IRIS ViewKit™ Programmer’s Guide

Document Number 007-2124-003

CONTRIBUTORS

Written by Ken Jones, Douglas B. O'Morain, and Sandra Motroni
Production by Gloria Ackley and Linda Rae Sande
Engineering contributions by Doug Young, Kim Rachmeler, Mike Yang, and Robert Blean
St Peter's Basilica image courtesy of ENEL SpA and InfoByte SpA. Disk Thrower image courtesy of Xavier Berenguer, Animatica.

IRIS ViewKit™ Programmer's Guide
Document Number 007-2124-003

Contents

Examples

Figures

Tables

Introduction

This guide describes how to create programs using IRIS ViewKit™, a C++ toolkit that provides commonly needed facilities for applications based on the IRIS IM™ user interface toolkit (the Silicon Graphics port of the industry-standard OSF/Motif™ user interface toolkit for use on Silicon Graphics workstations).

What This Guide Contains

The first two chapters of this guide provide an overview of ViewKit concepts:

Chapter 1, "Overview of ViewKit"
: describes the ViewKit toolkit and the advantages of using it compared to programming directly in IRIS IM and X, discusses the major elements of ViewKit, and provides instructions for compiling ViewKit programs.

Chapter 2, "Components"
: describes the ViewKit component class, gives instructions for using ViewKit components, and lists guidelines for creating new components.

The next six chapters describe the common ViewKit components that you use in practically every ViewKit program:

Chapter 3, "The ViewKit Application Class"
: explains the services provided by the ViewKit application class and gives instructions for controlling application-level services in your program.

Chapter 4, "ViewKit Windows"
: explains ViewKit's model for supporting multiple windows in an application, and describes how to create and manipulate application windows.

Chapter 5, "Creating Menus With ViewKit"
: describes how to create and manipulate different types of menus in a ViewKit application.

Chapter 6, "ViewKit Undo Management and Command Classes"
: explains how to implement support for "undoing" operations and describes how to implement actions as command classes.

Chapter 7, "Using Dialogs in ViewKit"
: discusses ViewKit's dialog management support, describes how to post and manipulate dialogs, and provides an overview of the different types of dialogs supported by ViewKit.

Chapter 8, "Preference Dialogs"
: describes how to use preference dialogs to maintain user preferences.

The rest of the book describes pre-built ViewKit components:

Chapter 9, "The ViewKit Graph Component"
: discusses the ViewKit component for creating and displaying arc-and-node graphs.

Chapter 10, "Miscellaneous ViewKit Display Classes"
: describes a variety of components that you use primarily to display information or to manage display items.

Chapter 11, "Miscellaneous ViewKit Data Input Classes"
: describes a variety of data input classes.

Appendix A, "ViewKit Interprocess Message Facility"
: describes ViewKit's support for interprocess communication.

Appendix B, "Contributed ViewKit Classes"
: gives you an idea of how you can expand ViewKit by describing some unsupported ViewKit classes that users have contributed.

Appendix C, "Using a Help System With ViewKit"
: explains how to use a help system with ViewKit applications. It also describes the basic help system provided with ViewKit.

What You Should Know Before Reading This Guide

This guide assumes that you are already an experienced C++ programmer. It also assumes that you are generally familiar with IRIS IM.

For a thorough discussion of the concepts on which the ViewKit toolkit is based, see:

- Young, Douglas A. *Object-Oriented Programming with C++ and OSF/Motif*. Englewood Cliffs, New Jersey: Prentice Hall, Inc., 1992.

For information on OSF / Motif, see:

- Open Software Foundation. *OSF/Motif Programmer's Guide, Revision 1.2*. Englewood Cliffs, New Jersey: Prentice Hall, Inc., 1992.
- Open Software Foundation. *OSF/Motif Programmer's Reference, Revision 1.2*. Englewood Cliffs, New Jersey: Prentice Hall, Inc., 1992.
- Open Software Foundation. *OSF/Motif Style Guide, Revision 1.2*. Englewood Cliffs, New Jersey: Prentice Hall, Inc., 1992.
- Heller, Dan. *Motif Programming Manual* (X Window System Series: Volume Six). Sebastopol, California: O'Reilly & Associates, Inc., 1992.

For information on IRIS IM enhancements to OSF / Motif and general tips for programming in IRIS IM on Silicon Graphics workstations, refer to the *IRIS IM Programming Notes*.

For comprehensive information on the X Window System™, Xlib, and Xt, see:

- Nye, Adrian. *Xlib Programming Manual* (X Window System Series: Volume One). Sebastopol, California: O'Reilly & Associates, Inc., 1992.
- O'Reilly & Associates, Inc. *Xlib Reference Manual* (X Window System Series: Volume Two). Sebastopol, California: O'Reilly & Associates, Inc., 1992.
- Nye, Adrian, and Tim O'Reilly. *X Toolkit Intrinsics Programming Manual* (X Window System Series: Volume Four). Sebastopol, California: O'Reilly & Associates, Inc., 1992.
- O'Reilly & Associates, Inc. *X Toolkit Intrinsics Reference Manual* (X Window System Series: Volume Five). Sebastopol, California: O'Reilly & Associates, Inc., 1992.

Conventions Used in This Guide

This section describes the conventions used for presenting information in this book.

Typographical Conventions

These type conventions and symbols are used in this guide:

Bold	C++ class names, C++ member functions, C++ data members, function names, literal command-line arguments (options and flags)
Italics	Filenames; onscreen button names; IRIX™ commands; executable files; manual and book titles; glossary entries; new terms; variable command-line arguments; program variables; and variables to be supplied by the user in examples, code, and syntax statements
`Screen type`	Onscreen text, prompts, error messages, examples, and code listings
`Bold screen type`	User input, including keyboard keys (printing and nonprinting); literals supplied by the user in examples, code listings, and syntax statements
““	(Double quotation marks) Onscreen menu items and references in text to document section titles
()	(Parentheses) Follow function names; also used to surround reference page (man page) section in which a command, function, or class is described
<>	(Angle brackets) Surround nonprinting keyboard keys, for example, **`<Esc>`**, **`<Ctrl-D>`**
#	IRIX shell prompt for the superuser (*root*)
%	IRIX shell prompt for users other than superuser

Reference pages (also known as man pages) are referred to by name and section number, in this format: name(section), where “name” is the name of a command, system call, library routine, or class; and “section” is the section number where the entry resides. For example, XtSetValues(3Xt) refers to the **XtSetValues()** reference page in section 3Xt.

Class Inheritance Graph Conventions

Most of the chapters in this book begin with a graph depicting the inheritance hierarchy of the classes described in that chapter. Figure i shows an example of a class inheritance graph that might appear at the beginning of a chapter.

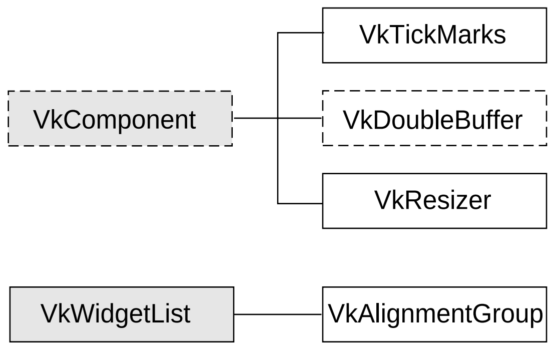


Figure i Example of a Class Inheritance Graph

In these inheritance graphs, classes are presented with the base classes to the left and the derived classes to the right. Abstract classes have dashed borders and non-abstract classes have solid borders. Classes described within the chapter appear in white boxes, whereas classes described elsewhere appear in shaded boxes.

In the inheritance graph shown in Figure i, **VkComponent** is an abstract base class. As indicated by its shaded box, it is not described within the chapter. The chapter describes three subclasses of **VkComponent**: **VkDoubleBuffer**, an abstract class; and **VkTickMarks** and **VkResizer**, non-abstract classes. The chapter also discusses the non-abstract class **VkAlignmentGroup**, which is derived from the non-abstract base class **VkWidgetList**.

Chapter 1

Overview of ViewKit

ViewKit is a C++ toolkit that makes it easier for you to develop applications. It provides a collection of high-level user interface components and other support facilities that you typically must implement in every application. For example, it provides high-level user interface components, such as windows, menus, and dialogs.

ViewKit does not replace IRIS IM or any other user interface toolkit. In fact, it uses IRIS IM widgets to implement all of its user interface components; also, you can directly call IRIS IM functions to create and manipulate widgets in a ViewKit application. The ViewKit architecture helps mask much of the complexity of programming with IRIS IM.

ViewKit offers you several benefits:

- It provides support for common user interface components such as windows, menus, and dialogs. It also provides specialty interface components for tasks such as displaying and managing arc-and-node graphs, displaying and managing toggle check boxes, and managing the layout of other widgets. Creating these elements using ViewKit is much simpler and faster than using low-level widgets to build them from scratch. Furthermore, by using the same basic components, applications that use ViewKit components have greater visual and behavioral consistency.
- It simplifies interaction with the X resource manager, allowing you to customize your application using resources more easily. By designing your application to use resource values rather than hard-coding the values in your program, you can easily modify the appearance of your application. This approach is particularly useful for preparing your application for internationalization.
- All user interface components in ViewKit are C++ classes, which provides a framework for using IRIS IM in a highly structured, object-oriented way. The ViewKit architecture encourages you to develop self-contained objects that you can re-use in multiple applications.
- It provides support for other common application services such as interprocess communication.

Major ViewKit Elements

You can think of ViewKit as consisting of several sets of classes: framework classes, interface components, interapplication communication, and convenience utilities. The following sections discuss these groups.

Framework Classes

ViewKit provides a small set of classes that are either essential for all applications or provide fundamental support for all other classes. The most basic of these classes is the **VkComponent** class, which defines the basic structure of all user interface components. All user interface classes are derived from **VkComponent**.

The framework classes also include support for features needed by nearly all applications, including application management and X server setup, top-level windows, menus, and dialog management. All classes are designed to implement as many typical features as possible. For example: all top-level windows and dialogs handle the window manager quit/close protocol; dialogs are cached to balance memory use and display speed; the menu system goes beyond simply constructing menus to support dynamically adding, removing, replacing items, and more.

The classes that make up the framework of ViewKit are closely integrated and work together to support essential features required by most applications as automatically as possible. Among the basic services supported by the core ViewKit framework are: single and multi-level undo; interruptible tasks; and an application-level callback mechanism that allows C++ classes to dynamically register member functions to be invoked by other C++ classes.

Interface Components

In addition to the basic user interface support provided by the core framework classes, ViewKit provides an assortment of ready-to-use interface components. Examples of these components are a graph viewer/editor, an input field that supports name expansion, and an outliner component for displaying and manipulating hierarchical information.

You are encouraged to use the architecture of ViewKit to create new components and extend existing components. Creating reusable, high-level components promotes consistency throughout a set of applications by providing elements that users can learn once and then easily recognize in multiple applications.

Interapplication Communication

Many applications need to be able to communicate with other applications to work effectively. ViewKit builds on the ToolTalk™ interapplication communications service to support simple and effective interapplication communication.

Convenience Utilities

ViewKit provides various utility functions and classes for your convenience. These utilities include simple functions that make it easier to load resources (including automatic type conversion), classes that support the use of icons, and other miscellaneous utilities.

Mixing ViewKit and Standard X and IRIS IM Functions

As stated earlier, ViewKit does not replace IRIS IM. It uses IRIS IM widgets to implement all of its user interface components, and you are free to make X and IRIS IM calls directly in a ViewKit application. ViewKit doesn't do anything that you can't do yourself using IRIS IM directly, but the advantage of using ViewKit is that many commonly needed services are already implemented for you.

Naturally, not all ViewKit services are appropriate for all applications at all times. If a situation arises in which a ViewKit facility doesn't meet your needs, you can use the lower-level IRIS IM, Xt, or Xlib facilities to perform the desired operation yourself.

Most ViewKit classes are optional; however, you should be aware that certain ViewKit classes depend on other classes. In particular, most classes depend on the existence of an instance of the **VkApp** class for application management. If you plan to use any ViewKit facilities, you should not attempt to bypass **VkApp** and open your own connection to the X server, or directly call **XtAppInitialize()** or an equivalent function. For best results, you should always allow **VkApp** to handle the Xt initialization and event dispatching. **VkApp** is described in detail in Chapter 3, "The ViewKit Application Class."

Also, you should use **VkSimpleWindow** or **VkWindow** for all top-level windows. These classes are described in detail in Chapter 4, “ViewKit Windows.”

As an example of some optional classes, consider the ViewKit dialog management facilities. These are intended to let you use dialogs easily and effectively. ViewKit automatically recycles dialogs (reusing the same dialog over and over for multiple purposes), which uses less memory and can lead to faster response times. It is also easy to add additional buttons to any dialog, to provide context-sensitive help on individual dialogs, and much more. The ViewKit dialog management facility is designed to be as flexible as possible, while minimizing the amount of work required of you. You can even write your own custom dialogs that take advantage of the dialog manager.

However, because the design of the ViewKit dialog management classes makes assumptions about the way typical applications use dialogs, the ViewKit dialog manager can’t offer the same control that you could obtain by directly constructing and manipulating an IRIS IM dialog. Should you encounter a situation where the behavior of the dialog manager doesn’t match your application’s needs, you can always take the same approach you would have to take if the dialog manager didn’t exist: create and manipulate your own IRIS IM dialog directly using IRIS IM and Xt functions. This doesn’t interfere with ViewKit in any way.

Before implementing your own mechanisms, you should be sure you understand the support offered by ViewKit. Situations in which it’s necessary to duplicate functionality supported by ViewKit should be rare. On the other hand, extending the class library by deriving new classes, or writing completely new classes to meet application-specific needs, is a natural part of developing any application based on ViewKit or any C++ class library.

Compiling and Linking ViewKit Programs

This section describes the software needed to compile and link ViewKit programs.

Required Packages

To compile and link with the ViewKit libraries, you must install the IRIS Development Option (IDO). This option includes the C compiler and the X Window System™ and IRIS IM development systems. You must also install the C++ Development Option, including the ViewKit development option subsystems. Consult the *ViewKit Release Notes* for a complete list of subsystems that you must install on your system to compile and link ViewKit programs.

The ViewKit development option contains the following subsystems:

ViewKit_dev.sw.base
: You are required to install this subsystem, which contains the optimized, unshared C++ ViewKit libraries and include files. (The shared ViewKit libraries are included in the IRIX system software as the *ViewKit_eoe.sw.base* subsystem.)

ViewKit_dev.sw.debug
: This subsystem contains the debug version of the optimized ViewKit libraries. You can optionally install this subsystem in addition to the *ViewKit_dev.sw.base* subsystem. Use this library for program debugging only.

ViewKit_dev.man.pages
: The complete set of C++ reference pages (man pages) for ViewKit. This subsystem is optional, but recommended.

ViewKit_dev.man.relnotes
: The online version of the *ViewKit Release Notes*. This subsystem is optional, but recommended.

ViewKit_dev.books.ViewKit_PG
: The IRIS Insight™ version of this guide. This subsystem is optional, but recommended.

ViewKit_dev.sw.demo
: Sample source code to various ViewKit programs. This subsystem is optional, but recommended.

The *ViewKit_dev.sw.base* subsystem installs the following libraries:

libvk.a	The basic ViewKit class library.
libvkmsg.a	Classes that support inter-process communication based on the ToolTalk library.
libXpm.a	A library that supports X pixmap creation. This is a freely available library, Copyright © 1990, 1991 GROUPE BULL. Some ViewKit classes use Xpm.

The *ViewKit_dev.sw.debug* subsystem installs the following libraries:

libvk_d.a	The debug version of the basic ViewKit class library.
libvkmsg_d.a	The debug version of the classes that support inter-process communication based on the ToolTalk library.

Required Header Files

All ViewKit header files appear in */usr/include/Vk*. In most cases, the header file for a given class is the class name followed by "*.h*". For example, the header file for the **VkWindow** class is <*Vk/VkWindow.h*>. Some minor classes are grouped together into single header files. For example, the header file for the **VkMenu** class automatically includes the header information for every type of menu supported by ViewKit. These cases are noted in the text where appropriate.

You need to include IRIS IM header files for only those IRIS IM widgets that you explicitly use in a ViewKit program. ViewKit automatically includes any X or IRIS IM header files required by ViewKit components that you use in your program.

Required Libraries

You must link all ViewKit programs with the ViewKit library, *libvk*, and the IRIS IM and X libraries. If you use an external help system with your application, you should link with the appropriate help library. (See Appendix C, "Using a Help System With ViewKit," for information on using a help system with ViewKit applications.)

For example, to compile a file *hello.c++* to produce the executable *hello,* you enter:

```
CC -o hello hello.c++ -lvk -lXm -lXt -lX11
```

If you are debugging a program, you might find it useful to compile your program with the debug libraries, which contain additional symbol table information.

Getting Started

This section gives you information on example programs that you might find helpful when getting started with ViewKit programming. It first describes the simplest ViewKit program, which displays a window containing a single label, and discusses the structure of the program. Then, it discusses the demonstration programs provided with ViewKit.

The Simplest ViewKit Program

Applications based on ViewKit must obey certain organizational conventions. To see how this organization works, consider a simple example of a ViewKit application that displays the label “hello” in a window.

Example 1-1 The Simplest ViewKit Program: *hello.c++*

```
#include <Vk/VkApp.h>
#include <Vk/VkSimpleWindow.h>
#include <Xm/Label.h>

// Define a top-level window class

class HelloWindow: public VkSimpleWindow {

  public:
    HelloWindow (const char *name);
    ~HelloWindow();
    virtual const char* className();
};

// Construct a single rooted widget tree, and designate the
// root of the tree as the window’s view.  This example is very
// simple, just creating a single XmLabel widget to display the
// string "hello".
```

```
HelloWindow::HelloWindow (const char *name) : VkSimpleWindow (name)
{
    Widget label =  XmCreateLabel (mainWindowWidget(), "hello",
                                   NULL, 0);
    addView(label);
}

const char* HelloWindow::className()
{
    return "HelloWindow";  // Identify this class
}

HelloWindow::~HelloWindow()
{
    // Empty
}


// Main driver. Just instantiate a VkApp and a top-level window,
// "show" the window and then "run" the application.

void main ( int argc, char **argv )
{
    VkApp        *app = new VkApp("Hello", &argc, argv);
    HelloWindow  *win = new HelloWindow("hello");

    win->show();
    app->run();
}
```

To build this example, simply compile the file *hello.c++* and link with the ViewKit library, and the IRIS IM and X libraries:

```
CC -o hello hello.c++ -lvk -lXm -lXt -lX11
```

Running the *hello* program displays a window that says "hello," as shown in Figure 1-1.

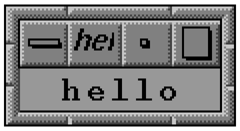


Figure 1-1 Result of Running *hello*

This example uses two classes: the **VkApp** class and an application-defined class, **HelloWindow**. The **HelloWindow** class is derived from the ViewKit **VkSimpleWindow** class.

First look at **main()**. All ViewKit applications start by creating an instance of **VkApp**. The arguments to this constructor specify the Xt-style class of the application, a pointer to *argc*, and the *argv* array. Instantiating a **VkApp** object opens a connection to the X server and initializes many other services needed by typical applications. **VkApp** is described in detail in Chapter 3, “The ViewKit Application Class.” Next, the *hello.c++* program instantiates a **HelloWindow** object that serves as the application’s top-level window. The constructor for this class requires only a name for the window. Finally, the application concludes by calling the **HelloWindow** object’s **show()** function and the **VkApp** object’s **run()** function. The **run()** method never returns. The bodies of most ViewKit programs are similar to this short example.

Now look at the **HelloWindow** class. ViewKit encourages you to create classes to represent all major elements of the user interface. In this simple example, the only major user interface component is a top-level window that contains a label widget. ViewKit provides a class, **VkSimpleWindow**, that supports many features common to all top-level windows and that works closely with the **VkApp** class to implement various ViewKit features. To use the **VkSimpleWindow** class, you derive a new subclass and create a single-rooted widget tree that the window displays as its *view*. ViewKit applications do not have to create shell widgets directly.

The *hello.c++* example is so simple that the **HelloWindow** class creates only a single XmLabel widget. The XmLabel widget is created in the constructor and then designated as the window’s view. More complex classes might create a manager widget and create other widgets as children, or might instantiate other objects, as well. Chapter 4, “ViewKit Windows,” describes how to create windows using ViewKit.

The **className()** member function is supported, by convention, by all ViewKit classes. This function is used by several ViewKit facilities and is discussed in “VkComponent Access Functions” on page 17.

Demonstration Programs

The *ViewKit_dev.sw.demo* subsystem installs in the */usr/share/src/ViewKit* directory several demonstration programs that illustrate different features of ViewKit. A few of the highlights include:

- */usr/share/src/ViewKit/ProgrammersGuide* contains several of the example programs from this guide.
- */usr/share/src/ViewKit/Components/CBrowser* contains the source for a component browser, which shows examples of many ViewKit components. You might find this particularly useful to run when you read the later chapters in this guide that describe the prebuilt components shipped with ViewKit.
- */usr/share/src/ViewKit/Applications/PhoneBook* creates PhoneBook, a full-fledged application that keeps track of names, phone numbers, and addresses.PhoneBook uses a variety of ViewKit classes.
- */usr/share/src/ViewKit/Applications/GLX* builds Rotate, a sample application that uses GLX to do GL rendering in an X window.
- */usr/share/src/ViewKit/Applications/Inventor* builds IvClock, a ViewKit implementation of the Inventor clock sample program from Inventor 2.0.

Chapter 2

Components

This chapter introduces the concept of ViewKit *components*: C++ classes that encapsulate sets of widgets along with convenient methods for their manipulation.

This chapter describes two ViewKit classes: **VkCallbackObject** and **VkComponent**. Figure 2-1 shows the inheritance graph for these classes.

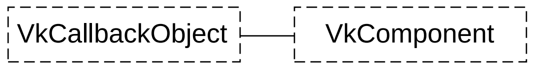


Figure 2-1 The Inheritance Graph for **VkCallbackObject** and **VkComponent**

Definition of a Component

Widget sets such as IRIS IM provide simple, low-level building blocks, like buttons, scrollbars, and text fields. However, to create interesting and useful applications, you must build collections of widgets that work together to perform given tasks. For example, many applications support a system of menus, which are constructed from several individual widgets. Just as the user thinks of the menu bar as a single logical component of the user interface, ViewKit builds abstractions that let applications deal with a "menu" rather than the individual pieces of the menu.

C++ allows you to do exactly this: to encapsulate collections of widgets and other objects as logical entities. By creating C++ classes and providing simple, convenient manipulation functions, you can avoid the complexity of creating widgets, specifying widget locations, setting resources, assigning callbacks, and other common tasks. Furthermore, for commonly used objects like menus, you can design general-purpose classes that you can easily use in many different applications.

In ViewKit, the general user interface classes are referred to as *components*. A component not only encapsulates a collection of widgets, but also defines the behavior of the overall component. ViewKit components are designed to implement as many commonly used features as possible. Typically, all you need to do to use a ViewKit component is create a subclass of the appropriate ViewKit class and define any application-specific behavior. Furthermore, using the ViewKit classes as a base, you can create your own library of reusable components.

The VkComponent Class

All ViewKit components are derived from the abstract base class **VkComponent**, which defines a basic structure and protocol for all components. When creating your own components, you should also derive them from **VkComponent** or one of its subclasses.

The **VkComponent** class enforces certain characteristics on components and expects certain behaviors of its subclasses. These characteristics and the features provided by **VkComponent** are discussed in detail in the following sections; the more important characteristics are summarized below:

- Widgets encapsulated by a component must form a single-rooted subtree. Components typically use a container widget as the root of the subtree; all other widgets are descendents of this widget. The root of the widget subtree is referred to as the *base widget* of the component.
- .You can create instances of components and use them in other components's widget subtrees. As a convenience, **VkComponent** defines an operator that allows you to pass a **VkComponent** object directly to functions that expect a widget. This operator is described further in "VkComponent Access Functions" on page 17.
- Components take a string as an argument (typically, the first argument) in the class constructor. This string is used as the name component's base widget. You should give each instance of a component a unique name so that you can identify each widget in an application by a unique path through the application's widget tree. If each widget can be uniquely identified, X resource values can be used to customize the behavior of each widget. ViewKit resource support is described in "Component Resource Support" on page 26.
- Components take a widget as an argument (typically, the second argument) in the class constructor. This widget is the parent of the component's base widget. Component constructors are discussed in "Component Constructors" on page 13.

- Most components should create the base widget and all other widgets in the class constructor. The constructor should manage all widgets except the base widget, which should be left unmanaged. You can then manage or unmanage a component's entire widget subtree using the member functions described in "Displaying and Hiding Components" on page 19.
- **VkComponent** provides an access function that retrieves the component's base widget. You might need to access the base widget, for example, to set constraint resources so that an XmForm widget can position the component. Normally, other widgets inside a component aren't exposed. Access functions are discussed in "VkComponent Access Functions" on page 17.
- Components must handle the destruction of widgets within the component's widget tree. The widgets encapsulated by the component must be destroyed when the component is destroyed. Component classes must also prevent dangling references by handling destruction of the widget tree without destruction of the component. **VkComponent** provides mechanisms for handling widget destruction which are described in "Handling Component Widget Destruction" on page 24.
- Components should define any Xt callbacks required by a class as private static member functions. Using Xt callbacks in ViewKit is discussed in "Using Xt Callbacks with Components" on page 21.
- All component classes must override the virtual **className()** member function so that it returns a string identifying the component's class. ViewKit uses this string for resource handling and other support functions. The **className()** member function is described in more detail in "VkComponent Access Functions" on page 17. "Component Resource Support" on page 26 describes ViewKit resource support.

Component Constructors

The **VkComponent** constructor has the following form:

```
VkComponent( const char *name )
```

The **VkComponent** constructor is declared protected and so can be called only from derived classes. Its primary purpose is to initialize component data members, in particular *_name* and *_baseWidget*.

Each component should have a unique name, which is used as the name of the component's base widget. The **VkComponent** constructor accepts a name as an argument, creates a copy of this string, and assigns the address of the copy to the *_name* data member.

The *_baseWidget* data member is the base widget of the component's widget subtree. The **VkComponent** constructor initializes *_baseWidget* to NULL.

Each derived class's constructor should take at least two arguments, the component's name and a widget that serves as the parent of the component's widget tree, and perform at least these initialization steps:

1. Pass the name to the **VkComponent** constructor to initialize the basic component data members.
2. Create the component's widget subtree and assign the base widget to the *_baseWidget* data member. The base widget should be a direct child of the parent widget passed in the constructor, and should have the same name as the component (as stored in *_name*) for the ViewKit resource support to work correctly. All other widgets in the component must be children or descendents of the base widget.
3. Immediately after creating the base widget, call **installDestroyHandler()** to set up a callback to handle widget destruction. This function is described further in "Handling Component Widget Destruction" on page 24.
4. Manage all widgets except the base widget, which should be left unmanaged.
5. Perform any other needed class initialization.

As an example, consider a user-defined component called **StartStopPanel** that implements a simple control panel containing *Start* and *Stop* buttons. The code fragment in Example 2-1 shows a possible constructor for this class.

Example 2-1 Example of a Component Constructor

```
/////////////////////////////
// StartStopPanel.h
/////////////////////////////

// Declare StartStopPanel as a subclass of VkComponent

class StartStopPanel: public VkComponent {

  public:
    StartStopPanel (const char *, Widget);
    ~StartStopPanel();
    // ...

  protected:
    Widget _startButton;
    Widget _stopButton;
    // ...
}

/////////////////////////////
// StartStopPanel.c++
/////////////////////////////


// Pass the name to the VkComponent constructor to initialize the
// basic component data members.

StartStopPanel::StartStopPanel(const char *name, Widget parent):VkComponent(name)
{
  // Create an XmRowColumn widget as the component's base widget
  // to contain the buttons. Assign the widget to the _baseWidget
  // data member.
  _baseWidget = XmCreateRowColumn ( parent, _name, NULL, 0 );
  // Set up callback to handle widget destruction

  installDestroyHandler();

  XtVaSetValues(_baseWidget, XmNorientation, XmHORIZONTAL, NULL);

  // Create all other widgets as children of the base widget.
  // Manage all child widgets.

  _startButton = XmCreatePushButton ( _baseWidget, "start", NULL, 0);
  _stopButton  = XtCreatePushButton ( _baseWidget, "stop", NULL, 0);

  XtManageChild(_startButton);
  XtManageChild(_stopButton);

  // Perform any other initialization needed (omitted in this example)
}
```

In this example, the **StartStopPanel** constructor passes the *name* argument to the **VkComponent** constructor to initialize the *_name* data member. The **VkComponent** constructor also initializes the *_baseWidget* data member to NULL. It then creates a RowColumn widget as the base widget to manage the other widgets in the component. The constructor uses the *_name* data member as the name of the base widget, uses the *parent* argument as the parent widget, and assigns the RowColumn widget to the *_baseWidget* data member. Immediately after creating the base widget, the constructor calls **installDestroyHandler()**. Then, it creates the two buttons as children of the base widget and manages the two child widgets.

A real constructor would then perform all other initialization needed by the class, such as setting up callbacks for the buttons and initializing any other data members that belong to the class. "Using Xt Callbacks with Components" on page 21 describes how you should set up Xt callbacks when working with ViewKit components.

Component Destructors

The virtual **VkComponent** destructor performs the following functions:

1. Triggers the *VkComponent::deleteCallback* ViewKit callback for that component. ViewKit callbacks are described in "ViewKit Callback Support" on page 34, and the *VkComponent::deleteCallback* is described in "Predefined ViewKit Callbacks" on page 40.
2. Removes the widget destruction handler described in "Handling Component Widget Destruction" on page 24.
3. Destroys the component's base widget, which in turn destroys the component's entire widget subtree.
4. Frees all memory allocated by the **VkComponent** constructor.
5. Sets to NULL all the data members defined by the **VkComponent** constructor.

The destructor for a derived class need free only the space that was explicitly allocated by the derived class, but of course it can perform any other cleanup your class requires.

For example, if your class allocates space for a string, you should free that space in your destructor, as shown in Example 2-2.

Example 2-2 Freeing Space in a Component Destructor

```
MyComponent: public VkComponent {

  public:
    MyComponent(const char *, Widget);
    ~MyComponent();
    // ...

  private:
    char *_label;
    //...
}
MyComponent::MyComponent(const char *name, Widget parent) : VkComponent(name)
{
    _label = strdup( label );
   // ...
}

MyComponent::~MyComponent()
{
    free ( _label );
}
```

Even if you don't need to perform any actions in a class destructor, you should still declare an empty one. If you don't explicitly declare a destructor, the C++ compiler creates an empty inline destructor for the class; however, because the destructor in the base class, **VkCallbackObject**, declares the destructor as virtual, the C++ compiler generates a warning because a virtual member function can't be inlined. The compiler then "un-inlines" the destructor and, to ensure that it's available wherever needed, puts a copy of it in every file that uses the class. Explicitly creating an empty destructor for your classes avoids this unnecessary overhead.

VkComponent Access Functions

VkComponent provides access functions for accessing some of the class's data members.

The **name()** function returns the name of a component as pointed to by the *_name* data member. This is the same as the name that you provided in the component's constructor. The syntax of the **name()** function is:

```
const char * name() const
```

The **className()** function returns a string identifying the name of the ViewKit class to which the component belongs. The syntax of **className()** is:

```
virtual const char *className()
```

All component classes should override this virtual function to return a string that identifies the name of the component's class. ViewKit uses this string for resource handling and other support functions. The class name for the **VkComponent** class is "VkComponent."

For example, if you create a **StartStopPanel** class, you should override the **StartStopPanel::className()** function as follows:

```
class StartStopPanel: public VkComponent {
  public:
    // ...
    virtual const char *className();
    // ...
}

const char* StartStopPanel::className()
{
    return "StartStopPanel";
}
```

The **baseWidget()** function returns the base widget of a component as stored in the *_baseWidget* data member:

```
Widget baseWidget() const
```

Normally, components are as encapsulated as possible, so you should avoid operating directly on a component's base widget outside the class. However, certain operations might require access to a component's base widget. For example, after instantiating a component as a child of an XmForm widget, you might need to set various constraint resources, as shown below:

```
Widget form = XmCreateForm(parent, "form", NULL, 0);
StartStopPanel *panel = new StartStopPanel("panel", form);
XtVaSetValues(panel->baseWidget(), XmNtopAttachment, XmATTACH_FORM, NULL);
```

As a convenience, **VkComponent** defines a Widget operator that allows you to pass a **VkComponent** object directly to functions that expect a widget. By default, the operator converts the component into its base widget. However, the operator is defined as a virtual function so that derived classes can override it to return a different widget. Note that you must use an object, not a pointer to an object, because of the way operators work in C++. For example, the Widget operator makes the following code fragment equivalent to the fragment presented above:

```
Widget form = XmCreateForm(parent, "form", NULL, 0);
StartStopPanel *panel = new StartStopPanel("panel", form);
XtVaSetValues(*panel, XmNtopAttachment, XmATTACH_FORM,
NULL);
```

Displaying and Hiding Components

The virtual member function **show()** manages the base widget of the component, displaying the entire component. The virtual member function **hide()** performs the inverse operation. You can call **show()** after calling **hide()** to redisplay a component. The syntax of these commands are simply:

```
virtual void show()
virtual void hide()
```

For example, the following lines display the component *panel*, an instance of the **StartStopPanel**:

```
StartStopPanel *panel = new StartStopPanel("panel", form);
panel->show();
```

You could hide this component with the line:

```
panel->hide();
```

If you're familiar with Xt, you can think of these functions as performing operations analogous to managing and unmanaging the widget tree; however, you shouldn't regard these functions simply as "wrappers" for the **XtManageChild()** and **XtUnmanageChild()** functions. First, these member functions show and hide an entire component, which typically consists of more than one widget. Second, other actions might be involved in showing a component. In general, the **show()** member function does whatever is necessary to make a component visible on the screen. You shouldn't circumvent these member functions and manage and unmanage components' base widgets directly. For example, some components might use **XtMap()** and **XtUnmap()** as well. Other components might not even create their widget subtrees until **show()** is called for the first time.

The **VkComponent** class also provides the protected virtual function **afterRealizeHook()**. This function is called after a component's base widget is realized, just before it's mapped for the first time. The default action is empty. You can override this function in a subclass if you want to perform actions after a component's base widget exists.

VkComponent Utility Functions

All ViewKit components provide the virtual member function **okToQuit()** to support "safe quit" mechanisms:

```
virtual Boolean okToQuit()
```

A component's **okToQuit()** function returns TRUE if it is "safe" for the application to quit. For example, you might want **okToQuit()** to return FALSE if a component is in the process of updating a file. By default, **okToQuit()** always returns TRUE; you must override **okToQuit()** for all components that you want to perform a check before quitting.

Usually only **VkSimpleWindow** and its subclasses use **okToQuit()**. When you call **VkApp::quitYourself()**, **VkApp** calls the **okToQuit()** function for all registered windows before quitting. If the **okToQuit()** function for any window returns FALSE, the application doesn't exit. "Quitting ViewKit Applications" on page 58 provides more information on how to quit a ViewKit application, and "Providing a "Safe Quit" Mechanism" on page 101 describes how to override **VkSimpleWindow::okToQuit()** to provide a "safe quit" mechanism for a window.

In some cases you might want to check one or more components contained within a window before quitting. To do so, override the **okToQuit()** function for that window to call the **okToQuit()** functions for all the desired components. Override the **okToQuit()** functions for the other components to perform whatever checks are necessary.

Another utility function provided by **VkComponent** is the static member function **isComponent()**:

```
static Boolean isComponent(VkComponent *component)
```

The **isComponent()** function applies heuristics to determine whether the pointer passed as an argument represents a valid **VkComponent** object. If *component* points to a **VkComponent** that has not been deleted, this function always returns TRUE; otherwise the function returns FALSE. It is possible, though highly unlikely, that this function could mistakenly identify a dangling pointer to a deleted object as a valid object. This could happen if another component were to be allocated at exactly the same address as the deleted object a pointer previously pointed to. The **isComponent()** function is used primarily for ViewKit internal checking, often within **assert()** macros.

Using Xt Callbacks with Components

Callbacks pose a minor problem for C++ classes. C++ member functions have a hidden argument, which is used to pass the *this* pointer to the member function. This hidden argument makes ordinary member functions unusable as callbacks for Xt-based widgets. If a member function were to be called from C (as a callback), the *this* pointer would not be supplied and the order of the remaining arguments might be incorrect.

Fortunately, there is a simple way to handle the problem, although it requires the overhead of one additional function call. The approach is to use a regular member function to perform the desired task, and then use a static member function for the Xt callback. A static member function does not expect a *this* pointer when it is called. However, it is a member of a class, and as such has the same access privileges as any other member function. It can also be encapsulated so it is not visible outside the class.

The only catch is that the static member function used as a callback needs a way to access the appropriate instance of the class. This can be provided by specifying a pointer to the component as the client data when registering the callback.

Generally, you should follow these guidelines for using Xt callbacks with ViewKit components:

- Define any Xt callbacks required by a component as static member functions of that class. You normally declare these functions in the private section of the class, because they are seldom useful to derived classes.
- Pass the *this* pointer as client data to all Xt callback functions installed for widgets. Callback functions should retrieve this pointer, cast it to the expected component type, and call a corresponding member function.
- Adopt a convention of giving static member functions used as callbacks the same name as the member function they call, with the word "Callback" appended. For example, the static member function **activateCallback()** should call the member function **activate()**. This convention is simply meant to make the code easier to read and understand. If you prefer, you can use your own convention for components you create, but this convention is used by all predefined ViewKit components.
- Member functions called by static member functions are often private, but they can instead be part of the public or protected section of the class. Occasionally it's useful to declare one of these functions as virtual, thereby allowing derived classes to change the function ultimately called as a result of a callback.

For example, the constructor presented in Example 2-1 for the simple control panel component described in "Component Constructors" on page 13 omitted the setup of callback routines to handle the activation of the buttons. To implement these callbacks you must:

1. Create regular member functions to perform the tasks desired in response to the user clicking the buttons.
2. Create static member functions that retrieve the client data passed by the callback, cast it to the expected component type, and call the corresponding member function.
3. Register the static member functions as callback functions in the class constructor.

Suppose that for the control panel, you want to call the member function **StartStopPanel::start()** when the user clicks the *Start* button, and to call **StartStopPanel::stop()** when the user clicks the *Stop* button:

```
void StartStopPanel::start(Widget w, XtPointer callData)
{
  // Perform "start" function
}
void StartStopPanel::stop(Widget w, XtPointer callData)
{
  // Perform "stop" function
}
```

You should then define the **StartStopPanel::startCallback()** and **StartStopPanel::stopCallback()** static member functions as follows:

```
void StartStopPanel::startCallback(Widget w, XtPointer clientData,
                                   XtPointer callData)
{
  StartStopPanel *obj = ( StartStopPanel * ) clientData;
  obj->start(w, callData);
}

void StartStopPanel::stopCallback(Widget w, XtPointer clientData,
                                  XtPointer callData)
{
  StartStopPanel *obj = ( StartStopPanel * ) clientData;
  obj->stop(w, callData);
}
```

Finally, you need to register the static member functions as callbacks in the constructor. Remember that you must pass the *this* pointer as client data when registering the callbacks. Example 2-3 shows the updated **StartStopPanel** constructor, which installs the Xt callbacks for the buttons.

Example 2-3 Component Constructor with Xt Callbacks

```
StartStopPanel::StartStopPanel(const char *name, Widget parent):VkComponent(name)
{
  // Create an XmRowColumn widget as the component’s base widget
  // to contain the buttons. Assign the widget to the _baseWidget
  // data member.

  _baseWidget = XmCreateRowColumn ( parent, _name, NULL, 0 );
  // Set up callback to handle widget destruction

  installDestroyHandler();

  XtVaSetValues(_baseWidget, XmNorientation, XmHORIZONTAL, NULL);

  // Create all other widgets as children of the base widget.
  // Manage all child widgets.

  _startButton = XmCreatePushButton ( _baseWidget, "start", NULL, 0);
  _stopButton  = XtCreatePushButton ( _baseWidget, "stop", NULL, 0);

  XtManageChild(_startButton);
  XtManageChild(_stopButton);

  // Install static member functions as callbacks for the pushbuttons

  XtAddCallback(_startButton, XmNactivateCallback,
              &StartStopPanel::startCallback, (XtPointer) this );

  XtAddCallback(_stopButton, XmNactivateCallback,
              &StartStopPanel::stopCallback, (XtPointer) this );
}
```

Handling Component Widget Destruction

When widgets are destroyed, it's easy to leave dangling references—pointers to memory that once represented widgets, but which are no longer valid. For example, when a widget is destroyed, its children are also destroyed. It's often difficult to keep track of the references to these children, so it's fairly easy to write a program that accidentally references the widgets in a class after the widgets have already been destroyed. In some cases, applications might try to delete a widget twice, which usually causes the program to crash. Calling **XtSetValues()** or other Xt functions with a widget that's been deleted is also an error that can occur easily in this situation.

To help protect the encapsulation of ViewKit classes, **VkComponent** provides a private static member function, **widgetDestroyedCallback()**, to register as an XmNdestroyCallback for the base widget so that the component can properly handle the deletion of its base widget. This callback can't be registered automatically within the **VkComponent** constructor because derived classes have not yet created the base widget when the **VkComponent** constructor is called.

As a convenience, rather than force every derived class to install the **widgetDestroyedCallback()** function directly, **VkComponent** provides a protected **installDestroyHandler()** function that performs this task:

```
void installDestroyHandler()
```

Immediately after creating a component's base widget in a derived class, you should call **installDestroyHandler()**. For example:

```
StartStopPanel::StartStopPanel(const char *name, Widget parent) :
                                                       VkComponent(name)
{
  _baseWidget = XmCreateRowColumn ( parent, _name, NULL, 0 );
  installDestroyHandler();
  // ...
}
```

When you link your program with the debugging version of the ViewKit library, a warning is issued for any class that does not install the **widgetDestroyedCallback()** function.

