programmer can define a *parse procedure* that will be invoked when certain text characters are read.

While parsing a text string, if an **XmNpattern** attached to a non–NULL **XmNinvokeParseProc** resource, matches part of the input string, the nominated parse procedure is invoked.

The **XmNinvokeParseProc** resource is a function of type **XmParseProc**, which is defined as follows:

```
XmIncludeStatus (*XmParseProc)(
        XtPointer       *text_in_out ,
        XtPointer       text_end ,
        XmTextType      type ,
        XmStringTag     tag ,
        XmParseMapping  entry ,
        int     pattern_length ,
        XmString        *str_include ,
        XtPointer       call_data )
```

The input text is a pointer to the first byte of the pattern that was matched to trigger the call to the parse procedure. The parse procedure may use, modify, or remove as many bytes of the input string as it needs, as long as upon return the input text pointer is advanced to the following byte. It returns a compound string to be included in the compound string being constructed, as well as an **XmIncludeStatus**, corresponding to the parse mapping resource, indicating how the returned compound string should be handled.

*text_in_out*

Specifies the text being parsed. The value is a pointer to the first byte of text matching the pattern that triggered the call to the parse procedure. When the parse procedure returns, this argument is set to the position in the text where parsing should resume––that is, to the byte following the last character parsed by the parse procedure.

*text_end*

Specifies a pointer to the end of the *text_in_out* string. If *text_end* is NULL, the string is scanned until a NULL character is found. Otherwise, the string is scanned up to but not including the character whose address is *text_end*.

*type*

Specifies the type of text and the tag type. If a locale tag should be created, *type* has a value of either **XmMULTIBYTE_TEXT** or **XmWIDECHAR_TEXT**. If a charset should be created, *type* has a value of **XmCHARSET_TEXT**.

*tag*

Specifies the tag to be used in creating the result string. The type of string tag created (charset or locale) depends on the text type and the *tag* value passed to the parse procedure. If the *tag* value is NULL and if *type* indicates that a charset tag should be created, the string tag has the value **Xm_FONTLIST_DEFAULT_TAG**. If *type* indicates a locale tag, the string tag has the value **_MOTIF_DEFAULT_LOCALE**.

*entry*

Specifies the parse mapping that triggered the call to the parse procedure.

*pattern_length*

Specifies the number of bytes in the input text, following *text_in_out*, that constitute the matched pattern.

*str_include*

Specifies a pointer to a compound string. The parse procedure creates a compound string to be included in the compound string being constructed. The parse procedure then returns the compound string in this argument.

*call_data*

Specifies data passed by the application to the parsing routine.

The parse procedure returns an **XmIncludeStatus** indicating how *str_include* is to be included in the compound string being constructed. Following are the possible values:

**XmINSERT**

Concatenate the result to the compound string being constructed and continue parsing.

**XmTERMINATE**

Concatenate the result to the compound string being constructed and terminate parsing.

Upon return, the function that invoked the parse procedure will continue looking for matching patterns from the position of the text pointer. If the parse procedure has not advanced the pointer, and the text and parse mappings also remain the same, the parse procedure will be called again, perhaps infinitely.

Two parse procedures are provided as part of Motif.

- **XmeGetNextCharacter**
- **XmeGetDirection**

The **XmeGetNextCharacter** procedure consumes the byte the input pointer indicates, and the following byte, and returns a compound string component containing only the second byte. The effect is to copy the byte following the trigger byte directly into the output compound string, even if the normal parsing would not have done so. In other words, the trigger byte has served as an "escape" character. The **XmeGetNextCharacter** return value is **XmINSERT**, so parsing will continue normally from the same spot.

The other parse procedure included as a part of Motif is **XmeGetDirection**. This function simply returns an **XmString** component of type **XmSTRING_COMPONENT_DIRECTION**, whose value depends on the input text.

# Use of a Parse Table

The following functions use a parse table to translate text into the compound string format and the reverse.

- **XmStringParseText**
- **XmStringUnparse**

To use a parse table, send it and a companion text string to the **XmStringParseText** function. This function also accepts a pointer to data that may be passed to a parse procedure, if one exists. The function will walk through the input text string, copying characters from it into a compound string; when it finds a pattern (specified by **XmNpattern** in the parse mappings) to match a pattern in the input parse table, it will not copy the pattern, but write the corresponding compound string (indicated by **XmNsubstitute** in the parse mapping) to the output **XmString**.

