

BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-92-M2

“Class Library for the Automation of Motif (CLAM) Programmer's Manual”

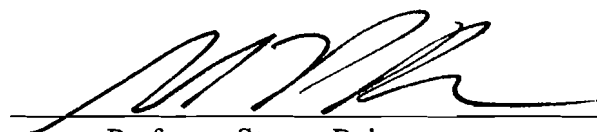
by
Peter Wagner

Class Library for the Automation of Motif (CLAM) Programmer's Manual

Peter Wagner
Masters Project
Brown University
Advisor: Professor Steven Reiss

January 25, 1991

This thesis by Peter Wagner is accepted in its present form by the Department of Computer Science
as satisfying the thesis requirement of the degree of Master of Science.



Professor Steven Reiss

2/21/92

Date

CLAM Programmer's Manual

Table of Contents

1. Introduction to CLAM

2. ClamPulldownMenu

- 2.1 Overview
- 2.2 Parameters
- 2.3 Menu Data Structure
- 2.4 Defining the Menu - the ClamMenu Data Array Format
 - 2.4.1 Submenus
- 2.5 Using ClamPulldownMenu
- 2.6 ClamPullDownMenu Methods
- 2.7 Using ClamPulldownMenu with Motif Code

3. ClamPopupMenu

- 3.1 Overview
- 3.2 Parameters
- 3.3 Using ClamPopupMenu
- 3.4 ClamPopupMenu Methods
- 3.5 Using ClamPopupMenu with Motif Code

4. ClamPanel

- 4.1 Overview
- 4.2 Parameters
- 4.3 ClamPanel Data Structure
- 4.4 Using ClamPanel
- 4.5 ClamPanel Methods
- 4.6 Using ClamPanel with Motif Code

5. ClamDialog

- 5.1 Overview
- 5.2 Parameters
- 5.3 Dialog Format Codes
- 5.4 Defining Dialogs
 - 5.4.1 Defining Widgets
 - 5.4.2 Widget Layout
 - 5.4.3 Region Layout
- 5.5 Special Features
- 5.6 Using ClamDialog
- 5.7 ClamDialog Methods
- 5.8 Using ClamDialog with Motif Code

6. BAUM

- 6.1 Overview
- 6.2 Using BAUM with CLAM

7. Callback Functions

7.1 Overview

7.2 CLAM Callbacks

7.3 ClamPulldownMenu & ClamPopupMenu Callback Types

7.4 ClamPanel Callback Types

7.5 ClamDialog Callbacks

8. Using Resource Files

8.1 Overview

8.2 Establishing a Resource File

8.3 Accessing a Widget's Resources

8.4 ClamPulldownMenu & ClamPopupMenu Resource Notes

8.5 ClamPanel Resource Notes

8.6 ClamDialog Resource Notes

9. Problems? - Common Pitfalls

10. Known Bugs

11. Future Enhancements

Appendix A - Example programs

1. Introduction to CLAM

The Class Library for the Automation of Motif is a set of classes that provides a simple way to create Motif user interfaces without learning Motif programming. The library contains classes which create pulldown menu systems, popup menus, panels (a panel is a window of buttons that resizes itself intelligently), widget layouts, and dialog boxes. Each class is instantiated with data which defines the interface component. Menus and panels are defined by an array of data structures, while dialogs are defined by a string containing special format codes.

CLAM was designed to allow the programmer who is not conversant in X Windows, Xt Intrinsics, or Motif to quickly and easily put together a graphical user interface. CLAM also provides the flexibility necessary to create intricate GUI components. Experienced Motif and Xt programmers will find CLAM useful in reducing the time spent developing the GUI component of a system.

While CLAM can be used without knowing how to write Motif and Xt Intrinsics programming, an understanding of Motif widgets and basic Xt concepts is important. In addition, a programmer using CLAM will need to use a few basic Xt Intrinsics functions. Use of these functions is described in context below and in the example programs.

Objects created with CLAM can easily be integrated into Xt and Motif code. Any CLAM object can be passed as a Widget to any function that takes a Widget parameter.

2. ClamPulldownMenu

2.1 Overview

A ClamPulldownMenu object provides the pulldown menu system for a Motif application. Typically a ClamPulldownMenu object will be installed as the menu bar in a Motif MainWindow widget.

ClamPulldownMenu methods provide an easy way to create and modify the menu bar, menus, and submenus.

2.2 Parameters

ClamPulldownMenu has two constructors (besides the copy constructor). The first takes no arguments and thus allows a default declaration of a ClamPulldownMenu object:

```
ClamPulldownMenu menu;
```

The second constructor takes four arguments:

```
ClamPulldownMenu(const char *name, const Widget parent, const void *data, const ClamMenuData  
menu[]).
```

The first argument is the name of the menu. This string will be the name of the top widget in the pulldown menu system. This name will be used to reference menu widgets when setting resources in a resource file.

The second argument is the Widget which is the parent of the menu system. In a typical scenario, this will be the MainWindow widget which serves as the main window of an application. Look at the example programs to see ClamPulldownMenu used in this fashion.

The third argument is the default data value which will be passed to the callback routines assigned to the menu buttons. This value will be overridden for a given menu button callback if a data value is provided for that button in the ClamMenuData record which defines the button (see below). The intention of this value is to provide an easy way to designate a common data value for all menu callbacks.

The fourth argument is the ClamMenuData array. This array defines the menu pads and their associated menus. This array contains an entry for every menu button and pad in the menu hierarchy. Note that while the ClamPulldownMenu class supports any number of levels of menus and submenus, submenu entries are not defined within the array passed to the ClamPulldownMenu constructor. Rather, an indication is made that a given menu entry is to have an associated submenu, and the submenu is attached to the menu after the ClamPulldownMenu is constructed (see ClamPulldownMenu::fillSubmenu()). The ClamMenuData structure is discussed in the next section.

2.3 Menu Data Structure

An array of ClamPulldownMenu structures is passed to the ClamPulldownMenu constructor as the specification for the menu pads and their associated menus. The same structure is also used to define the buttons in a submenu. An element in the array describes one menu button or menu pad. The ClamMenuData structure is:

```
typedef struct ClamMenuData {CLAM_MENU_STATE state;
                            char *name;
                            char *pixmap;
                            char *selectPix;
                            void (*routine)(Widget, void *, void *);
                            void *data}
```

The *state* entry is a bit field and describes several important parameters for a given menu button or pad. Various values are OR'd together to comprise the state field.

The low order bits indicate the type of the menu entry. One of the following values must be specified:

CLAM_PSTATE_END:	Indicates the last entry in an array of ClamMenuData.
CLAM_PSTATE_MENU:	The entry is for a menu pad and subsequent entries belong in this pad's menu.
CLAM_PSTATE_SEPARATOR:	The entry is a separator.
CLAM_PSTATE_BTN:	The entry is a menu button.
CLAM_PSTATE_SUBMENU:	The entry is a cascade button and will have a submenu.
CLAM_PSTATE_IGNORE:	The entry is to be ignored.
CLAM_PSTATE_TOGGLE:	The entry is a toggle button.

Only one of the above values should be OR'd into the state field. Behavior is undefined if more than one of the above values is used.

The middle order bits are optional and indicate several button options:

CLAM_PSTATE_DISABLE:	The entry is insensitive - grayed out.
CLAM_PSTATE_SELECT:	A toggle entry is indicated as set.
CLAM_PSTATE_DIAMOND_TOGGLE:	Toggle buttons are displayed as diamonds.
CLAM_PSTATE_INVISIBLE_TOGGLE:	No toggle indicator is displayed.

The high order bits are used for setting menu mnemonics, the keystroke alternative for clicking the menu button with the mouse. The following macros facilitate setting menu mnemonics:

CLAM_PSTATE_CHAR(c):	Specifies c as the mnemonic.
CLAM_PSTATE_META_CHAR(c):	Specifies Meta-c as the mnemonic.
CLAM_PSTATE_FCT_KEY(#):	Specifies Function key '#' as the mnemonic.
CLAM_PSTATE_SHIFT_KEY(c):	Specifies Shift-c as the mnemonic.
CLAM_PSTATE_CTRL_KEY(c):	Specifies Ctrl-c as the mnemonic.

One of the above macros should be used to set the mnemonic for a given menu option.