The **widgetDestroyedCallback()** function calls the virtual member function **widgetDestroyed()**:

```
virtual void widgetDestroyed()
```

By default, **widgetDestroyed()** sets the component's *_baseWidget* data member to NULL. You can override this function in derived classes if you want to perform additional tasks in the event of widget destruction; however, you should always call the base class's **widgetDestroyed()** function as well.

Occasionally, you might need to remove the destroy callback installed by **installDestroyHandler()**. For example, the **VkComponent** class destructor removes the callback before destroying the widget. To do so, you can call the **removeDestroyHandler()** function:

```
void removeDestroyHandler()
```

Component Resource Support

The X resource manager is a very powerful facility for customizing both applications and individual widgets. The resource manager allows the user or programmer to modify both the appearance and behavior of applications and widgets.

ViewKit provides a variety of utilities to simplify resource management. Using ViewKit, you can easily

- set resource values for a single component or an entire class of components
- initialize data members using values retrieved from the resource database
- programmatically set default resource values for a component
- obtain resource values

For ViewKit resource support to work properly, you must follow these two guidelines:

- You must override each components's virtual **className()** member functions, returning a string that identifies the name of each component's C++ class. For example, if you create a **StartStopPanel** component class, you must override **StartStopPanel::className()** as follows:

  ```
  const char* StartStopPanel::className()
  {
    return "StartStopPanel";
  }
  ```
- You must provide a unique component name when instantiating each component. This string must be used as the name of the component's base widget. Giving each instance of a component a unique name ensures a unique path through the application's widget tree for each widget. Widgets within a component can have hard-coded names because they can be qualified by the name of the root of the component subtree.

Setting Resource Values by Class or Individual Component

The structure of ViewKit allows you to specify resource values for either an individual component or for all components of a given class.

To set a resource for an individual instance of a component, refer to the resource using the syntax:

**name*resource*

In this case, “name” refers to the ViewKit component’s name that you pass as an argument to the component’s constructor, and “resource” is the name of the resource. A specification of this form works for setting both widget resources and “synthetic” resources that you use to initialize data member values. (“Initializing Data Members Based on Resource Values” on page 27 describes a convenience function for initializing data members from resource values.)

For example, you could set a “verbose” resource to TRUE for the instance named “status” of a hypothetical **ProcessMonitor** class with a resource entry such as:

```
*status*verbose: TRUE
```

To set a resource for an entire component class, refer to the resource using the syntax:

**className*resource*

In this case, “className” is the name of the ViewKit class returned by that class’s **className()** function, and “resource” is the name of the resource. A specification of this form works for setting “synthetic” resources only, not widget resources.[1]

For example, you can set a “verbose” resource for all instances of the hypothetical **ProcessMonitor** class to TRUE with a resource entry such as:

```
*ProcessMonitor*verbose: TRUE
```

Initializing Data Members Based on Resource Values

If you want to initialize data members in a class using values in the resource database, you can call the **VkComponent** member function **getResources()**:

```
void getResources ( const XtResourceList resources,
                    const int numResources )
```

[1] You can set resources for widgets within a component when you specify a component’s name because the name of component’s base widget is the same as the name of the component; the X resource manager can successfully determine a widget hierarchy based on widget names. On the other hand, a component’s class name has no relation to its base widget’s class name. If you use a component class name in a resource specification, the X resource manager cannot determine the widget hierarchy for widgets in the component.

The *resources* argument is a standard resource specification in the form of an XtResource list, and the *numResources* argument is the number of resources. You should define the XtResource list as a static data member of the class to encapsulate the resource specification with the class. You should call **getResources()** in the component constructor after creating your component's base widget.

getResources() retrieves the specified resources relative to the root of the component's widget subtree. For example, to set the value of a resource for a particular instance of a component, you would need to set the resource with an entry in the resource database of the form:

```
*name.resource:     value
```

where *name* is the component's name, *resource* is the name of the resource, and *value* is the resource value. To set the value of a resource for an entire component class, you would need to set the resource with an entry in the resource database of the form:

```
*className.resource:     value
```

where *className* is the component class name, *resource* is the name of the resource, and *value* is the resource value.

Example 2-4 demonstrates the initialization of a data member, *_verbose*, from the resource database. A default value is specified in the XtResource structure, but the ultimate value is determined by the value of the resource named "verbose" in the resource database.

Example 2-4 Initializing a Data Member from the Resource Database

```
// Header file: ProcessMonitor.h

#include <Vk/VkComponent.h>
#include <Xm/Frame.h>

class ProcessMonitor : public VkComponent
{
private:
  static XtResource _resources[];

protected:
  Boolean _verbose;
public:
  ProcessMonitor(const char *, Widget);
  ~ProcessMonitor();
  virtual const char *className();
};

// Source file: ProcessMonitor.c++

#include "ProcessMonitor.h"

XtResource ProcessMonitor::_resources [] = {
  {
  "verbose",
  "Verbose",
  XmRBoolean,
  sizeof ( Boolean ),
  XtOffset ( ProcessMonitor *, _verbose ),
  XmRString,
  (XtPointer) "FALSE",
  },
};

ProcessMonitor::ProcessMonitor(Widget parent, const char *name):VkComponent(name)
{
  _baseWidget = XtVaCreateWidget ( _name, xmFrameWidgetClass,
                                   parent, NULL ) ;
  installDestroyHandler();

  // Initialize members from resource database

  getResources ( _resources, XtNumber(_resources) );

  // ...
}
```

So, to initialize the *_verbose* data member to TRUE in all instances of the **ProcessMonitor** class, you need only to set the following resource in the resource database:

```
*ProcessMonitor.verbose: TRUE
```

To initialize *_verbose* to TRUE for an instance of **ProcessMonitor** named *conversionMonitor*, you could set the following resource in the resource database:

```
*conversionMonitor.verbose:  TRUE
```

Setting Default Resource Values for a Component

Often, you might want to specify default resource values for a component. A common way to accomplish this is to put the resource values in an application resource file. However, this makes the component dependent on that resource file; to use that component in another application, you must remember to copy those resources into the new application's resource file. This is especially inconvenient for classes that you reuse in multiple applications.

A better method of encapsulating default resources into a component is to use a ViewKit facility that allows you to specify them programmatically and then merge them into the resource database during execution. Although the resources are specified programmatically, they can be overridden by applications that use the class, or by end users in resource files. However, the default values are specified by the component class and cannot be separated from the class accidentally. If you later want to change the implementation of a component class, you can also change the resource defaults when necessary, knowing that applications that use the class will receive both changes simultaneously.

The **VkComponent** class provides the **setDefaultResources()** function for storing a collection of default resources in the application's resource database. The resources are loaded with the lowest precedence, so that these resources are true defaults. They can be overridden easily in any resource file. You should call this function in the component constructor before creating the base widget in case any resources apply to the component's base widget.

The **setDefaultResources()** function has the following syntax:

```
void setDefaultResources ( const Widget w,
                           const String *resourceSpec )
```

The first argument is a widget; you should always use the parent widget passed in the component's constructor.

The second argument is a NULL-terminated array of strings, written in the style of an X resource database specification. Specify all resources in the list relative to the root of the component's base widget, but do not include the name of the base widget. If you want to apply a resource to the base widget, simply use the name of the resource preceded by a "*" character. When resources are loaded, the value of *_name* is prefixed to all entries, unless that entry begins with the "-" character. As long as you use unique names for each component that you create of a given class, this results in resource specifications unique to each component. If you precede a resource value in this list with a "-" character, **setDefaultResources()** does not qualify the resource with the value of *_name*. This is useful in rare situations where you want to add global resources to the database.

You should declare the resource list as a static data member of the class. This encapsulates the set of resources with the class.

Note: Generally, setting resources using **setDefaultResources()** is most appropriate for components that you plan to reuse in multiple applications. In particular, it is a good method for setting resources for widget labels and other strings that your component displays. You should not use **setDefaultResources()** to set widget resources, such as orientation, that you would normally set programmatically. Typically you don't need to change these resources when you use the component in different applications, and so you save memory and execution time by not using **setDefaultResources()** to set these resources.

Example 2-5 builds on the **StartStopPanel** constructor from Example 2-3 to specify the default label strings "Start" and "Stop" for the button widgets.

Example 2-5 Setting a Component's Default Resource Values

```
// StartStopPanel.h

class StartStopPanel: public VkComponent {

  public:
    StartStopPanel (const char *, Widget);
    ~StartStopPanel();
    // ...

  private:
    static String _defaultResources[];
    // ...
}
```

```
// StatStopPanel.c++

String StartStopPanel::_defaultResources[] = {
  "*start.labelString:  Start",
  "*stop.labelString:   Stop",
  NULL
};

StartStopPanel::StartStopPanel(const char *name, Widget parent):VkComponent(name)
{
  // Load class-default resources for this object before creating base widget

  setDefaultResources(parent, _defaultResources );

  _baseWidget = XmCreateRowColumn ( parent, _name, NULL, 0 );

  installDestroyHandler();

  XtVaSetValues(_baseWidget, XmNorientation, XmHORIZONTAL, NULL);

  _startButton = XmCreatePushButton ( _baseWidget, "start", NULL, 0);
  _stopButton  = XtCreatePushButton ( _baseWidget, "stop", NULL, 0);

  // ...
}
```

Convenience Function for Retrieving Resource Values

ViewKit also provides **VkGetResource()**, a convenience function for retrieving resource values from the resource database. **VkGetResource()** is *not* a member function of any class. You must include the header file <*Vk/VkResource.h*> to use **VkGetResource()**.

VkGetResource() has two forms. The first is:

```
char * VkGetResource( const char * name,
                      const char * className )
```

This form returns a character string containing the value of the application resource you specify by name and class name. This function is similar to **XGetDefault(**3x**)** except that this form of **VkGetResource()** allows you to retrieve the resource by class name whereas **XGetDefault()** does not.

Note: Do not attempt to change or delete the value returned by **VkGetResource()**.

The second form of **VkGetResource()** is:

```
XtPointer VkGetResource( Widget w,
                         const char *name,
                         const char *className,
                         const char *desiredType,
                         const char *defaultValue)
```

This second form is similar to **XtGetSubresource(**3Xt**)** in that it allows you to retrieve a resource relative to a specific widget. You can specify the resource as a dot-separated list of names and classes, allowing you to retrieve "virtual" sub-resources. You can also specify a target type. **VkGetResource()** will convert the retrieved value, or the default value if no value is retrieved, to the specified type.

Note: Do not attempt to change or delete the value returned by **VkGetResource()**.

For example, suppose that you want to design an application for drawing an image and you want to allow the user to select various aspects of the style in which the image is drawn, such as color and fill pattern (a pixmap). You could specify each aspect of each style as a resource and retrieve the values as follows:

```
Widget canvas = XmCreateDrawingArea(parent, "canvas", NULL, 0);
Pixel fgOne = (Pixel) VkGetResource(canvas,
                                    "styleOne.foreground", "Style.Foreground",
                                    XmRString, "Black");
Pixel fgTwo = (Pixel) VkGetResource(canvas,
                                    "styleTwo.foreground", "Style.Foreground",
                                    XmRString, "Black");

Pixel bgOne = (Pixel) VkGetResource(canvas,
                                    "styleOne.background", "Style.Background",
                                    XmRString, "White");
Pixel bgTwo = (Pixel) VkGetResource(canvas,
                                    "styleTwo.background", "Style.Background",
                                    XmRString, "White");

Pixmap pixOne = (Pixmap) VkGetResource(canvas,
                                       "styleOne.pixmap", "Style.Pixmap",
                                       XmRString, "background");
Pixmap pixTwo = (Pixmap) VkGetResource(canvas,
                                       "styleTwo.pixmap", "Style.Pixmap",
                                       XmRString, "background");
```

Another common technique used in ViewKit programming is to use a string to search for resource value and, if no resource exists, use the string as the value. You can do this easily if you pass the string to **VkGetResource()** as the default value. For example, consider the following code:

```
char *timeMsg = "Time";
// ...
char *timeTitle = (char *) VkGetResource(_baseWidget, timeMsg, "Time",
                                          XmRString, timeMsg);
```

In this case, **VkGetResource()** searches for a resource (relative to the *_baseWidget* widget) whose name is specified by the character string *timeMsg*. If no such resource exists, **VkGetResource()** returns the value of *timeMsg* as the default value.

If you use this technique, you should not pass a string that contains a embedded spaces or newlines.

ViewKit Callback Support

All ViewKit components support ViewKit member function callbacks (also referred to simply as *ViewKit callbacks*). ViewKit callbacks are analogous to Xt-style callbacks supported by widget sets, but ViewKit callbacks are in no way related to Xt.

The ViewKit callback mechanism allows a component to define conditions or events, the names of which are exported as public static string constants encapsulated by that component. Any other component can register any of its member functions to be called when the condition or event associated with that callback occurs.

Unlike the case when registering ViewKit functions for Xt-style callbacks, the functions you register for ViewKit callbacks must be regular member functions, not static member functions.

ViewKit callbacks are implemented by the **VkCallbackObject** class. **VkComponent** is derived from **VkCallbackObject**, so all ViewKit components can use ViewKit callbacks. If you create a class for use with a ViewKit application, that class must be derived from **VkCallbackObject** or one of its subclasses (such as **VkComponent**) for you to be able to use ViewKit callbacks with that class.

Registering ViewKit Callbacks

The **addCallback()** function defined in **VkCallbackObject** registers a member function to be called when the condition or event associated with a callback occurs.

Caution: When registering a ViewKit callback, remember to call the **addCallback()** member function of the object that triggers the callback, not the object that is registering the callback.

The format of **addCallback()** for registering a member function is:

```
void addCallback(const char *name,
                 VkCallbackObject *component,
                 VkCallbackMethod callbackFunction,
                 void *clientData = NULL)
```

The arguments for this function are:

name
: The name of the ViewKit callback. You should always use the name of the public static string constant for the appropriate callback, not a literal string constant. (For example, use *VkComponent::deleteCallback*, not "deleteCallback".) This allows the compiler to catch any misspellings of callback names.

component
: A pointer to the object registering the callback function.

callbackFunction
: The member function to invoke when the condition or event associated with that callback occurs.

clientData
: A pointer to data to pass to the callback function when it is invoked.

For example, consider a member of a hypothetical **Display** class that instantiates another hypothetical component class, **Control**. The code fragment below registers a function to be invoked when the value set by the **Control** object changes and the **Control** object triggers its *valueChanged* callback:

```
Display::createControl()
{
  _control = new Control(_baseWidget, "control");
  _control->addCallback(Control::valueChanged, this,
                      (VkCallbackMethod) &Display::newValue);
}
```

In this example, the **Display** object requests that when the **Control** object triggers its *valueChanged* callback, it should call the *Display::newValue()* function of the **Display** object that created the **Control** object. The "(VkCallbackMethod)" cast for the callback function is required.

All ViewKit callback functions must have the form:

```
void memberFunctionCallback(VkCallbackObject *obj,
                            void *clientData,
                            void *callData)
```

The *obj* argument is the component that triggered the callback, which you must cast to the correct type to allow access to members provided by that class. The *clientData* argument is the optional client data specified when you registered the callback, and the *callData* argument is optional data supplied by the component that triggered the callback.

For example, you would define the **Display::newValue()** callback method used above as follows:

```
class Display : VkComponent {
  private:
    void newValue(VkCallbackObject *, void *, void *);
    // ...
};

void Display::newValue(VkCallbackObject* obj,
                       void *clientData,
                       void *callData);
{
  Control *controlObj = (Control *) obj;
  // Perform whatever operation that is needed to update
  // the Display object.  You can also access member
  // functions from the Control object (controlObj).
  // The clientData argument contains any information
  // you provided as clientData when you registered
  // this callback; cast it to the proper type to use it.
  // If the Control object passed the new value as the
  // callData argument, you can cast that to the proper
  // type and use it.
}
```

There is also a version of **addCallback()** for registering non-member functions. It's syntax is:

```
void addCallback(const char *name,
                 VkCallbackFunction callbackFunction,
                 void *clientData = NULL)
```

The arguments for this version are:

name
: The name of the ViewKit callback. You should always use the name of the public static string constant for the appropriate callback, not a literal string constant.

callbackFunction
: The non-member function to invoke when the condition or event associated with that callback occurs.

clientData
: A pointer to data to pass to the callback function when it is invoked.

The form of your non-member ViewKit callback functions must be:

```
void functionCallback(VkCallbackObject *obj,
                      void *clientData,
                      void *callData)
```

For example, if you have a non-member function **errorCondition()**:

```
void errorCondition(VkCallbackObject *obj,
                    void *clientData,
                    void *callData)
{
    // Handle error condition
}
```

You could register it for a ViewKit callback with the line such as

```
sample->addCallback(SampleComponent::errorCallback,
                    (VkCallbackFunction) &errorCondition);
```

The “(VkCallbackFunction)” cast for the callback function is required.

Removing ViewKit Callbacks

The **removeCallback()** function provided by the **VkCallbackObject** class removes previously registered callbacks. The following version of **removeCallback()** removes a member function registered as a callback:

```
void removeCallback(char *name,
                    VkCallbackObject *otherObject,
                    VkCallbackMethod memberFunction,
                    void *clientData = NULL)
```

The following version of **removeCallback()** removes a non-member function registered as a callback:

```
void removeCallback(const char *name,
                    VkCallbackFunction callbackFunction,
                    void *clientData = NULL)
```

To remove a callback, you must provide the same arguments specified when you registered the callback. For example, the following line removes the **Control** callback registered in the previous section:

```
_control->removeCallback(Control::valueChanged, this,
                      (VkCallbackMethod) &Display::newValue);
```

The **removeAllCallbacks()** function removes multiple ViewKit callbacks:

```
void removeAllCallbacks()
void removeAllCallbacks(VkCallbackObject *obj)
```

If you don’t provide an argument, this function removes all callbacks from an object, regardless of which components registered the callbacks. If you provide a pointer to a component, **removeAllCallbacks()** removes from an object all ViewKit callbacks that were set by the specified component. For example, the following would remove from the **Control** object *_control* all callbacks that the **Display** object had set:

```
_control->removeAllCallbacks(this);
```

Defining and Triggering ViewKit Callbacks

To create a ViewKit callback for a component class, define a public static string constant as the name of the callback. For clarity, you should use the string’s name as its value. For example, the following defines a callback, *StartStopPanel::actionCallback*, for the hypothetical **StartStopPanel** class discussed earlier in this chapter:

```
class StartStopPanel : public VkComponent {

  public:
    static const char *const actionCallback;
    // ...
}

const char *const StartStopPanel::actionCallback = "actionCallback";
```

The **callCallbacks()** member function triggers a specified callback, invoking all member functions registered for that callback:

```
callCallbacks(const char *callback, void *callData)
```

The first argument specifies the name of the callback. You should always use the name of the public static string constant for the appropriate callback, not a literal string constant. (For example, use *StartStopPanel::startCallback*, not “startCallback”.) This allows the compiler to catch any misspellings of callback names.

The second argument is used to supply any additional data that might be required.

For example, you could define the **StartStopPanel::start()** and **StartStopPanel::stop()** functions to trigger the *actionCallback* and pass an enumerated value as call data to indicate which button the user clicked:

```
enum PanelAction { START, STOP };

class StartStopPanel : public VkComponent {

  public:
    static const char *const actionCallback;
    // ...
}
```

```
const char *const StartStopPanel::actionCallback = "actionCallback";

void StartStopPanel::start(Widget w, XtPointer callData)
{
  callCallbacks(actionCallback, (void *) START);
}

void StartStopPanel::stop(Widget w, XtPointer callData)
{
  callCallbacks(actionCallback, (void *) STOP);
};
```

Predefined ViewKit Callbacks

The **VkComponent** class, and therefore all derived classes, includes the ViewKit callback *deleteCallback*, which is invoked when the component's destructor is called. You can use this callback to prevent dangling pointers when maintaining pointers to other components. The code fragment in Example 2-6 shows an example of this.

Example 2-6 Using the Predefined *deleteCallback* ViewKit Callback

```
class MainComponent : VkComponent {
  // ...
  AuxComponent *_aux;
  void createAux();
  void auxDeleted(VkCallbackObject *, void *, void *);
  // ...
};

// ...

void MainComponent::createAux()
{
_aux = new AuxComponent(_baseWidget, "auxilliary");
_aux->addCallback(VkComponent::deleteCallback, this,
              (VkCallbackMethod) &MainComponent::auxDeleted);
}

void MainComponent::auxDeleted(VkCallbackObject*,
                               void *, void *)
{
  _aux = NULL;
}
```

In the function **MainComponent::createAux()**, the **MainComponent** class creates an instance of the **AuxComponent** and then immediately registers **MainComponent::auxDeleted()** as a callback to be invoked when the **AuxComponent** object is deleted.

The **auxDeleted()** callback definition simply assigns NULL to the **AuxComponent** object pointer. All other **MainComponent** functions should test the value of *_aux* to ensure that it is not NULL before attempting to use the **AuxComponent** object. This eliminates the possibility that the **MainComponent** class would try to access the **AuxComponent** object after deleting it, or attempting to delete it a second time.

In most cases you should not need to use this technique of registering *deleteCallback* callbacks. It is necessary only if you need to create multiple pointers to a single object. In general, you should avoid multiple pointers to the same object, but *VkComponent::deleteCallback* provides a way to control situations in which you must violate this guideline.

Deriving Subclasses to Create New Components

This section demonstrates how to use the **VkComponent** class to create new components. It includes guidelines to follow when creating new components, an example of creating a new component, and an example of subclassing that component to create yet another component class.

Subclassing Summary

The following is a summary of guidelines for writing components based on the **VkComponent** class:

- Encapsulate all of your component’s widgets in a single-rooted subtree. While some extremely simple components might contain only a single widget, the majority of components must create some type of container widget as the root of the component’s widget subtree; all other widgets are descendents of this one.
- When you create your class’s base widget, assign it to the *_baseWidget* data member inherited from the **VkComponent** class.

- In most cases, create a component's base widget and all other widgets in the class constructor. The constructor should manage all widgets except the base widget, which should be left unmanaged. You can then manage or unmanage a component's entire widget subtree using the **show()** and **hide()** member functions.
- Accept at least two arguments in your component's constructor: a string to be used as the name of the base widget, and a widget to be used as the parent of the component's base widget. Pass the name argument to the **VkComponent** constructor, which makes a copy of the string. Refer to a component's name using the *_name* member inherited from **VkComponent** or the **name()** access function. Refer to a component's base widget using the *_baseWidget* member inherited from **VkComponent** or the **baseWidget()** access function.
- Override the virtual **className()** member function for your component classes to return a string consisting of the name of the component's C++ class.
- Define all Xt callbacks required by a component class as private static member functions. In exceptional cases, you might want to declare them as protected so that derived classes can access them.
- Pass the *this* pointer as client data to all Xt callback functions. Callback functions should retrieve this pointer, cast it to the expected component type and call a corresponding member function. For clarity, use the convention of giving static member functions used as callbacks the same name as the member function they call, with the word "Callback" appended. For example, name a static member function **startCallback()** if it calls the member function **start()**.
- Call **installDestroyHandler()** immediately after creating a component's base widget.
- If you need to specify default resources for a component class, call the function **setDefaultResources()** with an appropriate resource list before creating the component's base widget.
- If you need to initialize data members from values in the resource database, define an appropriate resource specification and call the function **getResources()** immediately after creating the component's base widget.

Example of Creating a New Component

To illustrate many of the features of the **VkComponent** base class, this chapter has shown how to build a simple class called **StartStopPanel**, which implements a control panel containing two buttons. Figure 2-2 shows the default appearance of a **StartStopPanel** object.

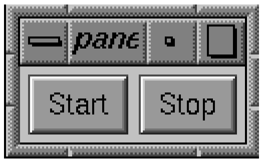


Figure 2-2 The Default Appearance of a **StartStopPanel** Component

Example 2-7 lists the full implementation of this class.

Example 2-7 A Simple User-Defined Component

```
//////////////////////////////////////////////////////////////
// StartStopPanel.h
//////////////////////////////////////////////////////////////

#ifndef _STARTSTOPPANEL_H
#define _STARTSTOPPANEL_H
#include <Vk/VkComponent.h>

enum PanelAction { START, STOP };

class StartStopPanel : public VkComponent {
  public:
    StartStopPanel (const char *, Widget);
    ~StartStopPanel();
    virtual const char *className();

    static const char *const actionCallback;

  protected:
    virtual void start(Widget, XtPointer);
    virtual void stop(Widget, XtPointer);

    Widget _startButton;
    Widget _stopButton;
  private:
    static void startCallback(Widget, XtPointer, XtPointer);
    static void stopCallback(Widget, XtPointer, XtPointer);
    static String _defaultResources[];
};

#endif
/////////////////////////////////////////////////////////////
// StartStopPanel.c++
/////////////////////////////////////////////////////////////

#include "StartStopPanel.h"
#include <Xm/RowColumn.h>
#include <Xm/PushB.h>

// These are default resources for widgets in objects of this class
// All resources will be prefixed by *<name> at instantiation,
// where <name> is the name of the specific instance, as well as the
// name of the baseWidget. These are only defaults, and may be
// overriden in a resource file by providing a more specific resource
// name

String StartStopPanel::_defaultResources[] = {
  "*start.labelString:  Start",
  "*stop.labelString:   Stop",
  NULL
};

const char *const StartStopPanel::actionCallback = "actionCallback";

StartStopPanel::StartStopPanel(const char *name, Widget parent):VkComponent(name)
{
  // Load class-default resources for this object before creating base widget

  setDefaultResources(parent, _defaultResources );

  // Create an XmRowColumn widget as the component’s base widget
  // to contain the buttons. Assign the widget to the _baseWidget
  // data member.

  _baseWidget = XmCreateRowColumn ( parent, _name, NULL, 0 );
  // Set up callback to handle widget destruction

  installDestroyHandler();

  XtVaSetValues(_baseWidget, XmNorientation, XmHORIZONTAL, NULL);

  // Create all other widgets as children of the base widget.
  // Manage all child widgets.

  _startButton = XmCreatePushButton ( _baseWidget, "start", NULL, 0);
  _stopButton  = XmCreatePushButton ( _baseWidget, "stop", NULL, 0);

  XtManageChild(_startButton);
  XtManageChild(_stopButton);

  // Install static member functions as callbacks for the buttons

  XtAddCallback(_startButton, XmNactivateCallback,
                &StartStopPanel::startCallback, (XtPointer) this );

  XtAddCallback(_stopButton, XmNactivateCallback,
                &StartStopPanel::stopCallback, (XtPointer) this );
}

StartStopPanel::~StartStopPanel()
{
  // Empty
}

const char* StartStopPanel::className()
{
  return "StartStopPanel";
}

void StartStopPanel::startCallback(Widget w, XtPointer clientData,
                                   XtPointer callData)
{
  StartStopPanel *obj = ( StartStopPanel * ) clientData;
  obj->start(w, callData);
}
```

```
void StartStopPanel::stopCallback(Widget w, XtPointer clientData,
                                  XtPointer callData)
{
  StartStopPanel *obj = ( StartStopPanel * ) clientData;
  obj->stop(w, callData);
}
void StartStopPanel::start(Widget, XtPointer)
{
  callCallbacks(actionCallback, (void *) START);
}
void StartStopPanel::stop(Widget, XtPointer)
{
  callCallbacks(actionCallback, (void *) STOP);
}
```

Examples of Using and Subclassing a Component Class

Example 2-7 slightly changes the **StartStopPanel** class from previous examples by declaring the member function **StartStopPanel::start()** and **StartStopPanel::stop()** as virtual functions. This allows you to use the **StartStopPanel** in two different ways: using the component directly and subclassing the component.

Example of Using a Component Class Directly

The simplest way to use the **StartStopPanel** class is to register callbacks with *StartStopPanel::actionCallback*. To do so, instantiate a **StartStopPanel** object in your application and register as a callback a member function that tests the value of the call data and performs some operation based on the value. This option avoids the additional work required to create a subclass of **StartStopPanel**. This technique of using a component class is most appropriate if the class already has all the functionality you require.

Example 2-8 shows a simple example of using the **StartStopPanel** directly. The **PanelWindow** class is a simple subclass of the **VkSimpleWindow** class, which is discussed in Chapter 4, “ViewKit Windows.” It performs the following activities in its constructor:

1. It instantiates a **StartStopPanel** object named “controlPanel” and assigns it to the *_controlPanel* variable.
1. It specifies a vertical orientation for the **StartStopPanel** object.
2. It installs **PanelWindow::statusChanged()** as a ViewKit callback function to be called whenever *StartStopPanel::actionCallback* triggers. In this example, **PanelWindow::statusChanged()** simply prints a status message to standard output whenever it is called.
3. It installs the *_controlPanel* object as the window’s “view.” Showing the **PanelWindow** object will now display the *_controlPanel* object. (“Creating the Window Interface” on page 85 describes how to create window interfaces.)

Example 2-8 Example of Using a Component Directly

```
//////////////////////////////////////////////////////////////
// PanelWindow.h
//////////////////////////////////////////////////////////////

#ifndef _PANELWINDOW_H
#define _PANELWINDOW_H

#include "StartStopPanel.h"
#include <Vk/VkSimpleWindow.h>

// Define a top-level window class

class PanelWindow: public VkSimpleWindow {

  public:
    PanelWindow(const char *name);
    ~PanelWindow();
    virtual const char* className();
  protected:
    void statusChanged(VkCallbackObject *, void *, void *);

    StartStopPanel * _controlPanel;
};

#endif

//////////////////////////////////////////////////////////////
// PanelWindow.c++
//////////////////////////////////////////////////////////////

#include "PanelWindow.h"
#include <iostream.h>

PanelWindow::PanelWindow(const char *name) : VkSimpleWindow (name)
{
    _controlPanel = new StartStopPanel( "controlPanel",
                                        mainWindowWidget() );

    XtVaSetValues(_controlPanel->baseWidget(),
                  XmNorientation, XmVERTICAL, NULL);

    _controlPanel->addCallback( StartStopPanel::actionCallback, this,
                         (VkCallbackMethod) &PanelWindow::statusChanged );

    addView(_controlPanel);
}

const char * PanelWindow::className()
{
    return "PanelWindow";
}

PanelWindow::~PanelWindow()
{
    // Empty
}
```

```
void PanelWindow::statusChanged(VkCallbackObject *obj,
                                void *, void *callData)
{

    StartStopPanel * panel = (StartStopPanel *) obj;
    PanelAction action = (PanelAction) callData;
    switch (action) {
        case START:
            cout << "Process started\n" << flush;
            break;
        case STOP:
            cout << "Process stopped\n" << flush;
            break;
        default:
            cout << "Undefined state\n" << flush;
    }
}
```

The following simple program displays the resulting **PanelWindow** object (Chapter 3, "The ViewKit Application Class," discusses the **VkApp** class):

```
//////////////////////////////////////////////////////////////
// PanelTest.c++
//////////////////////////////////////////////////////////////

#include <Vk/VkApp.h>
#include "PanelWindow.h"

// Main driver. Just instantiate a VkApp and the PanelWindow,
// "show" the window and then "run" the application.

void main ( int argc, char **argv )
{
    VkApp        *panelApp = new VkApp("panelApp", &argc, argv);
    PanelWindow  *panelWin = new PanelWindow("panelWin");

    panelWin->show();
    panelApp->run();
}
```

Figure 2-3 shows the resulting **PanelWindow** window displayed by this program.

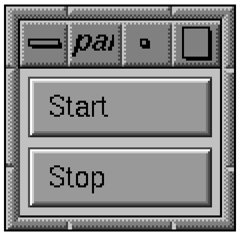


Figure 2-3 The Resulting **PanelWindow** Window

Example of Using a Component Class by Subclassing

Another way to use the **StartStopPanel** class is to derive a subclass and override the **StartStopPanel::start()** and **StartStopPanel::stop()** functions. This technique of using a component class is most appropriate if you need to expand or modify a component's action in some way.

Example 2-9 creates **ControlPanel**, a subclass of **StartStopPanel** that incorporates the features implemented in the **PanelWindow** class shown in Example 2-8.

Example 2-9 Example of Subclassing a Component

```
//////////////////////////////////////////////////////////////
// ControlPanel.h
//////////////////////////////////////////////////////////////

#ifndef _CONTROLPANEL_H
#define _CONTROLPANEL_H
#include "StartStopPanel.h"

class ControlPanel : public StartStopPanel {

  public:
    ControlPanel (const char *, Widget);
    ~ControlPanel();
    virtual const char *className();
  protected:
    virtual void start(Widget, XtPointer);
    virtual void stop(Widget, XtPointer);
};
#endif
```

```
//////////////////////////////////////////////////////////////
// ControlPanel.c++
//////////////////////////////////////////////////////////////

#include "ControlPanel.h"
#include <iostream.h>
ControlPanel::ControlPanel (const char *name , Widget parent) :
                                             StartStopPanel (name, parent)
{
    XtVaSetValues(_baseWidget, XmNorientation, XmVERTICAL, NULL);
}

ControlPanel::~ControlPanel()
{
    // Empty
}


const char* ControlPanel::className()
{
    return "ControlPanel";
}

void ControlPanel::start(Widget w, XtPointer callData)
{
    cout << "Process started\n" << flush;
    StartStopPanel::start(w, callData);
}

void ControlPanel::stop(Widget w, XtPointer callData)
{
    cout << "Process stopped\n" << flush;
    StartStopPanel::stop(w, callData);
}
```

The **ControlPanel** constructor uses the **StartStopPanel** constructor to initialize the component, creating the widgets and initializing the component's data members. Then, the **ControlPanel** constructor sets the orientation resource of the RowColumn widget, which is the component's base widget, to VERTICAL.

The **ControlPanel** class also overrides the virtual functions **start()** and **stop()** to perform the actions handled previously by the **PanelWindow** class. After performing these actions, the **ControlPanel::start()** and **ControlPanel::stop()** functions call **StartStopPanel::start()** and **StartStopPanel::stop()** respectively. While this may seem unnecessary for an example this simple, it helps preserve the encapsulation of the classes. You could now change the implementation of the **StartStopPanel** class, perhaps adding a status indicator to the component that the **StartStopPanel::start()** and **StartStopPanel::stop()** functions would update, and you would not have to change the **start()** and **stop()** function definitions in derived classes such as **ControlPanel**.

The following simple example creates a **VkSimpleWindow** object, adds a **ControlPanel** as the window's view, and then displays the window:

```
//////////////////////////////////////////////////////////////
// PanelTest2.c++
//////////////////////////////////////////////////////////////

#include <Vk/VkApp.h>
#include <Vk/VkSimpleWindow.h>
#include "ControlPanel.h"

// Main driver. Instantiate a VkApp, a VkSimpleWindow, and a
// ControlPanel, add the ControlPanel as the SimpleWindow's view,
// "show" the window and then "run" the application.

void main ( int argc, char **argv )
{
    VkApp *panelApp = new VkApp("panel2App", &argc, argv);
    VkSimpleWindow *panelWin = new VkSimpleWindow("panelWin");
    ControlPanel *control = new ControlPanel("control",
                                             panelWin->mainWindowWidget() );
    panelWin->addView(control);
    panelWin->show();
    panelApp->run();
}
```

Chapter 3

The ViewKit Application Class

This chapter describes the **VkApp** class, which handles application-level tasks such as Xt initialization, event handling, window management, cursor control, and application busy states. Figure 3-1 shows the inheritance graph for **VkApp** and an auxiliary class, **VkCursorList**.

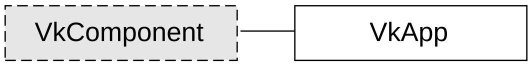


Figure 3-1 The Inheritance Graph for **VkApp**

Overview of the VkApp Class

The **VkApp** class, derived from the **VkComponent** class, provides facilities required by all ViewKit applications. In all of your ViewKit applications you must create a single instance of **VkApp** or a class derived from **VkApp**.

The primary responsibility of **VkApp** is to handle the initialization and event-handling operations common to all Xt-based applications. When you write a ViewKit application, instead of calling Xt functions such as **XtAppInitialize(3Xt)** and **XtAppMainLoop(3Xt)**, you simply instantiate and use a **VkApp** object.

The **VkApp** class also provides support for other application-level tasks. For example, **VkApp** provides functions for quitting your application; showing, hiding, iconifying, and opening all of the application’s windows; handling application busy states; maintaining product version information; and setting the aapplication’s cursor shape.

The **VkApp** class also stores some essential information that can be accessed throughout an application. This information includes: a pointer to the X Display structure associated with the application’s connection to the server; the XtAppContext structure required by many Xt functions; the application’s name; and the application’s class name. This information is maintained in the private portion of the class and is available through public access functions.

The VkApp Constructor

In all ViewKit applications you must create a single instance of the **VkApp** class (or a derived class) before instantiating any other ViewKit objects.The **VkApp** constructor initializes the Xt Intrinsics and creates a shell, which is never visible, to serve as the parent for all of the application's main windows. ViewKit supports a commonly-used multi-shell architecture as described in the book *X Window System Toolkit* (Asente and Swick, 1990). ViewKit creates all windows (using the **VkSimpleWindow** and **VkWindow** classes described in Chapter 4, "ViewKit Windows") as popup children of the shell created by **VkApp**.

When you create an instance of the **VkApp** class, the constructor assigns a pointer to the **VkApp** object to the global variable *theApplication*. The *<Vk/VkApp.h>* header file declares this global variable as follows:

```
extern VkApp *theApplication;
```

As a result, the *theApplication* pointer is available in any file that includes the *<Vk/VkApp.h>* header file. This provides easy use of **VkApp**'s facilities and data throughout your program.

The syntax of the **VkApp** constructor is:

```
VkApp(char *appClassName, int *argc, char **argv,
      XrmOptionDescRec *options = NULL,
      int numOptions = 0)
```

The *appClassName* argument designates the application class name, which is used when loading application resources. Note that **VkApp** differs from other ViewKit components in that you provide the application class name as an argument to the constructor rather than overriding the **className()** function. This allows you to set the application class name without creating a subclass of **VkApp**.**VkApp** also differs from other ViewKit components in that you do not provide a component name in the constructor; instead, ViewKit uses the command that you used to invoke your application (*argv[0]*) as the component name.

The second and third arguments to the **VkApp** constructor must be pointers to *argc* and the application's *argv* array. The **VkApp** constructor passes these arguments to **XtOpenDisplay(3Xt)**, which parses the command line according to the standard Xt command-line options, loads recognized options into the application's resource database, and modifies *argc* and *argv* to remove all recognized options.

You can specify additional command-line options to parse by passing an XrmOptionDescRec(3Xt) table as the *options* argument and specifying the number of entries in the table with the *numOptions* argument. This is sufficient for setting simple resource values from the command line; however, if you want to set application-level variables using either the command line or resource values, you should:

1. Derive a subclass of **VkApp**.
2. Use the protected member function **VkApp::parseCommandLine()** to parse command-line options.
3. Use **getResources()** to set the variables based on resource values.

This process is illustrated in Example 3-6 in "Deriving Classes from VkApp" on page 77.

Running ViewKit Applications

Once you have instantiated a **VkApp** object and set up your program's interface, call **VkApp::run()**:

```
virtual void run()
```

The **run()** function enters a custom main loop that supports dispatching raw events in addition to the normal Xt event handling. See "ViewKit Event Handling" on page 56 for more information on event handling.

Note: Do not call **XtMainLoop(3Xt)** or **XtAppMainLoop(3Xt)** in a ViewKit application.

Example 3-1 illustrates the typical use of the **VkApp** class in the main body of a ViewKit program.

Example 3-1 Typical Use of **VkApp** in a ViewKit Program

```
#include <Vk/VkApp.h>

// Application-specific setup

void main ( int argc, char **argv )
{
    VkApp   *myApp = new VkApp("MyApp", &argc, argv);
    // Application-specific code

    myApp->run();  // Run the application
}
```

ViewKit Event Handling

The **VkApp::run()** function is ViewKit's main event loop. **run()** implements the event handling normally supported by **XtAppMainLoop()** or **XtMainLoop()**. **run()** calls **run_first()** to do some internal initialization, and then enters a main loop that dispatches application events, raw X events, and normal Xt events. **run()** also allows for customized event handling. See "Customizing Event Handling" for more information.

Additionally, **run()** also supports events not normally handled by the Xt dispatch mechanism. For example, **run()** can handle events registered for non-widgets (such as a PropertyNotify event on the root window).

When **run()** receives an event not handled by the Xt dispatch mechanism, it calls the virtual function **VkApp::handleRawEvent()**:

`virtual void handleRawEvent(XEvent *`*event*`)`

The default action of **VkApp::handleRawEvent()** is to pass the event to the **handleRawEvent()** function of each instance of **VkSimpleWindow** (or subclass) in the application. By default, these members function are empty.

If you want to handle events through this mechanism, call **XSelectInput(3X)** to select the events that you want to receive, and override **handleRawEvent()** in a **VkApp** or **VkSimpleWindow** subclass to implement your event processing. Generally, in keeping with object-oriented practice, you should override **handleRawEvent()** in a **VkSimpleWindow** subclass rather than a **VkApp** subclass, unless your event processing has an application-wide effect. If you override

VkApp::handleRawEvent() in a derived class, call the base class's **handleRawEvent()** function after performing your event processing.

Note: If you explicitly call **XtNextEvent(3Xt)** and **XtDispatchEvent(3Xt)** in your application, you should pass any undispatched events to **handleRawEvent()**.

In addition to the automatic event dispatching provided by **run()**, you can force ViewKit to handle all pending events immediately by calling **VkApp::handlePendingEvents()**:

```
virtual void handlePendingEvents()
```

This function retrieves and dispatches all X events as long as there are events pending. Unlike **XmUpdateDisplay(3Xm)**, which handles only Expose events, **handlePendingEvents()** handles all events. In other words, **handlePendingEvents()** does not just refresh windows, it also handles all pending events including user input. You might want to call this function periodically to process events during a time-consuming task.

handlePendingEvents(), like **run()** can also be customized. See "Customizing Event Handling" for more information.