In the event it is desirable to translate a compound string back into a text string, the **XmStringUnparse** function is provided. This provides the precise converse to the **XmStringParseText** function, stepping through the input compound string and substituting text patterns for **XmString** components that match the **XmNsubstitute** resources in the parse table. If there is no parse procedure nominated in the parse table, the **XmStringUnparse** function called on the output of the **XmStringParseText** function will equal the input of **XmStringParseText**. A compound string created with a parse table that includes a parse procedure will not be correctly translated into simple text by a call to **XmStringUnparse**.

- **XmParseMappingFree**
- **XmParseTableFree**

The routines that create parse mappings allocate memory in which to store those mappings. It is the responsibility of the application, however, to free that memory when the mappings and tables are no longer needed. Use **XmParseMappingFree** and **XmParseTableFree** to free the memory

for other uses.

# Compound Strings in UIL

Text strings in UIL may be specified as simple string literals or as compound strings. Compound strings are created with the **COMPOUND_STRING** or the **COMPOUND_STRING_COMPONENT** functions. String literals are simply quoted text, and are stored either as NULL−terminated strings or as compound strings, depending on the complexity of the specified string. The UIL concatenation operator (*&*) concatenates both string literals and compound strings.

UIL also supports wide−character strings, which are documented in Chapter 11.

# Simple Text Strings

A simple text string, or string literal, consists of text, a character set, and a layout direction. The character set may be explicitly specified, or a default value will be used. UIL infers the layout direction from the character set.

Some string literals will actually be stored by UIL as compound strings. This happens when

- A string expression consists of two or more literals with different character sets or writing directions, or
- The literal or expression in question is used as a value that has a compound string data type (such as the value of a resource whose data type is **XmString**).

UIL recognizes a number of keywords specifying character sets. UIL associates parsing rules, including parsing direction and whether characters have 8 or 16 bits, for each character set it recognizes. It is also possible to define a character set by using the UIL **CHARACTER_SET** function.

The syntax of a string literal is one of the following:

```
'[character_string ]'
[#char_set ]"[character_string ]"
```

For each syntax, the character set of the string is determined as follows:

- For a string declared as *'string'*, the character set is the code set component of the **LANG** environment variable if it is set in the UIL compilation environment, or the value of **XmFALLBACK_CHARSET** if **LANG** is not set or has no code set. By default, the value of **XmFALLBACK_CHARSET** is ISO8859−1, but vendors may supply different values.
- For a string declared as *#char_set"string"*, the character set is *char_set*. If the *char_set* in a string specified in this form is not a built−in charset, and is not a user−defined charset, the charset of the string will be set to **XmFONTLIST_DEFAULT_TAG**, and an informational message will be issued to the user to note that this substitution has been made.
- For a string declared as *"string"*, the character set depends on whether or not the module has a *character_set* clause and on whether or not the UIL compiler's *use_setlocale_flag* is set:
  - If the module has a *character_set* clause, the character set is the one specified in that clause.
  - If the module has no *character_set* clause but the **uil** command was invoked with the *-s* option, or the *Uil* function was invoked with the *use_setlocale_flag* set, UIL calls *setlocale* and parses the string in the current locale. The character set of the resulting string is **XmFONTLIST_DEFAULT_TAG**.
  - If the module has no *character_set* clause and the **uil** command was invoked without the *-s* option, or the *Uil* function was invoked without the *use_setlocale_flag*, the character set is the code set component of the **LANG** environment variable if it is set in the UIL compilation environment; if **LANG** is not set or has no code set, the character set is the value of **XmFALLBACK_CHARSET**.

Note that certain predefined escape sequences, beginning with a backslash (), may appear in string literals, with these exceptions:

- A string in single quotes can span multiple lines, with each newline escaped by a backslash. A string in double quotes cannot span multiple lines.
- Escape sequences are processed literally inside a string that is parsed in the current locale (a localized string).

# Specifying Compound Strings

A programmer may create a compound string with the **COMPOUND_STRING** or the **COMPOUND_STRING_COMPONENT** functions. The **COMPOUND_STRING** function takes as arguments a string expression and optional specifications of a character set, direction, and whether or not to append a separator to the string. If no character set or direction is specified, UIL derives it from the string expression, in the same manner as for string literals, described in the previous section.