The *name* entry determines the label that will be displayed in the menu button and is the name of the widget to be referenced in a resource file.

The *pixmap* entry is the name of the pixmap that is to be displayed in the button. If a pixmap is not to be displayed, the *pixmap* entry must be set to 0.

The *selectPixmap* entry is only applicable to toggle buttons. It is the pixmap that is to be displayed when the toggle button is selected. If a select pixmap is not used, *selectPixmap* must be set to 0.

The *routine* entry is the callback routine that will be called when the user selects the menu entry.

The *data* entry is the data that will be passed to the above callback function. If this entry is zero, the default data passed to the ClamPulldownMenu constructor (or fillMenu() method) will be passed to the callback function.

2.4 Defining the Menu - the ClamMenu Data Array Format

The ClamMenuData array used to create a ClamPulldownMenu is a description of the menu pads and the buttons

in the menus that cascade from those pads. This array does not contain the definitions of any submenus that to not cascade directly from the menu pads. Submenus are added to the menu system via the method `ClamPulldownMenu::fillSubmenu()`.

Each element in the array has a certain designation which determines the function of that element. The designation is found in the *state* field of the `ClamMenuData` structure (see section 2.3 Menu Data Structure for a complete description of the possibilities for the state field). The array elements are arranged in the following general form (push buttons, separators, toggle buttons, and cascade/submenu buttons are interchangeable in the example below):

CLAM_PSTATE_MENU	1st Menu Pad
CLAM_PSTATE_BTN	1st Menu push button
CLAM_PSTATE_SEPARATOR	1st Menu separator
CLAM_PSTATE_BTN	1st Menu push button
...	...
CLAM_PSTATE_MENU	2nd Menu Pad
CLAM_PSTATE_TOGGLE	2nd Menu toggle button
CLAM_PSTATE_SUBMENU	2nd Menu cascade button
...	...
CLAM_PSTATE_MENU	3rd Menu Pad
...	...
CLAM_PSTATE_END	End of Menu Data

A menu pad will be created for each array element designated as `CLAM_PSTATE_MENU`. Each array element following an element designated as `CLAM_PSTATE_MENU` will create an entry in the menu triggered by that menu pad. A menu can have push buttons, toggle buttons, separators, and cascade buttons. Cascade buttons are created with the `CLAM_PSTATE_SUBMENU` designator. Cascade buttons look just like push buttons, however, they are set up to accept submenus (see section 2.4.1).

If a record designated as `CLAM_PSTATE_MENU` (a menu pad) has the name "Help", it will automatically be the rightmost menu pad and will be right justified.

2.4.1 Submenus

Creating menu systems with multiple levels of submenus is a two step process. First the toplevel menu, the menu pads and associated menus, is created. This step is accomplished by instantiating a `ClamPulldownMenu` with a `ClamMenuData` array or by the call to `ClamPulldownMenu::fillMenu()`. Next the method `ClamPulldownMenu::fillSubmenu()` is called:

```
fillSubmenu(const char *names[], const ClamMenuData menuData[], const void *data)
```

The first argument, *names*, is a NULL terminated array of strings. This array contains the series of cascade button names which leads to the cascade button to which the current submenu is to be attached. An example will make this description clearer. Suppose there is a **File** menu pad which brings up a menu with the choices **New**, **Open**, **Exit**.

```
File    Edit    Help
  •New
  •Open  <--- •Default
  •Exit      •Catalog
              •Other
```

We want to add a submenu to the **Open** button which contains the options **Default**, **Catalog**, **Other**. In this case, the names array would be {"File", "Open", NULL} - one has to select **File** then **Open** to get to the submenu which is being added. Note that the **Open** menu button to which the submenu is being attached must have been created with a designation of `CLAM_PSTATE_SUBMENU`.

The second argument, *menuData*, is a `ClamMenuData` array which defines the buttons in the submenu. The structure of this array is similar to the `ClamMenuData` array which defines the menubar and the top level menus. However, this array cannot contain any entries with a `CLAM_PSTATE_MENU` designation. *menuData* contains

any number of push buttons, separators, toggle buttons, and cascade/submenu buttons and is terminated by a CLAM_PSTATE_END record.

The last argument, *data*, is the default data value which will be passed to the callback routines assigned to the menu buttons in this submenu.

2.5 Using ClamPulldownMenu

Two steps are required before a menu system can be created using the ClamPulldownMenu class. First, the widget which is to contain the menu, its parent, must be created (in turn there are several things that must be done before the menu's parent widget is created - open the display, create the top level shell widget, etc. - see the example programs). This widget will probably be a Motif MainWindow widget. Second, an array of ClamMenuData structures must be filled with the appropriate data to create the desired menu configuration.

Once the above tasks are accomplished, the ClamPulldownMenu can be created. A ClamPulldownMenu is created in one of two ways, depending upon the method used to instantiate:

- 1) ClamPulldownMenu menu;
(or ClamPulldownMenu *menu = new ClamPulldownMenu;)
- 2) ClamPulldownMenu menu(name, parent, data, menuData);
(or ClamPulldownMenu *menu = new ClamPulldownMenu(name, parent, data, menuData);)

If the 2nd method is used, the menu system is installed upon instantiation. However, if the first method is used, it is still necessary to fill the ClamPulldown Menu. Use the method

ClamPulldownMenu::fillMenu()

This method takes exactly the same parameters as the non-default ClamPulldownMenu constructor. This two step method provides some flexibility in declaring ClamPulldownMenu variables.

To free memory used by a ClamPulldownMenu object, simply delete the object. Memory used by the widgets that comprise the menu system will be returned when the application destroys the widget that is the parent of the menu system.

2.6 ClamPullDownMenu Methods

Note: Many of the functions below take an argument which is an array of strings which indicates a menu entry or submenu on which to perform some action. For a description of how this string array is used, see section 2.4.1.

void **fillMenu**(const char *name, const Widget par, const void *data, const ClamMenuData menuData[])

Fills a menu that was instantiated without parameters. *name* is the name of the ClamPulldownMenu object (and is actually the name of the menubar Widget). *par* is the Widget that is the parent of the menu system. *data* is the default value to be passed as the *client_data* to the callback function that is called when a menu item is activated. *mData* is the array of data which specifies the structure of the menu system (see section 2.3).

int **fillSubmenu**(const char *names[], const ClamMenuData menuData[], const void *data)

Creates a submenu off of the cascade button designated by the array *names*. The *names* array is a NULL terminated array of strings which contains the series of cascade button names which leads to the cascade button to which the submenu is to be attached. *menuData* is the array which defines the buttons in the submenu. *data* is the default data value which will be passed to the callback routines assigned to the menu buttons in this submenu. Returns ERROR if the cascade button indicated by the *names* array does not exist, NO_ERROR otherwise.

int **enablePad**(const char *pad, const EnableDisable flag)

Based on the value of *flag*, Enables or disables the menu pad named *pad*. Returns ERROR if *pad* is not found, NO_ERROR otherwise.

int **enableEntry**(const char *entry[], const EnableDisable flag)

Based on the value of *flag*, Enables or disables the menu entry indicated by the NULL terminated string array *entry*. Returns ERROR if *entry* is not found, NO_ERROR otherwise.

int removePad(const char *pad)
Removes the menu pad named *pad*. Returns ERROR if *pad* is not found, NO_ERROR otherwise.

void removePads()
Removes all menu pads from the menu bar.

int removeEntry(const char *entry[])
Removes the menu entry specified by the string array *entry*. Returns ERROR if *entry* is not found, NO_ERROR otherwise.

int removeEntries(const char *submenu[])
Remove all entries from the menu indicated by the string array *submenu*. Returns ERROR if *submenu* is not found, NO_ERROR otherwise.

int setToggle(const char *entry[], const SelectDeselect flag)
Based on the value of *flag*, sets or unsets a the toggle button indicated by the *entry* array. Returns ERROR if *entry* is not found, NO_ERROR otherwise.

int setRadioBehavior(const char *submenu[], const Boolean flag)
Based on the value of *flag*, sets or unsets radio behavior for the menu indicated by the *submenu* array. A menu which has radio behavior set will enforce the rule that only one toggle button on the menu can be set at any given time. Returns ERROR if *submenu* is not found, NO_ERROR otherwise.

int getSize(const char *submenu[])
Returns the number of entries in the menu indicated by the *submenu* array. Returns ERROR if *submenu* is not found, NO_ERROR otherwise.