Customizing Event Handling

If you want to customize your application's event handling, you do not need to override **run()**. In fact, overriding **run()** is strongly discouraged. You can customize event handling in any of the following ways:

- use standard X mechanisms to add event handlers
- use one or more workprocs
- maintain your own queue of all that you need to do, and then dispatch that work in a single workproc
- use **run(Boolean(***appEventHandler***) (***XEvent &***))** to provide custom event handling

run(Boolean(*appEventHandler***) (***XEvent &***))** is the only safe way to customize ViewKit's event loop. It allows you to customize the event loop without taking responsibility for the entire process.

Each time through the event loop, before doing any event processing of its own, **run()** calls **appEventHandler()** with the event. **appEventHandler()** can then handle the event completely, handle it partially, or not handle it at all. If **appEventHandler()** has completely handled the event, it returns TRUE and no further handling of that event occurs. If the application decides not to handle the event, or if more handling is needed, then **appEventHandler()** returns FALSE and **run()** finishes the job.

If you really must override **run()**, then your application must call **run_first()** before entering its own event loop.

Much like **run(), handlePendingEvents()** can be customized by calling **handlePendingEvents(Boolean(***appEventHandler***)(***XEvent &***))**.

Quitting ViewKit Applications

If you want to exit a ViewKit application, but also want to give other parts of the application the option to abort the shutdown if necessary, call **VkApp::quitYourself()**:

```
virtual void quitYourself()
```

VkApp::quitYourself() calls the **okToQuit()** function for each top-level **VkSimpleWindow** (or subclass). All windows that return TRUE are deleted; however, if the **okToQuit()** function of any window returns FALSE, the shutdown is terminated and the windows returning FALSE are not deleted. **quitYourself()** queries the windows in the reverse order in which they were created, except that it checks the window designated as the *main window* last. (See “Managing Top-Level Windows” on page 60 for information on designating the main window.)

The default, as provided by **VkComponent**, is for the **okToQuit()** function to return TRUE in all cases. You must override **okToQuit()** for all components that you want to perform a check before quitting. For example, you could override the **okToQuit()** function for a window to post a dialog asking the user whether he or she really wants to exit the application and then abort the shutdown if the user says to do so. Another possibility would be to return FALSE if a component is in the process of updating a file.

Usually, only **VkSimpleWindow** and its subclasses use **okToQuit()**. In some cases, you might want to check one or more components contained within a window before quitting. To do so, override the **okToQuit()** function for that window to call the **okToQuit()** functions for all the desired components. Override the **okToQuit()** functions for the other components to perform whatever checks are necessary.

A ViewKit application automatically exits once all of its windows are deleted. This can occur as a result of any of the following circumstances:

- the application calls **quitYourself()**
- the application deletes all of its windows individually
- the user deletes all application windows through window manager interaction (for example, choosing the “Close” option in the window menu provided by the window manager)

Once all windows are deleted, the application exits by calling **VkApp::terminate()**:

```
virtual void terminate(int status = 0)
```

terminate() is a virtual function that calls **exit(2)**. **terminate()** is also called from within ViewKit when any fatal error is detected.

You can call **terminate()** explicitly to exit a ViewKit application immediately. Usually you would use this if you encounter a fatal error. If you provide a *status* argument, your application uses it as the exit value that the application returns.

You can override **terminate()** in a **VkApp** subclass to perform any cleanup operations that your application requires before aborting (for example, closing a database). If you override **terminate()** in a derived class, call the base class’s **terminate()** function after performing your cleanup operations.

Note: Even though you can override **quitYourself()** in a **VkApp** subclass, in most cases you should override **terminate()** instead. This ensures that any cleanup operations you add are performed no matter how the application exits (for example, by error condition or by user interaction with the window manager). If you decide to override **quitYourself()**, you must perform your cleanup operations before calling the base class’s **quitYourself()**: if **quitYourself()** succeeds in deleting all windows, your application calls **terminate()** and exits before ever returning from **quitYourself()**.

Managing Top-Level Windows

The **VkApp** object maintains a list of all windows created in an application. The **VkApp** object uses this list to manage the application’s top-level windows. So that **VkApp** can properly manage windows, you should always use the **VkSimpleWindow** and **VkWindow** classes to create top-level windows in your application. The classes are discussed in Chapter 4, “ViewKit Windows.”

Every application has a *main window*. By default, the first window you create is treated as the main window. You can use the **VkApp::setMainWindow()** function to specify a different window to treat as the main window:

```
void setMainWindow(VkSimpleWindow *window)
```

The access function **VkApp::mainWindow()** returns a pointer to the **VkSimpleWindow** (or subclass) object installed as the application’s main window:

```
VkSimpleWindow *mainWindow() const
```

Additionally, the **VkApp** class supports several operations that can be performed on all top-level windows in a multi-window application. All of the following functions take no arguments, have a void return value, and are declared virtual:

show()	Displays all of the application’s hidden, non-iconified windows.
hide()	Removes all of the application’s windows from the screen.
iconify()	Iconifies all visible windows in the application.
open()	Opens all iconified windows in the application.
raise()	Raises all visible windows in the application to the top of the window manager’s window stack.
lower()	Lowers all visible windows in the application to the bottom of the window manager’s window stack.

You can also specify whether or not your application’s windows start in an iconified state using **VkApp::startupIconified()**:

```
void startupIconified(const Boolean flag)
```

If *flag* is TRUE, then the application starts all windows in the iconified state.

Note: You must call **startupIconified()** before calling **run()**, otherwise it will not have any effect.

Setting Application Cursors

By default, **VkApp** installs two cursors for ViewKit applications: an arrow for normal use, and a watch for display during busy states. (See “Supporting Busy States” on page 68 for information on busy states in ViewKit applications.) The **VkApp** class also provides several functions for installing your own cursors and retrieving the currently installed cursors.

Setting and Retrieving the Normal Cursor

VkApp::setNormalCursor() sets the normal cursor for use in all of your application’s windows while the application is not busy:

```
void setNormalCursor(Cursor c)
```

You must provide **setNormalCursor()** with a Cursor argument. See the **XCreateFontCursor(3X)** man page for more information on creating an X cursor.

You can retrieve the current normal cursor with **VkApp::normalCursor()**:

```
virtual Cursor
normalCursor()
```

Setting and Retrieving the Busy Cursor

The **VkApp** class supports both fixed and animated busy cursors. A *fixed busy cursor* retains the same appearance throughout a busy state. An *animated busy cursor* is actually a sequence of Pixmaps that you can cycle through while in a busy state, giving the appearance of animation. “Animating the Busy Cursor” on page 71 describes the procedure to follow to cycle through an animated busy cursor’s Pixmaps. If you install an animated busy cursor but do not cycle it, **VkApp** simply uses the animated cursor’s current Pixmap as a fixed busy cursor.

The default busy cursor that **VkApp** installs, a watch, is actually an animated cursor.

Setting and Retrieving a Fixed Busy Cursor

VkApp::setBusyCursor() sets a fixed busy cursor for use in all of your application's windows while the application is busy:

```
void setBusyCursor(Cursor c)
```

You must provide **setBusyCursor()** with a Cursor argument.

You can retrieve the current busy cursor with **VkApp::busyCursor()**:

```
virtual Cursor busyCursor()
```

Creating, Setting, and Retrieving an Animated Busy Cursor

To create an animated busy cursor, you must create a subclass of the abstract base class **VkCursorList**. The syntax of the **VkCursorList** constructor is:

```
VkCursorList (int numCursors)
```

numCursors is the number of cursor Pixmaps in your animated cursor.The **VkCursorList** constructor uses this value to allocate space for an array of Cursor pointers. In your subclass constructor, you should perform any other initialization required by your cursor.

In your subclass, you must also override the pure virtual function **VkCursorList::createCursor()**:

```
virtual void createCursor(int index)
```

createCursor() creates the cursor for the given index in the animated cursor array. Cursors are numbered sequentially beginning with zero. When your application animates the cursor, it step through the cursor array sequentially. **createCursor()** must assign the cursor it creates to the *index* entry in the protected *_cursorList* array:

```
Pixmap *_cursorList
```

For example, Example 3-2 shows the code needed to create an animated hourglass busy cursor.

Example 3-2 Example of Creating an Animated Busy Cursor

```
#include <Vk/VkApp.h>
#include <Vk/VkResource.h>
#include <Vk/VkCursorList.h>

// Define an array of bit patterns that represent each frame of the cursor
// animation.

#define NUMCURSORS 8

static char time_bits[NUMCURSORS][32*32] = {
{
   0x00, 0x00, 0x00, 0x00, 0xfe, 0xff, 0xff, 0x7f, 0xfe, 0xff, 0xff, 0x7f,
   0x8c, 0x00, 0x00, 0x31, 0x4c, 0x00, 0x00, 0x32, 0x4c, 0xff, 0xff, 0x32,
   0x4c, 0xff, 0xff, 0x32, 0x4c, 0xff, 0xff, 0x32, 0x4c, 0xff, 0xff, 0x32,
   0x8c, 0xfe, 0x7f, 0x31, 0x0c, 0xfd, 0xbf, 0x30, 0x0c, 0xfa, 0x5f, 0x30,
   0x0c, 0xe4, 0x27, 0x30, 0x0c, 0x98, 0x19, 0x30, 0x0c, 0x60, 0x06, 0x30,
   0x0c, 0x80, 0x01, 0x30, 0x0c, 0x80, 0x01, 0x30, 0x0c, 0x60, 0x06, 0x30,
   0x0c, 0x18, 0x18, 0x30, 0x0c, 0x04, 0x20, 0x30, 0x0c, 0x02, 0x40, 0x30,
   0x0c, 0x01, 0x80, 0x30, 0x8c, 0x00, 0x00, 0x31, 0x4c, 0x00, 0x00, 0x32,
   0x4c, 0x00, 0x00, 0x32, 0x4c, 0x00, 0x00, 0x32, 0x4c, 0x00, 0x00, 0x32,
   0x4c, 0x00, 0x00, 0x32, 0x8c, 0x00, 0x00, 0x31, 0xfe, 0xff, 0xff, 0x7f,
   0xfe, 0xff, 0xff, 0x7f, 0x00, 0x00, 0x00, 0x00},
{
   0x00, 0x00, 0x00, 0x00, 0xfe, 0xff, 0xff, 0x7f, 0xfe, 0xff, 0xff, 0x7f,
   0x8c, 0x00, 0x00, 0x31, 0x4c, 0x00, 0x00, 0x32, 0x4c, 0x03, 0xc0, 0x32,
   0x4c, 0x3f, 0xfc, 0x32, 0x4c, 0xff, 0xff, 0x32, 0x4c, 0xff, 0xff, 0x32,
   0x8c, 0xfe, 0x7f, 0x31, 0x0c, 0xfd, 0xbf, 0x30, 0x0c, 0xfa, 0x5f, 0x30,
   0x0c, 0xe4, 0x27, 0x30, 0x0c, 0x98, 0x19, 0x30, 0x0c, 0x60, 0x06, 0x30,
   0x0c, 0x80, 0x01, 0x30, 0x0c, 0x80, 0x01, 0x30, 0x0c, 0x60, 0x06, 0x30,
   0x0c, 0x18, 0x19, 0x30, 0x0c, 0x84, 0x20, 0x30, 0x0c, 0x02, 0x41, 0x30,
   0x0c, 0x81, 0x80, 0x30, 0x8c, 0x00, 0x01, 0x31, 0x4c, 0x80, 0x00, 0x32,
   0x4c, 0x00, 0x01, 0x32, 0x4c, 0xfc, 0x3f, 0x32, 0x4c, 0xff, 0xff, 0x32,
   0x4c, 0x00, 0x00, 0x32, 0x8c, 0x00, 0x00, 0x31, 0xfe, 0xff, 0xff, 0x7f,
   0xfe, 0xff, 0xff, 0x7f, 0x00, 0x00, 0x00, 0x00},
{
   0x00, 0x00, 0x00, 0x00, 0xfe, 0xff, 0xff, 0x7f, 0xfe, 0xff, 0xff, 0x7f,
   0x8c, 0x00, 0x00, 0x31, 0x4c, 0x00, 0x00, 0x32, 0x4c, 0x00, 0x00, 0x32,
   0x4c, 0x03, 0xc0, 0x32, 0x4c, 0x1f, 0xf8, 0x32, 0x4c, 0x7f, 0xfe, 0x32,
   0x8c, 0xfe, 0x7f, 0x31, 0x0c, 0xfd, 0xbf, 0x30, 0x0c, 0xfa, 0x5f, 0x30,
   0x0c, 0xe4, 0x27, 0x30, 0x0c, 0x98, 0x19, 0x30, 0x0c, 0x60, 0x06, 0x30,
   0x0c, 0x80, 0x01, 0x30, 0x0c, 0x80, 0x01, 0x30, 0x0c, 0x60, 0x06, 0x30,
   0x0c, 0x18, 0x19, 0x30, 0x0c, 0x84, 0x20, 0x30, 0x0c, 0x02, 0x41, 0x30,
   0x0c, 0x81, 0x80, 0x30, 0x8c, 0x00, 0x01, 0x31, 0x4c, 0xc0, 0x07, 0x32,
   0x4c, 0xfc, 0x3f, 0x32, 0x4c, 0xfe, 0x7f, 0x32, 0x4c, 0xff, 0xff, 0x32,
   0x4c, 0x00, 0x00, 0x32, 0x8c, 0x00, 0x00, 0x31, 0xfe, 0xff, 0xff, 0x7f,
   0xfe, 0xff, 0xff, 0x7f, 0x00, 0x00, 0x00, 0x00},
{
   0x00, 0x00, 0x00, 0x00, 0xfe, 0xff, 0xff, 0x7f, 0xfe, 0xff, 0xff, 0x7f,
   0x8c, 0x00, 0x00, 0x31, 0x4c, 0x00, 0x00, 0x32, 0x4c, 0x00, 0x00, 0x32,
   0x4c, 0x00, 0x00, 0x32, 0x4c, 0x03, 0xc0, 0x32, 0x4c, 0x0f, 0xf0, 0x32,
   0x8c, 0x3e, 0x7c, 0x31, 0x0c, 0xfd, 0xbf, 0x30, 0x0c, 0xfa, 0x5f, 0x30,
   0x0c, 0xe4, 0x27, 0x30, 0x0c, 0x98, 0x19, 0x30, 0x0c, 0x60, 0x06, 0x30,
   0x0c, 0x80, 0x01, 0x30, 0x0c, 0x80, 0x01, 0x30, 0x0c, 0x60, 0x06, 0x30,
   0x0c, 0x18, 0x19, 0x30, 0x0c, 0x84, 0x20, 0x30, 0x0c, 0x02, 0x41, 0x30,
   0x0c, 0x81, 0x80, 0x30, 0x8c, 0xe0, 0x07, 0x31, 0x4c, 0xfc, 0x3f, 0x32,
   0x4c, 0xfe, 0x7f, 0x32, 0x4c, 0xff, 0xff, 0x32, 0x4c, 0xff, 0xff, 0x32,
   0x4c, 0x00, 0x00, 0x32, 0x8c, 0x00, 0x00, 0x31, 0xfe, 0xff, 0xff, 0x7f,
   0xfe, 0xff, 0xff, 0x7f, 0x00, 0x00, 0x00, 0x00},
{
   0x00, 0x00, 0x00, 0x00, 0xfe, 0xff, 0xff, 0x7f, 0xfe, 0xff, 0xff, 0x7f,
   0x8c, 0x00, 0x00, 0x31, 0x4c, 0x00, 0x00, 0x32, 0x4c, 0x00, 0x00, 0x32,
   0x4c, 0x00, 0x00, 0x32, 0x4c, 0x00, 0x00, 0x32, 0x4c, 0x03, 0xc0, 0x32,
   0x8c, 0x06, 0x60, 0x31, 0x0c, 0x1d, 0xb8, 0x30, 0x0c, 0x7a, 0x5e, 0x30,
   0x0c, 0xe4, 0x27, 0x30, 0x0c, 0x98, 0x19, 0x30, 0x0c, 0x60, 0x06, 0x30,
   0x0c, 0x80, 0x01, 0x30, 0x0c, 0x80, 0x01, 0x30, 0x0c, 0x60, 0x06, 0x30,
   0x0c, 0x18, 0x19, 0x30, 0x0c, 0x84, 0x20, 0x30, 0x0c, 0x82, 0x41, 0x30,
   0x0c, 0xf1, 0x8f, 0x30, 0x8c, 0xfc, 0x3f, 0x31, 0x4c, 0xfe, 0x7f, 0x32,
   0x4c, 0xff, 0xff, 0x32, 0x4c, 0xff, 0xff, 0x32, 0x4c, 0xff, 0xff, 0x32,
   0x4c, 0x00, 0x00, 0x32, 0x8c, 0x00, 0x00, 0x31, 0xfe, 0xff, 0xff, 0x7f,
   0xfe, 0xff, 0xff, 0x7f, 0x00, 0x00, 0x00, 0x00},
{
   0x00, 0x00, 0x00, 0x00, 0xfe, 0xff, 0xff, 0x7f, 0xfe, 0xff, 0xff, 0x7f,
   0x8c, 0x00, 0x00, 0x31, 0x4c, 0x00, 0x00, 0x32, 0x4c, 0x00, 0x00, 0x32,
   0x4c, 0x00, 0x00, 0x32, 0x4c, 0x00, 0x00, 0x32, 0x4c, 0x00, 0x00, 0x32,
   0x8c, 0x02, 0x40, 0x31, 0x0c, 0x05, 0xa0, 0x30, 0x0c, 0x1a, 0x58, 0x30,
   0x0c, 0x64, 0x26, 0x30, 0x0c, 0x98, 0x19, 0x30, 0x0c, 0x60, 0x06, 0x30,
   0x0c, 0x80, 0x01, 0x30, 0x0c, 0x80, 0x01, 0x30, 0x0c, 0x60, 0x06, 0x30,
   0x0c, 0x18, 0x19, 0x30, 0x0c, 0x84, 0x20, 0x30, 0x0c, 0xe2, 0x47, 0x30,
   0x0c, 0xf9, 0x9f, 0x30, 0x8c, 0xfe, 0x7f, 0x31, 0x4c, 0xff, 0xff, 0x32,
   0x4c, 0xff, 0xff, 0x32, 0x4c, 0xff, 0xff, 0x32, 0x4c, 0xff, 0xff, 0x32,
   0x4c, 0x00, 0x00, 0x32, 0x8c, 0x00, 0x00, 0x31, 0xfe, 0xff, 0xff, 0x7f,
   0xfe, 0xff, 0xff, 0x7f, 0x00, 0x00, 0x00, 0x00},
{
   0x00, 0x00, 0x00, 0x00, 0xfe, 0xff, 0xff, 0x7f, 0xfe, 0xff, 0xff, 0x7f,
   0x8c, 0xff, 0xff, 0x31, 0xcc, 0xff, 0xff, 0x33, 0x4c, 0x00, 0x00, 0x32,
   0x4c, 0x00, 0x00, 0x32, 0x4c, 0x00, 0x00, 0x32, 0x4c, 0x00, 0x00, 0x32,
   0x8c, 0x00, 0x00, 0x31, 0x0c, 0x01, 0x80, 0x30, 0x0c, 0x02, 0x40, 0x30,
   0x0c, 0x04, 0x20, 0x30, 0x0c, 0x18, 0x18, 0x30, 0x0c, 0x60, 0x06, 0x30,
   0x0c, 0x80, 0x01, 0x30, 0x0c, 0x80, 0x01, 0x30, 0x0c, 0x60, 0x06, 0x30,
   0x0c, 0x98, 0x19, 0x30, 0x0c, 0xe4, 0x27, 0x30, 0x0c, 0xfa, 0x5f, 0x30,
   0x0c, 0xfd, 0xbf, 0x30, 0x8c, 0xfe, 0x7f, 0x31, 0x4c, 0xff, 0xff, 0x32,
   0x4c, 0xff, 0xff, 0x32, 0x4c, 0xff, 0xff, 0x32, 0x4c, 0xff, 0xff, 0x32,
   0x4c, 0x00, 0x00, 0x32, 0x8c, 0x00, 0x00, 0x31, 0xfe, 0xff, 0xff, 0x7f,
   0xfe, 0xff, 0xff, 0x7f, 0x00, 0x00, 0x00, 0x00},
{
   0x00, 0x00, 0x00, 0x00, 0x06, 0x00, 0x00, 0x60, 0xfe, 0xff, 0xff, 0x7f,
   0xfe, 0xff, 0xff, 0x7f, 0x06, 0x00, 0x00, 0x60, 0x06, 0x00, 0x00, 0x60,
   0xf6, 0x01, 0x80, 0x6f, 0x0e, 0x02, 0x40, 0x78, 0xe6, 0x05, 0x20, 0x78,
   0xe6, 0x0b, 0x10, 0x78, 0xe6, 0x17, 0x08, 0x78, 0xe6, 0x2f, 0x04, 0x78,
   0xe6, 0x2f, 0x04, 0x78, 0xe6, 0x5f, 0x02, 0x78, 0xe6, 0x5f, 0x02, 0x78,
   0xe6, 0xbf, 0x01, 0x78, 0xe6, 0xbf, 0x01, 0x78, 0xe6, 0x5f, 0x02, 0x78,
   0xe6, 0x5f, 0x02, 0x78, 0xe6, 0x2f, 0x04, 0x78, 0xe6, 0x2f, 0x04, 0x78,
   0xe6, 0x17, 0x08, 0x78, 0xe6, 0x0b, 0x10, 0x78, 0xe6, 0x05, 0x20, 0x78,
   0x0e, 0x02, 0x40, 0x78, 0xf6, 0x01, 0x80, 0x6f, 0x06, 0x00, 0x00, 0x60,
   0x06, 0x00, 0x00, 0x60, 0xfe, 0xff, 0xff, 0x7f, 0xfe, 0xff, 0xff, 0x7f,
   0x06, 0x00, 0x00, 0x60, 0x00, 0x00, 0x00, 0x00}
};

// Masks used for this cursor. The last frame requires a different
// mask, but all other frames can use the same mask.

static  char time_mask_bits[] = {
   0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
   0x8e, 0xff, 0xff, 0x71, 0xce, 0xff, 0xff, 0x73, 0xce, 0xff, 0xff, 0x73,
   0xce, 0xff, 0xff, 0x73, 0xce, 0xff, 0xff, 0x73, 0xce, 0xff, 0xff, 0x73,
   0x8e, 0xff, 0xff, 0x71, 0x0e, 0xff, 0xff, 0x70, 0x0e, 0xfe, 0x7f, 0x70,
   0x0e, 0xfc, 0x3f, 0x70, 0x0e, 0xf8, 0x1f, 0x70, 0x0e, 0xe0, 0x07, 0x70,
   0x0e, 0x80, 0x01, 0x70, 0x0e, 0x80, 0x01, 0x70, 0x0e, 0xe0, 0x07, 0x70,
   0x0e, 0xf8, 0x1f, 0x70, 0x0e, 0xfc, 0x3f, 0x70, 0x0e, 0xfe, 0x7f, 0x70,
   0x0e, 0xff, 0xff, 0x70, 0x8e, 0xff, 0xff, 0x71, 0xce, 0xff, 0xff, 0x73,
   0xce, 0xff, 0xff, 0x73, 0xce, 0xff, 0xff, 0x73, 0xce, 0xff, 0xff, 0x73,
   0xce, 0xff, 0xff, 0x73, 0x8e, 0xff, 0xff, 0xf1, 0xff, 0xff, 0xff, 0xff,
   0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff};

#define time7_mask_width 32
#define time7_mask_height 32
#define time7_mask_x_hot 15
#define time7_mask_y_hot 15
static  char time7_mask_bits[] = {
   0x0f, 0x00, 0x00, 0xf0, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
   0xff, 0xff, 0xff, 0xff, 0x07, 0x00, 0x00, 0xe0, 0x07, 0x00, 0x00, 0xe0,
   0xf7, 0x01, 0x80, 0xef, 0xff, 0x03, 0xc0, 0xff, 0xff, 0x07, 0xe0, 0xff,
   0xff, 0x0f, 0xf0, 0xff, 0xff, 0x1f, 0xf8, 0xff, 0xff, 0x3f, 0xfc, 0xff,
   0xff, 0x3f, 0xfc, 0xff, 0xff, 0x7f, 0xfe, 0xff, 0xff, 0x7f, 0xfe, 0xff,
   0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x7f, 0xfe, 0xff,
   0xff, 0x7f, 0xfe, 0xff, 0xff, 0x3f, 0xfc, 0xff, 0xff, 0x3f, 0xfc, 0xff,
   0xff, 0x1f, 0xf8, 0xff, 0xff, 0x0f, 0xf0, 0xff, 0xff, 0x07, 0xe0, 0xff,
   0xff, 0x03, 0xc0, 0xff, 0xf7, 0x01, 0x80, 0xef, 0x07, 0x00, 0x00, 0xe0,
   0x07, 0x00, 0x00, 0xe0, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
   0xff, 0xff, 0xff, 0xff, 0x0f, 0x00, 0x00, 0xf0};


/////////////////////////////////////////////////
// Class declaration. Subclass VkCursorList
///////////////////////////////////////////////////

class HourGlassCursors : public VkCursorList {

  public:
    HourGlassCursors( );

  protected:
    void createCursor(int index);   // Overrides base class' pure virtual

  private:
    XColor  xcolors[2];
};


// The constructor gets two colors to use for the cursor.

HourGlassCursors::HourGlassCursors ( ) :  VkCursorList ( NUMCURSORS )
{
    xcolors[0].pixel= (Pixel) VkGetResource(theApplication->baseWidget(),
                                        "busyCursorForeground",
                                        XmCForeground,
                                        XmRPixel,
                                        (char *) "Black");

    xcolors[1].pixel= (Pixel) VkGetResource(theApplication->baseWidget(),
                                        "busyCursorBackground",
                                        XmCBackground,
                                        XmRPixel,
                                        char *) "White");
    XQueryColors (theApplication->display(),
                  DefaultColormapOfScreen(DefaultScreenOfDisplay(dpy)),
                  xcolors, 2);
}
```

```
// This function is called as needed, to create a new cursor frame.
// Just create the cursor corresponding to the requested index and
// install it in _cursorList.

void HourGlassCursors::createCursor(int index)
{
    Pixmap pixmap = 0, maskPixmap = 0;
    Display *dpy = theApplication->display();

    pixmap = XCreateBitmapFromData (dpy,
                                    DefaultRootWindow(dpy),
                                    time_bits[index],
                                    32, 32);

    if(index == 7)
        maskPixmap = XCreateBitmapFromData (dpy,
                                            DefaultRootWindow(dpy),
                                            time7_mask_bits,
                                            32, 32);
    else
        maskPixmap = XCreateBitmapFromData (dpy,
                                            DefaultRootWindow(dpy),
                                            time_mask_bits,
                                            32, 32);

    _cursorList[index] = XCreatePixmapCursor ( dpy, pixmap, maskPixmap,
                                               &(xcolors[0]), &(xcolors[1]),
                                               0, 0);
    if(pixmap)
        XFreePixmap (dpy, pixmap);
    if(maskPixmap)
        XFreePixmap (dpy, maskPixmap);
}
```

Once you have created an animated busy cursor, you can install it as your application's busy cursor using an overloaded version of the **VkApp::setBusyCursor()** function:

`void setBusyCursor(VkCursorList *`*animatedCursor*`)`

You should provide as the argument to **setBusyCursor()** a pointer to your animated busy cursor object.

When you use an animated busy cursor, the **busyCursor()** function returns the currently displayed Pixmap of your busy cursor.

Setting and Retrieving a Temporary Cursor

You can set a temporary cursor for use in all of your application's windows using **VkApp::showCursor()**:

```
void showCursor(Cursor c)
```

Calling **showCursor()** immediately displays the temporary cursor. The cursor stays in effect until the application enters or exits a busy state, or you reset the cursor back to the normal cursor by calling **showCursor()** with a NULL cursor argument.

You should use this function to display a cursor only briefly. If you want to change the cursor for an extended period, you should instead use **setNormalCursor()** or **setBusyCursor()**.

Supporting Busy States

This section describes ViewKit's support for *busy states*, when you lock out user input during an operation.

Entering and Exiting Busy States Using ViewKit

Whenever you expect a procedure to take considerable time to complete, you can call the **VkApp::busy()** function before entering the relevant region of code to lock out user input in all application windows:

```
virtual void busy(char *msg = NULL,
                  VkSimpleWindow window = NULL)
```

If you call **busy()** with no arguments, the application simply displays a busy cursor. If you provide a string as the first argument, the application posts a dialog to display the string. The string is treated first as a resource name that **busy()** looks up relative to the dialog widget. If the resource exists, its value is used as the message. If the resource does not exist, or if the string contains spaces or newline characters, **busy()** uses the string itself as the message.

)If you provide a **VkSimpleWindow** (or subclass) as the second argument, the application posts the dialog over this specified window. If you do not specify a window, the application posts the dialog over the main window. (See “Managing Top-Level Windows” on page 60 for instructions on setting the main window. See Chapter 7, “Using Dialogs in ViewKit,” for more details on dialog behavior.

The **VkApp::notBusy()** function undoes the previous call to **busy()**:

```
virtual void notBusy()
```

You can nest calls to **busy()**, but you must always have matching **busy()/notBusy()** pairs. An application exits the busy state only when the number of **notBusy()** calls matches the number of **busy()** calls.

Note: ViewKit does not “stack” nested busy dialogs, it simply displays the most recently posted busy dialog. Once you post a busy dialog, it remains displayed until the busy state is over or you replace it with another busy dialog.

Example 3-3 shows an example of setting busy dialog messages using resource values and using nested **busy()/notBusy()** calls. Note that this is not a complete example: it lists only the code relating to the busy states.

Example 3-3 Using Busy States in a ViewKit Application

```
class ReportWindow: public VkSimpleWindow {

  public:
    ReportWindow ( const char *name );
    ~ReportWindow();
    virtual const char* className();
    void report();
    void sort();

  private:
    static String _defaultResources[];
};
```

```
String _defaultResources[] = {
    "*sortDialogMsg:    Sorting records...",
    "*reportDialogMsg:  Generating report...",
    NULL
};

ReportWindow::ReportWindow(const char *name) : VkSimpleWindow ( name )
{
  setDefaultResources(theApplication->baseWidget(), _defaultResources);
  // Create window...
}

void ReportWindow::sort()
{
  theApplication->busy("sortDialogMsg");
  // Sort records...
  theApplication->notBusy();
}

void ReportWindow::report()
{
  theApplication->busy("reportDialogMsg");
  // Report generation...
  sort();
  // Report generation continued...
  theApplication->notBusy();
}
```

The **ReportWindow** class defines the busy dialog messages as resource values and loads these values using **setDefaultResources()** in the **ReportWindow** constructor.[1] The calls to **busy()** pass these resource names instead of passing the actual dialog text. This allows you to override these resource values in an app-defaults file should you need to.

[1] Unlike most ViewKit components, the **VkSimpleWindow** class constructor is not passed a parent widget. All ViewKit windows are children of the application's **VkApp** base widget. So, to access a window's parent widget, you must use the **VkApp::baseWidget()** access function as shown in this example.

When the application calls **ReportWindow::report()**, it posts the busy dialog shown in Figure 3-2.

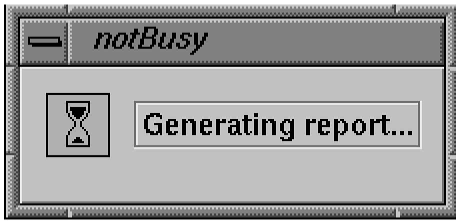


Figure 3-2 Example of Busy Dialog

When the application calls **ReportWindow::sort()**, it posts the busy dialog shown in Figure 3-3.

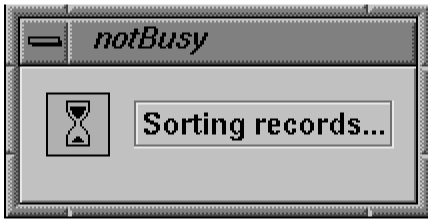


Figure 3-3 Example of Nested Busy Dialog

Note that the application continues to display the second busy dialog until reaching the **theApplication->notBusy()** statement in **ReportWindow::report()**.

Animating the Busy Cursor

To animate the busy cursor during a busy state, periodically call **VkApp::progressing()**:

```
virtual void progressing(const char *msg = NULL)
```

If you have an animated busy cursor installed, **progressing()** cycles to the next Pixmap in the cursor list. If you have a fixed cursor installed, **progressing()** has no effect on the busy cursor.

If you provide a character string argument, your application posts a dialog to display the message. The string is treated first as a resource name that **progressing()** looks up relative to the dialog widget. If the resource exists, its value is used as the message. If the resource does not exist, or if the string contains spaces or newline characters, **progressing()** uses the string itself as the message.

The code fragment in Example 3-4 performs a simulated lengthy task and periodically cycles the busy cursor.

Example 3-4 Example of Animating the Busy Cursor

```
int i;

// Start being "busy"

theApplication->busy("Busy", (BusyWindow *) clientData);

for(i=0; i<100; i++)
{
  // Every so often, update the busy cursor
  theApplication->progressing();
  sleep(1);
}

// Task done, so we're not busy anymore
theApplication->notBusy();
```

Installing Different Busy Dialogs

By default, **busy()** displays the dialog using *theBusyDialog*, a global pointer to an instantiation of the **VkBusyDialog** class[1] (described in "The Busy Dialog" on page 203). If you prefer to use a different dialog object, you can pass a pointer to the object to the **setBusyDialog()** function:

```
void setBusyDialog(VkBusyDialog *dialog)
```

[1] *theBusyDialog* is actually implemented as a compiler macro that invokes a **VkBusyDialog** access function to return a pointer to the unique instantiation of the **VkBusyDialog** class. Although you should never need to use this access function directly, you might encounter it while debugging a ViewKit application that uses the busy dialog.

This alternate busy dialog must be implemented as a subclass of **VkBusyDialog**. Calling **setBusyDialog()** with a NULL argument restores the default **VkBusyDialog** object.

Most frequently, you will use **setBusyDialog()** to install *theInterruptDialog*, a global pointer to an instantiation of the **VkInterruptDialog** class, which implements an interruptible busy dialog[1]. ("The Interruptible Busy Dialog" on page 204 describes the **VkInterruptDialog** class.) Example 3-5 shows a typical example of temporarily installing an interruptible busy dialog for a task.

Example 3-5 Temporarily Installing an Interruptible Busy Dialog

```
#inlcude <Vk/VkApp.h>
#include <Vk/VkInterruptDialog.h>

// ...

// Install theInterruptDialog as the busy dialog

theApplication->setBusyDialog(theInterruptDialog);
theApplication->busy("Generating report");  // Enter busy state

// Perform task...

theApplication->notBusy();                  // Exit busy state
theApplication->setBusyDialog(NULL);        // Install default busy dialog
```

[1] *theInterruptDialog* is actually implemented as a compiler macro that invokes a **VkInterruptDialog** access function to return a pointer to the unique instantiation of the **VkInterruptDialog** class. Although you should never need to use this access function directly, you might encounter it while debugging a ViewKit application that uses the interruptible busy dialog.

Maintaining Product and Version Information

The **VkApp** class provides several access functions and constant data members that you can use to identify your application and the current ViewKit release.

VkApp::ViewKitMajorRelease is a static integer constant that identifies the major release of ViewKit; *VkApp::ViewKitMinorRelease* is a static integer constant that identifies the minor release of ViewKit, and *VkApp::ViewKitReleaseString* is a static character array constant that contains the complete major and minor release information. For example, in a 1.2 release, the value of *VkApp::ViewKitMajorRelease* would be 1, the value of *VkApp::ViewKitMinorRelease* would be 2, and the value of *VkApp::ViewKitReleaseString* would be "ViewKit Release: 1.2". These values can be useful if you need to provide conditional statements in your code to handle different versions of the ViewKit library.

You can use **VkApp::setVersionString()** to set version information for an application based on ViewKit:

```
void setVersionString(const char *versionInfo)
```

You can retrieve the version string using **VkApp::versionString()**:

```
const char *versionString()
```

ViewKit displays this version string in the Product Information dialog that is posted when a user chooses "Product Information" from the default Help menu. (See "The ViewKit Help Menu" on page 165 for more information on the default Help menu.) For example, consider an application that you invoke with the command *MapMaker* that includes the following line of code:

```
theApplication->setVersionString("MapMaker 2.1");
```

If you choose “Product Information” from the default Help menu, your application posts the dialog shown in Figure 3-4.

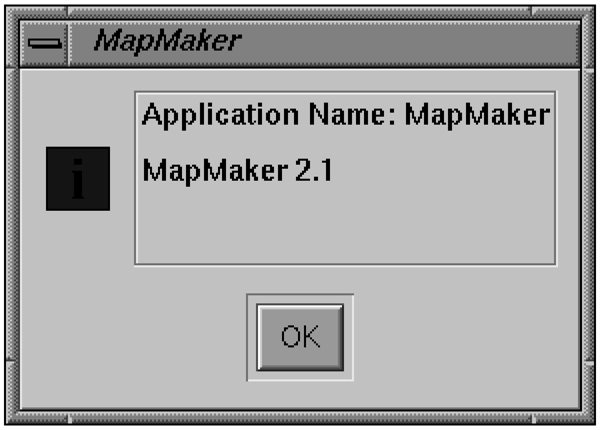


Figure 3-4 Example of the Product Information Dialog

You can use **VkApp::setAboutDialog()** to replace the standard Product Information dialog with your own custom dialog:

```
void setAboutDialog(VkDialogManager *dialog)
```

You must provide **setAboutDialog()** with a pointer to an object that is a subclass of **VkDialogManager**. Most frequently, you will actually create a subclass of **VkGenericDialog**, an abstract subclass of **VkDialogManager** that simplifies the process of creating custom dialogs. “Deriving New Dialog Classes Using the Generic Dialog” on page 211describes creating a custom dialog.

The **VkApp::aboutDialog()** function returns a pointer to the custom Product Information dialog you have installed:

```
VkDialogManager* aboutDialog()
```

Application Data Access Functions

VkApp provides several access functions for retrieving data useful for your application:

char ***name()** const
: Returns the command name you used to invoke the application (*argv[0]*).

char ***applicationClassName()** const
: Returns the application class name set in the **VkApp** constructor. This application class name is used when loading application resources.

virtual const char ***className()** const
: Returns the class name of the **VkApp** (or subclass) instance being used. By default, this is “VkApp”. Note that unlike all other ViewKit components, the **VkApp** class does not use the value returned by **className()** when loading resources; instead, it uses the application class name that you provide as an argument to the **VkApp** constructor. This allows you to set the application class name without creating a subclass of **VkApp**.

XtAppContext **appContext()** const
: Returns the application’s XtAppContext structure, which is required by many IRIS IM and Xt functions.

Display ***display()** const
: Returns a pointer to the X Display structure associated with the application’s connection to the X server.

char ***shellGeometry()** const
: Returns a string containing the geometry of the application’s base shell. You may want to use this information to size other windows in your application.

int **argc()** const
: Returns the number of items remaining in the *argv* array after all arguments recognized by Xt have been removed.

char ****argv()** const
: Called without arguments, this function returns a pointer to the *argv* array after all arguments recognized by Xt have been removed.

char ***argv(**int *index***)** const

Called with an integer argument, this function returns a single *argv* array item (after all arguments recognized by Xt have been removed) specified by the *index* argument.

Boolean **startupIconified()** const

Called with no arguments, this function returns the value TRUE if the application starts with all windows iconified and FALSE if it starts with all windows displayed normally.

Widget **baseWidget()**

For the **VkApp** class, **baseWidget()** returns the hidden shell widget.

Deriving Classes from VkApp

This section describes **VkApp** protected functions and data members that you can use in a **VkApp** subclass. Following that is an example of subclassing **VkApp** to parse command-line options.

VkApp Protected Functions and Data Members

You can use **VkApp::parseCommandLine()** to parse command line options:

```
int parseCommandLine(XrmOptionDescRec *options,
                     Cardinal numOptions)
```

You should call **parseCommandLine()** from within the constructor of your **VkApp** subclass. Provide an XrmOptionDescRec(3Xt) table as the *options* argument and specify the number of entries in the table with the *numOptions* argument. **parseCommandLine()** passes these arguments to **XtOpenDisplay(3Xt)**, which parses the command line and loads recognized options into the application's resource database. **parseCommandLine()** modifies *argv* to remove all recognized options and returns an updated value of *argc* which you must use to update the value of *argc*. Example 3-6 shows an example of using **parseCommandLine()**.

You can override **VkApp::afterRealizeHook()** to perform certain actions after all application windows have been realized:

```
virtual void afterRealizeHook()
```

For example, you could override **afterRealizeHook()** to install a colormap or set properties on the application's windows. By default, this function is empty.

When subclassing **VkApp**, you also have access to the protected data member *VkApp::_winList*:

```
VkComponentList _winList
```

This data member maintains the list of the application's top-level windows. Consult the VkComponentList(3Vk) reference page for more information on the **VkComponentList** class.

Example of Subclassing VkApp

The most common reason for creating a subclass of **VkApp** is to parse the command line and set global resources based on command line options. Also, rather than use global variables, you can store data that is needed throughout your application in data members of your **VkApp** subclass.

The program in Example 3-6 creates **MyApp**, a **VkApp** subclass that recognizes a `-verbose` command line argument and initializes a protected data member depending on whether or not the flag is present.

Note that this example uses the protected **VkApp** function **parseCommandLine()** to extract the flag if it exists. This function returns an updated value of *argc* which must be used to update the value of *argc* as passed by the calling application.