In order to create a complex compound string in UIL, it is necessary to use the **COMPOUND_STRING_COMPONENT** function. Use it to create compound strings in UIL consisting of single components, in a manner analogous to **XmStringComponentCreate**. This function lets you create simple compound strings containing components such as **XmSTRING_COMPONENT_TAB** and **XmSTRING_COMPONENT_RENDITION_BEGIN** that are not produced by the **COMPOUND_STRING** function. These components can

then be concatenated to other compound strings to build more complex compound strings. (Use the UIL *&* operator to concatenate compound strings.)

The first argument of the **COMPOUND_STRING_COMPONENT** function must be an **XmStringComponentType** enumerated constant. The type and interpretation of the second argument depends on the first argument. For example, if you specify any of the following enumerated constants for the first argument, then you should not specify a second argument:

- **XmSTRING_COMPONENT_SEPARATOR**
- **XmSTRING_COMPONENT_LAYOUT_POP**
- **XmSTRING_COMPONENT_TAB**
- **XmSTRING_COMPONENT_LOCALE**

However, if you specify an enumerated constant from the following group, then you must supply a *string* as the second argument:

- **XmSTRING_COMPONENT_TAG**
- **XmSTRING_COMPONENT_TEXT**
- **XmSTRING_COMPONENT_LOCALE_TEXT**
- **XmSTRING_COMPONENT_WIDECHAR_TEXT**
- **XmSTRING_COMPONENT_RENDITION_BEGIN**
- **XmSTRING_COMPONENT_RENDITION_END**

If **XmSTRING_COMPONENT_DIRECTION** is the first argument, the second argument must be an **XmStringDirection** enumerated constant. Finally, if you specify **XmSTRING_COMPONENT_LAYOUT_PUSH** as the first argument, then you must specify an **XmDirection** enumerated constant as the second argument.

For more information on UIL string and compound string syntax, see the **UIL(5X)** reference page in the *Motif Programmer's Reference*.

# Render Tables and Tab Lists

Render tables, renditions, tab lists, and tab stops are implemented as a special class of objects, in a form analogous to the widget class. Though a render table is not itself a widget, its specification lists resemble one, and a widget that wants to use a render table must specify the render table in its *controls* list, not just with the **XmNrenderTable** resource.

In the same manner, a render table *controls* list contains the names of the renditions that make it up, a rendition *controls* list contains the names of the tab lists it uses, and the tab list *controls* list contains the names of the tab stops it uses. Although none of these objects are actual widgets, the format reads exactly as if the render table were a widget controlling a rendition widget, which in turn controlled a tab list widget, which in turn controlled a tab stop widget.

The following sample of UIL code shows a scrolled list widget that uses a render table.

```
object
  Scrolled_List:  XmScrolledList {
     arguments {
        XmNlistSpacing =  10;
        XmNlistMarginWidth =  10;
        XmNlistMarginHeight =  10;
        XmNitems =  string_table
                ("item1", "item2", "item3", "item4",
                "item5", "item6", "item7");
        XmNitemCount =  7;
        XmNvisibleItemCount =  4;
        XmNselectionPolicy = XmSINGLE_SELECT;
        XmNrenderTable = rt1;
     };
     callbacks {
        XmNsingleSelectionCallback = procedure Report_Callback
                        ('singleSelectionCallback');
     };
     controls {
        XmRenderTable rt1;
     };
   };
```

```
object
  rt1: XmRenderTable {
     controls {
        XmRendition rend;
     };
  };

object
  rend: XmRendition {
     arguments {
        XmNtag = XmFONTLIST_DEFAULT_TAG;
        XmNfontName = '9x15';
        XmNfontType = XmFONT_IS_FONT;
     };
     controls {
        XmTabList tabl;
     };
  };

object
  tabl: XmTabList {
     controls {
        XmTab tabstop;
     };
  };

object
  tabstop: XmTab {
     arguments {
        XmNtabValue = 1.0;
        XmNunitType = XmCENTIMETERS;
        XmNoffsetModel = XmABSOLUTE;
     };
  };
```

The render table (*rt1*) in this sample only contains one rendition, with the tag **XmFONTLIST_DEFAULT_TAG**. If the tag were not explicitly specified, its value would default to the name of the rendition object, in this case *rend*. The rendition in question uses resources to specify a font, and names a tab list (*tabl*), which in turn names its only member tab stop (*tabstop*). The tab stop again uses resources in its *arguments* list to specify its properties.