WidgetList getEntries(const char *submenu[])
Returns a WidgetList containing the Widget id's of all of the entries in the menu specified by *submenu*. Returns NULL if *submenu* is not found.

Widget getWidget(const char*entry[])
Returns the Widget pointer of the menu entry specified by the *entry* array. Returns NULL if *entry* is not found.

Widget getPad(const char*pad)
Returns the Widget pointer of the menu pad specified by *pad*.

void addPad(const ClamMenuData menuData[], const char *after, const void *data)
Adds a menu to the top level of the menu system. The *menuData* array defines the menu that is to be added. The first record in the *menuData* array must be designated CLAM_PSTATE_MENU. The name indicated in the first record will be the name of the new menu pad. The rest of the records follow the format outlined in section 2.4.1. The *after* parameter is the name of the existing menu pad after which the new pad is to be added. If *after* is NULL, the new menu pad is added first. *data* is the default value to be passed as the *client_data* to the callback function that is called when a menu item on the new menu is activated.

int addEntry(const ClamMenuData entryData, const char *sub[], const char *after, const void *data)
Adds an entry described by *entryData* to the menu specified by the *sub* array. The new entry is added after the entry specified by *after*. If *after* is not found, the entry is placed first. *data* is the default value to be passed as the *client_data* to the callback function for the new entry. Returns ERROR if *sub* is not found, NO_ERROR otherwise.

int removeCallbacks(const char *entry[])

Removes all callbacks from the menu entry specified by the *entry* array. Returns ERROR if *entry* is not found, NO_ERROR otherwise.

int removeCallback(const char *entry[], BaumCallbackProc routine, void *data)

Removes one specific callback from the menu entry specified by the *entry* array. This function will remove the callback which matches both the *routine* and the *data* parameters. Returns ERROR if *entry* is not found, NO_ERROR otherwise.

int addEntryCallback(const char *entry[], BaumCallbackProc routine, void *data)

Adds the callback function *routine* to the menu entry specified by the *entry* array. *data* is the value passed to the callback function as the *client_data*. Returns ERROR if entry is not found, NO_ERROR otherwise.

2.7 Using ClamPulldownMenu with Motif Code

A ClamPulldownMenu is a Motif RowColumn widget with the XmNrowColumnType resource set to XmMENU_PULLDOWN. A ClamPulldownMenu instance can be passed to any function as a Widget, as the operator () is defined to return a pointer to the underlying Motif widget. The widget returned by the () operator on a ClamPulldownMenu is the menubar widget of the menu system.

3. ClamPopupMenu

3.1 Overview

ClamPopupMenu provides a simple facility for creating popup menus in an application. A popup menu is a menu which is not always visible (unlike a pulldown menu), but is brought up by an event, usually a certain key sequence.

3.2 Parameters

ClamPopupMenu takes exactly the same parameters as ClamPulldownMenu. The only difference in the construction of a ClamPulldownMenu and a ClamPopupMenu is in the format of the ClamMenuData array. While in a ClamPulldownMenu the data array can have several records designated as CLAM_PSTATE_MENU, a ClamPopupMenu does not have any such records. The first record in the array is for the first button in the menu.

3.3 Using ClamPopupMenu

Again, look at ClamPulldownMenu for details on how to instantiate and fill a ClamPopupMenu. The methods for both are the same.

The key to using ClamPopups is to get them to pop up when desired. Popping ClamPopups is done by setting up a callback for the event which is to cause the menu to popup. Usually this event will be an input event in a work window. The example program demonstrates the use of an input callback in the main window work area to bring up a popup menu when the middle mouse button is pressed.

Within the callback function there are two ways to bring up a ClamPopupMenu. If the menu is already defined, it is only necessary to call the ClamPopupMenu::manage() method. This method will bring the menu up, and once the user makes a selection for the menu or clicks outside of the menu with the right mouse button, the menu will automatically pop down (be unmanaged()). In order to call the manage() method, it is necessary to have a pointer to the ClamPopupMenu object. This pointer should be passed to the callback function via the *client_data*. Alternatively, the ClamPopupMenu object can be declared so that it is in the scope of the callback function (i.e. global).

The first method of bringing up a ClamPopupMenu assumes that the menu has already been instantiated at the time of the event. It is also possible to instantiate and manage the ClamPopupMenu all within the callback function. However, for performance reasons, this method is not recommended.

To remove a ClamPopupMenu from the system, call ClamPopupMenu::destroy() (inherited from BaumRowColumnWidget) before deleting the ClamPopupMenu object.

3.4 ClamPopupMenu Methods

int **fillPopupMenu**(const char *, const Widget, const void *, const ClamMenuData [], const int)

See ClamPulldownMenu::fillMenu.

int **addPopupEntry**(const ClamMenuData menuEntry, const void *data, const char *after)

Adds a new entry to the popup menu. Entry is defined by data in *menuEntry*. *data* is the default value to be passed as the *client_data* to the callback function for the new entry. *after* is the name of the menu button after which the new entry is to be added. If *after* is not found, the entry is placed first, and the function returns ERROR, otherwise the function returns NO_ERROR.

int **enableEntry**(const char *entry, const EnableDisable flag)

Based on the value of *flag*, Enables or disables the menu entry named *entry*. Returns ERROR if *entry* is not found, NO_ERROR otherwise.

int **setToggle**(const char *entry, const SelectDeselect flag)

Based on the value of *flag*, sets or unsets a the toggle button *entry*. Returns ERROR if *entry* is not found, NO_ERROR otherwise.

int **removeEntry**(const char *entry)

Removes the menu entry *entry*. Returns ERROR if *entry* is not found, NO_ERROR otherwise.

int **fillSubmenu**(const char *entry, const ClamMenuData menuData[], const void *data)

Creates a submenu off of the cascade button designated by *entry*. *data* is the default data value which will be passed to the callback routines assigned to the menu buttons in this submenu. Returns ERROR if the cascade button *entry* does not exist, NO_ERROR otherwise.

Widget **getWidget**(const char *entry)

Returns the Widget id of the menu entry specified by *entry*. Returns NULL if *entry* is not found.

int **removeEntries**()

Removes all entries from the menu.

int **removeCallbacks**(const char *entry)

Removes all callbacks from the menu entry specified by *entry*. Returns ERROR if *entry* is not found, NO_ERROR otherwise.

int **removeCallback**(const char *entry, BaumCallbackProc routine, void *data);

Removes one specific callback from the menu entry specified by *entry*. This function will remove the callback which matches both the *routine* and the *data* parameters. Returns ERROR if *entry* is not found, NO_ERROR otherwise.

int **addEntryCallback**(const char *entry, BaumCallbackProc routine, void *data)

Adds the callback function *routine* to the menu entry specified by *entry*. *data* is the value passed to the callback function as the *client_data*. Returns ERROR if *entry* is not found, NO_ERROR otherwise.

int **getSize**()

Returns the number of entries in the menu.

WidgetList **getEntries**()

Returns a WidgetList containing the Widget id's of all of the entries in the menu.

3.5 Using ClamPopupMenu with Motif Code

A ClamPopupMenu is a Motif RowColumn widget with the XmNrowColumnType resource set to XmMENU_POPUP. A ClamPopupMenu instance can be passed to any function as a Widget, as the operator () is defined to return a pointer to the underlying Motif widget. The widget returned by the () operator on a

ClamPopupMenu is the RowColumn widget that contains the menu options.

4. ClamPanel

4.1 Overview

A ClamPanel is a window of buttons that resizes itself “intelligently”, i.e. it chooses the best arrangement of buttons based on the size of the buttons and the size of the window. Any number of push and toggle buttons can comprise a ClamPanel. When a ClamPanel is initially realized, all buttons assume the size of the largest button within the panel. When a ClamPanel is resized, the buttons will shrink if necessary and will expand either 1) back to the original size but no larger or 2) beyond the original size so that all buttons are always touching. The button expansion behavior is set when the ClamPanel is created. It is also possible for the programmer to set the preferred button size.