Example 3-6 Deriving a Subclass from **VkApp**

```
‘#include <Vk/VkApp.h>
#include <Vk/VkResource.h>

class MyApp : public VkApp {

  public:
    MyApp(char             *appClassName,
          int              *arg_c,
          char            **arg_v,
          XrmOptionDescRec *optionList       = NULL,
          int               sizeOfOptionList = 0);
    Boolean verbose() { return _verbose; }       // Access function

  protected:
    Boolean   _verbose;                         // Data member to initialize

  private:
    static XrmOptionDescRec _cmdLineOptions[];  // Command-line options
    static XtResource _resources[];             // Resource descriptions
};


// Describe the command line options

XrmOptionDescRec MyApp::_cmdLineOptions[] =
{
    {
    "-verbose", "*verbose", XrmoptionNoArg, "TRUE",
    },
};

// Describe the resources to retrieve and use to initialize the class

XtResource MyApp::_resources [] = {
{
    "verbose",
    "Verbose",
    XmRBoolean,
    sizeof ( Boolean ),
    XtOffset ( MyApp *, _verbose ),
    XmRString,
    (XtPointer) "FALSE",
  },
};

MyApp::MyApp(char    *appClassName,
    int              *arg_c,
    char            **arg_v,
    XrmOptionDescRec *optionList,
    int               sizeOfOptionList) : VkApp(appClassName,
                                               arg_c,
                                               arg_v,
                                               optionList,
                                               sizeOfOptionList)
```

```
{
    // Parse the command line, loading options into the resource database

    *arg_c = parseCommandLine(_cmdLineOptions,
                              XtNumber(_cmdLineOptions));

    // Initialize this class from the resource data base

    getResources (_resources, XtNumber(_resources));
}
```

Chapter 4

ViewKit Windows

This chapter introduces the basic ViewKit classes needed to create and manipulate the top-level windows in a ViewKit application: **VkSimpleWindow** and **VkWindow**. Figure 4-1 shows the inheritance graph for these classes.

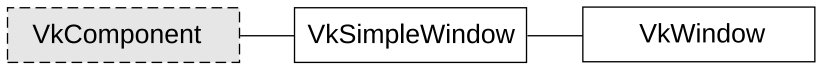


Figure 4-1 The Inheritance Graph for **VkSimpleWindow** and **VkWindow**

Overview of ViewKit Window Support

This section describes how ViewKit supports multiple top-level windows in an application, and then describes the ViewKit classes that implement these windows.

ViewKit's Multi-Window Model

There are several possible models for multi-window applications in Xt. One approach is to create a single top-level window used as the main window of the application. All other windows are then popup shells whose parent is the main window. Another approach is to create a single shell that never appears on the screen. All other windows are then popup children of the main shell. In this model, all top-level windows are treated equally, as siblings. One window may logically be the top-level window of the application, but as far as Xt is concerned, all windows are equal.

ViewKit follows the second model. The **VkApp** class, described in Chapter 3, "The ViewKit Application Class," creates a single widget that serves as the parent of all top-level windows created by the program. The **VkApp** base widget does not appear on the screen.

ViewKit Window Classes

All top-level windows in a ViewKit application must be instances of **VkSimpleWindow**, **VkWindow**, or a subclass of one of these classes. The **VkSimpleWindow** class supports a top-level window that does not include a menu bar. The **VkWindow** class, derived from **VkSimpleWindow**, adds support for a menu bar along the top of the window. You must create a separate instance of **VkSimpleWindow**, **VkWindow**, or a subclass of one of these classes for each top-level window in your application.

Instantiating a **VkSimpleWindow** or **VkWindow** object creates a popup shell as a child of the invisible shell created by your application's instance of **VkApp**. **VkSimpleWindow** and **VkWindow** also create a XmMainWindow widget as a child of the popup shell. You define the contents of a window by creating a widget or ViewKit component to use as the work area (or *view*) for the XmMainWindow widget. In most cases, you will create several widgets and / or ViewKit components as children of a container widget and then assign that container widget as the view of the XmMainWindow widget. "Creating the Window Interface" on page 85 describes how to assign a view to a window. Figure 4-2 shows an example of a widget hierarchy for the top-level windows of a simple ViewKit application with two top-level windows.

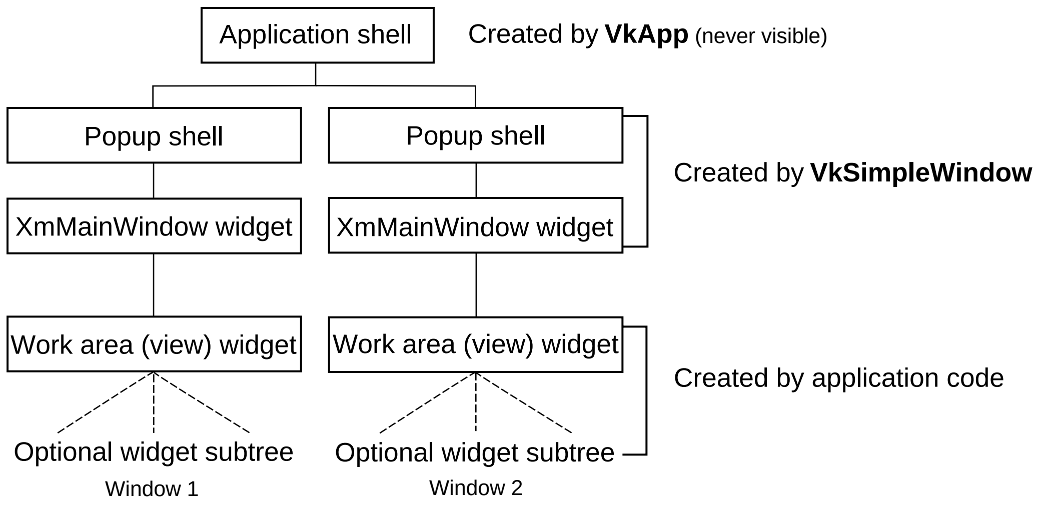


Figure 4-2 Widget Hierarchy of Top-Level Windows in ViewKit Applications

In most cases, directly instantiating a **VkSimpleWindow** or **VkWindow** object is not appropriate.[1] In addition to the widgets and components composing the window's interface, most windows require other data and support functions. In accordance with good object-oriented programming style, the functions and data associated with a window should be contained within that window's class. Therefore, the best practice to follow when creating a ViewKit application is to create a separate subclass for each window in your application. You can derive these subclasses from **VkWindow** for those windows that require menu bars, and from **VkSimpleWindow** for those windows that do not. "Deriving Window Subclasses" on page 101 describes in detail the process of deriving window subclasses.

In addition to creating shell and XmMainWindow widgets, the **VkSimpleWindow** and **VkWindow** classes set up various properties on the shell window and provide simple hooks for window manager interactions. "Window Manager Interface" on page 96 discusses the built-in window manager support.

The **VkSimpleWindow** and **VkWindow** classes provide simple functions to raise, lower, iconify, and open windows, as described in "Manipulating Windows" on page 95. The classes also provide several convenience functions for determining a window's state (for example, whether it is visible, iconified, and so on) and for retrieving other window information. These access functions are described in "Window Data Access Functions" on page 96.

The **VkSimpleWindow** and **VkWindow** classes also register their windows with the application's **VkApp** instance to support application-wide services such as setting the cursor for all of an application's windows, entering busy states, and manipulating all windows in an application. Chapter 3, "The ViewKit Application Class," describes how to use these application-wide services.

[1] There are exceptional cases for which you might choose to directly instantiate a **VkSimpleWindow** or **VkWindow** object and then associate a view with the window. For example, if you have a complex, self-contained component and need a window simply to display the component, you might find this method acceptable. "Adding a Window Interface to a Direct Instantiation of a ViewKit Window Class" on page 94 describes how to do this.

Window Class Constructors

The **VkSimpleWindow** and **VkWindow** constructors both have the same form:

```
VkSimpleWindow(const char *name,
               ArgList args = NULL,
               Cardinal argCount = 0)

VkWindow(const char *name,
         ArgList args = NULL,
         Cardinal argCount = 0)
```

Unlike most other ViewKit components, the **VkSimpleWindow** and **VkWindow** constructors do not require a parent widget as an argument: all ViewKit windows are automatically created as children of the invisible shell created by your application's instance of **VkApp**. You must specify a name for your window. Optionally, you can also provide a standard Xt argument list that the constructor will use when creating the window's popup shell.

Every application has a *main window*. By default, the first window you create is treated as the main window. To specify a different window to use as the main window, use the **VkApp::setMainWindow()** function described in "Managing Top-Level Windows" on page 60.

Because the first window you create is by default the main window, the window class constructors also set some shell resources on the popup shell widget of that window. The constructors obtain the geometry of the invisible application shell created by **VkApp** and assign that geometry to the window's popup shell widget. The constructors also set the XmNargc and XmNargv resources on the popup shell to the values of **VkApp::argc()** and **VkApp::argv()** respectively. ("Application Data Access Functions" on page 76 describes **VkApp::argc()** and **VkApp::argv()**.)

Finally, for all windows, the window class constructors register a callback function to handle messages from the window manager. The default action upon receiving a WM_DELETE_WINDOW message is to delete the window object. To change this behavior, override the **handleWmDeleteMessage()** member function as described in "Window Properties and Shell Resources" on page 98. The default action upon receiving a WM_QUIT_APP message is to quit the application. To change this behavior, override the **handleWmQuitMessage()** member function as described in "Window Properties and Shell Resources".

Window Class Destructors

The **VkSimpleWindow** and **VkWindow** destructors delete all privately allocated data and destroy the views associated with the windows. The **VkWindow** destructor also destroys any menu bar associated with the window, no matter how you added it (see "Menu Bar Support" on page 100). If you created a subclass, you should provide a destructor to free any space that you explicitly allocated in the derived class.

The **VkSimpleWindow** and **VkWindow** destructors also remove the window from the application's list of windows. If this window is the only window still associated with the application (for example, if it is the only window created or all other windows have also been deleted), then your application automatically calls **VkApp::terminate()** to quit itself. "Quitting ViewKit Applications" on page 58 describes **VkApp::terminate()**.

Creating the Window Interface

There are three methods that you can use to create the contents of a window:

- Create a subclass of **VkSimpleWindow** or **VkWindow** and define the interface in the class constructor
- Create a subclass of **VkSimpleWindow** or **VkWindow** and define the interface by overriding the virtual function **setUpInterface()**
- Create an instance of **VkSimpleWindow** or **VkWindow**, define the interface separately, and then add the interface as the window's view

These methods, and the advantages and disadvantages of each approach, are discussed in the following sections.

Creating the Window Interface in the Constructor

The preferred method of defining the contents of a window is to create the interface in the constructor of a **VkSimpleWindow** or **VkWindow** subclass. In this case, you simply create the widgets and components that you want to appear in your window in your subclass constructor. Remember that each window can have only one direct child widget as a view, so in most cases you must create a container widget and then create all other widgets and components as descendents of this direct child. Manage all widgets except the container widget, which you should leave unmanaged.

The parent widget of your view's top-level widget or component must be the window's XmMainWindow widget. You can retrieve this widget by calling the **mainWindowWidget()** function inherited from **VkSimpleWindow**. "Window Data Access Functions" on page 96 discusses the **mainWindowWidget()** function.

Note: The *_baseWidget* data member for **VkSimpleWindow** and derived classes is the window's popup shell widget. Do not assign any other widget to this data member in a derived class.

After creating your interface, call **addView()**:

```
void addView(Widget w)
void addView(VkComponent *component)
```

addView() accepts as an argument either a widget or a pointer to a component, which **addView()** installs as the view for the window.

Note: Some IRIS IM functions such as **XmCreateScrolledText(3Xm)** create a ScrolledWindow widget and a child widget, and then return the ID of the child widget. As a convenience for using these functions, **addView()** can automatically determine the correct parent widget if you provide the child widget ID instead of the ScrolledWindow ID.

Example 4-1 shows a simple example that defines **ScaleWindow**, which creates a window with a RowColumn widget containing three Scale widgets. Because **ScaleWindow** is derived from **VkSimpleWindow**, it does not support a menu bar. If you required a menu bar, you would instead derive this class from **VkWindow**.

Note that **ScaleWindow** includes default resources for the Scale widget labels. This encapsulation technique is a good object-oriented practice to follow when creating reusable components in ViewKit. For example, if you were to extend this class by adding callback functions to the Scale widgets, you should make the callback functions members of the **ScaleWindow** class.

Example 4-1 Example of Creating a Window Interface in the Class Constructor

```
///////////////////////////
// ScaleWindow.h
///////////////////////////

#include <Vk/VkSimpleWindow.h>

class ScaleWindow: public VkSimpleWindow {

  public:
    ScaleWindow (const char *);
    ~ScaleWindow();
    virtual const char* className();

  private:
    static String _defaultResources[];
};

///////////////////////////
// ScaleWindow.c++
///////////////////////////

#include "ScaleWindow.h"
#include <Xm/RowColumn.h>
#include <Xm/Scale.h>

String ScaleWindow::_defaultResources[] = {
    "*dayScale.titleString:    Days",
    "*weekScale.titleString:   Weeks",
    "*monthScale.titleString:  Months",
    NULL };

ScaleWindow::ScaleWindow (const char *name) : VkSimpleWindow (name)
{
    setDefaultResources(mainWindowWidget(), _defaultResources);

    Widget scales = XtCreateWidget("scales", xmRowColumnWidgetClass,
                               mainWindowWidget(), NULL, 0);

    Widget dayScale = XtCreateManagedWidget("dayScale", xmScaleWidgetClass,
                                        scales, NULL, 0);
    XtVaSetValues(dayScale,
                  XmNorientation, XmHORIZONTAL,
                  XmNminimum, 1,
                  XmNmaximum, 7,
                  XmNvalue, 1,
                  XmNshowValue, TRUE,
                  NULL);

    Widget weekScale = XtCreateManagedWidget("weekScale", xmScaleWidgetClass,
                                            scales, NULL, 0);
    XtVaSetValues(weekScale,
                  XmNorientation, XmHORIZONTAL,
                  XmNminimum, 1,
                  XmNmaximum, 52,
                  XmNvalue, 1,
                  XmNshowValue, TRUE,
                  NULL);
    Widget monthScale = XtCreateManagedWidget("monthScale", xmScaleWidgetClass,
                                             scales, NULL, 0);
    XtVaSetValues(monthScale,
                  XmNorientation, XmHORIZONTAL,
                  XmNminimum, 1,
                  XmNmaximum, 12,
                  XmNvalue, 1,
                  XmNshowValue, TRUE,
                  NULL);

    addView(scales);
}

ScaleWindow::~ScaleWindow()
{
    // Empty
}

const char* ScaleWindow::className()
{
    return "ScaleWindow";
}
```

```
///////////////////////////
// scaleApp.c++
///////////////////////////

#include "ScaleWindow.h"
#include <Vk/VkApp.h>

void main ( int argc, char **argv )
{
    VkApp *scaleApp = new VkApp("ScaleApp", &argc, argv);
    ScaleWindow *scaleWin = new ScaleWindow("scaleWin");

    scaleWin->show();
    scaleApp->run();
}
```

Running the *scaleApp* program shown above displays a **ScaleWindow**, as shown in Figure 4-3.

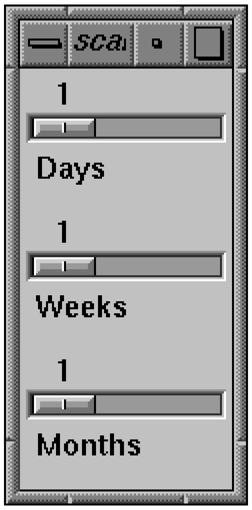


Figure 4-3 A Simple Example of a **VkSimpleWindow** Subclass

You can also create components and add them just as you would widgets. The constructor shown in Example 4-2 creates a **VkRadioBox(3Vk)** component and installs several items.

Example 4-2 Using a Component as a Window's View

```
//////////////////////////
// RadioWindow.h
//////////////////////////

#include <Vk/VkSimpleWindow.h>

class RadioWindow: public VkSimpleWindow {

  public:
    RadioWindow (const char *);
    ~RadioWindow();
    virtual const char* className();
  private:
    static String _defaultResources[];
};
//////////////////////////
// RadioWindow.c++
//////////////////////////

#include "RadioWindow.h"
#include <Vk/VkRadioBox.h>

String RadioWindow::_defaultResources[] = {
    "*color*label*labelString:  Color",
    "*red.labelString:    Red",
    "*green.labelString:  Green",
    "*blue.labelString:   Blue",
    NULL };

RadioWindow::RadioWindow (const char *name) : VkSimpleWindow (name)
{
    setDefaultResources(mainWindowWidget(), _defaultResources);

    VkRadioBox *rb = new VkRadioBox( "color", mainWindowWidget() );

    rb->addItem("red");
    rb->addItem("green");
    rb->addItem("blue");

    addView(rb);
}
```

```
RadioWindow::~RadioWindow()
{
    // Empty
}

const char* RadioWindow::className()
{
    return "RadioWindow";
}


///////////////////////////
// radioApp.c++
///////////////////////////

#include <Vk/VkApp.h>
#include "RadioWindow.h"

void main ( int argc, char **argv )
{
    VkApp *radioApp = new VkApp("RadioApp", &argc, argv);
    RadioWindow *radioWin = new RadioWindow("radioWin");

    radioWin->show();
    radioApp->run();
}
```

Running the *radioApp* program shown above displays a **RadioWindow** as shown in Figure 4-4.

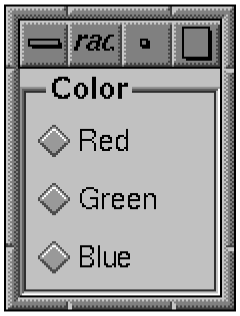


Figure 4-4 A Simple Example of Using a Component as a Window's View

Creating the Window Interface in the setUpInterface() Function

When you create your window interface in your window constructor using **addView()**, all setup overhead occurs when the window is instantiated. Additionally, your program allocates memory for all of the widgets created. Occasionally, you might need to instantiate a window so that your application can access some of its public functions, but not display it. If the window interface is large or complex, the time and memory consumed to create the interface is unnecessary if the user might not display it.

The ViewKit window classes provide a mechanism for delaying the creation of a window's interface until the window needs to be displayed. Rather than including the interface code in the window constructor, you can include the code in the definition of the protected virtual member function **setUpInterface()**.

When you call **show()** to display a window, **show()** checks to see whether you have already added a view to the window (for example, in the window's constructor). If not, **show()** calls **setUpInterface()** to create the window's interface.

Using this approach, you do not allocate memory for the window interface until your application actually displays the window for the first time—and you never allocate the memory if your application never displays the window. Additionally, this approach reduces your application's startup time. The trade-off is that the first time you display this window, the response time might be slow because your application must create the interface before displaying the window.

The syntax of **setUpInterface()** is:

```
virtual Widget setUpInterface(Widget parent)
```

show() passes the main window widget to **setUpInterface()** for you to use as the parent of the window's widget hierarchy. You must return a widget to be added as a view. Do not call **addView()** from within **setUpInterface()**.

Note: Some IRIS IM functions such as **XmCreateScrolledText(3Xm)** create a ScrolledWindow widget and a child widget, and then return the ID of the child widget. As a convenience for using these functions, **setUpInterface()** can automatically determine the correct parent widget if you provide the child widget ID instead of the ScrolledWindow ID.

Example 4-3 shows the **RadioWindow** example from Example 4-2 rewritten to use **setUpInterface()** instead of **addView()** in the constructor.

Example 4-3 Example of Creating a Window's Interface in the **setUpInterface()** Function

```
///////////////////////////
// RadioWindow2.h
///////////////////////////

#include <Vk/VkSimpleWindow.h>

class RadioWindow: public VkSimpleWindow {

  public:
    RadioWindow (const char *);
    ~RadioWindow();
    virtual const char* className();

  protected:
    Widget setUpInterface(Widget);

  private:
    static String _defaultResources[];
};


///////////////////////////
// RadioWindow2.c++
///////////////////////////

#include "RadioWindow2.h"
#include <Vk/VkRadioBox.h>

String RadioWindow::_defaultResources[] = {
    "*color*label*labelString:  Color",
    "*red.labelString:    Red",
    "*green.labelString:  Green",
    "*blue.labelString:   Blue",
    NULL };

RadioWindow::RadioWindow (const char *name) : VkSimpleWindow (name)
{
    // Empty
}
```

```
RadioWindow::~RadioWindow()
{
    // Empty
}

const char* RadioWindow::className()
{
    return "RadioWindow";
}

Widget RadioWindow::setUpInterface (Widget parent)
{
    setDefaultResources(mainWindowWidget(), _defaultResources);

    VkRadioBox *rb = new VkRadioBox( "color", parent );

    rb->addItem("red");
    rb->addItem("green");
    rb->addItem("blue");

    return(*rb);
}
```

Note that this example uses the Widget operator defined by **VkComponent** to return the **VkRadioBox**'s base widget in **setUpInterface()**. (See "VkComponent Access Functions" on page 17 for information on the Widget operator.) If you prefer, you could explicitly call **baseWidget()**:

```
return( rb->baseWidget() );
```

Adding a Window Interface to a Direct Instantiation of a ViewKit Window Class

There are exceptional cases for which you may choose to directly instantiate a **VkSimpleWindow** or **VkWindow** object and use **addView()** to associate a view with the window. For example, if you have a complex, self-contained component and need a window simply to display the component, you might find this method acceptable. Example 4-4 shows a simple example of adding a component to a direct instantation of the **VkSimpleWindow** class.

Example 4-4 Example of Adding a View to a Direct Instantiation of a ViewKit Window Class

```
VkSimpleWindow *roloWindow = VkSimpleWindow("roloWindow");
Rolodex *rolodex = Rolodex( "rolodex", roloWindow->mainWindowWidget() );
roloWindow->addView(rolodex);
```

In most cases, you should not use this technique because most windows require data and support functions that should be encapsulated by the window class to follow proper object-oriented programming style.

Replacing a Window’s View

Occasionally, you might want to replace the view of an existing window. To do so, you must first remove the current view using the **removeView()** function:

```
void removeView()
```

You should not call this function unless you have previously added a view to this window. **removeView()** does not destroy the view; if you no longer need the view, you should destroy it.

After removing a view, you can add another view using **addView()**.

Manipulating Windows

The **VkSimpleWindow** and **VkWindow** classes provide simple functions to show, hide, raise, lower, iconify, and open windows. All of the following functions take no arguments and have a void return value:

show()	Displays the window. **show()** has no effect if the window is currently iconified.
hide()	Removes the window from the screen.
iconify()	Iconifies the window.
open()	Opens the window if it is iconified.
raise()	Raises the window to the top of the application’s window stack.
lower()	Lowers the window to the bottom of the application’s window stack.

All of these functions are declared virtual. If you override them in a subclass, you should call the corresponding base class function after performing whatever operations your subclass requires.

Window Data Access Functions

The **VkSimpleWindow** and **VkWindow** classes support several data access functions:

- **mainWindowWidget()** returns the XmMainWindow widget created by the window constructor. Most frequently, you use **mainWindowWidget()** to obtain a parent widget for creating a view widget or component. You can also use this function to access and configure the window's XmMainWindow widget. For example, by default, the ViewKit window classes configure the window's XmMainWindow widget to not display scrollbars. You can use **mainWindowWidget()** to obtain the XmMainWindow widget and then use **XtSetValues(3Xt)** to enable the scrollbars.

  ```
  virtual Widget mainWindowWidget() const
  ```
- **viewWidget()** returns the widget currently installed as the window's view.

  ```
  virtual Widget viewWidget() const
  ```
- **visible()** returns TRUE if the window is currently displayed and FALSE if it is hidden.

  ```
  Boolean visible() const
  ```
- **iconic()** returns TRUE if the window is currently iconified and FALSE if it is not.

  ```
  Boolean iconic() const
  ```

Window Manager Interface

The **VkSimpleWindow** and **VkWindow** classes set up various properties on the shell window and provide simple hooks for window manager interactions.

Window and Icon Titles

The **VkSimpleWindow** and **VkWindow** classes provide easy-to-use functions to set your application's window and icon titles.

The **setTitle()** function sets the title of a window:

```
void setTitle(const char *newTitle)
```

The string is treated first as a resource name that **setTitle()** looks up relative to the window. If the resource exists, its value is used as the window title. If the resource does not exist, or if the string contains spaces or newline characters, **setTitle()** uses the string itself as the window title. This allows applications to dynamically change a window title without hard-coding the exact title names in the application code. Example 4-5 shows an example of setting a window title using a resource value.

You can retrieve the current window title using **getTitle()**:

```
const char *getTitle()
```

The **setIconName()** function sets the title of a window's icon:

```
void setIconName(const char *newTitle)
```

The string is treated first as a resource name that **setIconName()** looks up relative to the window. If the resource exists, its value is used as the window's icon title. If the resource does not exist, or if the string contains spaces or newline characters, **setIconName()** uses the string itself as the icon title. This allows applications to dynamically change a window's icon title without hard-coding the exact title names in the application code. Example 4-5 shows an example of setting a window's icon title using a resource value.

Example 4-5 Setting Window and Icon Titles Using Resource Values

```
class MainWindow : public VkSimpleWindow {

  public:
    MainWindow (const char *);
    // ...

  private:
    static String _defaultResources[];
    // ...
};

String _defaultResources[] = {
    "*winTitle:    Foobar Main Window",
    "*iconTitle:   Foobar",
    NULL
};
```

```
MainWindow::MainWindow(const char *name) : VkSimpleWindow(name)
{
    setDefaultResources(mainWindowWidget(), _defaultResources);

    setTitle("winTitle");
    setIconName("iconTitle");

    // ...
}
```

Window Properties and Shell Resources

The window class constructors automatically set up various window properties and shell resources when you create a window. The window classes also provide some hooks to allow you to set your own properties or change the window manager message handling in a derived class.

Because the first window you create is by default the main window, the window class constructors also set some shell resources on the popup shell widget of that window. The constructors obtain the geometry of the invisible application shell created by **VkApp** and assign that geometry to the window's popup shell widget. The constructors also set the XmNargc and XmNargv resources on the popup shell to the values of **VkApp::argc()** and **VkApp::argv()** respectively. ("Application Data Access Functions" on page 76 describes **VkApp::argc()** and **VkApp::argv()**.)

For all windows, the window class constructors register a callback function to handle WM_DELETE_WINDOW messages from the window manager. This callback function calls **handleWmDeleteMessage()**:

```
virtual void handleWmDeleteMessage()
```

By default, **handleWmDeleteMessage()** calls the window's **okToQuit()** function. If **okToQuit()** returns TRUE, then **handleWmDeleteMessage()** deletes the window. You can override **handleWmDeleteMessage()** to change how your window handles a WM_DELETE_WINDOW message. In most cases, you should simply perform any additional actions that you desire and then call the base class's **handleWmDeleteMessage()** function.

The window class constructors also register a callback function to handle WM_QUIT_APP messages from the window manager. This callback function calls **handleWmQuitMessage()**:

```
virtual void handleWmQuitMessage()
```

By default, **handleWmQuitMessage()** calls the application's **quitYourself()** function to quit the application. You can override **handleWmQuitMessage()** to change how your windows handles a WM_QUIT_APP message. In most cases, you should simply perform any additional actions that you desire and then call the base class's **handleWmQuitMessage()** function to exit your application.

If you want to set any additional properties on a window, you can override **setUpWindowProperties()**:

```
virtual void setUpWindowProperties()
```

setUpWindowProperties() is called after realizing a window's popup shell widget but before mapping it. Subclasses that wish to store other properties on windows can override this function and perform additional actions. If you override this function, you should set all desired properties and then call the base class's **setUpWindowProperties()** function.

Note that you should use **setUpWindowProperties()** to set window properties instead of **VkComponent::afterRealizeHook()** as described in "Displaying and Hiding Components" on page 19. The difference between the two is that **setUpWindowProperties()** is guaranteed to be called before the window manager is notified of the window's existence. Because of race conditions, this might not be true of **afterRealizeHook()**.

You can also change the value of the window manager class hint stored on a window using **setClassHint()**:

```
void setClassHint(const char *className)
```

setClassHint() sets the class resource element of the XA_WM_CLASS property stored on this window to the string you pass as an argument.

Menu Bar Support

The **VkSimpleWindow** class is useful for windows that require only a work area; however, windows frequently require menus. The **VkWindow** class extends the **VkSimpleWindow** class by providing support for a menu bar along the top of the window.

In ViewKit, the **VkMenuBar(3Vk)** class provides support for menu bars. Chapter 5, "Creating Menus With ViewKit," describes in depth the process of creating and manipulating menus; "Menu Bar" on page 149 describes additional functions specific to the **VkMenuBar** class and provides an example of constructing a menu bar for an application. This section describes only those functions provided by **VkWindow** for installing and manipulating a menu bar.

You install a menu bar using **setMenuBar()**:

```
void setMenuBar(VkMenuBar *menuObj)
void setMenuBar(VkMenuDesc *menudesc)
```

If you provide a pointer to an existing **VkMenuBar** object, **setMenuBar()** installs that menu bar. If you prove a VkMenuDesc static menu description, **setMenuBar()** creates a menu bar from that description and then installs the menu bar.

Once you have installed a menu bar, **menu()** will return a pointer to the menu bar object:

```
virtual VkMenuBar *menu() const
```

You can add a menu pane to the menu bar using **addMenuPane()**:

```
VkSubMenu *addMenuPane(const char *name)
VkSubMenu *addMenuPane(const char *name, VkMenuDesc *menudesc)
```

addMenuPane() creates a **VkSubMenu(3Vk)** object and adds it to the window's menu bar. If you provide a VkMenuDesc static menu description, **addMenuPane()** uses it to create the menu pane. Additionally, **addMenuPane()** automatically creates and installs a menu bar if the window does not currently have one.

You can add a menu pane that enforces radio behavior on the toggle items it contains using **addRadioMenuPane()**:

```
VkRadioSubMenu *addRadioMenuPane(const char *name)
VkRadioSubMenu *addRadioMenuPane(const char *name,
                                 VkMenuDesc *menudesc)
```

addRadioMenuPane() creates a **VkRadioSubMenu(3Vk)** object and adds it to the window's menu bar. If you provide a VkMenuDesc static menu description, **addRadioMenuPane()** uses it to create the menu pane. Additionally, **addRadioMenuPane()** automatically creates and installs a menu bar if the window does not currently have one.

Deriving Window Subclasses

This section summarizes how to create subclasses from the ViewKit window classes. It describes additional virtual functions and data members not covered in previous sections, provides a window creation checklist, and shows an example of deriving a window subclass.

Additional Virtual Functions and Data Members

In addition to those functions described in previous sections, the ViewKit window classes provide a number of virtual functions and data members that you can access from window subclasses. These functions and data allow you to:

- provide a "safe quit" mechanism for your window
- determine your window's state and perform actions on state changes
- perform actions after realizing a window
- handle raw events not normally handled by the Xt dispatch mechanism

Providing a "Safe Quit" Mechanism

The **VkComponent** class provides the virtual function **okToQuit()** to support "safe quit" mechanisms:

```
virtual Boolean okToQuit()
```

A component's **okToQuit()** function returns TRUE if it is "safe" for the application to quit. For example, you might want **okToQuit()** to return FALSE if a component is in the process of updating a file. By default, **okToQuit()** always returns TRUE; you must override **okToQuit()** for all components that you want to perform a check before quitting. Usually, only **VkSimpleWindow** and its subclasses use **okToQuit()**.

When you call **VkApp::quitYourself()**, **VkApp** calls the **okToQuit()** function for all registered windows before quitting. If the **okToQuit()** function for any window returns FALSE, the application does not exit. ("Quitting ViewKit Applications" on page 58 describes **VkApp::quitYourself()**.)

Also, the window's **handleWmDeleteMessage()** function calls **okToQuit()** when the window receives a WM_DELETE_WINDOW message from the window manager. This determines whether it is safe to delete the window. ("Window Properties and Shell Resources" on page 98 describes **handleWmDeleteMessage()**.)

If you want to perform a test to see whether it is safe to delete a window, override the window's **okToQuit()** function. If you want to check one or more components contained within a window, you can override the window's **okToQuit()** function so that it calls the **okToQuit()** functions for all the desired components. You can then override the **okToQuit()** functions for the other components so you can perform whatever checks or shutdown actions are necessary. For example, you could post a blocking dialog asking whether the user wants to save data before quitting. (Chapter 7, "Using Dialogs in ViewKit," describes how to use ViewKit dialogs.

Determining Window States

The ViewKit window classes provide the following protected data members for determining the current states of a window:

IconState *_iconState*
: Contains an enumerated constant of type IconState that describes the current iconification state of the window. This variable contains OPEN if the window is not iconified, CLOSED if it is iconified, and ICON_UNKNOWN if it is in an unknown state. (Typically, the unknown state is used only internally to the **VkSimpleWindow** class.)

VisibleState *_visibleState*
: Contains an enumerated constant of type VisibleState that describes the current visibility state of the window. This variable contains VISIBLE if the window is visible, HIDDEN if it is not visible, and VISIBLE_UNKNOWN if it is in an unknown state. (Typically, the unknown state occurs only before you add a view to your window.)

StackingState *_stackingState*

Contains an enumerated constant of type StackingState that describes the current stacking state of the window relative to the application. This variable contains RAISED if the window is at the top of the application's window stack, LOWERED if it is at the bottom of the window stack, and STACKING_UNKNOWN if it is in an unknown state (the state before you make any calls to **raise()** or **lower()** on this window).

If you need to perform any operations when your window changes its iconification state, you can override **stateChanged()**:

```
virtual void stateChanged(IconState newState)
```

stateChanged() is called whenever the window's iconification state changes, whether programmatically (by calls to **iconify()** and **open()**) or through window manager interaction. Because this function is responsible for maintaining the window's state information, if you override this function in a subclass you should call the base class's **stateChanged()** function before performing any additional operations.

Performing Actions After Realizing a Window

If you want to perform certain actions only after a window exists, you can override the **afterRealizeHook()** function inherited from **VkComponent**:

```
virtual void afterRealizeHook()
```

Note that you should use **setUpWindowProperties()** to set window properties instead of **afterRealizeHook()**. The difference between **afterRealizeHook()** and **setUpWindowProperties()** is that **setUpWindowProperties()** is guaranteed to be called before the window manager is notified of the window's existence. Because of race conditions, this might not be true of **afterRealizeHook()**. **afterRealizeHook()** is appropriate for performing actions that do not affect the window's interaction with the window manager.

Handling Raw Events

You can handle events not normally handled by the Xt dispatch mechanism by overriding the window's **handleRawEvent()** function:

```
virtual void handleRawEvent(XEvent *event)
```

As described in “ViewKit Event Handling” on page 56, **VkApp::run()** supports events not normally handled by the Xt dispatch mechanism. For example, **VkApp::run()** can handle client messages and events registered for non-widgets (such as a PropertyNotify event on the root window).

When **run()** receives an event not handled by the Xt dispatch mechanism, it calls the virtual function **VkApp::handleRawEvent()**, which passes the event to the **handleRawEvent()** function of each instance of **VkSimpleWindow** (or subclass) in the application. By default, these member functions are empty.

If you want a window to handle events through this mechanism, call **XSelectInput(3X)** to select the events that you want to receive, and override **handleRawEvent()** in the **VkSimpleWindow** subclass to implement your event processing.

Additional Data Members

The ViewKit window classes also provide the protected data member *_mainWindowWidge*t:

```
Widget _mainWindowWidget
```

*_mainWindowWidge*t contains the XmMainWindow widget created by the window constructor. In a subclass, you can use this data member instead of calling **mainWindowWidget()**, although this is not recommended.

Window Creation Summary

The following is a summary of guidelines for creating subclasses of the ViewKit window classes:

- Decide whether this window requires a menu bar. If it does, derive your subclass from **VkWindow**; otherwise, derive it from **VkSimpleWindow**.
- In most cases where you provide a menu bar for your window, you should create it in the window class when you create the rest of your window’s interface.

- Determine whether users will often use your application without displaying this window even after the object is instantiated. If so, and the window interface is large or complex, you might consider creating the window interface using **setUpInterface()** to reduce the time it takes to start your application; otherwise, create the interface in the window's constructor.
- Implement the window interface as a single-rooted widget subtree whose parent is the window's XmMainWindow widget (obtained by the **mainWindowWidget()** function). While some windows might contain only a single complex component, the majority of windows must create some type of container widget as the root of the window's interface; all other widgets and components are descendents of this widget.
- Do not assign any widget to the *_baseWidget* data member. The ViewKit window classes assign the window's popup shell widget to *_baseWidget*.
- Wherever appropriate, use resource values to set labels, other interface characteristics, and user-configurable component behavior. Define a default resource list as a static member variable of your window class, and call **setDefaultResources()** to set your window's default resources before creating the window interface.
- Override the **className()** function to return the name of your window's class.
- In addition to the widgets and components composing the window's interface, encapsulate any other required data and support functions as members of your window class.
- If you explicitly allocate any memory in your derived window class, remember to free it in the window's destructor.
- To explicitly set your window's title or its icon's title, call **setTitle()** or **setIconName()** respectively. You can also set these characteristics using the normal resource mechanisms.
- To provide a "safe quit" mechanism for your window, override **okToQuit()** to perform any checking you want to perform before deleting the window.
- To change how your window handles a WM_DELETE_MESSAGE from the window manager, override **handleWmDeleteMessage()**.
- To change how your window handles a WM_QUIT_APP from the window manager, override **handleWmQuitMessage()**.

- To set any additional properties on your window, override **setUpWindowProperties()**.
- To change the value of the window manager class hint stored on a window, call **setClassHint()**.
- To perform certain actions only after the window exists, override **afterRealizeHook()**.
- To handle events not normally handled by the Xt dispatch mechanism, call **XSelectInput(3X)** to select the events that you want to receive, and override **handleRawEvent()** in your window subclass to implement your event processing.

Window Subclassing Example

The program in Example 4-6 creates **ColorWindow**, a **VkSimpleWindow** subclass that implements a simple utility for determining the results of mixing primary ink colors when printing. The user can use toggles to select any of the three primary colors—cyan, magenta, and yellow—and the window reports the resulting color.

Figure 4-5 shows the widget hierarchy of the **ColorWindow** subclass. The **VkSimpleWindow** constructor creates the window's popup shell and XmMainWindow widget. The **ColorWindow** constructor creates a Form widget to serve as the window's view. The constructor adds a **VkCheckBox** component as a child of the Form to provide the toggle buttons. The constructor then adds a Frame widget as a child of the Form widget, and creates two Label gadgets as children of the Frame: one to serve as a title, and one to report the resulting color. The constructor manages all of these widgets except for the top-level Form widget. (The constructor manages the **VkCheckBox** component by calling its **show()** member function.)

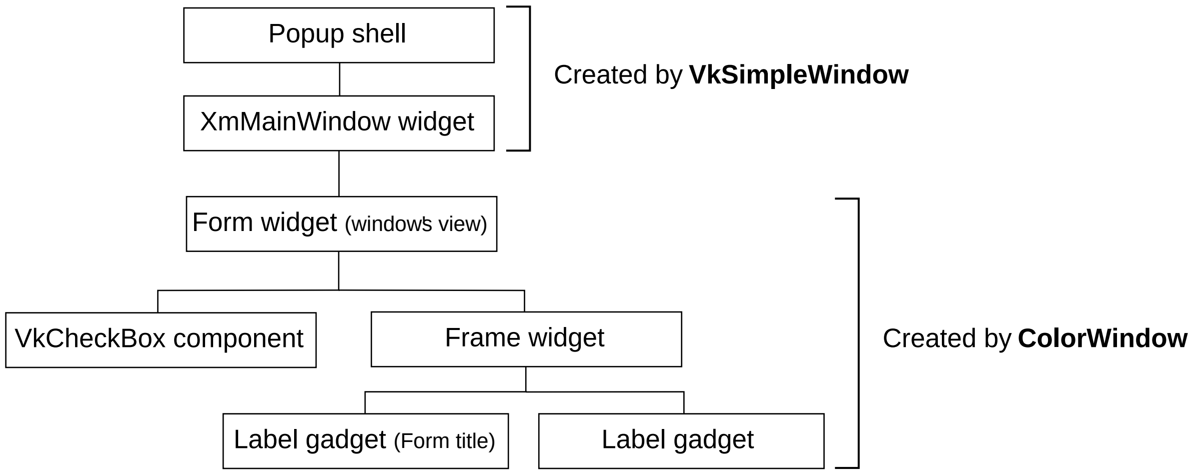

Popup shell
XmMainWindow widget
Created by **VkSimpleWindow**
Form widget (window's view)
VkCheckBox component
Frame widget
Label gadget (Form title)
Label gadget
Created by **ColorWindow**

Figure 4-5 Widget Hierarchy of **ColorWindow** Subclass

This example illustrates a number of object-oriented techniques that you should follow when programming in ViewKit. Note that all data and utility functions used by the window are declared as members of the **ColorWindow** class. Also note that **ColorWindow** uses resources to set all the text that it displays. It includes a set of default values, but you can override these values in a resource file (for example, to provide German-language equivalents for all the strings).

Example 4-6 Example of Creating a Window Subclass

```
///////////////////////////
// ColorWindow.h
///////////////////////////

#include <Vk/VkSimpleWindow.h>
#include <Vk/VkCheckBox.h>

class ColorWindow: public VkSimpleWindow {

  public:
    ColorWindow (const char *);
    ~ColorWindow();
    virtual const char* className();
  private:
    void displayColor(char *);
    void colorChanged(VkCallbackObject *, void *, void *);
    static String _defaultResources[]; // Default resource values
    static String _colors[];           // Array of possible resulting colors
    Widget _resultColor;               // Label to display resulting color
    VkCheckBox *_primaries;            // Checkbox for setting colors
    int _colorStatus;                  // Bit-wise color status variable
                                       //     Bit 0: Cyan
                                       //     Bit 1: Magenta
                                       //     Bit 2: Yellow
                                       //   Also used as index into _colors[]
};


///////////////////////////
// ColorWindow.c++
///////////////////////////

#include "ColorWindow.h"
#include <Xm/RowColumn.h>
#include <Xm/Form.h>
#include <Xm/Frame.h>
#include <Xm/LabelG.h>
#include <Vk/VkCheckBox.h>
#include <Vk/VkResource.h>

// Default ColorWindow class resource values.

String ColorWindow::_defaultResources[] = {
    "*windowTitle:                  Color Mixer",
    "*iconTitle:                    Color Mixer",
    "*primaries*label*labelString:  Primary Colors",
    "*cyan.labelString:             Cyan",
    "*magenta.labelString:          Magenta",
    "*yellow.labelString:           Yellow",
    "*resultLabel.labelString:      Resulting Color",
    "*cyan:                         Cyan",
    "*magenta:                      Magenta",
    "*yellow:                       Yellow",
    "*blue:                         Blue",
    "*red:                          Red",
    "*green:                        Green",
    "*white:                        White",
    "*black:                        Black",
    NULL };
```

```
// Set _colors array to correspond to color values indicated by the
// bits in the _colorStatus variable.

String ColorWindow::_colors[] = {
    "white",
    "cyan",
    "magenta",
    "blue",
    "yellow",
    "green",
    "red",
    "black" };

ColorWindow::ColorWindow (const char *name) : VkSimpleWindow (name)
{
    Arg args[5];
    int n;

    // Set default resources for the window.

    setDefaultResources(mainWindowWidget(), _defaultResources);

    // Create a Form widget to use as the window's view.

    Widget _form = XmCreateForm(mainWindowWidget(), "form", NULL, 0);


    // Create a VkCheckBox object to allow users to select primary colors.
    // Add toggle buttons and set their intial values to FALSE (unselected).
    // The labels for the checkbox frame and the toggle buttons are set
    // by the resouce database.

    _primaries = new VkCheckBox( "primaries", _form );
    _primaries->addItem("cyan", FALSE);
    _primaries->addItem("magenta", FALSE);
    _primaries->addItem("yellow", FALSE);
    _primaries->addCallback(VkCheckBox::itemChangedCallback, this,
                            (VkCallbackMethod) &ColorWindow::colorChanged);
    _primaries->show();

    // Set constraint resources on checkbox's base widget.

    n = 0;
    XtSetArg(args[n], XmNtopAttachment, XmATTACH_FORM); n++;
    XtSetArg(args[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg(args[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetValues(_primaries->baseWidget(), args, n);

    // Create a frame to display the name of the resulting blended color.

    n = 0;
    XtSetArg(args[n], XmNtopAttachment, XmATTACH_FORM); n++;
    XtSetArg(args[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg(args[n], XmNrightAttachment, XmATTACH_FORM); n++;
    XtSetArg(args[n], XmNleftAttachment, XmATTACH_WIDGET); n++;
    XtSetArg(args[n], XmNleftWidget, _primaries->baseWidget()); n++;
    Widget _result = XmCreateFrame(_form, "result", args, n);
    XtManageChild(_result);

    // Create a frame title label.  The label text is set by the resource
    // database.

    n = 0;
    XtSetArg(args[n], XmNchildType, XmFRAME_TITLE_CHILD); n++;
    Widget _resultLabel = XmCreateLabelGadget( _result, "resultLabel", args, n);

    // Create the label to display the blended color name.

    _resultColor = XmCreateLabelGadget( _result, "resultColor", NULL, 0);

    // Set intial value of _colorStatus and label string to white (all off).

    _colorStatus = 0;
    displayColor(_colors[_colorStatus]);

    XtManageChild(_resultLabel);
    XtManageChild(_resultColor);

    // Add the top-level Form widget as the window's view.

    addView(_form);

    // Set the window title and the icon title.

    setTitle("windowTitle");
    setIconName("iconTitle");
}
```

```
ColorWindow::~ColorWindow()
{
    // Empty
}

const char* ColorWindow::className()
{
    return "ColorWindow";
}

// Given a color name, update the label to display the color

void ColorWindow::displayColor(char *newColor)
{
    Arg args[2];
    int n;

    // Common resource trick in ViewKit applications.
    // Given a string, check the resource database for a corresponding
    // value.  If none exists, use the string as the value.

    char *_colorName = (char *) VkGetResource(_baseWidget, newColor, "Color",
                                              XmRString, newColor);

    // Update the label

    XmString _label = XmStringCreateSimple(_colorName);
    n = 0;
    XtSetArg(args[n], XmNlabelString, _label); n++;
    XtSetValues(_resultColor, args, n);
    XmStringFree(_label);
}

// When the user changes the value of one of the toggles, update the
// display to show the new blended color.

void ColorWindow::colorChanged(VkCallbackObject *obj, void *, void *callData)
{
    ColorWindow *win = (ColorWindow *) obj;
    int index = (int) callData;

    // Update color status based on toggle value.  Set or rest the
    // status bit corresponding to the respective toggle.
    if (_primaries->getValue(index))
        _colorStatus |= 1<<index;
    else
        _colorStatus &= ~(1<<index);

    // Update the display to show the new blended color, using
    // _colorStatus as an index.

    displayColor(_colors[_colorStatus]);
}


///////////////////////////
// colors.c++
///////////////////////////

#include <Vk/VkApp.h>
#include "ColorWindow.h"

void main ( int argc, char **argv )
{
    VkApp *colorApp = new VkApp("ColorApp", &argc, argv);
    ColorWindow *colorWin = new ColorWindow("colorWin");

    colorWin->show();
    colorApp->run();
}
```

Figure 4-6 shows the **ColorWindow** window displayed by the *colors* program.

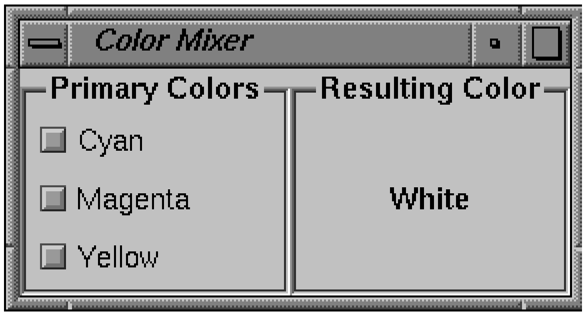


Figure 4-6 Example of the **ColorWindow** Window Subclass

QuickHelp

QuickHelp is a facility that displays a string when the pointer enters a widget. Help can be displayed in a message line at the bottom of the window, in a small balloon that pops up next to the pointer (i.e. “balloon help” or “popup help”), or both. Each can have its own separate help text—typically a brief phrase for popup help, and a more detailed message for the message line.

QuickHelp availability is controlled by the resources showHelp, showPopupHelp, and showMsgLineHelp:

- If showHelp is FALSE, no QuickHelp will be shown. This provides an easy way to enable or disable the entire QuickHelp system.
- If showPopupHelp is FALSE, popup help will not be shown. If showPopupHelp and showHelp are both TRUE, then popup help will be shown.
- If showMsgLineHelp is FALSE, no message line help will be shown. If showMsgLineHelp and showHelp are both TRUE, then message line help will be shown.

You must set these resources when you create the VkSimpleWindow**.** Space will only be allocated for the message line if message line help is enabled when the window is first created.

QuickHelp usability includes getting balloons promptly when you want them, but not getting them when you do not want them. This is a “Do What I Mean” problem, and so has no perfect solution. In an attempt to come as close as possible, QuickHelp has several timers.

The timers control how soon and how long a balloon will be displayed once the pointer enters a widget. The delay time before a balloon is displayed depends on whether the user is deemed to be in browse mode or non-browse mode. The user is considered to be in browse mode when the pointer enters two or more widgets in succession at a relatively slow speed. In this mode, all balloons after the first will be displayed more quickly. If the user stops browsing for a set length of time, the application will return to non-browse mode.

Since these timings greatly affect the usability of QuickHelp, they have been carefully set to minimize both the number of unwanted balloons, and the length of time users must wait to receive wanted help. If the default timings do not work for your application, you may reset them.

helpTextWaitTime
: The delay after entering a widget, when not in browse mode, before the QuickHelp balloon will be posted.

helpTextBrowseWaitTime
: The delay after entering a widget, when in browse mode, before the QuickHelp balloon will be posted.

helpTextTimeUp
: The length of time a QuickHelp balloon will remain posted.

helpTextBrowseCancelTime
: The length of time after leaving a widget before browse mode is cancelled.

helpTextBrowseVelocity
: The pointer velocity below which users will be considered to be browsing and above which they will be considered to be in transit. A QuickHelp balloon will be posted when users are browsing, but not when they are in transit.

QuickHelp also provides some miscellaneous resources:

helpTextInsensitive
: Controls whether or not QuickHelp is given when entering insensitive widgets.

smallWidget
: Determines where the help balloon will be displayed, in relation to the widget. If either dimension of a widget is below the number of pixels specified in the resource smallWidget, then the widget is considered to be a small widget.

  For a small widget, if the narrow dimension is it’s height, the balloon will be displayed below the widget (for example, a horizontal scroll bar). If the narrow dimension is its width, the balloon will be displayed beside the widget (for example, a vertical scroll bar).

  For large widgets, the balloon will be displayed near the part of the widget where the pointer first entered it.

Two other resources are intended for developers to use in debugging but may also be useful to some end-users.

dumpTree — Prints the name and class for each of the widgets in the widget tree at the time the dump is done. This can be useful as a starting point for creating the QuickHelp text for each widget.

Note: A common error is to forget that this cannot dump any widget that has not yet been created at the time of the dump. For example, unless you disable it, ViewKit creates menus later in a workProc. If dumpTree runs before the workProc is completed, those menus will not be included in the widget tree, because the tree will be dumped before the menus exist. For more information on creating menus, see VkMenu(3X), useWorkProcs(*Boolean)*.

showWidgetInfo

Causes QuickHelp to display the widget name, rather than any QuickHelp text. This can be useful when trying to figure out just what a widget is called so you can set a resource for it. For this to work, showHelp must be set to TRUE. If you want the widget name to be displayed in a popup balloon, showPopupHelp must be set to TRUE. If you want the widget name to be displayed on the message line showMsgLineHelp must be set to TRUE.

And, finally, there are two per-widget resources that provide the actual help strings, msgLineHelpText and popupHelpText. Both of these are of resource class QuickHelpText. If one of these resources is not set for any given widget, the user will not be shown that type of help message, even if showHelp, showPopupHelp, and showMsgLineHelp are all set to TRUE

Chapter 5

Creating Menus With ViewKit

This chapter introduces the basic ViewKit classes needed to create and manipulate the menus in a ViewKit application. Figure 5-1 shows the inheritance graph for these classes.

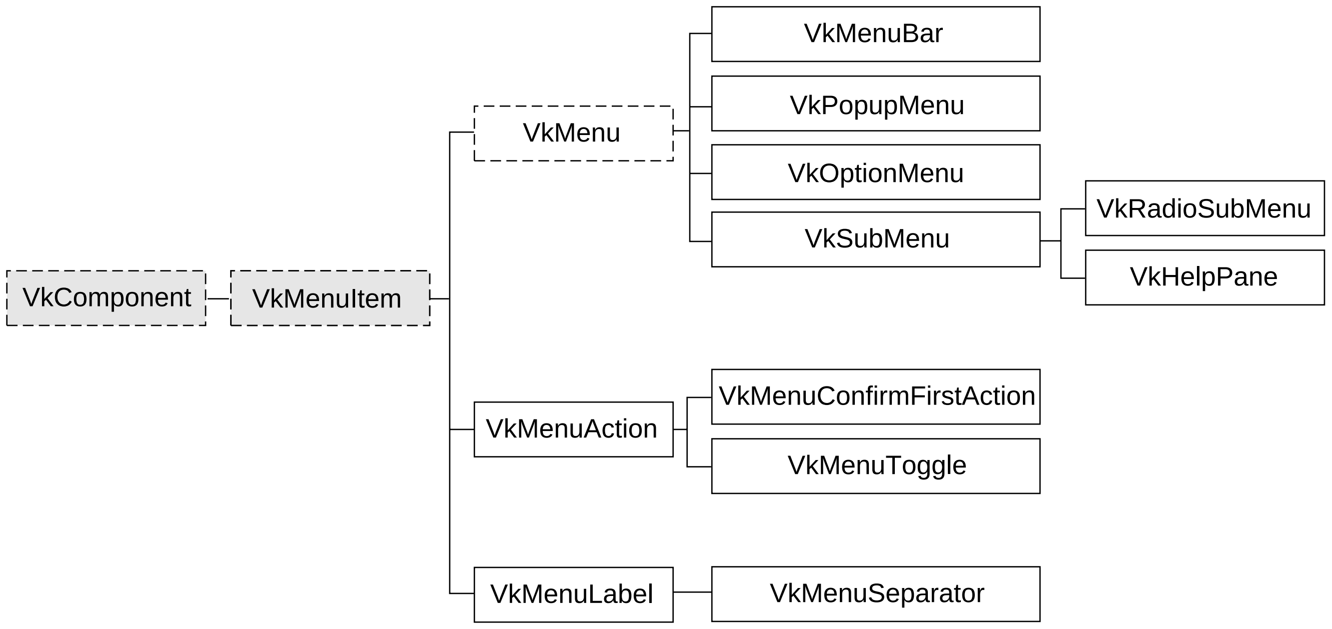


Figure 5-1 The Inheritance Graph for the ViewKit Menu Classes

Overview of ViewKit Menu Support

IRIS IM provides the components for building menus (buttons, menu shells, and so on) but does little to make menu construction easy. ViewKit provides a set of classes that facilitate common operations on menus, including creating menu bars, menu panes, popup menus, option menus, and cascading menu panes. The ViewKit menu package also provides an object-oriented interface for activating and deactivating menu items, dynamically adding, removing, or replacing menus items or menu panes, and performing other operations.

The basis for all ViewKit menu classes is the abstract class **VkMenuItem**, which is derived from **VkComponent**. There are two types of classes derived from **VkMenuItem**. The first serve as containers and correspond to the menu types supported by IRIS IM: popup menus, pulldown menu panes, menu bars, and option menus. The second type of derived classes are individual menu items: actions, toggles, labels, and separators.

The classes derived from **VkMenuItem** correspond closely with IRIS IM widgets and gadgets. For example, an action implemented as a **VkMenuAction** object represents a XmPushButton gadget along with an associated callback. However, the ViewKit menus offer several advantages over directly using IRIS IM widgets and gadgets. You can manipulate the menu objects more easily than widgets. You can display, activate, and deactivate items with a single function call. You can also easily move or replace items.

Caution: ViewKit implements menu items as gadgets rather than widgets. This causes a problem in callbacks and other situations if you try to use certain Xt functions (such as **XtDisplay(3Xt)**, **XtScreen(3Xt)**, and **XtWindow(3Xt)**), which expect widgets as arguments. Therefore, use the more general functions (such as **XtDisplayofObject(3Xt)**, **XtScreenofObject(3Xt)**, and **XtWindowofObject(3Xt)**) when you need this information for ViewKit menu items.

VkMenu, derived from **VkMenuItem**, is the abstract base class that implements the functionality needed to create and manipulate menus. It provides support for creating menus and adding, removing, replacing, finding, activating, and deactivating menu items.

Separate subclasses of **VkMenu** implement the various types of menus supported by ViewKit:

VkMenuBar Menu bars designed to work with the **VkWindow** class.

VkPopupMenu
Popup menus that automatically pop up when the user clicks the right mouse button over a widget.

VkOptionMenu
Option menus.

VkSubMenu Pulldown menu panes that can be used either as pulldown panes in a menu bar or pull-right panes in a popup or other pulldown menu.

VkRadioSubMenu
A subclass of **VkSubMenu** used to enforce radio behavior on toggle items that it contains.

VkHelpPane A ready-made menu pane that provides an interface to the standard help protocol supported by all ViewKit applications.

Individual menu items are implemented as subclasses derived from **VkMenuItem**:

VkMenuAction
A selectable menu item that performs an action, implemented as a PushButtonGadget.

VkMenuConfirmFirstAction
A selectable menu item that performs an action that the user must confirm before it is executed. When the user chooses this type of menu item, the application posts a question dialog asking the user for confirmation. The application performs the action only if the user VkHelpPane.3confirms it.

VkMenuToggle
A two-state toggle button gadget. To enforce radio behavior on a group of toggles, you must add them to a **VkRadioSubMenu** object.

VkMenuLabel A non-selectable label.

VkMenuSeparator
A non-selectable separator.

The ViewKit Menu Item Classes

This section describes the features of the ViewKit menu item classes. First it describes the features implemented by **VkMenuItem**, which are common to all the menu item classes. Then it describes the unique features of each individual menu item class.

Submenus are described in “Submenus” on page 151 and “Radio Submenus” on page 152.

Note: The header file <*Vk/VkMenuItem.h*> contains the declarations for all menu item classes.

Common Features of Menu Items

Individual menu items are implemented as subclasses derived from **VkMenuItem**, which provides a standard set of functions for accessing and manipulating menu items.

Unlike with many other ViewKit classes, you should never need to directly instantiate a menu item class. ViewKit automatically instantiates menu item objects as needed when you create menus, as described in “Constructing Menus” on page 127. Therefore, this guide does not describe the menu item constructors and destructors.

Keep in mind that ViewKit implements menu items as gadgets rather than widgets. If you need to directly access menu item gadgets, remember to use Xt functions that accept gadgets as well as widgets as arguments.

Displaying and Hiding Menu Items

The **VkMenuItem::show()** function makes a menu item visible when you display the menu to which it belongs:

```
void show()
```

By default, all menu items are visible when they are created (that is, they appear when you display the menu to which they belong). You do not have to explicitly call a menu item’s **show()** function to display it. You can call **show()** to display a menu item after you have hidden it with **hide()**.

The **VkMenuItem::hide()** function makes a menu item invisible when you display the menu to which it belongs:

```
void hide()
```

hide() does not remove the menu item from the menu, it simply unmanages the widget or gadget associated with a menu item. You can display a hidden menu item by calling its **show()** function.

If you want to remove a menu item from a menu, you can call **VkMenuItem::remove()**:

```
void remove()
```

remove() does not destroy a menu item, it simply removes the item from the menu hierarchy.

Note that instead of retaining pointers to all of your menu items and using **VkMenuItem::remove()** to remove menu items, you can instead use **VkMenu::removeItem()**. The effect is the same no matter which function you use, though typically you will find it easier to use the **VkMenu** function. “Removing Items From a Menu” on page 144 describes **VkMenu::removeItem()**.

Activating and Deactivating Menu Items

The **VkMenuItem::activate()** function makes a menu item sensitive so that it accepts user input (that is, a user can choose the item):

```
void activate()
```

By default, all menu items are activated (sensitive) when they are created.

The **VkMenuItem::deactivate()** function makes a menu item insensitive so that it does not accept user input (that is, a user cannot choose the item):

```
void deactivate()
```

When it is insensitive, the menu item appears “grayed out” when you display the menu to which it belongs. You can reactivate a menu item by calling its **activate()** function.

Note that instead of retaining pointers to all of your menu items and using **VkMenuItem::activate()** and **VkMenuItem::deactivate()** to activate and deactivate menu items, you can instead use **VkMenu::activateItem()** and **VkMenu::deactivateItem(),** respectively. The effect is the same no matter which functions you use, though typically it is easier to use the **VkMenu** functions. "Activating and Deactivating Items in a Menu" on page 143 describes **VkMenuItem::activate()** and **VkMenuItem::deactivate()**.

Setting Menu Item Labels

Generally, you set the label for a menu item by setting a value in the resource database for that item's XmNlabelString resource. For example, if you have a menu item named "addPage," you can set the label for that item by including a resource specification such as:

```
*addPage.labelString:    Add Page
```

If you do not set the menu item's XmNlabelString resource, ViewKit uses the item's name.

In some cases, you might need to set the label of an item programmatically. For example, in a page layout system, you might want to change the labels for the items in an Edit menu to reflect the type of object the user has currently chosen.
You can change a menu item's label programmatically with the **setLabel()** function:

```
virtual void setLabel(const char * str)
```

The string is treated first as a resource name that **setLabel()** looks up relative to the menu item's widget. If the resource exists, its value is used as the item's label. If the resource does not exist, or if the string contains spaces or newline characters, **setLabel()** uses the string itself as the item's label. This allows applications to dynamically set and change menu item labels without hard-coding the exact label strings in the application code.

Setting the Position of Menu Items

By default, ViewKit inserts items into a menu in the order you specify them. Therefore, the easiest way to set the positions of menu items is to add them to the menu in the order that you want them to appear.

Occasionally you might need to explicitly set the position of a menu item. To do so, use **VkMenuItem::setPosition()**:

```
void setPosition(int position)
```

setPosition() sets the item's position in the menu. You can specify any integer value from zero to the number of items in the menu; a value of zero specifies the first position in the menu. **setPosition()** ignores invalid values.

Note: **setPosition()** is effective only before ViewKit realizes the menu to which the menu item belongs. If you call **setPosition()** after realizing a menu, it has no effect. For example, if you create a menu bar in a window's constructor, you can safely use **setPosition()** to position menu items; however, after calling the window's **show()** function, **setPosition()** has no effect.

Menu Items Utility Functions

You can use **MenuItem::menuType()** to determine the specific menu item type when given a pointer to a **VkMenuItem** object:

```
virtual VkMenuItemType menuType()
```

menuType() returns one of the following enumerated values of type VkMenuItem::VkMenuItemType:

ACTION	A **VkMenuAction** object
CONFIRMFIRSTACTION	A **VkMenuConfirmFirstAction** object
TOGGLE	A **VkMenuToggle** object
LABEL	A **VkMenuLabel** object
SEPARATOR	A **VkMenuSeparator** object
SUBMENU	A **VkSubMenu** object
RADIOSUBMENU	A **VkRadioSubMenu** object
BAR	A **VkMenuBar** object

OPTION	A **VkOptionMenu** object
POPUP	A **VkPopupMenu** object
OBJECT	A user-defined subclass of **VkMenuActionObject** (described in "Command Classes" on page 180)

You can also determine when an object pointed to by a **VkMenuItem** pointer is a menu by calling **MenuItem::isContainer()**:

```
virtual Boolean isContainer()
```

isContainer() returns TRUE if the **VkMenuItem** object is an item that can "contain" other menu items (in other words, a menu).

Menu Actions

The **VkMenuAction** class provides a selectable menu item that performs an action. A **VkMenuAction** object is implemented as a PushButtonGadget.

A **VkMenuAction** object has associated with it a callback function that performs an operation and, optionally, a callback function that "undoes" the operation. You specify these callback functions when you add the item to a menu using one of the methods described in "Constructing Menus" on page 127. Consult that section for information on using **VkMenuAction** objects in a menu.

VkMenuAction provides a couple of public functions in addition to those implemented by **VkMenuItem**:

- You can determine whether an action has an undo callback associated with it by calling **VkMenuAction::hasUndo()**:

  ```
  Boolean hasUndo()
  ```

  hasUndo() returns TRUE if the object has an associated undo callback function.
- If an object has an undo callback function, you can call it programmatically using **VkMenuAction::undo()**:

  ```
  virtual void undo()
  ```

Typically, you won't have any need to call **undo()** explicitly. ViewKit provides automatic undo handling for your application using the **VkUndoManager** class, as described in Chapter 6, "ViewKit Undo Management and Command Classes." All you have to do is provide undo callback functions for your **VkMenuAction** objects and create an instance of **VkUndoManager** as described in Chapter 6.

Confirmable Menu Actions

The **VkMenuConfirmFirstAction** class, which is derived from **VkMenuAction**, provides a selectable menu item that performs an action. When the user chooses this type of menu item, the application posts a question dialog asking the user for confirmation. The application performs the action only if the user confirms it.

Because the **VkMenuConfirmFirstAction** class is intended for irrecoverable actions (for example, deleting a file), **VkMenuConfirmFirstAction** objects do not support undo callback functions.

The **VkMenuConfirmFirstAction** class uses a PushButtonGadget to implement the menu choice and the **VkQuestionDialog(3Vk)** to implement the question dialog. (See "The Question Dialog" on page 206 for more information on the **VkQuestionDialog** class.)

The question displayed in the confirmation dialog is determined by the value of the resource "noUndoQuestion," which ViewKit looks up relative to the menu item's widget. For example, if you have a menu item named "quit," set the question text for that item by including a resource specification such as:

```
*quit.noUndoQuestion:    Do you really want to quit?
```

If you do not provide a value for this resource, ViewKit uses the default question: "This action cannot be undone. Do you want to proceed anyway?"

Menu Toggles

The **VkMenuToggle** class, which is derived from **VkMenuAction**, provides a two-state toggle as a menu item. To enforce radio behavior on a group of toggles, you must add them to a **VkRadioSubMenu** object; otherwise, **VkMenuToggle** objects exhibit simple checkbox-style behavior. A **VkMenuToggle** object is implemented as a ToggleButtonGadget.

In addition to the public functions provided by **VkMenuItem**, **VkMenuToggle** provides functions for setting and retrieving the toggle state:

- You can set the visual state of a **VkMenuToggle** object, without activating its associated callback, using **VkMenuToggle::setVisualState()**:

  ```
  void setVisualState(Boolean state)
  ```

  setVisualState() selects the toggle if *state* is TRUE, and deselects the toggle if *state* is FALSE.
- You can set the visual state of a **VkMenuToggle** object and activate its associated callback with **VkMenuToggle::setStateAndNotify()**:

  ```
  void setStateAndNotify(Boolean state)
  ```
- You can retrieve the current value of a **VkMenuToggle** object using **VkMenuToggle::getState()**:

  ```
  Boolean getState()
  ```

  getState() returns TRUE if the toggle is currently selected, and FALSE if it is currently deselected.

Menu Labels

The **VkMenuLabel** class provides a non-selectable label as a menu item. A **VkMenuLabel** object is implemented as a LabelGadget.

The **VkMenuLabel** class does not provide any public functions other than those implemented by **VkMenuItem**.

Menu Separators

The **VkMenuSeparator** class provides a non-selectable separator as a menu item. A **VkMenuSeparator** object is implemented as a SeparatorGadget.

The **VkMenuSeparator** class does not provide any public functions other than those implemented by **VkMenuItem**.

The ViewKit Menu Base Class

This section describes the abstract **VkMenu** class, which provides the basic features of the ViewKit menu classes. It describes how to construct menus, manipulate items contained in the menus, and use the menu access functions. Because all ViewKit menu classes are derived from **VkMenu**, the functions and techniques described in this section apply to all menu classes.

Constructing Menus

The methods of constructing menus are the same for all types of menus (menu bars, options menus, and so on). The examples in this section use the **VkMenuBar** class, but the principles are similar for any of the ViewKit menu classes.

You can build menus either by passing a static menu description to the class constructor for a menu, or by adding items dynamically through function calls. You can mix the two approaches, initially defining a static menu structure and then dynamically adding items as needed.

Constructing Menus From a Static Description

To construct a menu from a static description, you must create a VkMenuDesc array that describes the contents of the menu and then pass that array as an argument to an appropriate menu constructor. This section describes the format of the VkMenuDesc structure and provides examples of its use.

The VkMenuDesc Structure

The definition for the VkMenuDesc structure is:

```
struct VkMenuDesc {
    VkMenuItemType    menuType;
    char             *name;
    XtCallbackProc    callback;
    VkMenuDesc       *submenu;
    XtPointer         clientData;
    XtCallbackProc    undoCallback;
};
```

The purposes of the VkMenuDesc fields are:

menuType	The type of menu item. The value of this field must be one of the enumerated constants listed below.
name	The menu item's name, which is also used as the menu item's default label.
callback	An Xt-style callback procedure that is executed when this menu item is activated.
submenu	A pointer to an array of a VkMenuDesc structures that describes the contents of a submenu.
clientData	Data that is passed to the callback procedure when it is executed.
undoCallback	A callback procedure that can be executed to undo the effects of the actions of the activation callback. Implementation of support for undoing actions is described in Chapter 6, "ViewKit Undo Management and Command Classes."

The *menuType* parameter is an enumerated value of type VkMenuItemType. Possible values are:

ACTION	A selectable menu item, implemented as a **VkMenuAction** object
CONFIRMFIRSTACTION	A selectable menu item, implemented as a **VkMenuConfirmFirstAction** object, which performs an action that the user must confirm before it is executed
TOGGLE	A two-state toggle button gadget, implemented as a **VkMenuToggle** object
LABEL	A label, implemented as a **VkMenuLabel** object
SEPARATOR	A separator, implemented as a **VkMenuSeparator** object
SUBMENU	A cascading submenu, implemented as a **VkSubMenu** object
RADIOSUBMENU	A cascading submenu that acts as a radio-style pane, implemented as a **VkRadioSubMenu** object
END	A constant that must terminate all menu descriptions.

Not all fields are used for each menu item type. Table 5-1 summarizes the optional and required fields for each menu item type.

Table 5-1 Required and Optional Parameters in a Static Menu Description

menuType	**name**	**callback**	**submenu**	**clientData**[a]	**undoCallback**
ACTION	R	O[b]	I	O	O
CONFIRMFIRSTACTION	R	O[b]	I	O	I
TOGGLE	R	O[b]	I	O	I
LABEL	R	I	I	I	I
SEPARATOR	I	I	I	I	I
SUBMENU	R	I	R	O[c]	I
RADIOSUBMENU	R	I	R	O[c]	I
END	R	I	I	I	I

R = Required parameter

O = Optional parameter

I = Ignored parameter

a. If you provide a default client data argument to the menu constructor, that value is used for all menu items for which you do not explicitly provide a client data parameter.

b. While this parameter is optional, the menu item is useless unless you provide a callback function.

c. If you provide a client data parameter, that value is used as default client data for all menu items in the submenu.

For example, consider the following array definition:

```
class EditWindow: public VkWindow {
  private:
    static VkMenuDesc editMenu[];
    // ...
};
```

```
VkMenuDesc EditWindow::editMenu[] = {
  { ACTION,   "Cut",   &EditWindow::cutCallback,
                              NULL,   NULL,   &EditWindow::undoCutCallback },
  { ACTION,   "Copy",  &EditWindow::copyCallback,
                              NULL,   NULL,   &EditWindow::undoCopyCallback },
  { ACTION,   "Paste", &EditWindow::pasteCallback,
                              NULL,   NULL,   &EditWindow::undoPasteCallback },
  { ACTION,   "Search" &EditWindow::searchCallback }
  { SEPARATOR },
  { CONFIRMFIRSTACTION, "Revert", &EditWindow::revertCallback },
  { END }
};
```

The *editMenu* array describes a simple menu for editing in an application. The menu consists of five actions and a separator. The menu's "Cut" item calls the **cutCallback()** function when it is activated with no client data passed to it. "Cut" also supports an undo action through the **undoCutCallback()** function. The "Copy" and "Paste" items work similarly.

The "Search" action does not support an undo action. Presumably, the action performed by this item is either too complex to undo or is meaningless to undo.

The "Revert" item is implemented as a CONFIRMFIRSTACTION. When the user activates this item, the application posts a confirmation dialog to warn the user that the action cannot be undone.

As a more complex example, consider a menu that contains two submenus, each of which contains two selectable items. You could describe this menu with definitions such as:

```
class TextWindow: public VkWindow {
  private:
    static VkMenuDesc menu[];
    static VkMenuDesc applicationPane[];
    static VkMenuDesc editPane[];
    // ...
};

VkMenuDesc TextWindow::applicationPane[] = {
    { ACTION, "Open", &TextWindow::openCallback },
    { ACTION, "Save", &TextWindow::saveCallback },
    { END }
};
```

```
VkMenuDesc TextWindow::editPane[] = {
    { ACTION, "Cut",   &TextWindow::cutCallback },
    { ACTION, "Paste", &TextWindow::pasteCallback },
    { END }
};

VkMenuDesc TextWindow::menu[] = {
    { SUBMENU, "Application", NULL, applicationPane },
    { SUBMENU, "Edit",        NULL, editPane },
    { END }
};
```

After constructing a static menu description, you create it by passing it as an argument to a menu constructor. For example, to implement the menus defined above as a menu bar, you can execute:

```
VkMenuBar *menubar = new VkMenuBar(menu);
```

You can implement the same menu as a popup menu simply by passing the definition to a popup menu constructor:

```
VkPopupMenu *popup = new VkPopupMenu(menu);
```

Special Considerations for Xt Callback Client Data When Using Static Menu Descriptions

As described in “Using Xt Callbacks with Components” on page 21, when using Xt-style callbacks in ViewKit, pass the *this* pointer as client data to all Xt callback functions. Callback functions then retrieve this pointer, cast it to the expected component type, and call a corresponding member function.

However, you cannot use the *this* pointer when you define a static data member. To get around this limitation, menu constructors accept a *defaultClientData* argument. If you provide a value for this argument, any menu item that does not provide a client data argument uses this argument instead. This allows you to specify menus statically while still allowing you to use an instance pointer with Xt callbacks. The code fragment Example 5-1 illustrates this technique.

Example 5-1 Providing Default Client Data When Using Static Menu Descriptions

```
class SampleWindow: public VkWindow {

  private:
    static void oneCallback(Widget,  XtPointer, XtPointer);
    static void twoCallback(Widget,  XtPointer, XtPointer);
    static void cutCallback(Widget,  XtPointer, XtPointer);
    static void pasteCallback(Widget,  XtPointer, XtPointer);

    static VkMenuDesc applicationPane[];
    static VkMenuDesc editPane[];
    static VkMenuDesc menu[];

  public:
    SampleWindow(const char *name);

    // Other members
};
SampleWindow::SampleWindow(char *name) : VkWindow(name)
{
     setMenuBar(new VkMenuBar(menu, (XtPointer) this));

     // Other actions
}
```

Note: **VkWindow::addMenuPane()**, **VkWindow::addRadioMenuPane()**, and the form of the **VkWindow::setMenuBar()** function that accepts a VkMenuDesc array as an argument all automatically use the *this* pointer as default client data for the menu bars and menu panes that they create.

Example of a Creating a Menu Bar Using a Static Description

Example 5-2 illustrates using a static description of a menu tree to create a menu bar. The program creates its main window using **MyWindow**, a subclass of **VkWindow**. The menu description and all menu callbacks are contained within the **MyWindow** subclass definition.

Example 5-2 Example of Creating a Menu Bar Using a Static Description

```
#include <Vk/VkApp.h>
#include <Vk/VkWindow.h>
#include <Vk/VkMenu.h>
#include <iostream.h>
#include <Xm/Label.h>

class MyWindow: public VkWindow {
  private:
    static void sampleCallback( Widget,  XtPointer , XtPointer);
    static void quitCallback( Widget,  XtPointer , XtPointer);

    void quit();
    void sample();

    static VkMenuDesc subMenu[];
    static VkMenuDesc sampleMenuPane[];
    static VkMenuDesc appMenuPane[];
    static VkMenuDesc mainMenuPane[];

  public:
    MyWindow( const char *name);
    ~MyWindow();

    virtual const char* className();
};
MyWindow::MyWindow( const char *name) : VkWindow( name)
{
    Widget label =  XmCreateLabel(mainWindowWidget(), "a menu",
                                  NULL, 0);
    setMenuBar(mainMenuPane);
    addView(label);
}

MyWindow::~MyWindow()
{
    // Empty
}

const char* MyWindow::className()
{
    return "MyWindow";
}
```

```
// The menu bar is essentially a set of cascading menu panes, so the
// top level of the menu tree is always defined as a list of submenus

VkMenuDesc  MyWindow::mainMenuPane[] = {
  { SUBMENU, "Application",  NULL, MyWindow::appMenuPane },
  { SUBMENU, "Sample",       NULL, MyWindow::sampleMenuPane },
  { END }
};

VkMenuDesc MyWindow::appMenuPane[] = {
  { ACTION,   "One",     &MyWindow::sampleCallback },
  { ACTION,   "Two",     &MyWindow::sampleCallback },
  { ACTION,   "Three",   &MyWindow::sampleCallback },
  { SEPARATOR },
  { ACTION,   "Quit",    &MyWindow::quitCallback },
  { END },
};

VkMenuDesc MyWindow::sampleMenuPane[] = {
  { LABEL,    "Test Label" },
  { SEPARATOR },
  { ACTION,    "An Action",       &MyWindow::sampleCallback },
  { ACTION,    "Another Action",  &MyWindow::sampleCallback },
  { SUBMENU,   "A Submenu",       NULL, MyWindow::subMenu },
  { END },
};

VkMenuDesc MyWindow::subMenu[] = {
  { ACTION,  "foo",   &MyWindow::sampleCallback },
  { ACTION,  "bar",   &MyWindow::sampleCallback },
  { ACTION,  "baz",   &MyWindow::sampleCallback },
  { END },
};

void MyWindow::sample()
{
    cout << "sample callback" <<  "\n" << flush;
}

void MyWindow::sampleCallback(Widget, XtPointer clientData, XtPointer)
{
    MyWindow *obj = (MyWindow *) clientData;
    obj->sample();
}
```

```
void MyWindow::quitCallback ( Widget, XtPointer, XtPointer )
{
    theApplication->quitYourself();
}

void main(int argc, char **argv)
{
  VkApp     *myApp    = new VkApp("Menudemo",  &argc,  argv);
  MyWindow  *menuWin  = new MyWindow("MenuWindow");

  menuWin->show();
  myApp->run();
}
```

When you run this program, you see the window shown in Figure 5-2.

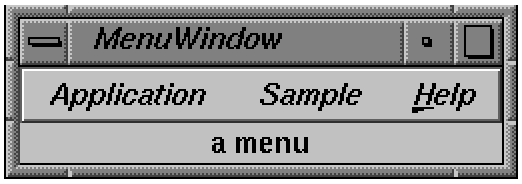


Figure 5-2 Main Window with Menu Bar Created by Static Description

The first pane, shown in Figure 5-3, contains three selectable entries (actions), followed by a separator, followed by a fourth action. The first three menu items simply invoke a stub function when chosen. The fourth item calls **quitCallback()**, which exits the application.

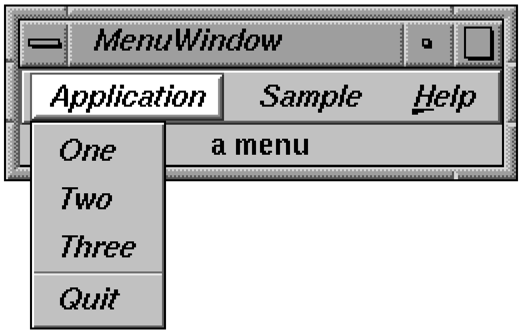


Figure 5-3 A Menu Pane Created by a Static Description

The second menu pane, shown in Figure 5-4, demonstrates a non-selectable label, a separator, and a cascading submenu.

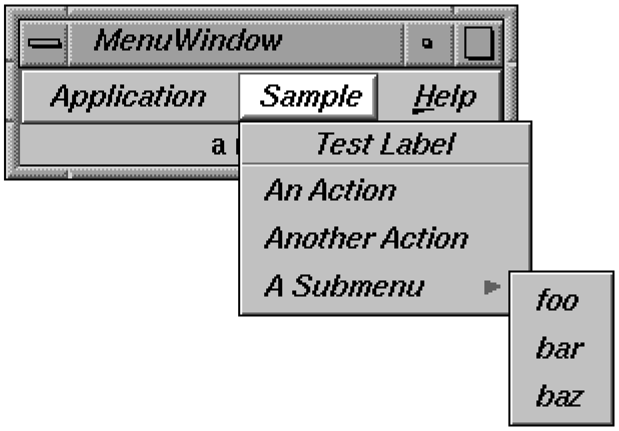


Figure 5-4 A Menu Pane Containing a Label and a Submenu

In addition to implementing these application-defined menu panes, ViewKit can automatically add a Help menu to a menu bar, which provides a user interface to a help system. “The ViewKit Help Menu” on page 165 describes the Help menu. Appendix C, “Using a Help System With ViewKit,” describes how to add an interface to an external help system to a ViewKit application.

Constructing Menus Dynamically

In addition to the static description approach demonstrated in the previous section, ViewKit allows applications to construct menus and menu items dynamically using functions defined in **VkMenu**. This section describes the menu-creation functions and provides examples of their use.

Functions for Dynamically Creating Menus

The **VkMenu** class provides a number of member functions for creating menus. Each function adds a single menu item to a given menu. You can use these functions at any time in your program. Even if you created a menu using a static definition, you can use these functions to add items to the menu.

VkMenu::addAction() adds to a menu a selectable menu action, implemented as a **VkMenuAction** object:

```
VkMenuAction *addAction(const char *name,
                        XtCallbackProc actionCallback = NULL,
                        XtPointer clientData = NULL,
                        int position = -1)

VkMenuAction *addAction(const char *name,
                        XtCallbackProc actionCallback,
                        XtCallbackProc undoCallback,
                        XtPointer clientData,
                        int position = -1)
```

addAction() creates a **VkMenuAction** object named *name* and adds it to the menu. By default, **addAction()** adds the item to the end of the menu; if you specify a value for *position*, **addAction()** adds the item at that position. *actionCallback* is the callback function that performs the item's action, and *undoCallback* is the callback function that undoes the action. If you do not provide an undo callback, the action cannot be undone and does not participate in the ViewKit undo mechanism as described in Chapter 6. *clientData* is client data passed to the callback functions. Following ViewKit conventions as described in "Using Xt Callbacks with Components" on page 21, pass the *this* pointer as client data so that the callback functions can retrieve the pointer, cast it to the expected component type, and call a corresponding member function.

VkMenu::addConfirmFirstAction() adds to a menu a selectable menu action, implemented as a **VkMenuConfirmFirstAction** object:

```
VkMenuConfirmFirstAction *
      addConfirmFirstAction(const char *name,
                            XtCallbackProc actionCallback = NULL,
                            XtPointer clientData = NULL,
                            int position = -1)
```

addConfirmFirstAction() creates a **VkMenuConfirmFirstAction** object named *name* and adds it to the menu. By default, **addConfirmFirstAction()** adds the item to the end of the menu; if you specify a value for *position*, **addConfirmFirstAction()** adds the item at that position. *actionCallback* is the callback function that performs the item's action, and *clientData* is client data passed to the callback function. As described above, pass the *this* pointer as client data.

VkMenu::addToggle() adds to a menu a selectable menu toggle, implemented as a **VkMenuToggle** object:

```
VkMenuToggle *addToggle(const char *name,
                        XtCallbackProc actionCallback = NULL,
                        XtPointer clientData = NULL,
                        int state = -1)
                        int position = -1)
```

addToggle() creates a **VkMenuToggle** object named *name* and adds it to the menu. By default, **addToggle()** adds the item to the end of the menu; if you specify a value for *position*, **addToggle()** adds the item at that position. If you provide a *state* argument, **addToggle()** sets the initial state of the toggle to that value. *actionCallback* is the callback function that performs the item's action, and *clientData* is client data passed to the callback function. As described above, pass the *this* pointer as client data.

VkMenu::addLabel() adds to a menu a non-selectable menu label, implemented as a **VkMenuLabel** object:

```
VkMenuLabel *addLabel(const char *name,
                      int position = -1)
```

addLabel() creates a **VkMenuLabel** object named *name* and adds it to the menu. By default, **addLabel()** adds the item to the end of the menu; if you specify a value for *position*, **addLabel()** adds the item at that position.