4.2 Parameters

ClamPanel has two constructors (besides the copy constructor). The first takes no arguments and thus allows a default declaration of a ClamPanel object:

```
ClamPanel menu;
```

The second constructor takes six arguments:

```
ClamPanel(const char *, const Widget, const ClamPanelBtn [], const void *,  
          const CLAM_PANEL_OPTIONS, const ClamShellType);
```

The first argument is the name of the panel. This string will be the name of the form widget that manages the buttons. This name will be used to reference menu widgets when setting resources in a resource file.

The second argument is the widget which is the parent of the ClamPanel. In a typical scenario, this will probably be the Main Window widget which serves as the main window of the application or the Top Level Shell widget which is the parent of the Main Window widget. The example programs demonstrate a ClamPanel parented in this fashion.

The third argument is the ClamPanelBtn array. This array defines the panel buttons and contains an entry for every button in the panel. The data structure ClamPanelBtn is described in detail in the next section.

The fourth argument is the default data value which will be passed to the callback routines assigned to the panel buttons. This value will be overridden for a given panel button callback if a data value is provided for that button in the ClamPanelBtn record which defines the button (see below). The intention of this value is to provide an easy way to designate a common data value for all panel button callbacks.

The fifth argument is a bit field that holds the values for several ClamPanel parameters that affect the behavior/look of all the buttons in the panel. Any or all of the below values can be OR'd together in this bit field:

const int CLAM_PANEL_PUSH = 1;	Indicates that all buttons are push buttons
const int CLAM_PANEL_TOGGLE = 2;	Indicates that all buttons are toggle buttons
const int CLAM_PANEL_BUTTON_BORDER = 4;	Indicates that a border is to surround all buttons
const int CLAM_PANEL_BUTTON_NOESIZE = 8;	Indicates that buttons are not to expand beyond original size

The sixth argument indicates the type of shell that is to hold the ClamPanel. The options are:

```
SHELL_DIALOG  
SHELL_TOP_LEVEL  
SHELL_APPLICATION  
SHELL_NONE
```

The choice of shell is a subtle issue. If a title is not required on the ClamPanel window, choose SHELL_DIALOG. If a title is required, one of the other shell types if appropriate. Consult a manual which describes shell widget types for help in making this decision.

4.3 ClamPanel Data Structure

The ClamPanelBtn structure is the same as the ClamMenuData structure only without the *state* field:

```
typedef struct ClamPanelBtn {char *name;
                           char *pixmap;
                           char *selectPixmap;
                           void (*routine)(Widget, void *, void *);
                           void *data;}
```

The *name* entry is the name for the push or toggle button widget. This name will be used when referencing a particular button in a resource file. *name* is also the text that will appear in the button if a pixmap is not specified. Setting *name* to 0 serves to terminate the ClamPanelBtn array. For example, a panel that contains four buttons will have a ClamPanelBtn array in which the fifth record has name set to 0, thereby indicating to ClamPanel that the button array ends after the fourth button.

The *pixmap* entry is the name of the pixmap that is to be displayed in the button. If a pixmap is not to be displayed, the *pixmap* entry must be set to 0.

The *selectPixmap* entry is only applicable to toggle buttons. It is the pixmap that is to be displayed when the toggle button is selected. If a select pixmap is not used, *selectPixmap* must be set to 0.

The *routine* entry is the callback routine that will be called when the user presses the button.

The *data* entry is the data that will be passed to the above callback function. If this entry is zero, the default data passed to the ClamPanel constructor (or fillPanel method) will be passed to the callback function.

4.4 Using ClamPanel

Two steps are required before a ClamPanel can be created. First, the widget which is to be the parent of the ClamPanel must be created (in turn there are several things that must be done before the menu's parent widget is created - open the display, create the top level shell widget, etc. - see the example programs). This widget will probably be a Motif MainWindow widget or a Top Level Shell widget. Second, an array of ClamPanelBtn structures must be filled with the appropriate data to create the desired ClamPanel.

Once the above tasks are accomplished, the ClamPanel can be created. A ClamPanel is created in one of two ways, depending on the method used for instantiation:

- 1) ClamPanel panel;
(or ClamPanel *panel = new ClamPanel;)
- 2) ClamPanel panel(name, parent, panelData, data, options, shellType);
(or ClamPanel *panel = new ClamPanel(name, parent, panelData, data, options, shellType);)

Note that it is not necessary to specify a shellType parameter. If no shellType is specified, the shell type will default to SHELL_DIALOG.

If the 2nd method is used, the ClamPanel is created upon instantiation. However, if the first method is used, it is still necessary to fill the ClamPanel. Use the method

```
ClamPanel::fillPanel()
```

This method takes exactly the same parameters as the non-default ClamPanel constructor. This two step method allows some flexibility in declaring ClamPanel variables.

Once a panel has been created, it is still necessary to pop it up in order to make it visible. Popping up a panel is accomplished by calling the method ClamPanel::popUp(). Call ClamPanel::popDown to make the panel invisible. Note that calling ClamPanel::popDown does not destroy the panel - ClamPanel::popUp() can be called to make it visible again.

To remove a ClamPanel from the system, call ClamPanel::destroy() (inherited from BaumFormWidget) before deleting the ClamPanel object.

4.5 ClamPanel Methods

void **fillPanel** - see sections 4.2 and 4.4.

void **setButtonSize**(const int width, const int height)

Only applies if panel is designated PANEL_BUTTON_NORESIZE - Sets the button size for all buttons in the panel. The size set is the maximum size for the buttons. This size will only be realized at the time of the call if the panel is large enough to accomodate all of the buttons at this size. Otherwise, this size will be applied as the maximum size that the buttons can become when the panel is enlarged.

void **size**(int width, int height)

Sets the size of the panel to *width x height*.

void **setPlace**(const int x, const int y)

Positions the panel at the coordinates (x, y), relative to the upper left corner of the screen.

void **getPlace**(int& x, int& y)

Returns the position of the panel in the parameters x and y.

void **addButton**(const ClamPanelBtn btn, char *name, void *data, const CLAM_PANEL_OPTIONS opts)

Adds a button to the panel. *name* is the name of the new button (to be referenced in the resource file). *btn* defines the new button. *data* is the value that will be passed to the button's callback routine as the client_data. *opts* determine whether the button is a push or toggle button and whether or not the new button has a border. Note that by varying *opts* it is possible to have different button types within a panel. It is not possible to create different button types without using this function.

Widget **getButton**(const char *name)

Returns the Widget ID of the button named *name*.

int **removeButton**(const char *)

Removes the button named *name* from the panel. Returns ERROR if the button named *name* is not found, NO_ERROR otherwise.

int **setToggle**(const char *name, const SelectDeselect flag)

Depending on the value of *flag*, sets or unsets the toggle button named *name*. Returns ERROR if the button named *name* is not found, NO_ERROR otherwise.

int **resetPanel**(const ClamPanelBtn panelData[], const void *data, const CLAM_PANEL_OPTIONS opts)

Removes all buttons in the panel and creates new buttons based on the data in *panelData*. *data* and *opts* are used as in the constructor and fillMenu() functions. Returns ERROR if the panel has been instantiated but not yet filled, NO_ERROR otherwise.

void **popUp**()

Pops the panel up - makes it visible.

void **popDown**()

Pops the panel down - makes it invisible.

4.6 Using ClamPanel with Motif Code

A ClamPanel is a Motif Form widget containing a set of buttons. A ClamPanel instance can be passed to any function as a Widget, as the operator () is defined to return a pointer to the underlying Motif widget. The widget returned by the () operator on a ClamPanel is the Form widget that contains the buttons. The shell widget that contains the form can be determined by calling XtParent() on the ClamPanel instance (or by calling <ClamPanel instance>.parent() - see section 6, Baum, for an explanation of Baum methods available for Clam objects).

5. ClamDialog

5.1 Overview

ClamDialog provides a formatted string interface for widget layout. A ClamDialog is created with a string that contains special codes and data which specify which widgets to create and where. ClamDialog has been designed to provide the flexibility to produce virtually any dialog box. In addition, ClamDialog can be used to lay out widgets within any Motif manager widget by specifying that the dialog is not to create its own shell (shellType of SHELL_NONE) but is to lay out the widgets within the parent widget.