VkMenu::addSeparator() adds to a menu a non-selectable menu separator, implemented as a **VkMenuSeparator** object:

```
VkMenuSeparator *addSeparator(const char *name,
                              int position = -1)
```

addSeparator() creates a **VkMenuSeparator** object named *name* and adds it to the menu. By default, **addSeparator()** adds the item to the end of the menu; if you specify a value for *position*, **addSeparator()** adds the item at that position.

VkMenu::addSubmenu() adds to a menu a submenu, implemented as a **VkSubMenu** object:

```
VkSubMenu *addSubmenu(VkSubMenu *submenu,
                      int position = -1)

VkSubMenu *addSubmenu(const char *name,
                      int position = -1)
```

```
VkSubMenu *addSubmenu(const char *name,
                      VkMenuDesc *menuDesc)
                      XtPointer *defaultClientData = NULL)
                      int position = -1)
```

addSubmenu() is overloaded so that you can: 1) add an existing **VkSubMenu** object; 2) create and add a **VkSubMenu** object containing no items; or 3) create and add a **VkSubMenu** object from the static menu description, *menuDesc*. If you create and add the submenu using the static menu description, you can also provide a *defaultClientData* value that is used as the default client data for all items contained by the submenu. By default, **addSubmenu()** adds the item to the end of the menu; if you specify a value for *position*, **addSubmenu()** adds the item at that position.

Note: The m in **addSubmenu()** is lowercase, whereas the M in **VkSubMenu** is uppercase.

VkMenu::addRadioSubmenu() adds to a menu a submenu that enforces radio-style behavior on the toggle items it contains:

```
VkRadioSubMenu *addRadioSubmenu(VkRadioSubMenu *submenu,
                                int position = -1)

VkRadioSubMenu *addRadioSubmenu(const char *name,
                                int position = -1)

VkRadioSubMenu *addRadioSubmenu(const char *name,
                              VkMenuDesc *menuDesc)
                              XtPointer *defaultClientData = NULL)
                              int position = -1)
```

addRadioSubmenu() is overloaded so that you can: 1) add an existing **VkRadioSubMenu** object; 2) create and add a **VkRadioSubMenu** object containing no items; or 3) create and add a **VkRadioSubMenu** object from the static menu description, *menuDesc*. If you create and add the submenu using the static menu description, you can also provide a *defaultClientData* value that is used as the default client data for all items contained by the submenu. By default, **addSubmenu()** adds the item to the end of the menu; if you specify a value for *position*, **addSubmenu()** adds the item at that position.

Note: The m in **addRadioSubmenu()** is lowercase, whereas the M in **VkRadioSubMenu** is uppercase.

VkMenu::add() adds an existing menu item to a menu:

void add(VkMenuItem **item*, int *position* = -1)

By default, **add()** adds the item to the end of the menu; if you specify a value for *position*, **add()** adds the item at that position. Though you can use **add()** to add any type of menu item to a menu, you typically need it to add only the ViewKit undo manager and **VkMenuActionObject** objects. "Undo Management" on page 169 describes the ViewKit undo manager, and "Command Classes" on page 180 describes the **VkMenuActionObject** class.

Example of Creating a Menu Bar Dynamically

Example 5-3 is functionally equivalent to Example 5-2. It constructs a menu by adding items one at a time to the window's menu bar and to individual menu panes.

Example 5-3 Example of Creating a Menu Bar Dynamically

```
#include <Vk/VkApp.h>
#include <Vk/VkWindow.h>
#include <Vk/VkSubMenu.h>
#include <Vk/VkMenu.h>
#include <Xm/Label.h>
#include <iostream.h>

class MyWindow: public VkWindow {
  private:
    static void sampleCallback( Widget, XtPointer, XtPointer);
    static void quitCallback( Widget, XtPointer, XtPointer);

  protected:
    void sample();

  public:
    MyWindow( const char *name);
    ~MyWindow();

    virtual const char* className();
};
```

```
MyWindow::MyWindow( const char *name) : VkWindow( name)
{
    Widget label =  XmCreateLabel(mainWindowWidget(), "a menu", NULL, 0);

    // Add a menu pane

    VkSubMenu *appMenuPane = addMenuPane("Application");

    appMenuPane->addAction("One",   &MyWindow::sampleCallback,
                           (XtPointer) this);
    appMenuPane->addAction("Two",   &MyWindow::sampleCallback,
                           (XtPointer) this);
    appMenuPane->addAction("Three", &MyWindow::sampleCallback,
                           (XtPointer) this);
    appMenuPane->addSeparator();
    appMenuPane->addAction("Quit", &MyWindow::quitCallback,
                           (XtPointer) this);

    // Add a menu second pane

    VkSubMenu *sampleMenuPane = addMenuPane("Sample");

    sampleMenuPane->addLabel("Test Label");
    sampleMenuPane->addSeparator();
    sampleMenuPane->addAction("An Action",
                              &MyWindow::sampleCallback,
                              (XtPointer) this);
    sampleMenuPane->addAction("Another Action",
                              &MyWindow::sampleCallback,
                              (XtPointer) this);

    // Create a cascading submenu

    VkSubMenu *subMenu = sampleMenuPane->addSubmenu("A Submenu");

    subMenu->addAction("foo", &MyWindow::sampleCallback,
                       (XtPointer) this);
    subMenu->addAction("bar", &MyWindow::sampleCallback,
                       (XtPointer) this);
    subMenu->addAction("baz", &MyWindow::sampleCallback,
                       (XtPointer) this);

    addView(label);
}
```

```
MyWindow::~MyWindow()
{
    // Empty
}

const char* MyWindow::className() { return "MyWindow";}

void MyWindow::sampleCallback(Widget, XtPointer clientData, XtPointer)
{
    MyWindow *obj = (MyWindow *) clientData;
    obj->sample();
}

void MyWindow::sample()
{

    cout << "sample callback" << "\n" << flush;
}

void MyWindow::quitCallback ( Widget, XtPointer, XtPointer )
{
    theApplication->quitYourself();
}

void main(int argc, char **argv)
{
    VkApp  *myApp = new VkApp("Menu", &argc, argv);
    MyWindow  *w1  = new MyWindow("menuWindow");

    w1->show();
    myApp->run();
}
```

Manipulating Items in Menu

One of the advantages of the ViewKit menu system is the ability to manipulate the items in a menu after the menu has been created. The ViewKit menu system allows menu items to be manipulated by sending messages to any menu item. Menu items can also be found and manipulated by name.

Finding Items in a Menu

The **VkMenu::findNamedItem()** function allows you to find an item in a menu given its component name:

```
VkMenuItem *findNamedItem(const char *name,
                          Boolean caseless = FALSE)
```

findNamedItem() finds and returns a pointer to a menu item of the specified name belonging to the menu object or any submenus of the menu object. You can also pass an optional Boolean argument specifying whether or not the search is case-sensitive. If **findNamedItem()** finds no menu item with the given name, it returns NULL. If multiple instances of the same name exist, **findNamedItem()** returns the first name found in a depth-first search.

Note: Remember that you need to cast the return value if you need to access a member function provided by a **VkMenuItem** subclass. For example, if you search for a toggle item, remember to cast the return value to **VkMenuToggle** before calling a member function such as **VkMenuToggle::setVisualState()**.

Activating and Deactivating Items in a Menu

The **VkMenu::activateItem()** function makes a menu item sensitive so that it accepts user input (that is, a user can choose the item):

```
VkMenuItem *activateItem(const char *name)
```

You provide as an argument to **activateItem()** the name of the menu item to activate. This is the same name that you gave the menu item when you created it. **activateItem()** returns a **VkMenuItem** pointer to the item activated (or NULL if you did not provide a valid menu item name). By default, all menu items are activated (sensitive) when they are created.

The **VkMenu::deactivateItem()** function makes a menu item insensitive so that it does not accept user input (that is, a user cannot choose the item):

```
VkMenuItem *deactivateItem(const char *name)
```

You provide as an argument to **deactivateItem()** the name of the menu item to deactivate. This is the same name that you gave the menu item when you created it. **deactivateItem()** returns a **VkMenuItem** pointer to the item deactivated (or NULL if you did not provide a valid menu item name). When it is insensitive, the menu item appears "grayed out" when you display the menu. You can reactivate a menu item by calling **deactivateItem()** on that item.

Note that instead of using **VkMenu::activateItem()** and **VkMenu::deactivateItem()** to activate and deactivate menu items, you could retain pointers to all of your menu items and use **VkMenuItem::activate()** and **VkMenuItem::deactivate(),** respectively. The effect is the same no matter which functions you use, though typically it is easier to use the **VkMenu** functions. "Activating and Deactivating Menu Items" on page 121 describes **VkMenuItem::activate()** and **VkMenuItem::deactivate()**.

Removing Items From a Menu

If you want to remove a menu item from a menu, you can call **VkMenu::removeItem()**:

```
VkMenuItem *removeItem(const char *name)
```

You provide as an argument to **removeItem()** the name of the menu item to remove from the menu. This is the same name that you gave the menu item when you created it. **removeItem()** returns a **VkMenuItem** pointer to the item removed. **removeItem()** does not destroy a menu item; it simply removes the item from the menu hierarchy.

Note that instead of using **VkMenu::removeItem()**, you can retain pointers to all of your menu items and use **VkMenuItem::remove()**. The effect is the same no matter which functions you use, though typically you it is easier to use the **VkMenu** functions. "Displaying and Hiding Menu Items" on page 120 describes **VkMenuItem::remove()**.

Replacing Items in a Menu

You can replace an item in a menu with another menu item using **VkMenu::replace()**:

```
VkMenuItem *replace(const char *name, VkMenuItem *newItem)
```

replace() first uses **VkMenu::findNamedItem** to find the item specified by *name*. Then it removes that item from the menu and adds the menu item specified by *newItem* in its place. **replace()** returns a pointer to the menu item that you replaced.

Example of Manipulating Menu Items

The program in Example 5-4 allows users to dynamically add and remove items from a menu, and also to activate and deactivate items.

Example 5-4 Example of Manipulating Menu Items

```
#include <Vk/VkApp.h>
#include <Vk/VkWindow.h>
#include <Vk/VkMenu.h>
#include <Vk/VkSubMenu.h>
#include <Xm/Label.h>
#include <stream.h>
#include <stdlib.h>

class MyWindow: public VkWindow {

  private:
    static void addOneCallback       (Widget, XtPointer, XtPointer);
    static void removeOneCallback    (Widget, XtPointer, XtPointer);
    static void activateOneCallback  (Widget, XtPointer, XtPointer);
    static void deactivateOneCallback(Widget, XtPointer, XtPointer);
    static void sampleCallback       (Widget, XtPointer, XtPointer);
    static void quitCallback         (Widget, XtPointer, XtPointer);

  protected:
    VkSubMenu *_appMenuPane;
    VkSubMenu *_menuPaneTwo;

    void addOne();
    void removeOne();
    void activateOne();
    void deactivateOne();
    void sample();

  public:
    MyWindow( const char *name);
    ~MyWindow();
    virtual const char* className();
};
```

```
MyWindow::~MyWindow()
{
    // Empty
}

const char* MyWindow::className() { return "MyWindow";}

void MyWindow::sampleCallback(Widget, XtPointer clientData, XtPointer)
{
    MyWindow *obj = ( MyWindow * ) clientData;
    obj->sample();
}

void MyWindow::sample()
{
    cout << "sample callback" << "\n" << flush;
}

void MyWindow::addOneCallback(Widget, XtPointer clientData, XtPointer)
{
    MyWindow *obj = ( MyWindow * ) clientData;
    obj->addOne();
}

void MyWindow::addOne()
{
    _menuPaneTwo->addAction("A New Action", &MyWindow::sampleCallback,
                            (XtPointer) this);
}

void MyWindow::removeOneCallback(Widget, XtPointer clientData,
                                 XtPointer)
{
    MyWindow *obj = (MyWindow *) clientData;
    obj->removeOne();
}

void MyWindow::removeOne()
{
    _menuPaneTwo->removeItem("A New Action");
}
```

```
void MyWindow::activateOneCallback(Widget, XtPointer clientData,
                                   XtPointer)
{
    MyWindow *obj = (MyWindow *) clientData;
    obj->activateOne();
}

void MyWindow::activateOne()
{
    _menuPaneTwo->activateItem("A New Action");
}

void MyWindow::deactivateOneCallback(Widget, XtPointer clientData,
                                     XtPointer)
{
    MyWindow *obj = (MyWindow *) clientData;
    obj->deactivateOne();
}

void MyWindow::deactivateOne()
{
    _menuPaneTwo->deactivateItem("A New Action");
}

void MyWindow::quitCallback (Widget, XtPointer, XtPointer)
{
    theApplication->quitYourself();
}

MyWindow::MyWindow( const char *name) : VkWindow( name)
{
    Widget label =  XmCreateLabel(mainWindowWidget(), "a menu",
                                  NULL, 0);

    // Add a menu pane

    _appMenuPane = addMenuPane("Application");

    _appMenuPane->addAction("Add One",
                            &MyWindow::addOneCallback,
                            (XtPointer) this);
    _appMenuPane->addAction("Remove One",
                            &MyWindow::removeOneCallback,
                            (XtPointer) this);
    _appMenuPane->addAction("Activate One",
                            &MyWindow::activateOneCallback,
                            (XtPointer) this);
    _appMenuPane->addAction("Deactivate One",
                            &MyWindow::deactivateOneCallback,
                            (XtPointer) this);
    _appMenuPane->addSeparator();
    _appMenuPane->addAction("Quit",
                            &MyWindow::quitCallback,
                            (XtPointer) this );

    // Add a menu second pane

    _menuPaneTwo = addMenuPane("PaneTwo");

    addView(label);
}


void main(int argc, char **argv)
{
  VkApp  *myApp = new VkApp("MenuDemo3",  &argc,  argv);
  MyWindow  *menuWin  = new MyWindow("menuWindow");

  menuWin->show();
  myApp->run();
}
```

Menu Access Functions

The **VkMenu** class also provides access functions to help manipulate menu items.

You can determine the number of items currently associated with a menu by using **VkMenu::numItems()**:

```
int numItems() const
```

You can determine the position of an item in a menu with **VkMenu::getItemPosition()**:

```
int getItemPosition(VkMenuItem * item)
int getItemPosition(char *name)
int getItemPosition(Widget w)
```

You can specify the menu item by pointer, name, or widget. **getItemPosition()** returns the position of the item within the menu, with zero representing the first position in the menu.

As a convenience, you can also access items in a menu using standard array subscript notation:

```
VkMenuItem * operator[] (int index) const
```

For example, you can use **VkMenu::numItems()** with the array subscript notation to loop through an entire menu and perform an operation on all of the items it contains. For example, if *menubar* is a menu, the following code prints the name and class of each item in the *menubar* menu:

```
for ( i=0; i < menubar->numItems(); i++ )
    cout << "Name: " << (*menubar)[i]->name() << "\t"
         << "Class: " << (*menubar)[i]->className() << "\n";
```

Using ViewKit Menu Subclasses

This section describes the features of each ViewKit menu subclass. In addition to specific member functions listed, each class also supports all functions provided by the **VkMenu** class.

Menu Bar

The **VkMenuBar** class provides a menu bar designed to work with the **VkWindow** class. In addition to the functions described in this section, the **VkWindow** class provides some member functions for installing a **VkMenuBar** object as a menu bar. “Menu Bar Support” on page 100 describes the functions provided by **VkWindow**.

Examples of menu bar construction were given in “Example of a Creating a Menu Bar Using a Static Description” on page 132 (Example 5-2) and “Example of Creating a Menu Bar Dynamically” on page 140 (Example 5-3).

Menu Bar Constructors

There are four different versions of the **VkMenuBar** constructor:

```
VkMenuBar(Boolean showHelpPane = TRUE)

VkMenuBar(const char *name,
          Boolean showHelpPane = TRUE);

VkMenuBar(VkMenuDesc *menuDesc,
          XtPointer defaultCientData= NULL,
          Boolean showHelpPane = TRUE)

VkMenuBar(const char *name,
          VkMenuDesc *menuDesc,
          XtPointer defaultCientData = NULL,
          Boolean showHelpPane = TRUE)
```

To work with Silicon Graphics' color schemes, give the menu bar the name "menuBar." (For information on schemes, consult the schemes(3x) reference page.) The forms of the constructor that do not take a *name* argument automatically use the name "menuBar." You can specify another name, but schemes does not work correctly if you do.

If you use a form of the **VkMenuBar** constructor that accepts a *menuDesc* argument, the constructor creates a menu from the VkMenuDesc structure you provide.

Some forms of the constructor also accept an optional *defaultClientData* argument. If this argument is provided, any menu item that does not provide a client data argument uses this argument instead. This allows menus to be specified statically, while still allowing an instance pointer to be used with callbacks, as described in "Special Considerations for Xt Callback Client Data When Using Static Menu Descriptions" on page 131.

The last argument to each version of the constructor is a Boolean value that specifies whether the constructor should create a help pane that interfaces to the Silicon Graphics help system. The default is to automatically provide the help pane. The help pane is implemented by the **VkHelpPane** class, described in "The ViewKit Help Menu" on page 165.

Menu Bar Access Functions

The **VkMenuBar** class also provides the **helpPane()** member function:

```
VkHelpPane *helpPane() const
```

helpPane() returns a pointer to the menu bar's help pane. If the menu bar does not have a help pane, **helpPane()** returns NULL.

Submenus

The VkSubMenu class supports pulldown menu panes. You can use these menu panes within a menu bar (a **VkMenuBar** object), or as a cascading, pull-right menu in a popup or other pulldown menu.

Submenu Constructor

You should seldom need to instantiate a **VkSubMenu** object directly. You can add a submenu to any type of menu by calling that menu's **addSubmenu()** member function. You can also add menu panes to the menu bar of a **VkWindow** object by calling **VkWindow::addMenuPane()**.

For those cases where you need to instantiate a **VkSubMenu** object directly, the form of the constructor to use is:

```
VkSubMenu(const char *name,
          VkMenuDesc *menuDesc = NULL,
          XtPointer defaultClientData = NULL)
```

name specifies the name of the submenu. If you provide the optional *menuDesc* argument, the constructor creates a menu from the VkMenuDesc structure you provide. If you provide the optional *defaultClientData* argument, any menu item that does not provide a client data argument uses this argument instead. This allows menus to be specified statically, while still allowing an instance pointer to be used with callbacks, as described in "Special Considerations for Xt Callback Client Data When Using Static Menu Descriptions" on page 131.

Submenu Utility and Access Functions

The **VkSubMenu** class provides a couple of additional public member functions:

- IRIS IM supports tear-off menus, which enable the user to retain a menu pane on the screen. If tear-off behavior is enabled for a menu pane, a tear-off button, which has the appearance of a dashed line, appears at the top of the menu pane. The user can tear off the pane by clicking the tear-off button.

  By default, tear-off behavior is disabled for all menu panes. You can change the tear-off behavior of a submenu using **VkSubMenu::showTearOff()**:

  ```
  void showTearOff(Boolean showIt)
  ```

  If you pass the Boolean value TRUE to **showTearOff()**, the submenu displays the tear-off button; if you pass the value FALSE, it hides the tear-off button.

  You can also enable tear-off behavior for a menu by setting its XmNtearOffModel resource to XmTEAR_OFF_ENABLED (for example, in a resource file).

- You can access the RowColumn widget used to implement the submenu's pulldown pane by calling **VkSubMenu::pulldown()**:

  ```
  Widget pulldown()
  ```

Note: The **baseWidget()** function of a **VkSubMenu** object returns the CascadeButton widget required by IRIS IM pulldown menus.

Radio Submenus

The **VkRadioSubMenu** class, derived from **VkSubMenu**, supports pulldown menu panes. Its function is similar to that of **VkSubMenu**, but the RowColumn widget used as a menu pane is set to exhibit radio behavior. This class is intended to support one-of-many collections of **VkToggleItem** objects. You can use **VkRadioSubMenu** objects as menu panes within a menu bar (a **VkMenuBar** object), or as a cascading, pull-right menu in a popup or other pulldown menu.

It is seldom necessary to directly create a **VkRadioSubMenu** object. You can add radio submenus to any **VkMenuBar**, **VkPopupMenu**, or **VkSubMenu** by calling those classes' **addRadioSubmenu()** member function. You can also add menu panes to a **VkWindow** by calling **VkWindow::addRadioMenuPane()**.

Radio Submenu Constructor

You seldom need to instantiate a **VkRadioSubMenu** object directly. You can add a radio submenu to any type of menu by calling that menu's **addRadioSubmenu()** member function. You can also add radio menu panes to the menu bar of a **VkWindow** object by calling **VkWindow::addRadioMenuPane()**.

For those cases where you need to instantiate a **VkRadioSubMenu** object directly, the form of the constructor to use is:

```
VkRadioSubMenu(const char *name,
               VkMenuDesc *menuDesc = NULL,
               XtPointer defaultClientData = NULL)
```

name specifies the name of the radio submenu. If you provide the optional *menuDesc* argument, the constructor creates a menu from the VkMenuDesc structure you provide. If you provide the optional *defaultClientData* argument, any menu item that does not provide a client data argument uses this argument instead. This allows menus to be specified statically, while still allowing an instance pointer to be used with callbacks, as described in "Special Considerations for Xt Callback Client Data When Using Static Menu Descriptions" on page 131.

Radio Submenu Utility and Access Functions

The **VkRadioSubMenu** class does not provide any public member functions in addition to those provided by the **VkSubMenu** class. For information on the utility and access functions provided by **VkSubMenu**, see "Submenu Utility and Access Functions" on page 152.

Radio Submenu Example

Example 5-5 shows an example of using a **VkRadioSubMenu** class.

Example 5-5 Example of Using a **VkRadioSubMenu** Object

```
#include <Vk/VkApp.h>
#include <Vk/VkWindow.h>
#include <Vk/VkSubMenu.h>
#include <Vk/VkRadioSubMenu.h>
#include <Vk/VkMenu.h>
#include <Xm/Label.h>
#include <stream.h>
```

```
#include <stdlib.h>

class MyWindow: public VkWindow {

  private:

    static void sampleCallback( Widget,  XtPointer , XtPointer);
    static void quitCallback( Widget,  XtPointer , XtPointer);

  protected:

    void sample();

  public:

    MyWindow( const char *name);
    ~MyWindow();

    virtual const char* className();
};


MyWindow::~MyWindow()
{
    // Empty
}

void MyWindow::sampleCallback( Widget,  XtPointer clientData , XtPointer)
{
    MyWindow *obj = (MyWindow *) clientData;
    obj->sample();
}

const char* MyWindow::className() { return "MyWindow";}

void MyWindow::sample()
{

    cout << "In Sample Callback" << "\n" << flush;
}
```

```
void MyWindow::quitCallback ( Widget, XtPointer, XtPointer )
{
    exit(0);
}

MyWindow::MyWindow( const char *name) : VkWindow( name)
{
    Widget label =  XmCreateLabel(mainWindowWidget(), "a menu", NULL, 0);

    // Add a menu pane

    VkSubMenu *appMenuPane = addMenuPane("Application");

    appMenuPane->addAction("One",  &MyWindow::sampleCallback, (XtPointer) this);
    appMenuPane->addAction("Two",  &MyWindow::sampleCallback, (XtPointer) this);
    appMenuPane->addSeparator();
    appMenuPane->addAction("Quit", &MyWindow::quitCallback, (XtPointer) this);

    // Add a menu second pane

    VkSubMenu *sampleMenuPane = addMenuPane("Sample");

    sampleMenuPane->addLabel("Test Label");
    sampleMenuPane->addSeparator();
    sampleMenuPane->addAction("An Action", &MyWindow::sampleCallback,
                              (XtPointer) this);

    // Create a cascading submenu

    VkRadioSubMenu *subMenu = sampleMenuPane->addRadioSubmenu("A Submenu");

    subMenu->addToggle("foo", &MyWindow::sampleCallback, (XtPointer) this);
    subMenu->addToggle("bar", &MyWindow::sampleCallback, (XtPointer) this);
    subMenu->addToggle("baz", &MyWindow::sampleCallback, (XtPointer) this);

    addView(label);
}
```

```
void main(int argc, char **argv)
{
    VkApp  *myApp = new VkApp("Menu",  &argc,  argv);
    MyWindow  *w1  = new MyWindow("menuwindow");

    w1->show();

    myApp->run();
}
```

Option Menus

The **VkOptionMenu** class supports option menus. You can use this component anywhere in your interface.

Note: Unlike many other ViewKit components, **VkOptionMenu** objects are automatically visible when you create them; you do not need to call **show()** initially to display a **VkOptionMenu** object.

Option Menu Constructors

There are two different versions of the **VkOptionMenu** constructor that you can use:

```
VkOptionMenu(Widget parent,
             VkMenuDesc *menuDesc,
             XtPointer defaultClientData = NULL)

VkOptionMenu(Widget parent,
             const char *name = "optionMenu",
             VkMenuDesc *menuDesc = NULL,
             XtPointer defaultClientData = NULL)
```

You must provide a *parent* argument specifying the parent widget of the option menu.

To work with Silicon Graphics’ color schemes, give the option menu the name “optionMenu.” (For information on schemes, consult the schemes(3x) reference page.) The forms of the constructor that do not take a *name* argument automatically use the name “optionMenu.” You can specify another name, but schemes does work correctly if you do.

If you provide the optional *menuDesc* argument, the constructor creates a menu from the VkMenuDesc structure you provide.

If you provide the optional *defaultClientData* argument, any menu item that does not provide a client data argument uses this argument instead. This allows menus to be specified statically, while still allowing an instance pointer to be used with callbacks. This is described in “Special Considerations for Xt Callback Client Data When Using Static Menu Descriptions” on page 131.

Setting the Option Menu Label

To specify the string that is displayed as the option menu’s label, you must set the XmNlabelString resource for the menu’s label widget. To do so you can:

- Use the **VkComponent::setDefaultResources()** function to provide default resource values.
- Set resource values in an external app-defaults resource file. Any values you provide in an external file override values that you set using the **VkComponent::setDefaultResources()** function. This is useful when your application must support multiple languages; you can provide a separate resource file for each language supported.
- Set the resource value directly using the **XtSetValues()** function. Values you set using this method override any values set using either of the above two methods. You should generally avoid using this method as it “hard codes” the resource values into the code, making them more difficult to change.

Because all option menus must be named “optionMenu” to work with Silicon Graphics’ color schemes, if you set the label through a resource value, qualify the resource specifications with the name of a parent widget or component so that the X resource database can distinguish between instances of **VkOptionMenu**. For example, you can use resource specifications such as “*mainWindow*optionMenu*labelString” and “*graphWindow*optionMenu*labelString” to distinguish between an option menu that is a descendant of a “mainWindow” component and one that is a descendant of a “graphWindow” component, respectively.

Selecting Items in an Option Menu

You can programmatically set the selected item in an option menu using **VkOptionMenu::set()**:

```
void set(char* name)
void set(int index)
void set(VkMenuItem *item)
```

You can specify the selected item either by a pointer to the item, the item’s component name, or the item’s index (position) in the option menu, where the top item in the menu has an index of zero.

Determining Selected Items in an Option Menu

There are two functions that you can use to determine which item is selected in an option menu:

- You can retrieve the index (position) of the currently selected menu item using **VkOptionMenu::getIndex()**:

  ```
  int getIndex()
  ```

  getIndex() returns the index (position) of the selected item, where the top item in the menu has an index of zero.

- You can retrieve a pointer to the currently selected menu item using **VkOptionMenu::getItem()**:

  ```
  VkMenuItem *getItem()
  ```

Option Menu Utility Functions

Normally, the width of the option menu is set to be that of the largest item it contains. You can force the option menu to a different width using **VkOptionMenu::forceWidth()**

void forceWidth(int *width*)

forceWidth() sets all of the items in the option menu to be *width* pixels wide.

Option Menu Example

Example 5-6 shows an example of using a **VkOptionMenu** class.

Example 5-6 Example of Using a **VkOptionMenu** Object

```
////////////////////////////////////////////////////////////////////
// Demonstrate viewkit interface to option menus
///////////////////////////////////////////////////////////////////
#include <Vk/VkApp.h>
#include <Vk/VkSimpleWindow.h>
#include <Vk/VkOptionMenu.h>
#include <stream.h>
```

```
#include <Xm/RowColumn.h>

class MyWindow: public VkSimpleWindow {

  private:

    static void sampleCallback( Widget,  XtPointer , XtPointer);

    static VkMenuDesc MyWindow::optionPaneDesc[];

  protected:

    void sample(Widget, XtPointer);
    VkOptionMenu *_optionMenu;

  public:

    MyWindow( const char *name);
    ~MyWindow( );

    virtual const char* className();
};
VkMenuDesc MyWindow::optionPaneDesc[] = {
  { ACTION,   "Red",   &MyWindow::sampleCallback},
  { ACTION,   "Green", &MyWindow::sampleCallback},
  { ACTION,   "Blue",  &MyWindow::sampleCallback},
  { END},
};

MyWindow::MyWindow( const char *name) : VkSimpleWindow( name)
{
    Widget rc = XmCreateRowColumn(mainWindowWidget(), "rc", NULL, 0);

    _optionMenu = new VkOptionMenu(rc, optionPaneDesc, (XtPointer) this);
    _optionMenu->set("Green");

    addView(rc);
}

MyWindow::~MyWindow( )
{

}

const char* MyWindow::className() { return "MyWindow";}
```

```
void MyWindow::sampleCallback( Widget w, XtPointer clientData, XtPointer callData
)
{
    MyWindow *obj = (MyWindow *) clientData;
    obj->sample(w, callData);
}

void MyWindow::sample(Widget, XtPointer)
{
    cout << "Selected item's index = "
    << _optionMenu->getIndex()
    << ",  name = "
    <<  _optionMenu->getItem()->name()
    << "\n"
    << flush;
}
void main(int argc, char **argv)
{
  VkApp     *app    = new VkApp("Option",  &argc,  argv);
  MyWindow  *win  = new MyWindow("OptionMenu");


  win->show();

  app->run();
}
```

Popup Menus

The **VkPopupMenu** class supports popup menus. You can attach a ViewKit popup menu to one or more widgets in your application so that it pops up automatically whenever the user clicks any of those widgets with the right mouse button. You can also pop up the menu programmatically.

Popup Menu Constructors

There are four versions of the **VkPopupMenu** constructor:

```
VkPopupMenu(VkMenuDesc *menuDesc,
            XtPointer defaultClientData = NULL)

VkPopupMenu(const char *name = "popupMenu",
            VkMenuDesc *menuDesc = NULL,
            XtPointer defaultClientData = NULL)

VkPopupMenu(Widget parent,
            VkMenuDesc *menuDesc = NULL,
            XtPointer defaultClientData = NULL)

VkPopupMenu(Widget parent,
            const char *name = "popupMenu",
            VkMenuDesc *menuDesc = NULL,
            XtPointer defaultClientData = NULL)
```

The forms of the constructor that do not take a *name* argument automatically use the name “popupMenu.” You can specify another name, but schemes does not work correctly if you do.

If you provide the optional *menuDesc* argument, the constructor creates a menu from the VkMenuDesc structure you provide.

If you provide the optional *defaultClientData* argument, any menu item that does not provide a client data argument uses this argument instead. This allows menus to be specified statically, while still allowing an instance pointer to be used with callbacks. This is described in “Special Considerations for Xt Callback Client Data When Using Static Menu Descriptions” on page 131.

If you use a form of the **VkPopupMenu** constructor that accepts a *parent* argument, the constructor automatically attaches the menu to the widget. This builds the menu as a child of the widget and installs an event handler to pop up the menu whenever the user clicks the widget with the right mouse button. For more information on attaching a popup menu to a widget, see the description of **VkPopupMenu::attach()** in “Attaching Popup Menus to Widgets” on page 161.

Attaching Popup Menus to Widgets

The **VkPopupMenu::attach()** function attaches a popup menu to a widget:

```
virtual void attach(Widget w)
```

The first call to **attach()** creates all widgets in the popup menu, using the given widget as the parent of the menu. **attach()** then adds an event handler to post the menu automatically whenever the user clicks the widget with the right mouse button. Subsequent calls to **attach()** add the ability to post the menu over additional widgets.

Popping Up Popup Menus

Once you have attached a popup menu to one or more widgets in your application, ViewKit automatically posts the menu whenever the user clicks any of those widgets with the right mouse button.

You can also post the menu programmatically even if you have not attached the popup menu to a widget, by first building the menu using **VkPopupMenu::build()**:

```
virtual void build(Widget parent)
```

build() builds the menu as a child of the *parent* widget, but does not install an event handler to post the menu.

Once you have built the menu, you can post it with **VkPopupMenu::show()**:

```
virtual void show(XEvent *buttonPressEvent)
```

show() requires an X ButtonPress event as an argument to position the menu on the screen. This requires you to register your own event handler to handle the ButtonPress events.

build() and **show()** support applications that wish to control the posting of menus directly. Normally, **attach()** provides an easier way to use popup menus.

Popup Menu Example

Example 5-7 shows an example of using a **VkPopupMenu** class.

Example 5-7 Example of Using a **VKPopupMenu** Object

```
//////////////////////////////////////////////////////////////////
// Sample program that demonstrates how to create a popup menu
/////////////////////////////////////////////////////////////////
#include <Vk/VkApp.h>
#include <Vk/VkWindow.h>
#include <Vk/VkPopupMenu.h>
#include <stream.h>
```

```
#include <Xm/Label.h>

class MyWindow: public VkWindow {

  private:

    VkPopupMenu *_popup;

    static void sampleCallback( Widget,  XtPointer , XtPointer);
    void sample();

    static VkMenuDesc subMenu[];
    static VkMenuDesc sampleMenuPane[];

  protected:


  public:

    MyWindow( const char *name);
    ~MyWindow();

    virtual const char* className();
};

MyWindow::MyWindow( const char *name) : VkWindow( name)
{
    Widget label =  XmCreateLabel(mainWindowWidget(), "a menu", NULL, 0);

    _popup = new VkPopupMenu(label, sampleMenuPane, (XtPointer) this);

    addView(label);
}

MyWindow::~MyWindow( )
{

}

const char* MyWindow::className() { return "MyWindow";}

// The menu bar is essentially a set of cascading menu panes, so the
// top level of the menu tree is always defined as a list of submenus
```

```
VkMenuDesc MyWindow::sampleMenuPane[] = {
  { LABEL,    "Test Label"},
  { SEPARATOR },
  { ACTION,    "An Action",       &MyWindow::sampleCallback},
  { ACTION,    "Another Action",  &MyWindow::sampleCallback},
  { SUBMENU,   "A Submenu",       NULL, MyWindow::subMenu},
  { END},
};


VkMenuDesc MyWindow::subMenu[] = {
  { ACTION,  "foo",   &MyWindow::sampleCallback},
  { ACTION,  "bar",   &MyWindow::sampleCallback},
  { ACTION,  "baz",   &MyWindow::sampleCallback},
  { END},
};


void MyWindow::sample()
{
    cout << "sample callback" <<  "\n" << flush;
}
void MyWindow::sampleCallback( Widget,  XtPointer clientData  , XtPointer)
{
    MyWindow *obj = (MyWindow *) clientData;
    obj->sample();
}

void main(int argc, char **argv)
{
  VkApp     *myApp    = new VkApp("Menudemo",  &argc,  argv);
  MyWindow  *menuWin  = new MyWindow("MenuWindow");

  menuWin->show();

  myApp->run();
}
```

The ViewKit Help Menu

The Help menu, implemented by the **VkHelpPane** class, provides a simple user interface to a help system. For more information on integrating a help system with your application, consult Appendix C, “Using a Help System With ViewKit.”

Implementation of the Help Menu

VkHelpPane is a subclass of **VkSubMenu**. **VkHelpPane** automatically provides five standard menu items, as shown in Figure 5-5:

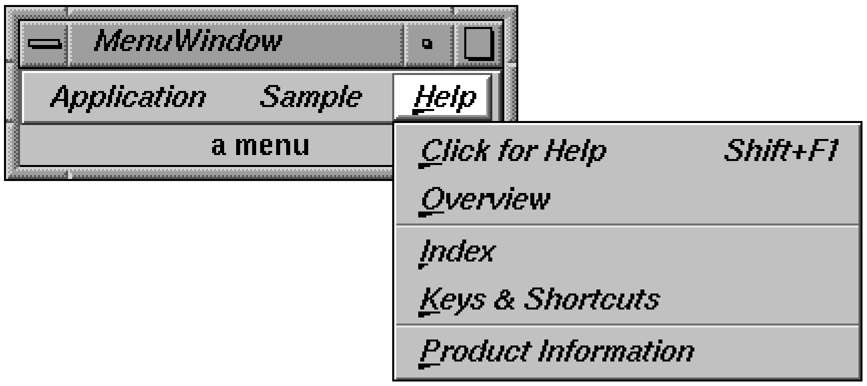


Figure 5-5 The ViewKit Help Menu

The first four items interface to a help system. This help system must provide help request handling and appropriate help messages for the menu item selected:

“Click for Help”	Provides context-sensitive help. When the user chooses this item, the cursor changes into a question mark. The user can then click any widget in the application.
“Overview”	Requests overview help.
“Index”	Requests an index of available help topics.
“Keys & Shortcuts”	Requests help on keys and shortcuts.

“Product Information” displays the Product Information dialog described in “Maintaining Product and Version Information” on page 74. The Product Information dialog has no connection to the help system.

Because **VkHelpPane** is a subclass of **VkSubMenu**, you can also use the functions provided by **VkSubMenu** to add custom Help menu items and delete predefined Help menu items.

Adding the Help Pane to a Menu

The **VkMenuBar** constructor, described in “Menu Bar Constructors” on page 150, accepts a *showHelpPane* argument. If this argument is TRUE, the default, the **VkMenuBar** constructor automatically creates a **VkHelpPane** object and installs it in the menu bar.

You can create a **VkHelpPane** object and add it to another menu, for example a popup menu, but you should rarely need to do this.

X Resources Associated With the Help Pane

The following X resources affect the appearance and behavior of the **VkHelpPane** class:

*helpMenu.labelString	The label for the Help menu (default value “Help”)
*helpMenu.mnemonic	The Help menu mnemonic (default value “H”)
*helpMenu.helpOnContextMenuItem.labelString	The label for the context-sensitive help item (default value “Click for Help”)
*helpMenu.helpOnContextMenuItem.mnemonic	The context-sensitive help item mnemonic (default value “C”)
*helpMenu.helpOnContextMenuItem.accelerator	The context-sensitive help item accelerator (default value “Shift<Key>F1”)
*helpMenu.helpOnContextMenuItem.acceleratorText	The context-sensitive help item accelerator label (default value “Shift+F1”)
*helpMenu.helpOverviewMenuItem.labelString	The label for the help overview item (default value “Overview”)

*helpMenu.helpOverviewMenuItem.mnemonic	The help overview item mnemonic (default value "O")
*helpMenu.helpIndexMenuItem.labelString	The label for the help index item (default value "Index")
*helpMenu.helpIndexMenuItem.mnemonic	The help index item mnemonic (default value "I")
*helpMenu.helpKeysMenuItem.labelString	The label for the keys and shortcuts item (default value "Keys & Shortcuts")
*helpMenu.helpKeysMenuItem.mnemonic	The keys and shortcuts item mnemonic (default value "K")
*helpMenu.helpVersionMenuItem*labelString	The label for the product information item (default value "Product Information")
*helpMenu.helpVersionMenuItem*mnemonic	The product information item mnemonic (default value "P")

Pulldown Menus in the Overlay Planes

By default, pulldown menus appear in the normal planes. ViewKit pulldown menus, however, may be explicitly placed in the overlay planes. Doing so prevents the menus from causing expose events and disturbing such things as complex GL rendering in the normal planes.

There are three ways to enable pulldowns in the overlay planes:

- Call **VkMenu::useOverlayMenus(***TRUE***).** This will force pulldowns into the overlay planes, with no way to put them back in the normal planes without recompiling.
- Put the resource string "*useOverlayMenus: True" in your application's default file. This will put pulldowns in the overlay planes by default, but allow users to use the normal planes by changing their .Xdefaults file.
- Have users add the -*useOverlayMenus* command line switch when they run your application if they wish to use the overlay planes for menus.

If you do decide to place menus in the overlay planes, here are some factors to consider:

- Menus will be placed in the deepest available overlay planes: generally 4 or 8 bit planes, occasionally 2 bit planes.
- If the deepest available overlay is 2 bits, any menus placed in that visual may not look right. Because the colormap in the 2 bit overlay planes only has three color entries (the fourth being a transparent pixel), any items in the menu pulldowns other than labels (for example cascade or toggle buttons) may look odd.
- Other applications using the overlay planes may display in the wrong colors when the menu is activated. The colors in those applications may flash when a pulldown menu appears because the menu's colormap will get installed and replace any previous overlay colormap.
- Tear-off menus may display in the wrong colors. Since tear-off menus are no longer transient, they may be susceptible to color distortions as in previous examples.

Chapter 6

ViewKit Undo Management and Command Classes

Many applications offer users the ability to reverse or "undo" various actions. This chapter describes how ViewKit provides undo support. It also describes how ViewKit supports *command classes*, commands implemented as classes.

Figure 6-1 shows the inheritance graph for ViewKit classes that support undo management and command classes.

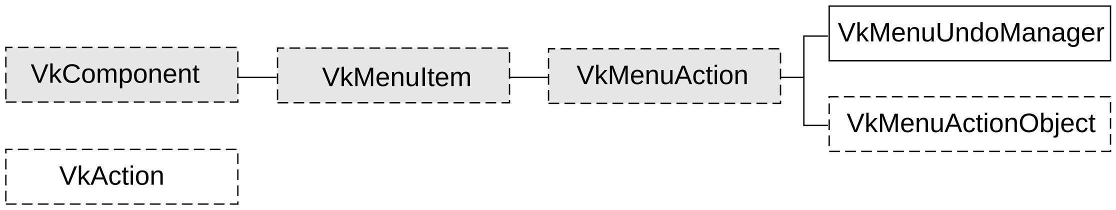


Figure 6-1 The Inheritance Graph for the ViewKit Classes Supporting Undo Management and Command Classes

Undo Management

This section describes the ViewKit undo manager, which supports reversing or "undoing" actions.