Note that in the sections below, the terms *widget* and *region* are used to indicate different entities. A region is an area that contains widgets. A widget is any one of the various GUI components that CLAM supports. However, this difference is only conceptual. A region, technically speaking, is a Motif Form widget. The discussions below, however, assume that regions and widgets are different types of objects.

5.2 Parameters

ClamDialog has three constructors (besides the copy constructor). The first takes no arguments and thus allows a default declaration of a ClamDialog object:

```
ClamDialog dialog;
```

The other two constructors take nine arguments:

```
ClamDialog(const char *, const Widget, const char *, const short, const ClamShellType, const Boolean, const Boolean, const XtCallbackProc, const void *)
```

```
ClamDialog(const char *, const Widget, const char **, const short, const ClamShellType, const Boolean, const Boolean, const XtCallbackProc, const void *)
```

The only difference between the two constructors is in the third argument, `char *` vs. `char **`.

The first argument is the name of the dialog. This string will be the name of the shell widget that contains the dialog and also the name of the outermost form widget in the dialog. This name will be used to reference dialog widgets when setting resources in a resource file.

The second argument is the Widget which is the parent of the dialog system. In a typical scenario, this will be the MainWindow widget which serves as the main window of an application. However, just about any widget can serve as the parent of a dialog. Look at the example programs to see a couple of ways in which dialogs can be parented.

The third argument is the format string which defines the widgets of the dialog. Note that this variable can be either a `char *` or a `char **`. ClamDialog assumes that a `char **` format string is a NULL terminated array of strings. If a `char **` format string is passed to ClamDialog, the string array is simply concatenated into a single `char *` before the format string is parsed. Thus the string can be broken up into any set of strings that makes sense to the programmer.

See section 5.3 for a detailed description of the format codes.

The fourth argument is the modality of the dialog. This setting affects whether other windows are active while the dialog is up. Modality can be set to

```
XmDIALOG_MODELESS
XmDIALOG_PRIMARY_APPLICATION_MODAL
XmDIALOG_FULL_APPLICATION_MODAL
XmDIALOG_SYSTEM_MODAL
```

See a Motif manual for a description of these settings.

The fifth argument is the shell type for the dialog. This setting determines the type of shell that holds the dialog. Shell type settings can be

SHELL_DIALOG	creates DialogShell
SHELL_TOP_LEVEL	creates TopLevelShell
SHELL_APPLICATION	creates ApplicationShell
SHELL_NONE	no shell created, widgets laid out in parent window

See a Motif manual for a description of these shell types. SHELL_NONE allows ClamDialog to be used to lay out

application windows that are not dialog boxes.

The sixth argument is a Boolean which determines whether the dialog follows its parent window when the parent window is moved.

The seventh argument is a Boolean which determines whether the dialog is raised when its parent is raised.

The eighth argument is the callback routine that will be called whenever the user changes something in the dialog. This callback will be passed the last argument (see below) as the *client_data* *except* when the widget is an Option Menu. In the case of Option Menus, the *client_data* to the callback is the button number of the button selected from the menu. To identify which widget the user has changed and thus determine the appropriate callback action, the programmer must check the name of the widget passed to the callback function. The widget's name is determined by calling the Xt function `XtName(Widget w)`. See the example programs for details on how this is function is used in this context.

The last argument is the data that will be passed to the dialog's callback routine as the *client_data*.

5.3 Dialog Format Codes

A ClamDialog is defined by a string which contains special codes which designate widget types and parameters. All codes begin with a '%'. Widget and region codes are capital letters (except for the separator widget which has a code of %-) Widget and region parameter codes are all lower case letters.

Note that ClamDialogs are region based. That is, the composition of a dialog is broken up into regions to simplify widget layout (see section 5.4.3). A ClamDialog is composed of one or more regions. Regions can nest, thus every region that is specified in the dialog string must have a matching end region marker. The format string must begin with a '%R' and end with a '%E'.

5.3.1 Widget & Region Codes

%R	Start Region
%E	End Region
%L	Label
%P	Push Button
%G	Toggle Button
%D	Drawn Button
%O	Option Menu
%S	Scrolled List
%T	Text
%F	Text Field
%C	Scale
%H	Separator
%A	Drawing Area
%M	Command Area
%B	Scroll Bar
%W	Arrow Button
%U	Cascade Button
%I	File Selection Box

5.3.2 Codes for All Widget Types

The following can be specified for all widgets:

%p	position - followed by 0-4, if %p not indicated position = 0 (see section 5.4) 0: BELOW 1: FIRST 2: TO_THE_RIGHT 3: WITHIN_BELOW 4: WITHIN_TO_THE_RIGHT
%h	horizontal tabs - followed by # (see section 5.5)

%v vertical tabs - followed by # (see section 5.5)
 %n name - followed by string, if not present widget is given name of "NoName"
 %s stretch horizontally - if indicated stretch to the right, else don't stretch (see section 5.5)
 %j stretch vertically - if indicated stretch the bottom, else don't stretch (see section 5.5)
 %f don't attach left side of widget - if indicated override normal left attachment (see section 5.5)
 %k don't attach top of widget - if indicated override normal top attachment (see section 5.5)
 %< attach left widget's right side - if indicated (see section 5.5)
 %^ attach top widget's bottom side - if indicated (see section 5.5)
 %z make widget insensitive - if indicated widget is insensitive, else widget is sensitive

5.3.3 Codes for Region (%R)

%b border type - followed by 0-4, if %b not indicated border type = 0
 These correspond to the Motif *borderType* resource for Frame widgets
 0: XmNO_LINE
 1: XmSHADOW_IN
 2: XmSHADOW_OUT
 3: XmSHADOW_ETCHED_IN
 4: XmSHADOW_ETCHED_OUT

 %r radioBehavior - if indicated yes, else no (see section 5.5)
 %m menu - if indicated, create a RowColumn manager for pulldown menus (see section 5.5)
 %e spread - if indicated yes, else no (see section 5.5)
 %c number of Columns for RowColumn region - followed by #, 1 if not indicated.
 %t type - if indicated XmPanedWindow, XmForm if not indicated (default) or XmRowColumn if
 other codes requiring XmRowColumn manager are present (%r, %m). Note that topmost
 region cannot be an XmPanedWindow. Enclose entire dialog in an additional region and make
 the second region an XmPanedWindow to get desired behavior (see section 5.5).
 %q packing - 0-3, if not indicated XmNO_PACKING (see section 5.5)
 These correspond to the Motif *packing* resource for RowColumn widgets
 0: XmNO_PACKING
 1: XmPACK_COLUM
 2: XmPACK_TIGHT
 3: XmPACK_NONE

5.3.4 Codes for Label and its Descendants (Push Button, Toggle Button, Cascade Button, Option Menu)

These are for widgets that have a label:

%l label string - followed by a string
 %a alignment - followed by 0-2, if not indicated alignment = 0
 These correspond to the Motif *alignment* resource for Label widgets
 0: XmALIGNMENT_BEGINNING
 1: XmALIGNMENT_MIDDLE
 2: XmALIGNMENT_END

 %x pix map name - followed by a string which is the name of the file containing the pix map to be
 displayed in the label.

5.3.5 Codes for Label Only

%d default - if indicated the button will be shown as the default

5.3.6 Codes for Toggle Button Only

%y set - if indicated the button will be set when created
 %m select pix map name - followed by a string which is the name of the file containing the pix map
 pixmap to displayed when the toggle button is set

5.3.7 Codes for Cascade Button Only

None

5.3.8 Codes for Drawn Button

%w width - followed by a number which is the width in pixels
%g height - followed by a number which is the height in pixels

5.3.9 Codes for Option Menu

%o options - followed by a string of options, comma delimited (Ex. %oone,two,three)

5.3.10 Codes for Scrolled List

%i items - followed by a string of items, comma delimited (Ex. %ione,two,three)
%u number of items in list - followed by #
%b number of visible items - followed by #
%e selection policy - followed by 0-3, if %e not indicated policy = Xm_SINGLE_SELECT
These correspond to the Motif *selectionPolicy* resource for List widgets
0: XmSINGLE_SELECT
1: XmBROWSE_SELECT
2: XmMULTIPLE_SELECT
3: XmEXTENDED_SELECT

5.3.11 Codes for Text and TextField

%l contents - followed by string to be displayed in text widget
%c number of columns - followed by # (width in characters of text widget)
%r number of rows - followed by #, 1 if not indicated
%b scroll bars - display scroll bars if this option indicated

5.3.12 Codes for Scale

%m minimum value on scale - followed by #, 0 if not specified
%u maximum value on scale - followed by #, 100 if not specified
%i initial value - followed by #, minimum if not specified
%o orientation - vertical if specified, horizontal by default
%w show value - show value if specified, don't show value if not specified

5.3.13 Codes for Separator

%o orientation - vertical if specified, horizontal by default
%t type - followed by 0-6, if not indicated type = XmSINGLE_LINE
These correspond to the Motif *separatorType* resource for Separator widgets
0: XmSHADOW_ETCHED_IN
1: XmSHADOW_ETCHED_OUT
2: XmSINGLE_LINE
3: XmDOUBLE_LINE
4: XmSINGLE_DASHED_LINE
5: XmDOUBLE_DASHED_LINE
6: XmNO_LINE

%g length in tabs - followed by #, if not specified separator is stretched across its parent form
%w width in pixels - followed by #, if the separator doesn't show up, this parameter may not have been specified.

5.3.14 Codes for Drawing Area

%w width in pixels - followed by #
%g height in pixels - followed by #

5.3.15 Codes for Command Area

%w width in pixels - followed by #
%g height in pixels - followed by #

%x max # of history items - followed by #
 %i # history items visible - followed by #
 %m prompt - followed by string, no prompt if not specified

5.3.16 Codes for ScrollBar

%w width in pixels - followed by #
 %g height in pixels - followed by #
 %o orientation - vertical if specified, horizontal by default

5.3.17 Codes for Arrow Button

%d direction - followed by 0-3, if not indicated direction = XmARROW_UP
 These correspond to the Motif *arrowDirection* resource for ArrowButton widgets
 0: XmARROW_UP
 1: XmARROW_DOWN
 2: XmARROW_LEFT
 3: XmARROW_RIGHT

5.3.18 Codes for File Selection Box

None

5.3.19 Special Codes

%% '%' in string
 %, ',' in item within list
 %. ends a label or list (so there can be white space following)

Use '%%' to indicate a label that contains a percent sign.

Use '%, ' to indicate a comma within an item within a list.

Use '%.' to terminate a label that is followed by white space.

5.4 Defining Dialogs

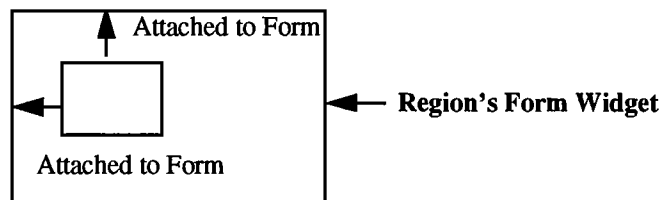
5.4.1 Defining Widgets

Defining individual widgets is simple - just follow the codes outlined in section 5.3. More involved is the process of laying out widgets and regions.

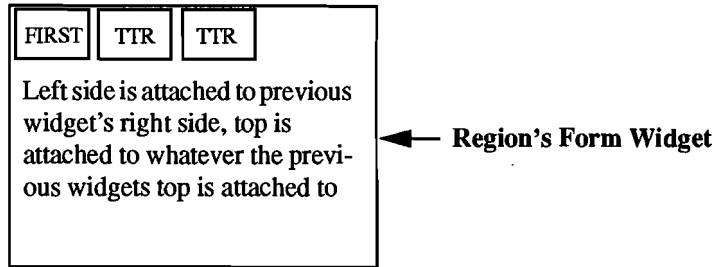
5.4.2 Widget Layout

ClamDialog uses a very simple method of laying out widgets. A widget can be placed either FIRST in its region (need not be specified, the parser figures out whether a widget or region is FIRST), TO_THE_RIGHT of the previous widget, or BELOW the widgets in the current row of widgets.

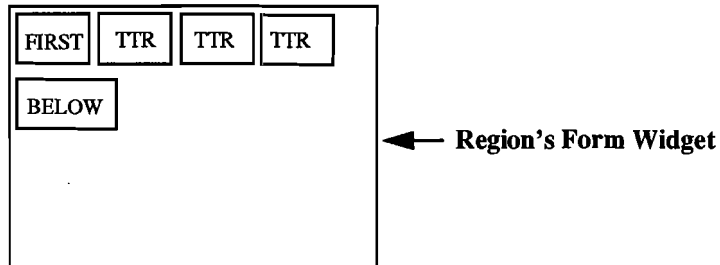
FIRST



TO_THE_RIGHT



BELOW



A widget placed BELOW is attached on the top to the first widget in the previous row and on the left to whatever the first widget in the previous row is attached to on the left.

This simple layout scheme does not in itself provide the mechanism whereby any conceivable dialog can be formed. However, by breaking dialogs down into simple regions, and placing the regions with a mechanism similar to that used for laying out widgets (the same three options plus two options unique to regions, WITHIN_BELOW and WITHIN_TO_THE_RIGHT), virtually any dialog layout can be achieved.

5.4.3 Region Layout

As indicated above, regions are laid out according to rules that are very similar to those used for widgets. Three options are the same - FIRST, BELOW, and TO_THE_RIGHT. Note that when these options are used, regions are placed relative to other regions, not relative to individual widgets.

The two options specific to regions are WITHIN_BELOW and WITHIN_TO_THE_RIGHT. These options take care of situations where a region is to be placed relative to a widget, not relative to another region. The rules for these options are identical to their widget counterparts explained in section 5.4.2.

Most complex dialogs can be broken up into regions in a variety of ways. Usually a solution is fairly straightforward. The example programs demonstrate several dialog layouts.

A dialog format string *always* starts with a %R and ends with a %E. In addition, special region formatting codes (%s, %e, %j) cannot be used on the top level region. If these options are needed on the top level, enclose the entire dialog string in another region. ("%R%R%s%e%j ... %E%E").

5.5 Special Features

There are a number of format codes that affect the layout of widgets and regions. There are also some global constants that can determine widget spacing. Below is a more detailed explanation of these flags than is found in section 5.3, and an explanation of the format constants.

5.5.1 Special Codes for All Widget Types

Non-attachment codes %f, %k: These codes indicate that the normal left (%f) and top (%k) attachments are not to be performed. This allows a widget's placement to be completely under the control of the resource file. It is also common for these codes to be used in conjunction with the stretch codes %s and %j, and the special attachment codes %^ and %< (see below), if a widget is to move instead of stretch when a window is resized.

Attachment codes %s, %j: (Note that this applies to both Regions and Widgets) By default, a widget's left and top sides are attached to the widget(s) it is adjacent to. The right and bottom sides are not attached to anything. If %s is specified, the widget's right side is attached to the Form widget that is the widget's parent. %j indicates that the widget's bottom side is to be attached to the Form widget that is the widget's parent. These parameters greatly facilitate lining up widgets and sensible resize behavior. Note that if a widget is not stretching to the extent that is expected, it's parent probably needs to be stretched itself. Check to see that the parent Form extends where expected. It is often necessary to specify these parameters for both the widget and its parent.

Attachment codes %^, %<: These codes can be helpful when designing a format which is to resize correctly when its shell is resized. If the current widget is to remain the same size, but the adjacent widget(s) is/are to respond to a shell resize, a combination of these attachment codes with the non-attachment codes can be used to specify this behavior. This situation can occur when the widget to be resized is not adjacent to the edge of the form, and the widget which is to remain the same size is.

%^ indicates that the bottom side of the widget above the current widget is to be attached to the top of the current widget. This flag is used in conjunction with the non-attachment code %k, as it doesn't make sense to attach the top of one widget to the bottom of another widget and vice versa.

%< indicates that the right side of the widget to the left of the current widget is to be attached to the left of the current widget. This flag is used in conjunction with the non-attachment flag %f.

Pads: There are several global constants that can be used to space widgets uniformly. These constants and their defaults are:

```
const short DIALOG_VERTICAL_PAD = 0;
const short DIALOG_HORIZONTAL_PAD = 0;
const short DIALOG_REGION_HORIZONTAL_PAD = 0;
const short DIALOG_REGION_VERTICAL_PAD = 0;
```

When set to 0, these constants have no effect. When > 0, these constants determine the left and top offsets for widgets and regions, and the offsets cannot be overridden by resource file settings. The first two constants apply only to widgets, while the last two apply only to regions. By setting these greater than 0, one can achieve uniform spacing throughout the dialog. However, the flexibility provided by resource file settings is lost.