Overview of ViewKit Undo Management

The **VkMenuUndoManager** class is the basis of ViewKit's undo manager. The ViewKit undo manager provides an easy-to-use method for users to undo commands that they issue to your application.

The user interface to the ViewKit undo manager is a single menu item that you add to one of your application's menus. By default, the label of that menu item is "Undo: *last_command*", where *last_command* is the name of the last command the user issued. Whenever the user issues a command, the undo manager automatically updates the menu item to reflect the latest command. To undo the command, the user simply chooses the undo manager's menu item.

By default, ViewKit's undo manager provides multi-level undo support. The undo manager keeps commands on a stack. When the user undoes a command, the undo manager pops it from the stack, revealing the previously executed command. Once a user has undone at least one command, executing any new command clears the undo stack. Also, executing any non-undoable command clears the undo stack. If you choose, you can also force the undo manager to provide only single-level undo support, where it remembers only the last command the user issued.

You can use the undo manager to support undoing any command, regardless of whether the user issues the command through a menu or through other interface methods (for example, pushbuttons). The undo manager also supports undoing command classes as implemented by the **VkAction(3Vk)** and **VkMenuActionObject(3Vk)** classes described in "Command Classes" on page 180. In most cases, all you need to provide for each command is a callback function that reverses the effects of that command.

Using ViewKit's Undo Manager

The programmatic interface to the undo manager is simple to use. Because the **VkMenuUndoManager** class is a subclass of **VkMenuItem**, you can add it to a menu and manipulate it as you would any other menu item.

To add undo support for an undoable menu item (**VkMenuAction(3Vk)** and **VkMenuToggle(3Vk)** items), simply provide an undo callback function (a function that reverses the effects of the item's action) when you either statically or dynamically define the menu item. Similarly, to add undo support for a command class (**VkAction** and **VkMenuActionObject** objects), you provide a member function to undo the effects of the command. For those action that are not implemented in your application as menu items or action classes, you can add undo callbacks directly to the undo stack.

Instantiating ViewKit's Undo Manager

Do not directly instantiate a **VkMenuUndoManager** object in your program. If you provide an undo callback to any menu item or if you use a subclass of **VkAction** or **VkMenuActionObject** in your program, ViewKit automatically creates an instance of **VkMenuUndoManager** named "Undo". ("Command Classes" on page 180 describes the **VkAction** and **VkMenuActionObject** classes.) The *<Vk/VkMenuItem.h>* header file provides *theUndoManager*, a global pointer to this instance. To access the ViewKit undo manager, simply use this global pointer.[1]

Adding the Undo Manager to a Menu

You add the undo manager to a menu just as you would any other menu item: using the **VkMenu::add()** function of the menu object to which you want to add the undo manager. For example, the following line adds the undo manager to a menu pane specified by the variable *edit*:

```
edit->add(theUndoManager);
```

You cannot include the undo manager in a static menu description; however, you can add the undo manager to a statically-defined menu after creating the menu. To specify the position of the undo manager within the menu, include a position parameter when you add the undo manager. For example, the following line adds the undo manager to the top of a menu pane specified by the variable *edit*:

```
edit->add(theUndoManager, 0);
```

Providing Undo Support for Actions That Are Menu Items

To add undo support for an undoable menu item (**VkMenuAction** and **VkMenuToggle** items), simply provide an undo callback function when you define the menu item. The undo callback function should reverse the effects of the item's action.

[1] *theUndoManager* is actually implemented as a compiler macro that invokes a **VkUndoManager** access function to return a pointer to the unique instantiation of the **VkUndoManager** class. Although you should never need to use this access function directly, you might encounter it while debugging a ViewKit application that uses the undo manager.

For example, the following static description describes a “Cut” menu item that executes the callback function **cutCallback()** when the user chooses the item and **undoCutCallback()** when the user undoes the command:

```
class EditWindow: public VkWindow {
  private:
    static VkMenuDesc editPane[];
    static void cutCallback(Widget, XtPointer, XtPointer);
    static void undoCutCallback(Widget, XtPointer, XtPointer);
    // ...
};

VkMenuDesc EditWindow::editPane[] = {
  { ACTION,   "Cut",   &EditWindow::cutCallback,
                              NULL,   NULL,   &EditWindow::undoCutCallback },
  { END }
};
```

You could do the same thing by adding the menu item dynamically:

```
class EditWindow: public VkWindow {
  private:
    static VkSubMenu *editMenu;
    static void cutCallback(Widget, XtPointer, XtPointer);
    static void undoCutCallback(Widget, XtPointer, XtPointer);
    // ...
};

EditWindow::EditWindow(char *name) : VkWindow(name)
{
  // ...
  editMenu->addAction("Cut", &EditWindow::cutCallback,
                      &EditWindow::undoCutCallback, this);
}
```

Providing Undo Support for Actions That Are Not Menu Items

Sometimes you might want to provide undo support for an action not implemented as a menu item (for example, an action invoked by a pushbutton). ViewKit allows you to do this by adding the action directly to the undo stack using **VkMenuUndoManager::add()**:

```
void add(const char *name,
          XtCallbackProc undoCallback,
          XtPointer clientData)
```

The *name* argument provides a name for the action to appear in the undo manager's menu item. The *undoCallback* argument must be an Xt-style callback function that the undo manager can call to undo the action. The undo manager passes the *clientData* argument to the undo callback function as client data when it invokes the callback. Following ViewKit conventions as described in "Using Xt Callbacks with Components" on page 21, you should pass the *this* pointer as client data so that the callback function can retrieve the pointer, cast it to the expected component type, and call a corresponding member function.

Note: add() simply adds an action to the undo stack; it does not "register" a permanent undo callback for an action. Once the undo stack is cleared, the undo information for that action is deleted. If you later perform the action again and you want to provide undo support for that action, you must use **add()** again to add the action to the undo stack.

Example 6-1 shows a simple example of adding an action to the undo stack. The **MyComponent** constructor creates a pushbutton as part of its widget hierarchy and registers **actionCallback()** as the button's activation callback function. **actionCallback()**, in addition to performing an action, adds **undoActionCallback()** to the undo stack.

Example 6-1 Adding a Non-Menu Item Directly to the Undo Stack

```
MyComponent: public VkComponent {

  public:
    MyComponent(const char *, Widget);
    void actionCallback(Widget, XtPointer, XtPointer);
    void undoActionCallback(Widget, XtPointer, XtPointer);
    // ...
};

MyComponent::MyComponent(const char *, Widget parent)
{
  // ...
  Widget button = XmCreatePushButton(viewWidget, "button", NULL, 0);
  XtAddCallback(button, XmNactivateCallback,
                &MyWindow::actionCallback, (XtPointer) this);
  // ...
}
```

```
void MyComponent::actionCallback(Widget w, XtPointer clientData,
                                     XtPointer callData)
{
  // Perform action...

  theUndoManager->add("Action", &MyComponent::undoActionCallback, this);
}
```

Providing Undo Support for Command Class Objects

The ViewKit classes that support command classes, **VkAction** and **VkMenuActionObject**, both require you to override the pure virtual function **undoit()**, which the undo manager calls to undo an action implemented as a command class. “Command Classes” on page 180 describes how to use **VkAction** and **VkMenuActionObject** to implement command classes.

Enabling and Disabling Multi-level Undo Support

By default, **VkMenuUndoManager** provides multi-level undo support. The undo manager keeps commands on a stack. When the user undoes a command, the undo manager pops it from the stack, revealing the previously executed command. Once a user has undone at least one command, executing any new command clears the undo stack. Also, executing any undoable command clears the undo stack.

Supporting multi-level undo in your application can be difficult. If you prefer to support undoing only the last command executed, you can change the behavior of the undo manager with the **VkMenuUndoManager::multiLevel()** function:

`void multiLevel(Boolean` *flag*`)`

If *flag* is FALSE, the undo manager remembers only the last command executed.

Clearing the Undo Stack

You can force the undo manager to clear its command stack with the **VkMenuUndoManager::reset()** function:

```
void reset()
```

Examining the Undo Stack

You can examine the contents of the undo manager's command stack using **VkMenuUndoManager::historyList()**:

```
VkComponentList *historyList()
```

historyList() returns a list of objects representing commands that have been executed and are available to be undone. Commands are listed in order of execution; the last command executed is the last item in the list. All of the objects in the list are subclasses of **VkMenuItem**. Commands added directly to the undo stack (as described in "Providing Undo Support for Actions That Are Not Menu Items" on page 172) or commands implemented as **VkAction** objects (as described in "Command Classes" on page 180) appear as **VkMenuActionStub** objects. **VkMenuActionStub** is an empty subclass of **VkMenuAction**.

Setting the Label of the Undo Manager Menu Item

The label that the undo manager menu item displays is of the form *Undo_label*:*Command_label*. *Undo_label* is the value of the labelXmNlabelString resource of the undo manager. By default, this value is "Undo". You can change this string (for example, for a German-language app-defaults file) by providing a different value for the XmNlabelString resource. For example, you could set the resource as follows:

```
*Undo.labelString:    Annul
```

Command_label is the label for the last executed command registered with the undo manager, determined as follows:

- For commands executed by menu items—**VkMenuAction**, **VkMenuToggle**, **or VkMenuActionObject** (described in "Command Classes" on page 180) objects—the label is the item's XmNlabelString resource.
- For **VkAction** objects (described in "Command Classes" on page 180), the undo manager uses the object's "labelString" resource if one is defined, otherwise it uses the **VkAction** object's name as the label.
- For actions that you add directly to the undo stack (described in "Providing Undo Support for Actions That Are Not Menu Items" on page 172), the undo manager uses the action name that you provided when you added the action.

Example of Using ViewKit’s Undo Manager

Example 6-2 shows an example of using the undo manager.

Example 6-2 Example of Using the Undo Manager

```
////////////////////////////////////////////////////////////////
// Simple example to exercise Vk undo facilities
///////////////////////////////////////////////////////////////
#include <Vk/VkApp.h>
#include <Vk/VkWindow.h>
#include <Vk/VkMenu.h>
#include <Vk/VkMenuItem.h>
#include <Vk/VkSubMenu.h>
#include <stream.h>
#include <Xm/Label.h>
#include <Xm/RowColumn.h>
#include <Xm/PushB.h>

class MyWindow: public VkWindow {

  private:

    static void pushCallback( Widget,  XtPointer, XtPointer);
    static void undoPushCallback( Widget,  XtPointer, XtPointer);

    static void oneCallback( Widget,  XtPointer , XtPointer);
    static void twoCallback( Widget,  XtPointer , XtPointer);
    static void threeCallback( Widget,  XtPointer , XtPointer);

    static void undoOneCallback( Widget,  XtPointer , XtPointer);
    static void undoTwoCallback( Widget,  XtPointer , XtPointer);
    static void undoThreeCallback( Widget,  XtPointer , XtPointer);

    static void quitCallback( Widget,  XtPointer , XtPointer);

    void quit();
    void one();
    void two();
    void three();
    void undoOne();

    void undoTwo();
    void undoThree();
    static VkMenuDesc appMenuPane[];
    static VkMenuDesc mainMenuPane[];

  public:

    MyWindow( const char *name);
    ~MyWindow( );
    virtual const char* className();
};
MyWindow::MyWindow( const char *name) : VkWindow( name)
{
    Widget rc =  XmCreateRowColumn(mainWindowWidget(), "rc", NULL, 0);
    Widget label =  XmCreateLabel(rc, "an undo test", NULL, 0);
    Widget pb =  XmCreatePushButton(rc, "push", NULL, 0);

    XtAddCallback(pb, XmNactivateCallback, &MyWindow::pushCallback,
                  (XtPointer) this);
    XtManageChild(label);
    XtManageChild(pb);

    setMenuBar(mainMenuPane);

    VkSubMenu *editMenuPane = addMenuPane("Edit");

    editMenuPane->add(theUndoManager);

    addView(rc);
}

MyWindow::~MyWindow()
{

}

const char* MyWindow::className()
{
 return "MyWindow";
}
```

```
// The menu bar is essentially a set of cascading menu panes, so the
// top level of the menu tree is always defined as a list of submenus

VkMenuDesc  MyWindow::mainMenuPane[] = {
  { SUBMENU, "Application",  NULL, MyWindow::appMenuPane},
  { END}
};

VkMenuDesc MyWindow::appMenuPane[] = {
  { ACTION,   "Command One",   &MyWindow::oneCallback, NULL, NULL,
                                                  &MyWindow::undoOneCallback },
  { ACTION,   "Command Two",   &MyWindow::twoCallback, NULL, NULL,
                                                  &MyWindow::undoTwoCallback },
  { ACTION,   "Command Three", &MyWindow::threeCallback, NULL, NULL,
                                                  &MyWindow::undoThreeCallback },
  { SEPARATOR },
  { CONFIRMFIRSTACTION,  "Quit",    &MyWindow::quitCallback},
  { END},
};

void MyWindow::one()
{
    cout << "Command One executed" <<  "\n" << flush;
}

void MyWindow::two()
{
    cout << "Command Two executed" <<  "\n" << flush;
}

void MyWindow::three()
{
    cout << "Command Three executed" <<  "\n" << flush;
}

void MyWindow::undoOne()
{
    cout << "Undoing Command One" <<  "\n" << flush;
}

void MyWindow::undoTwo()
{
    cout << "UNdoing Command Two" <<  "\n" << flush;
}
```

```
void MyWindow::undoThree()
{
    cout << "Undoing Command Three" <<  "\n" << flush;
}

void MyWindow::oneCallback( Widget,  XtPointer clientData, XtPointer)
{
    MyWindow *obj = (MyWindow *) clientData;
    obj->one();
}
void MyWindow::twoCallback( Widget,  XtPointer clientData, XtPointer)
{
    MyWindow *obj = (MyWindow *) clientData;
    obj->two();
}
void MyWindow::threeCallback( Widget,  XtPointer clientData, XtPointer)
{
    MyWindow *obj = (MyWindow *) clientData;
    obj->three();
}

void MyWindow::undoOneCallback( Widget,  XtPointer clientData, XtPointer)
{
    MyWindow *obj = (MyWindow *) clientData;
    obj->undoOne();
}
void MyWindow::undoTwoCallback( Widget,  XtPointer clientData, XtPointer)
{
    MyWindow *obj = (MyWindow *) clientData;
    obj->undoTwo();
}
void MyWindow::undoThreeCallback( Widget,  XtPointer clientData, XtPointer)
{
    MyWindow *obj = (MyWindow *) clientData;
    obj->undoThree();
}

void MyWindow::quitCallback ( Widget, XtPointer clientData, XtPointer )
{
    MyWindow *obj = (MyWindow*) clientData;
    delete obj;
}
```

```
void MyWindow::pushCallback( Widget,  XtPointer clientData, XtPointer)
{
    cout << "doing a push command\n" << flush;

    theUndoManager->add("Push", &MyWindow::undoPushCallback, (XtPointer) clientDa
ta);
}



void MyWindow::undoPushCallback( Widget,  XtPointer clientData, XtPointer)
{
    cout << "undoing the push command\n" << flush;
}


main(int argc, char **argv)
{
  VkApp     *app    = new VkApp("Menudemo",  &argc,  argv);
  MyWindow  *win  = new MyWindow("MenuWindow");

  win->show();
  app->run();
}
```

Command Classes

This section describes the **VkAction** class, which supports ViewKit *command classes*. Command classes allow you to implement actions as objects.

Overview of Command Classes

Nearly every user action in an interactive application can be thought of as a “command.” Programmers typically implement commands as functions (callback functions, for example) that are invoked as a result of some user action. This section explores an approach in which each command in a system is modelled as an object.

Representing commands as objects has many advantages. Many commands have some state or data associated with the command, while others may involve a set of related functions. In both cases, a class allows the data and functions associated with a single logical operation to be encapsulated in one place. Because command objects are complete and self-contained, you can queue them for later execution, store them in “history” lists, re-execute them, and so on. Representing commands as objects can also facilitate undoing the command. For example, to prepare to undo a command, you might need to save some state data before executing the command. When you model commands as objects, you can store this information in data members.

The **VkMenuAction** class (described in “Menu Actions” on page 124) implements the command class model to a certain extent in that it allows you to specify callback functions both for performing an action and undoing that action. But the **VkMenuAction** class does not provide a true command class in that it does not allow you to encapsulate any data or support functions the action might need within a discrete object. Furthermore, you must use the **VkMenuAction** class within a menu; it does not allow you to implement command classes activated by pushbuttons, text fields, or other input mechanisms.

ViewKit provides two abstract classes to implement command classes in an application: **VkAction** and **VkMenuActionObject**. **VkAction** supports commands that do not appear in menus and **VkMenuActionObject** supports commands that appear in menus. **VkAction** does not inherit from any other classes, whereas **VkMenuActionObject** is a subclass of **VkMenuAction**, which allows you to add instances of it to a menu and manipulate them as you would any other menu item.

You can encapsulate with a subclass of **VkAction** or **VkMenuActionObject** any data or support functions required to perform an action. Additionally, commands implemented as subclasses of **VkAction** and **VkMenuActionObject** automatically register themselves with the ViewKit undo manager whenever you execute them.

Using Command Classes in ViewKit

To use command classes in ViewKit, you must create a separate subclass for each command in your application.

Command Class Constructors

The syntax of the **VkAction** constructor is:

```
VkAction(const char *name)
```

Each class derived from **VkAction** should provide a constructor that takes at least one argument: the object’s name. All derived class constructors should pass the name to the **VkAction** constructor to initialize the basic class data members, and then initialize any subclass-specific data members.

The syntax of the **VkMenuActionObject** constructor is:

```
VkMenuActionObject(const char *name, XtPointer clientData = NULL)
```

Each class derived from **VkMenuActionObject** should provide a constructor that takes two arguments: the object’s name and optional client data. All derived class constructors should pass the name and the client data to the **VkMenuActionObject** constructor to initialize the basic class data members, and then initialize any subclass-specific data members.

The **VkMenuActionObject** constructor stores the client data in the protected data member *_clientData*:

```
void *_clientData
```

VkMenuActionObject objects do not use the *_clientData* data member for callback functions. Instead it is simply an untyped pointer that you can use to pass any information your command object might need. For example, you could pass a pointer to another object, a value, a string, or any other value. You can access and manipulate *_clientData* from member functions of your command subclass.

Overriding Virtual Functions

Both **VkAction** and **VkMenuActionObject** have two protected pure virtual functions that you must override: **doit()** and **undoit()**:

```
virtual void doit()
virtual void undoit()
```

doit() performs the command class’s action; **undoit()** undoes the action.

Using Command Classes as Menu Items

You can use command classes derived only from **VkMenuActionObject** in a ViewKit menu. Because **VkAction** is not derived from **VkMenuItem**, it does not provide the services required of a menu item.

cannot specify **VkMenuActionObject** objects in a static menu description; you must add them dynamically using **VkMenu::add()**, which is described in “Functions for Dynamically Creating Menus” on page 136.

Activating Command Classes

When a user chooses a **VkMenuActionObject** command object from a menu, ViewKit executes the command by calling the object’s **doit()** function. ViewKit also automatically registers the command with the undo manager.

To activate a command object that is a subclass of **VkAction**, call that action’s **execute()** member function:

```
void execute()
```

execute() calls the object’s **doit()** function. **execute()** also registers the command with the undo manager.

Note: Do not call a command object’s **doit()** function directly. If you do, ViewKit cannot register the command with the undo manager.

Setting the Label Used by Command Classes

You can set the label of a **VkMenuActionObject** command object as you would any other **VkMenuItem** item: by setting the object’s XmNlabelString resource or by calling the object’s **setLabel()** function. “Setting Menu Item Labels” on page 122 describes how to set the label for a menu item.

Because **VkAction** objects are command classes and not interface classes, they technically do not have labels; however, the undo manager requires a label that it can display after you have executed a **VkAction** command. Therefore, ViewKit allows you to set the value of a “labelString” resource for **VkAction** objects, qualified by the object’s name. For example, if you have an instance of a **VkAction** named “formatPara”, you can set the label for this object by providing a value for the “formatPara.labelString” resource:

```
*formatPara:    Format Paragraph
```

If you do not provide a value for a **VkAction** object's "labelString" resource, the undo manager uses the object's name as the label.

Note: The **VkAction** "labelString" resource is a "synthetic" resource, not a widget resource. The only way that you can set the value of this resource is through a resource file. You can't use **XtSetValues()** because the object contains no widgets, and you can't use **setDefaultResources()** because **VkAction** is not a subclass of **VkComponent**.

Chapter 7

Using Dialogs in ViewKit

This chapter introduces the basic ViewKit classes needed to create and manipulate the dialogs in a ViewKit application. Figure 7-1 shows the inheritance graph for these classes.

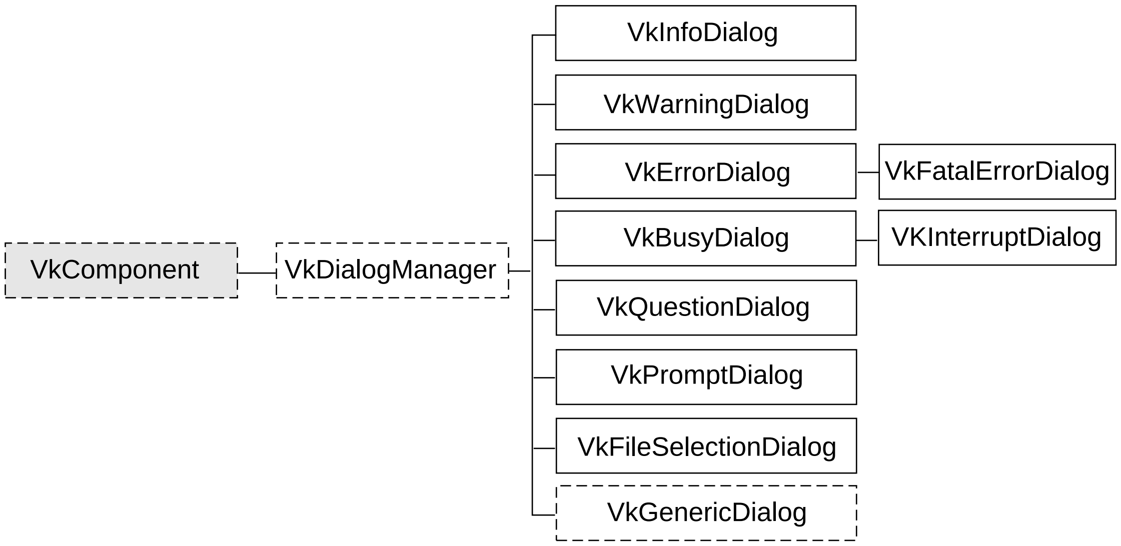


Figure 7-1 The Inheritance Graph for the ViewKit Dialog Classes

Overview of ViewKit Dialog Management

Creating all of the dialogs your application uses when you start the application is inefficient: the dialogs, which might or might not be displayed, take time to create, consume memory, and tie up server resources. If an application does not create a dialog until it is needed, it is smaller and has faster initial startup time; however, depending on the performance of the system, there may be an unacceptable delay in posting each dialog because the application must create a new dialog for each message.

The compromise used by ViewKit is to cache dialogs when they are created. When a particular dialog is no longer needed, the application unmanages that dialog but retains it in the cache. Then, if the cache contains an unused dialog widget when the application needs to post a dialog, the application reuses the cached dialog widget; otherwise it creates a new dialog widget. ViewKit caches up to one dialog of each class for each window in the application (for example, information dialogs and question dialogs are cached separately).

The ViewKit dialog classes also offer the following features:

- single function mechanisms for posting dialogs
- ability to post any dialog in non-blocking, non-modal mode; modal mode; and two blocking modes
- positioning in multi-window applications
- posting of dialogs even when windows are iconified, if desired
- correct handling of dialog references when widgets are destroyed

ViewKit Dialog Class Overview

ViewKit encapsulates dialog management, including caching, in the abstract **VkDialogManager** class that serves as a base class for other, specific dialog classes. Each type of dialog in ViewKit has a separate class derived from **VkDialogManager**. Each class is responsible for managing its own type of dialog (for example, each class maintains its own dialog cache).

The dialog classes provided by ViewKit fall into three categories: information and error dialogs; busy dialogs; and data input dialogs.

The information and error dialogs provide feedback to the user about actions or conditions in the application. The dialog classes in this category are:

VkInfoDialog	Displays information
VkWarningDialog	Warns the user about the consequences of an action (for example, that an action will irretrievably delete items)

VkErrorDialog Informs the user of an invalid action (such as entering out-of-range data) or a potentially dangerous condition (for example, the inability to create a backup file)

VkFatalErrorDialog
Informs the user of a fatal error; the application terminates when the user acknowledges the dialog

The busy dialogs inform the user that an action is underway which might take considerable time. While a busy dialog is displayed, the user cannot interact with the application. The dialog classes in this category are:

VkBusyDialog Busy or working dialog displayed while the application is busy

VkInterruptDialog
Busy or working dialog that allows the user to interrupt the action

The data input dialogs allow the application to request input from the user. The dialog classes in this category are:

VkQuestionDialog
Allows the user to choose among simple choices by clicking pushbuttons

VkPromptDialog
Prompts the user to enter a text string

VkFileSelectionDialog
Allows the user to interactively browse and select a file or directory

VkPrefDialog Supports preference dialogs capable of displaying a wide variety of program-configurable controls that allow the user to observe and set values used by the program. Chapter 8, "Preference Dialogs," discusses preference dialogs.

Additionally, ViewKit provides the **VkGenericDialog** class, an abstract class providing a convenient interface for creating custom dialogs that use the ViewKit interface.

Do not directly instantiate dialog manager objects in your program for the predefined dialog types. ViewKit automatically creates an instance of an appropriate dialog manager if you attempt to use a predefined dialog type in your program.

The header file for each dialog class provides a global pointer to the instance of that class's dialog manager. The name of the pointer consists of "the" followed by the dialog type. For example, the global pointer to the information dialog manager declared in *<Vk/VkInfoDialog.h>* is *theInfoDialog*, the global pointer to the error dialog manager declared in *<Vk/VkErrorDialog.h>* is *theErrorDialog*, and so forth. To access the dialog managers in your application, simply use these global pointers.[1]

Note: VkGenericDialog, being an abstract class designed for creating customized dialogs, does not automatically create a dialog manager or provide a global pointer.

The ViewKit Dialog Base Class

This section describes the dialog management features provided by the abstract **VkDialogManager** base class. It describes how to post dialogs, unpost dialogs, set dialog titles, and set dialog button labels. Because all ViewKit dialog management classes are derived from **VkDialogManager**, the functions and techniques described in this section apply to all dialog management classes.

Posting Dialogs

This section describes the various methods of posting dialogs and provides some simple examples.

Methods of Posting Dialogs

ViewKit offers four different functions for posting dialogs:

post()	Posts a non-blocking, non-modal dialog. The function immediately returns, and the application continues to process user input in all windows.
postModal()	Posts a non-blocking, full-application-modal dialog. The function immediately returns, but the user cannot interact with any application windows until after dismissing the dialog.

[1] These global pointers are actually implemented as compiler macros that invokes access functions to return pointers to the unique instantiation of the dialog managers. Although you should never need to use these access functions directly, you might encounter them while debugging a ViewKit application that uses dialogs.

postBlocked() Posts a blocking, full-application-modal dialog. The user cannot interact with any application windows until after dismissing the dialog. Furthermore, the function does not return until the user dismisses the dialog.

postAndWait() Posts a blocking, full-application-modal dialog. The user cannot interact with any application windows until after dismissing the dialog. Furthermore, the function does not return until the user dismisses the dialog. **postAndWait()** is simpler to use than **postBlocked()**, but it does not allow as much programming flexibility.

post(), **postModal()**, and **postBlocked()** accept the same arguments. They are also overloaded identically to allow for almost any combination of arguments without resorting to using NULLs as placeholders. Consult the **VkDialogManager(3Vk)** reference page for a complete listing of the overloaded versions of the **post()**, **postModal()**, and **postBlocked()** functions. The most general form of the **post()** function is:

```
virtual Widget post ( const char      *msg = NULL,
                      XtCallbackProc   okCB = NULL,
                      XtCallbackProc   cancelCB = NULL,
                      XtCallbackProc   applyCB = NULL,
                      XtPointer        clientData = NULL,
                      const char      *helpString = NULL,
                      Widget          *parent = NULL)
```

The arguments for these methods are:

msg The message to display in the dialog. This string is first treated as a resource name which is looked up relative to the dialog widget. If it exists, the resource value is used as the message. If the resource does not exist, or if the string contains spaces or newline characters, the string itself is used as the message.

Most dialogs are not useful if you do not provide a message argument: they display no text. **VkFileDialog** and **VkPreferenceDialog** are exceptions in that they provide their own complex interfaces.

okCB An Xt-style callback function executed when the user clicks the *OK* button. (All dialogs except for the **VkBusyDialog** and **VkInterruptDialog** dialogs display an *OK* button by default.)

cancelCB	An Xt-style callback function executed when the user clicks the *Cancel* button. For many of the dialog classes, ViewKit does not display a *Cancel* button unless you provide this callback.
applyCB	An Xt-style callback function executed when the user clicks the *Apply* button. For many of the dialog classes, ViewKit does not display an *Apply* button unless you provide this callback.
clientData	Client data to pass to the button callback functions. Following ViewKit conventions as described in "Using Xt Callbacks with Components" on page 21, you should normally pass the *this* pointer as client data so that the callback functions can retrieve the pointer, cast it to the expected component type, and call a corresponding member function.
helpString	A help string to pass to the help system. See Appendix C, "Using a Help System With ViewKit," for information on the help system. If you provide a string, the dialog displays a *Help* button.
parent	The widget over which ViewKit should display the dialog. If you do not provide a widget, or if the given widget is hidden or iconified, ViewKit posts the dialog over the main window if it is managed and not iconified. ("Managing Top-Level Windows" on page 60 describes how the main window is determined.) If both the widget you specify and the main window are hidden or iconified, ViewKit posts the dialog as a child of the hidden application shell created by the **VkApp** class. Also see the description of **VkDialogManager::centerOnScreen()** in "Dialog Access and Utility Functions" on page 198.

All versions of the **post()**, **postModal()**, and **postBlocked()** functions return the widget ID of the posted dialog. You should rarely need to use this value.

Note: The arguments that you provide apply only to the dialog posted by the current call to **post()**, **postModal()**, and **postBlocked()**; they have no effect on subsequent dialogs. For example, if you provide an apply callback function to a call to **post()**, it is used only for the dialog posted by that call. If you want to use that callback for subsequent dialogs, you must provide it as an argument every time you post a
dialog.

postAndWait() provides a simpler method for posting blocking, application-modal dialogs than using **postBlocked()**. The most general form of the **postAndWait()** function is:

```
virtual VkDialogReason postAndWait ( const char      *msg = NULL,
                                     Boolean          ok = TRUE,
                                     Boolean          cancel = TRUE,
                                     Boolean          apply = FALSE,
                                     const char      *helpString = NULL,
                                     Widget          *parent = NULL)
```

msg is the message to display in the dialog. As with the other posting functions, **postAndWait()** first treats the string as a resource name which it looks up relative to the dialog widget. If the resource exists, **postAndWait()** uses the resource value as the message. If **postAndWait()** finds no resource, or if the string contains spaces or newline characters, it uses the string itself as the message. The next three arguments determine which buttons the dialog should display. A TRUE value displays the button and a FALSE value hides the button. *helpString* and *parent* specify a help string and a parent window, just as with the other posting functions.

Note: The arguments that you provide apply only to the dialog posted by the current call to **postAndWait()**; they have no effect on subsequent dialogs.

When you call **postAndWait()**, ViewKit posts the dialog, enters a secondary event loop, and does not return until the user dismisses the dialog. Unlike **postBlocked()**, **postAndWait()** handles all callbacks internally and simply returns an enumerated value of type VkDialogReason indicating which button the user chose. The possible return values are VkDialogManager::OK, VkDialogManager::CANCEL, or VkDialogManager::APPLY. **postAndWait()** is useful for cases in which it is necessary or convenient not to go on to the next line of code until the user dismisses the dialog. For example:

```
if ( theFileSectionDialog->postAndWait() == VkDialogManager::OK )
    int fd = open( theFileSelectionDialog->fileName(), O_RDONLY);
```

Note: **postAndWait()** posts dialogs as full-application modal dialogs to minimize potential problems that can be caused by the secondary event loop, but you should be aware that the second event loop is used and be sure that no non-re-entrant code can be called.

As with the other functions for posting a dialog, **postAndWait()** is overloaded to allow for almost any combination of arguments without resorting to using NULLs as placeholders. Consult the **VkDialogManager** reference page for a complete listing of the overloaded versions of **postAndWait()**.

Note: Under certain circumstances, using **postAndWait()** can cause some unexpected consequences. If you have your own custom dialog, and you delete a widget within it from an event handler such as **prePost()**, the widget will not be destroyed until the event handler returns. Therefore, widgets that you destroyed will still appear in the dialog. This is because the phase 2 destroy does not happen until the return from the XtDispatch. There are several workarounds you can try if this proves to be a problem:

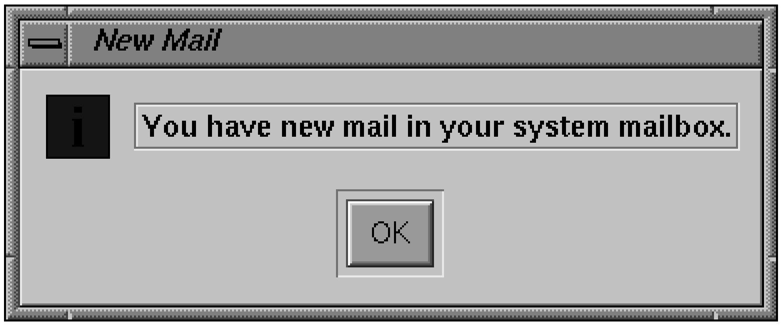
- Do not use **postAndWait()**. Simply post the dialog, return from your event handler, then do whatever you need to do. This may result in flashing, since widgets may be momentarily posted before they are destroyed.
- Unmanage any widget that should not appear. The object will still be there, but will not be visible.
- Keep the dialog cleaned up as you go along. Set up the dialog initially with only permanent items. Then, whenever the dialog is posted, add whatever objects you need. Finally, whenever that dialog is taken down, return it to the original state. You can handle this by catching both OK and Cancel callbacks (to get a Cancel callback, there must be a cancel button).

Examples of Posting Dialogs

The following line posts a simple non-modal, non-blocking information dialog over the application's main window:

```
theInfoDialog->post("You have new mail in your system mailbox");
```

Figure 7-2 shows the appearance of this dialog when posted. Because the call did not provide any callback for the *OK* button, when the user clicks the button, ViewKit simply dismisses the dialog.



You have new mail in your system mailbox.

Figure 7-2 Example Information Dialog

You could also specify the message as an X resource. In the above example, you could name the resource something such as newMailMessage and set it in a resource file with the line:

```
*newMailMessage:  You have new mail in your system mailbox
```

Then you could use the following line to post the information dialog:

```
theInfoDialog->post("newMailMessage");
```

The following line displays a non-modal, non-blocking question dialog over the application's main window:

```
void MailWindow::newMail()
{
  // ...
    theQuestionDialog->post("Read new mail?",
                            &MailWindow::readMailCallback,
                            (XtPointer) this);
  // ...
}
```

Figure 7-3 shows the appearance of this dialog when posted. If the user clicks the *OK* button, the program dismisses the dialog and executes the **MailWindow::readMailCallback()** function. Following ViewKit conventions as described in "Using Xt Callbacks with Components" on page 21, the client data argument is set to the value of the *this* pointer so that **MailWindow::readMailCallback()** can retrieve the pointer, cast it to the expected component type, and call a corresponding member function.

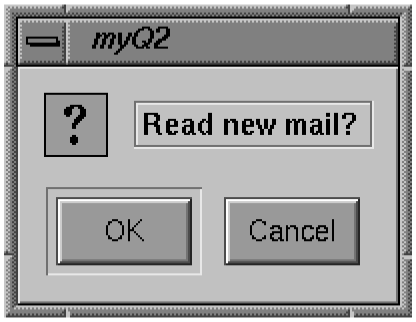


Figure 7-3 Example Question Dialog

Because the call to **post()** did not provide any callback for the *Cancel* button, when the user clicks the button, ViewKit simply dismisses the dialog. If instead you needed to perform some type of cleanup operation when the user clicks the *Cancel* button, you would need to provide a callback for the *Cancel* button:

```
void MailWindow::newMail()
{
  // ...
    theQuestionDialog->post("Read new mail?",
                            &MailWindow::readMailCallback,
                            &MailWindow::cleanupMailCallback,
                            (XtPointer) this);
  // ...
}
```

In general, you should try to encapsulate all dialog callbacks and related information in the subclass of the object with which they are associated. For example, for dialogs that are associated with a specific window, you include all the code related to those dialogs in the subclass definition for that window.

This technique is illustrated in Example 7-1, which shows a simple example of using the **VkWarningDialog** class to post a warning dialog.

Example 7-1 Example of Posting a Dialog

```
#include <Vk/VkApp.h>
#include <Vk/VkSimpleWindow.h>
#include <Vk/VkWarningDialog.h>
#include <Xm/PushB.h>

class MyWindow: public VkSimpleWindow {

  protected:
    static void postCallback(Widget, XtPointer, XtPointer);

  public:
    MyWindow (const char *name);
    ~MyWindow ( );
    virtual const char* className();
};

MyWindow::MyWindow (const char *name) : VkSimpleWindow (name)
{
    Widget button =  XmCreatePushButton (mainWindowWidget(), "Push Me",
                                         NULL, 0);
    XtAddCallback(button, XmNactivateCallback,
                  &MyWindow::postCallback,
                  (XtPointer) this);
    addView(button);
}

const char* MyWindow::className() { return "MyWindow"; }

MyWindow::~MyWindow()
{
    // Empty
}

void MyWindow::postCallback(Widget, XtPointer clientData, XtPointer)
{
    theWarningDialog->post("Watch Out!!!", NULL,
                            (MyWindow *) clientData);
}

void main ( int argc, char **argv )
{
    VkApp     *app  = new VkApp("Dialog", &argc, argv);
    MyWindow  *win  = new MyWindow("Dialog");

    win->show();
    app->run();
}
```

Unposting Dialogs

After posting a dialog, you might encounter situations in which you want to unpost it even though the user has not acknowledged and dismissed it. For example, your application might post an information dialog that the user doesn't bother to acknowledge. At some later point, the information presented in the dialog might no longer be valid, in which case the application should unpost the dialog. In situations such as these, you can use the **VkDialogManager::unpost()** function to remove the dialog:

```
void unpost()
void unpost(Widget w)
```

If you provide the widget ID of a specific dialog, **unpost()** dismisses that dialog. Otherwise, **unpost()** dismisses the most recent dialog of that class posted.

If you want to dismiss all dialogs of a given class, you can call the **VkDialogManager::unpostAll()** function:

```
void unpostAll()
```

For example, the following dismisses all information dialogs currently posted:

```
theInformationDialog->unpostAll();
```

Setting the Title of the Dialog

By default, ViewKit sets the title of a dialog (displayed in the window manager title bar for the dialog) to the name of the application; however, you have the ability to set dialog titles on both a per-class and per-dialog basis.

If you want all dialogs of a certain class to have a title other than the default, you can specify the title with an X resource. For example, you could set the title of all warning dialogs in an application to “Warning” by including the following line in a resource file:

```
*warningDialog.dialogTitle: Warning
```

You can use the **VkDialogManager::setTitle()** function to set the title for the next dialog of that class that you post:

```
void setTitle(const char *nextTitle = NULL)
```

setTitle() accepts as an argument a character string. **setTitle()** first treats the string as a resource name which it looks up relative to the dialog widget. If the resource exists, **setTitle()** uses the resource value as the dialog title. If **setTitle()** finds no resource, or if the string contains spaces or newline characters, it uses the string itself as the dialog title.

setTitle() affects only the next dialog posted; subsequent dialogs revert to the default title for that class.

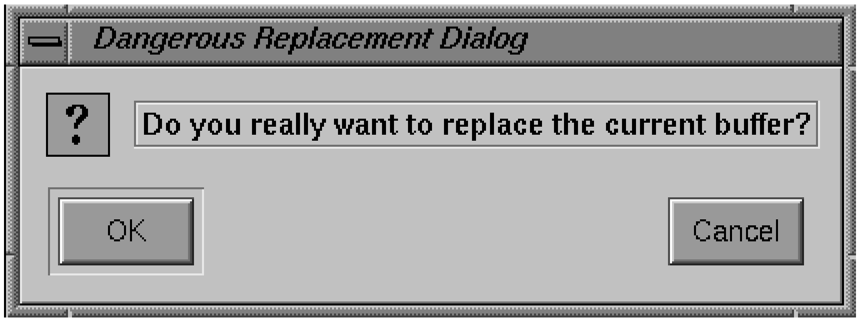
For example, imagine an editor that uses the question dialog to post two dialogs, one that asks “Do you really want to replace the current buffer?” and one that asks “Do you really want to exit?” If you want different titles for each dialog, you could define resources for each:

```
*replaceTitle: Dangerous Replacement Dialog
*exitTitle: Last Chance Before Exit Dialog
```

Then to post the question dialog for replacing the buffer, call:

```
theQuestionDialog->setTitle("replaceTitle");
theQuestionDialog->post("Do you really want to replace the current buffer?",
                        &EditWindow::replaceBufferCallback,
                        XtPointer) this);
```

Figure 7-4 shows the resulting dialog.



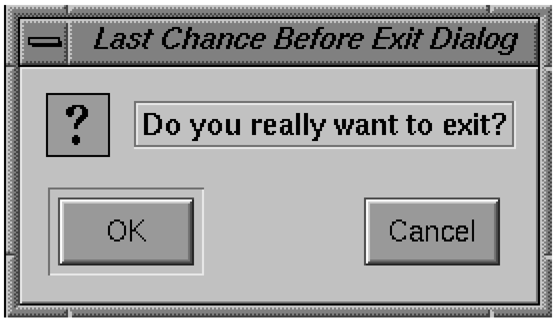
Do you really want to replace the current buffer?