Tabs %h, %v: A widget or region does not have to be placed relative to its neighbors. An absolute position can be set by using tabs. Both horizontal and vertical tabs can be set. The following constants determine the tab sizes:

```
const short DIALOG_VERTICAL_TAB = 5;
const short DIALOG_HORIZONTAL_TAB = 5;
const short DIALOG_REGION_VERTICAL_TAB = 5;
const short DIALOG_REGION_HORIZONTAL_TAB = 5;
```

The first two constants apply only to widgets, while the the last two apply only to regions. The smaller the tab sizes, the more exactly one can place widgets and regions.

5.5.2 Special Codes for Regions

Radio Behavior %r, %q: The %r flag automatically ensures that only one toggle button can be set in a region. The %q flag can be used in conjunction with the %r flag. When the %r flag is specified, the region's manager widget is a RowColumn widget instead of a Form widget. The %q flag indicates what sort of packing (see RowColumn *packing* resource) the RowColumn manager should use.

Menu Manager %m: If cascade buttons are to be used within the dialog for bringing up menus, the region that contains the cascade button must be of type RowColumn and the *rowColumnType* resource must be set to XmMENU_PULLDOWN. Specifying %m for the region causes a RowColumn manager of XmMENU_PULLDOWN type to be created.

Spread %e: The spread flag %e indicates that the widgets within the region are to be arranged at even intervals accross the region. This flag is usually used in conjunction with the %s flag to stretch the region

across a certain area and to have the widgets within the region arranged at even intervals across the region. Only Label, Push Button, and Toggle Button widgets can be used inside a region with a spread designation. Behavior is undefined if other widget types are included. Note that the constant

```
const short TIGHTNESS = 10;
```

determines how tightly the widgets are arranged. The greater the value of TIGHTNESS, the closer together the widgets are placed.

Paned Window Manager %t: %t indicates the type of window manager to be used for a region. By default, a Form widget is used as the manager. The %r and %m flags cause a RowColumn widget to be created as the region manager. If the %t flag is present, the region manager created will be a PanedWindow widget. This flag will most often be used for the main region of an application to allow resizing of regions via resize sashes. Note that %t is incompatible with both %r and %m, and will override the expected behavior specified by %r and %m.

5.6 Using ClamDialog

Before a dialog can be created, the widget which is to be the parent of the dialog, must be created (in turn there are several things that must be done before the dialog's parent widget is created - open the display, create the top level shell widget, etc. - see the example programs). Once the widget which is to be the dialog's parent exists, the ClamDialog object can be created. A ClamDialog can be created in one of two ways, depending upon the method used to instantiate:

- 1) ClamDialog dialog;
(or ClamDialog *dialog = new ClamDialog;)
- 2) ClamDialog dialog(name, parent, dialogString, modality, shellType, follow, raise, routine);
(or ClamDialog *dialog = new ClamDialog(name, parent, dialogString, modality, shellType, follow, raise, routine);)

If the 2nd method is used, the dialog is created upon instantiation. However, if the first method is used, it is still necessary to fill the ClamDialog. Use the method

```
ClamDialog::fillDialog()
```

This method takes exactly the same parameters as the non-default ClamDialog constructor. This two step method allows some flexibility in declaring ClamDialog variables. Note that fillDialog will also accept a char * or a char ** format string.

Once a dialog has been created, it is still necessary to pop it up in order to make it visible, unless it was created with a shell type of SHELL_NONE (in which case it will come up as soon as it is filled). Popping up a dialog is accomplished by calling the method ClamDialog::popUp(). Call ClamDialog::popDown to make the dialog invisible. Note that calling ClamDialog::popDown does not destroy the dialog - ClamDialog::popUp() can be called to make it visible again. Note that calling ClamDialog::popUp() or ClamDialog::popDown() on a dialog with shell type of SHELL_NONE results in a fatal error.

To remove a ClamDialog from the system, call ClamDialog::destroy() (inherited from BaumFormWidget) before deleting the ClamDialog object.

5.7 ClamDialog Methods

int **fillDialog()** - see sections 5.2 & 5.5.

Widget **getWidget(char *name)**

Returns the widget ID of the widget within the dialog named *name*. Returns NULL if named widget is not found.

int **removeWidget(char *name)**

Removes the widget named *name* from the dialog. Returns ERROR if named widget is not found, NO_ERROR otherwise.

int **sensitizeWidget(char *name, const Boolean flag)**

Sensitized/desensitizes widget named *name* based on the values of *flag*. Returns ERROR if named widget is not found, NO_ERROR otherwise.

int **addWidgetCallback**(char *name, const XtCallbackProc routine, void *data, char *callbackType)

Adds *routine* as a callback to the widget named *name*. *data* is the value passed as the client_data to the callback. *callbackType* determines the type of callback added to the widget (“valueChangedCallback”, “activateCallback”, etc.). Returns ERROR if named widget is not found, NO_ERROR otherwise.

int **removeWidgetCallback**(char *name, const XtCallbackProc routine, void *data, char*callbackType)

Removes the callback matching *routine*, *data*, and *callbackType*. Returns ERROR if named widget is not found, NO_ERROR otherwise.

int **removeWidgetCallbacks**(char *name, char *callbackType)

Removes all callbacks of type *callbackType* (“activateCallback”, “valueChangedCallback”, etc.) for named widget. Returns ERROR if named widget is not found, NO_ERROR otherwise.

void **position**(int x, int y)

Positions the upper left corner of the dialog box at coordinates (x, y).

void **popUp**(XtGrabKind grab)

Pops up the dialog. *grab* is usually XtGrabNone, but can also be XtGrabNonexclusive or XtGrabExclusive. This setting affects the modality of the dialog. See an Xt Ininsics manual for more information on XtGrabKind.

void **popDown**()

Pops down the dialog.

5.8 Using ClamDialog with Motif Code

A ClamDialog is a Motif DialogShell, ApplicationShell, TopLevelShell, or manager widget (depending on shell type parameter) which contains first a Form widget and then some number of other widgets depending on the dialog definition. Every region in a ClamDialog corresponds to a Form, PanedWindow, or RowColumn widget. If a border has been specified for a region, that region will be contained in a Frame widget which has the same name as the region. The individual items within a ClamDialog are each separate widgets which are attached to each other or to their enclosing form. Use getWidget() to get the widget ID of an item within a ClamDialog.

A ClamDialog instance can be passed to any function as a Widget, as the operator () is defined to return a pointer to the underlying Motif Widget. The widget returned by the () operator on a ClamDialog is the topmost Form within the layout. The shell widget that contains the form (assuming shell type is not SHELL_NONE) can be determined by calling XtParent() on the ClamDialog instance (or by calling <ClamDialog instance>.parent() - see section 6 and the Baum manual for an explanation of Baum methods available for Clam objects).

6. BAUM

6.1 Overview

BAUM (Brown Augmented Utilities for Motif) is a set of C++ wrappers for Motif widgets. BAUM facilitates Motif programming in C++ by providing methods for setting and getting all resources, creating and managing widgets, and common Xt functions. CLAM’s internals rely heavily on BAUM. See the BAUM documentation for details on how to use the BAUM class library.

6.2 Using BAUM with CLAM

It is helpful to recognize that all CLAM classes inherit from some BAUM class. Therefore, BAUM methods can be called with CLAM objects.

```
class ClamPulldownMenu : public BaumRowColumnWidget
class ClamPopupMenu : public ClamMenu
class ClamMenu : BaumRowColumnWidget
```

```
class ClamPanel : public BaumFormWidget
class ClamDialog : public BaumFormWidget
```

7. Callback Functions

7.1 Overview

All of the work done by a Motif application is accomplished through callback functions. The interface is set up to process events (button clicks, key presses, mouse movement, time passage, etc.) and to translate these events into function calls. Once the interface is set up, the Xt Intrinsics takes over. The program sits in a simple loop waiting for events to happen. When an event happens, the appropriate callback is executed, passing control from the user interface to the application's routines.

The above model, that of Event Driven Programming, places certain restrictions on the way a program must operate. For example, a callback function should execute very quickly (approximately one second or less) in order for events to be processed smoothly (there are several ways to handle a situation where there is more work to be done than can be accomplished in under a second. See *Motif Programming Manual* by Dan Heller, O'Reilly & Associates, Inc., Chapter 20 - "Advanced Dialog Programming"). It is recommended that one read about events and callbacks in the Xt Intrinsics manual and O'Reilly Volume 1, *Xlib Programming Manual* before developing a system using Clam.

The following sections assume that the programmer is familiar with callback semantics.

7.2 CLAM Callbacks

The callback routines used by CLAM objects are of type XtCallbackProc. Note that a function of type BaumCallbackProc is exactly the same as one of type XtCallbackProc, however, these functions cannot be used interchangeably without casting, since the C++ compiler will not accept an XtPointer as a void *, even though they are exactly the same thing (typedef void * XtPointer;). A simple cast of BaumCallbackProc to XtCallbackProc or vice versa gets around this problem.

CLAM assigns callback functions to widgets as indicated in sections 7.3 - 7.5. Also, all CLAM classes have routines for adding additional callbacks to any widget and for removing callbacks.

7.3 ClamPulldownMenu & ClamPopupMenu Callback Types

The callback routine associated with a menu button is assigned to a push button as the *activateCallback* and to a toggle button as the *valueChange* callback. Cascade buttons (submenu buttons) cannot have callbacks.

7.4 ClamPanel Callback Types

Same as for menus.

7.5 ClamDialog Callbacks

Dialog callbacks are always called with the name of the affected widget as the *client_data*. The name thus serves to identify which of the widgets in the dialog has been changed. Many callbacks also contain important information about the user action in the *call_data*. The data structure passed as the *call_data* varies depending on the type of callback. Consult the Motif and X manuals for information on *call_data* structures.

The following list indicates how callbacks are assigned to each type of dialog widget:

• Label	no callback
• Push Button	activateCallback
• CascadeButton	cascadingCallback
• Toggle Button	valueChangedCallback
• Drawn Button	inputCallback
• Arrow Button	activateCallback
• Option Menu	simpleCallback = activateCallback for each button in menu
• Scrolled List	defaultActionCallback
• Text	valueChangedCallback
• TextField	valueChangedCallback

• Scale	valueChangedCallback
• Separator	no callback
• Drawing Area	inputCallback
• Command	commandEnteredCallback
• Scroll Bar	valueChangedCallback
• File Selection Box	okCallback

8. Using Resource Files

8.1 Overview

Every widget's behavior is determined by a set of parameters called resources. Typical resources are width, height, background color, foreground color, etc. Resources can be set by the program, in which case they cannot be changed at runtime, or there are several ways that resource values can be set at runtime. Consult an X reference for further information on the resource database and establishing resource files.

It is the intention that CLAM be flexible yet also very easy to use. With that end in mind, a minimal number of resources are hard coded in CLAM, allowing the programmer great flexibility in modifying the user interface components that CLAM generates. Sections 8.4 - 8.6 describe which resources *are* hard coded in CLAM, and certain important resources that a programmer may well want to customize.

8.2 Establishing a Resource File

There are many ways to change an application's resources. The following set of suggestions is by no means an exhaustive list.

- 1) If the Baum library is used, the BaumDisplay method BaumDisplay::resourceFile(<resource file name>) provides an easy way to indicate a resource file to be read when the program is executed.
- 2) Resources can be specified in the user's ~/.Xdefaults (see section 8.3 for naming conventions).
- 3) Resources can be specified in a file in the users root directory that has the same name as the application's top level shell widget.

8.3 Accessing a Widget's Resources

A particular widget is referenced by its name preceded by the names of the widget's ancestors as the widget tree is traversed up to the top level of the application. Names are separated by periods. A wildcard "*" can be used to replace any name or series of names. See the Motif manual for a complete list of resources for each Motif widget type.

Example: Top level of application named "app"
Main Window widget of application named "main", parent is "app"
Dialog box of application named "dialog", parent is "main"
Outermost region of dialog named "region0", parent is "dialog"
Push button widget in "region0" named "push1", parent is "region0"
Push button widget in "region0" named "push2", parent is "region0"

The push button widget "push1" can be referenced by the name

app.main.dialog.region0.push1

If the foreground and background colors of the push button were to be set, the format would be:

```
app.main.dialog.region0.push1.foreground:    <color for foreground>
app.main.dialog.region0.push1.background:    <color for background>
```

It is not necessary to specify the complete name hierarchy when referencing a widget. Wildcards can replace one or more names. The above could be shortened to:

```
*push1.foreground:    <color for foreground>
*push1.background:    <color for background>
```

If the foreground and background of all widgets in region0 were to be set, the format could be:

*region0*foreground: <color for foreground>
*region0*background: <color for background>

Note that without examining the CLAM code, it is not always possible to know the complete ancestor tree for a given widget. Use wildcards to work around any unknowns in the widget tree.

Also note that widgets that have scrollbars (scrolled list and scrolled text) are actually contained inside an additional widget, a container that holds the scrollbars and the text or list area. The containing widget has the name specified in the format string with the letters 'SW' appended to the end. For example, if it were necessary to specify the right attachment of a scrolled text widget named "scrolledText", one would reference

*scrolledTextSW*rightAttachment:

in the resource file.

8.4 ClamPulldownMenu & ClamPopupMenu Resource Notes

8.4.1 Hard Coded Resources

8.4.1.1 Push Button

- sensitive (if designated not sensitive)
- mnemonic (if mnemonic designated)
- labelType (if pixmap specified)
- labelPixmap (if pixmap specified)

8.4.1.2 Toggle Button

Same as for push buttons plus:

- selectPixmap (if selectPixmap specified)
- visibleWhenOff
- indicatorType (if diamond shape designated)
- set (if designated as selected)

8.4.1.3 Cascade Button (Submenu button)

Same as for push buttons.

8.5 ClamPanel Resource Notes

8.5.1 Hard Coded Resources

8.5.1.1 Push Button

- labelType (if pixmap specified)
- labelPixmap (if pixmap specified)

8.5.1.2 Toggle Button

Same as for push buttons plus:

- selectPixmap (if selectPixmap specified)

8.5.2 Recommended Resources to Change

8.5.2.1 Toggle Button

- indicatorOn (set to False to remove indicator from toggle button)

8.6 ClamDialog Resource Notes

8.6.1 Hard Coded Resources

A widgets left and top attachments are hardcoded unless the %f and %k parameters are specified. No other resources are hardcoded unless a parameter is specified.

8.6.2 Recommended Resources to Change

There are any number of resources that can be changed to alter the appearance and behavior of the widgets in

a dialog box. However, there are two important resources that bear mention - *leftOffset* and *topOffset*.

The widgets within a ClamDialog are all attached on the top and left sides to another widget (unless %f and/or %k is used). Thus all widgets are placed rather snugly inside the dialog box. Widgets can be moved from the default position by adjusting the top and left offsets. Note that these offsets will not be affected by resource file settings if they are being controlled by tabs, pads, or the spread parameter.

9. Problems? - Common Pitfalls

9.1 Menus

- Check the ClamMenuData array! Make sure that all fields in the structure have been initialized.

9.2 Panels

- Check the ClamPanelBtn array! Make sure that all fields in the structure have been initialized.

9.3 Dialogs

- Check the format string for inconsistencies!
- Can't set resources? Check that names followed by white space are terminated with '%.'.
- 'Stretched' widgets not stretching? Perhaps the containing region also needs to stretch.

10. Known Bugs

10.1 Menus

10.2 Panels

10.2.1 Resize: ClamPanels are sometimes "one event behind" when resizing. The ClamPanel's buttons do not reflect the current size of the window, rather they reflect the size of the window before the last resize event.

10.3 Dialogs

10.3.1 Dialog follow: If a dialog is configured to follow its parent window when the parent is moved, the dialog can get confused by movement within a virtual window manager.

11. Future Enhancements

11.1 Menus

- To be specified with a string like ClamDialog
- To accept submenus upon initialization, i.e. no need to call fillSubmen() method

11.2 Panels

- To be specified with a string like ClamDialog

11.3 Dialogs

Appendix A - Example programs

Look for the examples directory where CLAM is installed.