Figure 7-4 Example of Setting the Dialog Title

To post the exit question dialog as a modal dialog, call:

```
theQuestionDialog->setTitle("exitTitle");
theQuestionDialog->postModal("Do you really want to exit?",
                             &EditWindow::replaceBufferCallback,
                             (XtPointer) this);
```

Figure 7-5 shows the resulting dialog.



Do you really want to exit?

Figure 7-5 Another Example of Setting the Dialog Title

Setting the Button Labels

The button labels (the text that appears on the buttons) used for a dialog are controlled by the XmNokLabelString, XmNcancelLabelString, and XmNapplyLabelString resources. The default values of these resources are respectively “OK”, “Cancel”, and “Apply”.

You can use the **VkDialogManager::setButtonLabels()** function to set the button labels for the next dialog that you post:

```
void setButtonLabels(const char *ok = NULL,
                     const char *cancel = NULL,
                     const char *apply = NULL)
```

setButtonLabels() accepts as arguments up to three character strings: the first string controls the label for the *OK* button, the second the label for the *Cancel* button, and the third the label for the *Apply* button. If you pass NULL as an argument for any of these strings, the corresponding button uses the default label. **setTitle()** first treats each string as a resource name which it looks up relative to the dialog widget. If the resource exists, **setTitle()** uses the resource value as the button label. If **setTitle()** finds no resource, or if the string contains spaces or newline characters, it uses the string itself as the button label.

setButtonLabels() affects only the next dialog posted; subsequent dialogs revert to the default button labels.

Dialog Access and Utility Functions

The **VkDialogManager** class also provides some access and utility functions to help manipulate dialogs.

The **VkDialogManager::centerOnScreen()** function controls the algorithm that ViewKit uses to determine where on the screen to post a dialog:

```
void centerOnScreen(Boolean flag)
```

If *flag* is TRUE, ViewKit uses the following algorithm:

1. If you provide a parent window argument when you call one of the posting functions, and that window is visible and not iconified, ViewKit posts the dialog over that window.
2. If a) you provide a parent window argument but the window is hidden or iconified, or b) you do not provide a parent window argument, ViewKit creates the dialog as a child of the hidden application shell created by the **VkApp** class and posts the dialog over that shell. Unless you or the user explicitly sets the geometry for the application, ViewKit centers the application shell on the screen, so the dialog appears centered on the screen.

If *flag* is FALSE, ViewKit uses the following algorithm, which is the default algorithm:

1. If you provide a parent window argument when you call one of the posting functions, and that window is visible and not iconified, ViewKit posts the dialog over that window.
2. If a) you provide a parent window argument but the window is hidden or iconified, or b) you do not provide a parent window argument, ViewKit attempts to create the dialog as a child of the application's main window and post the dialog over that window. ("Managing Top-Level Windows" on page 60 describes how the main window is determined.)
3. If the main window is hidden or iconified, ViewKit creates the dialog as a child of the hidden application shell created by the **VkApp** class and posts the dialog over that shell. Unless you or the user explicitly sets the geometry for the application, ViewKit centers the application shell on the screen, so the dialog appears centered on the screen.

The **VkDialogManager::lastPosted()** returns the widget ID of the last dialog posted of that class:

```
Widget lastPosted()
```

Using the ViewKit Dialog Subclasses

This section describes the features of each ViewKit dialog subclass. In addition to specific member functions listed, each class also supports all functions provided by the **VkDialogManager** class.

Information Dialogs

The **VkInfoDialog** class supports standard IRIS IM information dialogs. The global pointer to the information dialog manager, declared in <*Vk/VkInfoDialog.h*>, is *theInfoDialog*.

Use information dialogs to display useful information. Do not use information dialogs to display error messages, which should be handled by the **VkErrorDialog**, **VkWarningDialog**, or **VkFatalErrorDialog** class.

Because the message contained in an information dialog should not require any decision to be made by the user, information dialogs display only the *OK* button by default. If you need the user to make a selection, you should use another dialog class such as **VkQuestionDialog**.

VkInfoDialog does not provide any additional functions beyond those offered by the **VkDialogManager**.

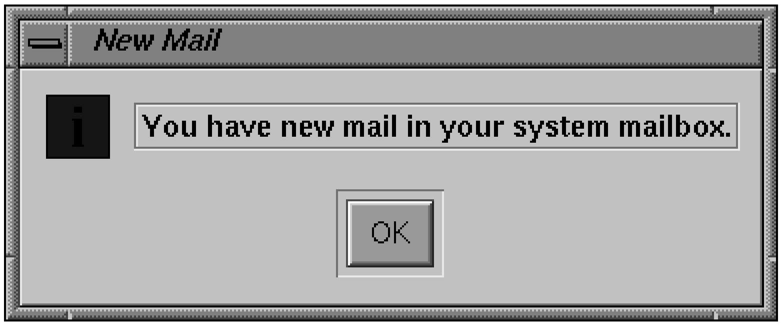
Example 7-2 shows a simple example of posting an information dialog. Note that the window subclass that posts the dialog defines the dialog title and message as resource values.

Example 7-2 Example of Posting an Information Dialog

```
#include <Vk/VkWindow.h>
#include <Vk/VkInfoDialog.h>

class MailWindow: public VkWindow {
  public:
    MailWindow(const char*);
    void newMail();
    // ...

  private:
    static String _defaultResources[];
    // ...
};

String MailWindow::_defaultResources[] = {
    "-*newMailMsg:     You have new mail in your system mailbox.",
    "-*newMailTitle:   New Mail",
    NULL
};
```

```
MailWindow::MailWindow(const char *name) : VkSimpleWindow (name)
{
    setDefaultResources( mainWindowWidget(), _defaultResources );
    // ...
}

void MailWindow::newMail()
{
    // ...
    theInfoDialog->setTitle("newMailTitle");
    theInfoDialog->post("newMailMsg");
    // ...
}
```

Figure 7-6 shows the appearance of the resulting dialog.



You have new mail in your system mailbox.

Figure 7-6 Example of an Information Dialog

Warning Dialogs

The **VkWarningDialog** class supports standard IRIS IM warning dialogs. The global pointer to the warning dialog manager, declared in *<Vk/VkWarningDialog.h>*, is *theWarningDialog*.

Use **VkWarningDialog** to warn the user of the consequences of an action. For example, **VkWarningDialog** is appropriate for warning the user that an action will irretrievably delete information.

By default, the dialogs posted by **VkWarningDialog** contain only an *OK* button; however, according to Open Software Foundation style guidelines, if you have posted a warning dialog to warn the user about an unrecoverable action, you must allow the user to cancel the destructive action. To add a *Cancel* button to your warning dialog, simply provide a cancel callback function when you post the dialog.

Tip: If you perform the action in the warning dialog's OK callback, you can simply define an empty function as a cancel callback. If the user clicks the warning dialog's *OK*, button, the ok callback performs the action; if the user clicks the *Cancel* button, ViewKit dismisses the dialog without performing any action.

VkWarningDialog does not provide any additional functions beyond those offered by the **VkDialogManager**.

Error Dialogs

The **VkErrorDialog** class supports standard IRIS IM error dialogs. The global pointer to the error dialog manager, declared in *<Vk/VkErrorDialog.h>*, is *theErrorDialog*.

Use **VkErrorDialog** to inform the user of an invalid action (such as entering out-of-range data) or potentially dangerous condition (for example, the inability to create a backup file).

The messages contained in the error dialogs should not require any decision to be made by the user. Therefore, the error dialogs display only the *OK* button by default. If you need the user to make a selection, you should use another dialog class such as **VkQuestionDialog**.

VkErrorDialog does not provide any additional functions beyond those offered by the **VkDialogManager**.

Fatal Error Dialogs

The **VkFatalErrorDialog** class supports an error dialog that terminates the application when the user dismisses it. The global pointer to the fatal error dialog manager, declared in *<Vk/VkFatalErrorDialog.h>*, is *theFatalErrorDialog*.

Use **VkFatalErrorDialog** only for those errors from which your program cannot recover. For example, **VkFatalErrorDialog** is appropriate if an application terminates because it cannot open a necessary data file. When the user acknowledges the dialog posted by **VkFatalErrorDialog**, the application terminates by calling **VkApp::terminate()** with an error value of 1. “Quitting ViewKit Applications” on page 58 describes the **terminate()** function.

The messages contained in a fatal error dialog should not require any decision to be made by the user. Therefore, the fatal error dialog displays only the *OK* button by default.

VkFatalErrorDialog does not provide any additional functions beyond those offered by the **VkDialogManager**.

The Busy Dialog

The **VkBusyDialog** class supports a busy dialog (also called a working dialog in IRIS IM) that is displayed when the application is busy. The global pointer to the busy dialog manager, declared in *<Vk/VkBusyDialog.h>*, is *theBusyDialog*.

Unlike most other dialog classes, you should not directly post and unpost the busy dialog. **VkBusyDialog** is used by the **VkApp** object to display a busy dialog when you place the application in a busy state. The busy dialog is displayed automatically when you call **VkApp::busy()**, and dismissed automatically when you make a corresponding call to **VkApp::notBusy()**. **VkApp** also allows you to use the **VkApp::setBusyDialog()** function to use a busy dialog other than that provided by **VkBusyDialog**. Consult “Supporting Busy States” on page 68 for more information about how **VkApp** handles busy states.

Because the busy dialog is intended to lock out user input during a busy state, by default the busy dialog does not display any buttons. If you want to allow the user to interrupt the busy state, you should use the **VkApp::setBusyDialog()** function to substitute the **VkInterruptDialog** class object for the normal busy dialog.

VkBusyDialog does not provide any additional functions beyond those offered by the **VkDialogManager**.

The Interruptible Busy Dialog

The **VkInterruptDialog** class supports an interruptible busy dialog that you can substitute for the normal busy dialog. The dialog posted by the **VkInterruptDialog** class includes a *Cancel* button that the user can click to cancel the current action. The global pointer to the interruptible busy dialog manager, declared in *<Vk/VkInterruptDialog.h>*, is *theInterruptDialog*.

In addition to those functions offered by the **VkDialogManager** class, **VkInterruptDialog** provides the **wasInterrupted()** member function:

```
Boolean wasInterrupted()
```

Applications that use **VkInterruptDialog** must periodically call **wasInterrupted()** to determine whether the user has clicked the dialog's *Cancel* button since the last time the function was called. The period of time between checks is up to the application, which must weigh responsiveness against time spent checking.

Note that **wasInterrupted()** also calls **VkApp::handlePendingEvents()** to process any events that have occurred while the application was busy. Because checking for interrupts involves entering a secondary event loop for a short time, you should beware of any problems with re-entrant code in any callbacks that could be invoked.

Also note that you are responsible for performing any cleanup operations required by your application if the user interrupts a process before it is finished (that is, before you would normally call **VkApp::notBusy()** to end the busy state).

VkInterruptDialog also provides the ViewKit callback *VkInterruptDialog::interruptedCallback*. This callback allows objects to register a member function to be called when the user clicks the *Cancel* button of a **VkInterruptDialog** dialog. This callback can be called only if the application calls **VkInterruptDialog::wasInterrupted()**.

Unlike most other dialog classes, you should not directly post and unpost the interruptible busy dialog. You can use the **VkApp::setBusyDialog()** function to instruct the **VkApp** object to use the interruptible busy dialog rather than the normal busy dialog provided by the **VkBusyDialog** class. The following line shows how you could do this in a program:

```
theApplication->setBusyDialog(theInterruptDialog);
```

The following line instructs the **VkApp** object to revert back to the normal busy dialog:

```
theApplication->setBusyDialog(NULL);
```

If you instruct the **VkApp** object to use the interruptible busy dialog, it is displayed automatically when you call **VkApp::busy()**, and dismissed automatically when you make a corresponding call to **VkApp::notBusy()**. Consult "Supporting Busy States" on page 68 for more information about how **VkApp** handles busy states.

The code fragment in Example 7-3 installs the interruptible busy dialog and performs a simulated lengthy task, checking for interrupts periodically. After completing the task, the code reinstalls the normal busy dialog.

Example 7-3 Example of Using the Interruptible Busy Dialog

```
int i;

// Install the interruptible dialog as the dialog
// to post when busy

theApplication->setBusyDialog(theInterruptDialog);

// Start being "busy"

theApplication->busy("Very Busy", (BusyWindow *) clientData);

for(i=0; i<10000; i++)
{
  // Every so often, see if the task was interrupted

  if( theInterruptDialog->wasInterupted() )
  {
    break; // kick out of current task if user interrupts
  }
  sleep(1);
}

// Task done, so we”re not busy anymore

theApplication->notBusy();

// Restore the application’s busy dialog as the default

theApplication->setBusyDialog(NULL);
```

The Question Dialog

The **VkQuestionDialog** class supports standard IRIS IM question dialogs. These allow the user to select among simple choices by clicking pushbuttons. The global pointer to the question dialog manager, declared in <*Vk/VkQuestionDialog.h*>, is *theQuestionDialog*.

As described in “Posting Dialogs” on page 188, the **post()**, **postModal()**, and **postBlocked()** functions allow you to specify callback functions to be executed when the user clicks the *OK*, *Cancel*, or *Apply* button. These callbacks apply only to the dialog posted by the current function call; they do not affect any subsequent dialog postings. You can also provide client data that is passed to all of the callbacks. Following ViewKit conventions as described in “Using Xt Callbacks with Components” on page 21, you should normally pass the *this* pointer as client data so that the callback functions can retrieve the pointer, cast it to the expected component type, and call a corresponding member function.

For the **postAndWait()** function, instead of providing callbacks, you simply pass a Boolean value for each button specifying whether or not it is displayed. Unlike the other posting functions, the value returned by **postAndWait()** is an enumerated constant of type VkDialogReason (defined in **VkDialogManager**). This value is CANCEL, OK, or APPLY, corresponding to the button the user clicked.

By default, **VkQuestionDialog** displays only the *OK* and *Cancel* buttons. **VkQuestionDialog** displays the *Apply* button only if you provide a callback for that button.

VkQuestionDialog does not provide any additional functions beyond those offered by the **VkDialogManager**.

The Prompt Dialog

The **VkPromptDialog** supports standard IRIS IM prompt dialogs that allow the user to enter a text string. The global pointer to the prompt dialog manager, declared in <*Vk/VkPromptDialog.h*>, is *thePromptDialog*.

You can use **VkPromptDialog** any time you need to prompt the user to enter a single piece of information. If you need the user to enter more than one value, you should consider whether it is more appropriate to create a preference dialog as described in Chapter 8, “Preference Dialogs.” Another option is to create your own custom dialog using **VkGenericDialog** as described in "Deriving New Dialog Classes Using the Generic Dialog" on page 211.

By default, **VkPromptDialog** displays only the *OK* and *Cancel* buttons. **VkPromptDialog** displays the *Apply* button only if you provide a callback for that button.

One method of obtaining the prompt dialog's text string is to extract it and use it in the OK callback function (and the apply callback function if you provide one). Example 7-4 demonstrates this technique.

Example 7-4 Extracting the Text String from a Prompt Dialog

```
void MailWindow::okCallback(Widget w, XtPointer, clientData, XtPointer callData)
{
    MailWindow *obj = (MailWindow *) clientData;
    obj->ok(w, callData);
}

void MailWindow::ok(Widget dialog, XtPointer callData);
{
    char *_text;
    XmSelectionBoxCallbackStruct *cbs =
                                  (XmSelectionBoxCallbackStruct *) callData;

    XmStringGetLtoR(cbs->value,
                    XmFONTLIST_DEFAULT_TAG,
                    &_text );
    // ...
}
```

Another method of obtaining the text string is to call **VkPromptDialog::text()** after the user has dismissed the dialog:

```
const char *text()
```

If the user clicks the *OK* button, the dialog accepts the currently displayed text as input and uses that string as the return value of **text()**. If the user clicks the *Cancel* button, the dialog discards the currently displayed value and any previously-displayed string the dialog might have contain is returned as the value of **text()**. Do not attempt to free the string returned by **text()**. Typically, you should call **text()** only if you post the dialog using **postAndWait()** and **postAndWait()** returns a value of VkDialogManager::OK.

Note: Do not use **text()** from within one of the **VkPromptDialog** callback functions. **VkPromptDialog** sets the value returned by **text()** using its own OK callback function. Because IRIS IM does not guarantee the calling order of callback functions, you cannot be certain that **text()** will return the correct value from within another callback function.

Caution: Be aware that subsequent posting of *thePromptDialog* can alter the text value. In rare conditions, if you post non-modal, non-blocking dialogs, this could occur even before you retrieved the value using **text()**. To prevent this, either retrieve the text string in the OK callback function as shown in Example 7-4, or call **text()** only after posting the dialog using **postAndWait()** and verifying that **postAndWait()** returned the value VkDialogManager::OK).

The File Selection Dialog

The **VkFileSelectionDialog** class supports standard IRIS IM file selection dialogs (an example of which is shown in Figure 7-7). These allow the user to interactively browse and select a file or directory. The global pointer to the file selection dialog manager, declared in *<Vk/VkFileSelectionDialog.h>*, is *theFileSelectionDialog*.

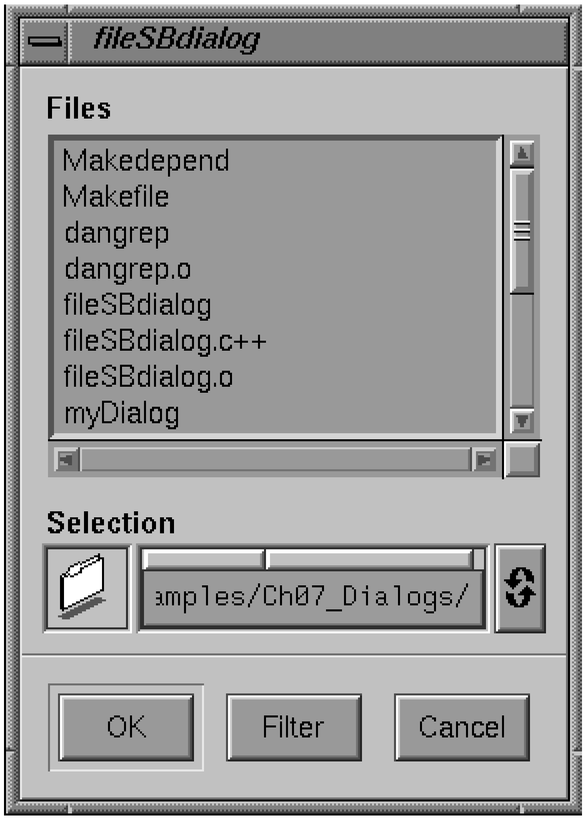


Figure 7-7 Example of a File Selection Dialog

You can set the initial directory displayed by the dialog using **VkFileSelectionDialog::setDirectory()**:

```
void setDirectory(const char *directory)
```

If you do not explicitly set a directory, the dialog defaults to the current directory.

You can set the initial filter pattern used by the dialog, which determines the files displayed in the list box by using **VkFileSelectionDialog::setFilterPattern()**:

```
void setFilterPattern(const char *pattern)
```

If you do not explicitly set a selection, the dialog displays all files in a directory.

You can set the initial selection used of the dialog using **VkFileSelectionDialog::setSelection()**:

```
void setSelection(const char
*selection)
```

One method of obtaining the selection string of the file selection dialog is to extract it and use it in the OK callback function. Example 7-5 demonstrates this technique.

Example 7-5 Extracting the Text String from a File Selection Dialog

```
void MailWindow::okCallback(Widget w, XtPointer, clientData, XtPointer callData)
{
    MailWindow *obj = (MailWindow *) clientData;
    obj->ok(w, callData);
}

void MailWindow::ok(Widget dialog, XtPointer callData);
{
    char *_text;
    XmFileSelectionBoxCallbackStruct *cbs =
                              (XmFileSelectionBoxCallbackStruct *) callData;

    XmStringGetLtoR(cbs->value,
                    XmFONTLIST_DEFAULT_TAG,
                    &_text );
    // ...
}
```

Another method of obtaining the selection string is to call **VkFileSelectionDialog::fileName()** after the user has dismissed the dialog:

```
const char* fileName()
```

If the user clicks the *OK* button, the dialog accepts the currently displayed text as input and uses that string as the return value of **fileName()**. If the user clicks the *Cancel* button, the dialog discards the currently displayed value, and any previously-displayed string the dialog might have contained is returned as the value of **fileName()**. Do not attempt to free the string returned by **fileName()**. Typically, you should call **fileName()** only if you post the dialog using **postAndWait()**, and **postAndWait()** returns a value of VkDialogManager::OK.

Note: Do not use **fileName()** from within one of the **VkFileSelectionDialog** callback functions. **VkFileSelectionDialog** sets the value returned by **fileName()** using its own OK callback function. Because IRIS IM does not guarantee the calling order of callback functions, you cannot be certain that **fileName()** will return the correct value from within another callback function.

Caution: Be aware that subsequent posting of *theFileSelectionDialog* can alter the selection value. In rare conditions, if you post non-modal, non-blocking dialogs, this could occur even before you retrieve the value using **fileName()**. To prevent this, either retrieve the selection string in the OK callback function, or call **fileName()** only after posting the dialog using **postAndWait()**, and verifying that **postAndWait()** returned the value VkDialogManager::OK).

The following code fragment shows a simple example of using the **VkFileSelectionDialog** class:

```
#include <iostream.h>
#include <Vk/VkFileSelectionDialog.h>

// ...

theFileSelectionDialog->setDirectory("/usr/tmp");

if(theFileSelectionDialog->postAndWait( ) == VkDialogManager::OK)
  cout << "File name: " << theFileSelectionDialog->fileName()
       << '\n' << flush;
```

Deriving New Dialog Classes Using the Generic Dialog

The **VkGenericDialog** class is an abstract subclass of **VkDialogManager**. It provides a convenient interface for creating custom dialogs that use the ViewKit interface. Custom dialogs that you derive from this class automatically support caching and all the other features supported by **VkDialogManager**. You can post and manipulate your custom dialogs using the functions provided by **VkDialogManager**.

Minimally, when you derive a new dialog class, you must override the **VkGenericDialog::createDialog()** function to create the dialog used by your class:

```
virtual Widget createDialog(Widget parent)
```

ViewKit passes to **createDialog()** the parent widget for the dialog, and **createDialog()** must return the dialog you create. Your overriding function must first call **VkGenericDialog::createDialog()**, which creates a MessageBox dialog template. By default, the dialog displays *OK* and *Cancel* buttons. Then, you simply add the interface to the MessageBox widget.

You can change the buttons displayed by default and other characteristics for your custom dialog by setting certain protected data members:

Boolean *_showOK*
: Set this value to TRUE (the default) to force the *OK* button to always appear in your custom dialog. If you set *_showOK* to FALSE, the *OK* button appears only if you provide an OK callback function when posting the dialog.

Boolean *_showCancel*
: Set this value to TRUE (the default) to force the *Cancel* button to always appear in your custom dialog. If you set *_showCancel* to FALSE, the *Cancel* button appears only if you provide a cancel callback function when posting the dialog.

Boolean *_showApply*
: Set this value to TRUE to force the *Apply* button to always appear in your custom dialog. If you set *_showApply* to FALSE (the default), the *Apply* button appears only if you provide an apply callback function when posting the dialog.

Boolean *_allowMultipleDialogs*
: The default behavior of the **VkDialogManager** class is to allow multiple dialogs of any given type to be posted at once. The **VkDialogManager** class calls derived classes's **createDialog()** member function as needed to create additional widgets. For some types of dialogs, it makes more sense to allow only one instance of a particular dialog type to exist at any one time. For example, multiple nested calls to **VkApp::busy()** should not normally produce multiple dialogs. If you set *_allowMultipleDialogs* to FALSE, the **VkDialogManager** class does not create additional dialogs, but reuses an existing dialog in all cases.

Boolean *_minimizeMultipleDialogs*

Normally, **VkDialogManager** caches dialogs on a per-top-level window basis. If there are many top-level windows, this could result in having many dialogs of the same type, which may be undesirable for some types of dialogs, particularly if they are expensive to create. If you set *_minimizeMultipleDialogs* TRUE, **VkDialogManager** reuses any existing dialog that is not currently displayed. **VkDialogManager** creates a new dialog only if all existing instances of the dialog type are currently displayed.

Also, by default ViewKit dismisses your dialog whenever the user clicks either the *OK* or *Cancel* button, and keeps the dialog posted whenever the user clicks the *Apply* button. You can change this behavior by overriding the functions **VkDialogManager::ok()**, **VkDialogManager::cancel()**, and **VkDialogManager::apply()** respectively:

```
virtual void ok(Widget dialog, XtPointer callData)
virtual void cancel(Widget dialog, XtPointer callData)
virtual void apply(Widget dialog, XtPointer callData)
```

ViewKit calls these functions whenever the user clicks one of the buttons in the dialog. By default, **ok()** and **cancel()** unpost the dialog and **apply()** is empty. You can override these functions to change the unposting behavior or to perform any other actions you want.

Chapter 8

Preference Dialogs

This chapter introduces the basic ViewKit classes needed to create and manipulate *preference dialogs* in a ViewKit application. Figure 8-1 shows the inheritance graph for these classes.

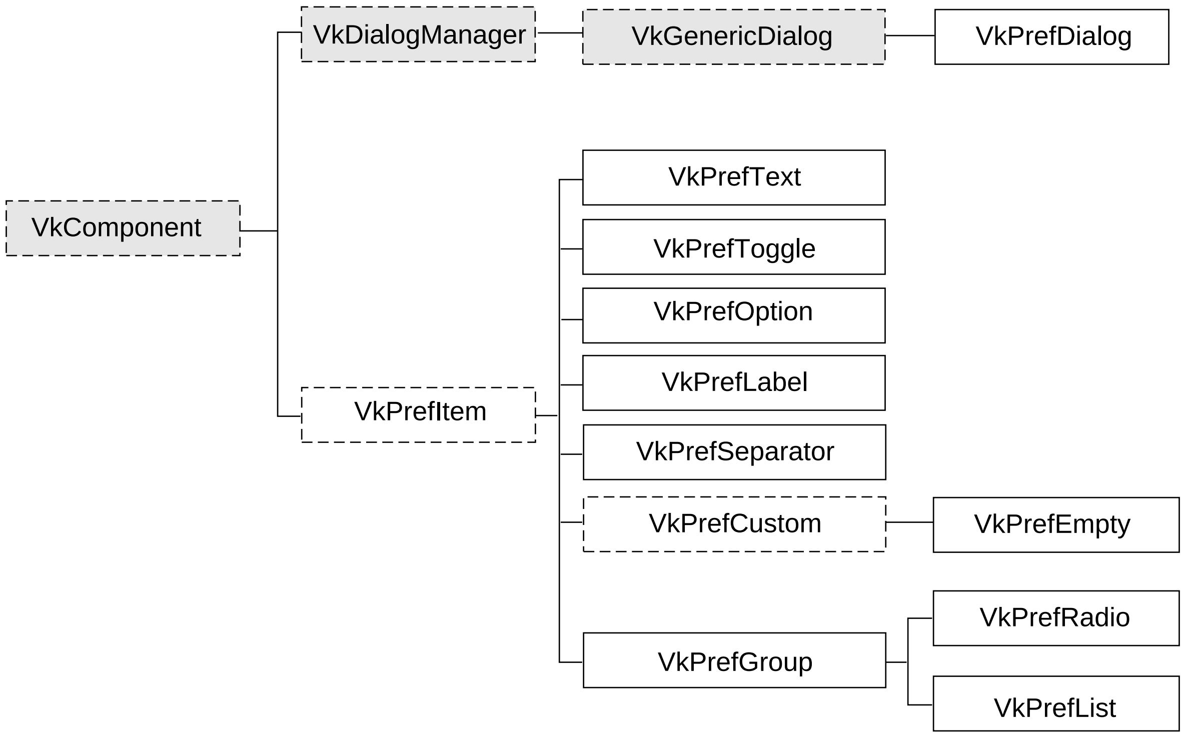


Figure 8-1 The Inheritance Graph for the ViewKit Preference Dialog Classes

Overview of ViewKit Preference Dialogs

Preference dialogs allow users to customize the behavior of an application. Without high-level support, preference dialogs can take considerable time and effort to write because they can involve large numbers of text input fields, labels, toggle buttons, and other controls. A user expects preference dialogs to work in a specific way, as well. Usually, a user sets a number of preferences and then clicks an *Apply* button or an *OK* button to apply all changes at once. A user also expects to be able to click *Cancel* and return all preferences to their previous state, regardless of how many changes the user might have made.

ViewKit supports an easy-to-use collection of classes for building preference dialogs. Rather than dealing directly with widgets, their placement, callbacks, and so on, programmers who use ViewKit can simply create groups of *preference items*. These items maintain their own states, which allows an application to simply query each item to see if it has been changed. Layout is handled automatically, and ViewKit provides the ability to apply or revert all preferences to their previous state.

The ViewKit Preference Dialog Class

In ViewKit, preference dialogs are implemented as a specialized class of dialog. Specifically, the base preference dialog class, **VkPrefDialog**, is a subclass of **VkGenericDialog**, which is in turn a subclass of **VkDialogManager**. Thus, the **VkPrefDialog** class inherits all of the functions and data members provided by these base classes.

However, there are some significant differences in the way you use preference dialogs in your programs compared to the other dialog classes. For the other dialog classes, a single, reusable instance of each type of dialog is sufficient. Details such the message, the button labels, or the dialog title change from posting to posting, but the general dialog behavior remains the same.

On the other hand, individual postings of preference dialogs often vary significantly; they usually have greatly different preference items and data structures associated with each preference item. Therefore, unlike the other dialog classes, **VkPrefDialog** does not create a global instance of a preference dialog. Instead, you must create a separate instance of **VkPrefDialog** for each preference dialog that you want to display in your program. For very simple preference dialogs (for example, just a few toggle buttons), you might be able to directly instantiate a **VkPrefDialog** object; however, in most cases you should create a separate subclass of **VkPrefDialog** for each preference dialog in your application.

For each preference dialog, you create a collection of preference items and associate them with the dialog. Each preference item maintains its own state or value, and your program can query the value of preference items as needed. Users can change the values associated with any number of preference items, then click the *Apply* button to apply all changes and keep the dialog up, or the *OK* button to apply all changes and dismiss the dialog. Users can also click the *Cancel* button to return all preferences to their last applied values and dismiss the dialog.

The **VkPrefDialog** class also supplies a ViewKit callback named *prefCallback*. The preference dialog activates this callback whenever the user clicks the dialog's *Apply*, *OK*, or *Cancel* button.

The ViewKit Preference Item Classes

The basis for all ViewKit preference item classes is the abstract class **VkPrefItem**, which is derived from **VkComponent**. All preference items are derived from the base class **VkPrefItem**, which provides a common set of manipulation functions.

Preference items can be divided into three groups: those that implement various controls such as text fields, toggles, and option menus; those that are "ornamental"; and those that arrange other preference items and manage them as a group.

The following preference items implement controls:

VkPrefText	A text field
VkPrefToggle	A single toggle button (you can group multiple toggle buttons into a **VkPrefRadio** item, described below, to enforce radio-style behavior of the buttons)
VkPrefOption	An option menu

The following preference items are ornamental:

VkPrefLabel	A text label
VkPrefSeparator	A separator
VkPrefEmpty	A "null" item that you can use to add extra space between other items

The following preference items create groups of items:

VkPrefGroup	Defines a group of related items. You can specify either vertical or horizontal layout; the default is vertical. With a vertical layout, **VkPrefGroup** pads items so that they take equal space. You have the option of displaying a label for the group.
VkPrefRadio	A subclass of **VkPrefGroup** for managing a group of toggle items in a radio box style. You can specify either vertical or horizontal layout; the default is vertical. Items are always padded so that they take equal space. You have the option of displaying a label for the group.
VkPrefList	Defines a group of related items. The **VkPrefList** class arranges its items vertically. Unlike **VkPrefGroup**, items are not padded so that they take equal space; instead, each item takes only as much space as it needs. Also in contrast to **VkPrefGroup**, **VkPrefList** does not display any label for the group.

Each preference item maintains its own state or value, and your program can query the value of preference items as needed. Preference items automatically handle updating their stored values when the user clicks the preference dialog's *Apply* or *OK* button, and reverting to their previous values when the user clicks the dialog's *Cancel* button.

Example of Building a ViewKit Preference Dialog

Figure 8-2 shows an example of a preference dialog created using the ViewKit classes.

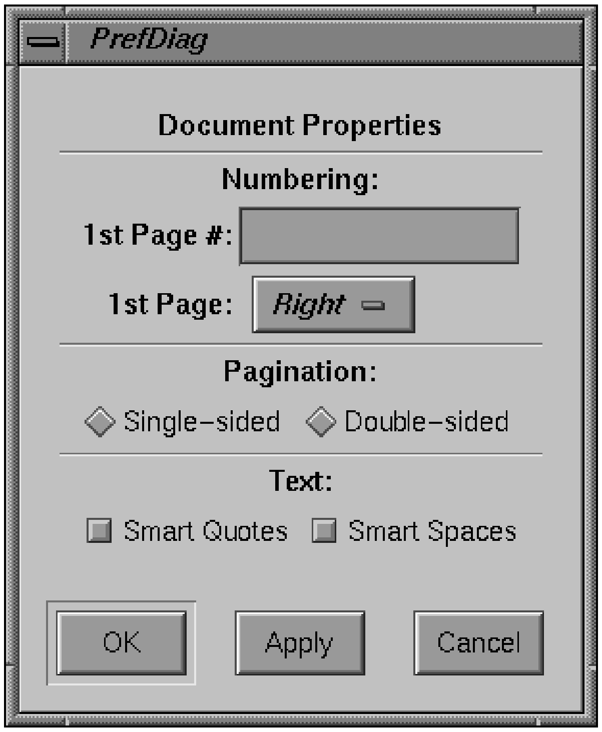


Figure 8-2 Example of a ViewKit Preference Dialog

Example 8-1 lists the code used to create this preference dialog.

Example 8-1 Example of Creating a ViewKit Preference Dialog

```
/////////////////////
//  DocPrefDiag.c++
/////////////////////

#include <Vk/VkApp.h>
#include <Vk/VkPrefDialog.h>
#include <Vk/VkPrefItem.h>
```

```
class DocPrefDialog: public VkPrefDialog {

 protected:

 VkPrefLabel *dialogName;
 VkPrefSeparator *sep1;
 VkPrefText *firstPageNumber;
 VkPrefOption *firstPageSide;
 VkPrefGroup *numberGroup;
 VkPrefSeparator *sep2;
 VkPrefToggle *paginSingleSide;
 VkPrefToggle *paginDoubleSide;
 VkPrefRadio *paginationGroup;
 VkPrefSeparator *sep3;
 VkPrefToggle *textQuotes;
 VkPrefToggle *textSpaces;
 VkPrefGroup *textGroup;
 VkPrefList *docList;

 static String _defaultResources[];
 virtual Widget createDialog(Widget parent);

 public:

 DocPrefDialog ( const char *name );
 ~DocPrefDialog();
 virtual const char* className();
};

String DocPrefDialog::_defaultResources[] = {
 "*dialogNameBase.labelString:Document Properties",
 "*numberGroupLabel.labelString:Numbering:",
 "*firstPageNumberLabel.labelString:1st Page #:",
 "*firstPageSideLabel.labelString:1st Page:",
 "*firstPageRight:Right",
 "*firstPageLeft:Left",
 "*paginationGroupLabel.labelString:Pagination:",
 "*paginSingleSideBase.labelString:Single-sided",
 "*paginDoubleSideBase.labelString:Double-sided",
 "*textGroupLabel.labelString:Text:",
 "*textQuotesBase.labelString:Smart Quotes",
 "*textSpacesBase.labelString:Smart Spaces",
 NULL
};
```

```
DocPrefDialog::DocPrefDialog ( const char *name ) : VkPrefDialog ( name )
{
 // Empty
}

Widget DocPrefDialog::createDialog(Widget parent) {

 setDefaultResources(parent, _defaultResources);

 VkPrefLabel *dialogName = new VkPrefLabel(“dialogName”);

 VkPrefSeparator *sep1 = new VkPrefSeparator(“sep1”);

 VkPrefText *firstPageNumber = new VkPrefText(“firstPageNumber”);

 VkPrefOption *firstPageSide = new VkPrefOption(“firstPageSide”, 2);
 firstPageSide->setLabel(0, “firstPageRight”);
 firstPageSide->setLabel(1, “firstPageLeft”);

 VkPrefGroup *numberGroup = new VkPrefGroup(“numberGroup”);
 numberGroup->addItem(firstPageNumber);
 numberGroup->addItem(firstPageSide);

 VkPrefSeparator *sep2 = new VkPrefSeparator(“sep2”);

 VkPrefToggle *paginSingleSide = new VkPrefToggle(“paginSingleSide”);
 VkPrefToggle *paginDoubleSide = new VkPrefToggle(“paginDoubleSide”);

 VkPrefRadio *paginationGroup = new VkPrefRadio(“paginationGroup”, TRUE);
 paginationGroup->addItem(paginSingleSide);
 paginationGroup->addItem(paginDoubleSide);

 VkPrefSeparator *sep3 = new VkPrefSeparator(“sep3”);

 VkPrefToggle *textQuotes = new VkPrefToggle(“textQuotes”);
 VkPrefToggle *textSpaces = new VkPrefToggle(“textSpaces”);

 VkPrefGroup *textGroup = new VkPrefGroup(“textGroup”, TRUE);
 textGroup->addItem(textQuotes);
 textGroup->addItem(textSpaces);

 VkPrefList *docList = new VkPrefList(“docList”);
 docList->addItem(dialogName);
 docList->addItem(sep1);
 docList->addItem(numberGroup);
```

```
    docList->addItem(sep2);
    docList->addItem(paginationGroup);
    docList->addItem(sep3);
    docList->addItem(textGroup);

    setItem(docList);

    Widget base = VkPrefDialog::createDialog(parent);

    return(base);
}

DocPrefDialog::~DocPrefDialog()
{
    // Empty
}

const char* DocPrefDialog::className()
{
    return "DocPrefDialog";
}


void main ( int argc, char **argv )
{
    VkApp *app = new VkApp("PrefDialogDemoApp", &argc, argv);
    DocPrefDialog *docPrefs = new DocPrefDialog("docPrefs");

    docPrefs->show();
    app->run();
}
```

To post this dialog, you simply create an instance of the **DocPrefDialog** class and use one of the **post()** functions described in "Posting Dialogs" on page 188. For example:

```
DocPrefDialog *docPref = new DocPrefDialog("docPref");
// ...
docPref->post();
```

You can retrieve the value of a preference item with the **getValue()** function as described in "Getting and Setting Preference Item Values" on page 225. For example:

```
Boolean smartSpaces;
// ...
smartSpaces = docPref->textSpaces->getValue();
```

The ViewKit Preference Item Base Class

All preference items are derived from an abstract base class, **VkPrefItem**, which defines the structure of ViewKit preference items and provides a common set of manipulation functions.

Preference Item Labels

Most preference items contain two top-level widgets: a base widget and a label widget. The base widget implements the preference items "control" mechanism (for example, a text field, an option menu, or a toggle button). The label widget (actually implemented as a gadget) displays a text label for the item.

The name of the base widget is the string "Base" appended to the name of the preference item as given in its constructor. The name of the label widget is the string "Label" appended to the name of the preference item as given in its constructor. So, if you create a **VkPrefText** object named "firstName," the name of the base widget is "firstNameBase" and the name of the label widget is "firstNameLabel."

To specify the string that is displayed as the label, you must set the XmNlabelString resource for the label widget. There are various ways to do this:

- Use the **VkComponent::setDefaultResources()** function to provide default resource values. See "Creating Preference Dialog Subclasses" on page 249 for information on using the **setDefaultResources()** function when you create a subclass of **VkPrefDialog**.
- Set resource values in an external app-defaults resource file. Any values you provide in an external file will override values that you set using the **VkComponent::setDefaultResources()** function. This is useful when your application must support multiple languages; you can provide a separate resource file for each language supported.
- Set the resource value directly using the **XtSetValues()** function. Values you set using this method override any values set using either of the above two methods. You should avoid using this method as it "hard codes" the resource values into the code, making them more difficult to change.

The code fragment below sets the labels for two **VkPrefText** items using the first method:

Example 8-2 Setting Default Resource Values for Preference Items

```
#include <Vk/VkPrefDialog.h>
#include <Vk/VkPrefItem.h>

class NameDialog: public VkPrefDialog {
  public:
    VkPrefText *firstName;
    VkPrefText *lastName;
    // ...

  protected:
    Widget createDialog(Widget)

  private:
    static String _defaultResources[];
    // ...
};

String NameDialog::_defaultResources[] = {
 "*firstNameLabel.labelString:  First Name:",
 "*lastNameLabel.labelString:   Last Name:",
};

Widget NameDialog::createDialog(Widget parent)
{
  setDefaultResources(mainWindowWidget(), _defaultResources);

  firstName = new VkPrefText("firstName");
  lastName = new VkPrefText("lastName");
  VkPrefList *nameList = new VkPrefList("nameList");
  // ...
}
```

Not all items display a label. **VkPrefSeparator** is an example of this type of preference item. Some preference items, such as **VkPrefGroup**, allow you to specify in the constructor whether or not you want to display a label for the item. The sections appearing later in this chapter that describe individual preference items discuss how each item uses its label widget.

Getting and Setting Preference Item Values

Preference items that allow the user to input information—**VkPrefText**, **VkPrefToggle**, and **VkPrefOption**—have values associated with them. Each such item stores its own value internally. This value might or might not match the value currently displayed in the preference dialog. Because users can click the *Cancel* button to return all preferences to their last applied values, a preference item must not immediately store a new value that a user enters. Only when the user clicks the dialog's *Apply* button or *OK* button do preference items update their internally stored values to match the values displayed on the screen.

Preference items provide a **getValue()** function that updates the internally-stored value with the currently displayed value and returns the updated value. The **getValue()** function is not actually declared in the **VkPrefItem** base class because different types of preference items use different types of values (for example, **VkPrefToggle** uses a Boolean value whereas **VkPrefText** uses a character string). Each preference item with an associated value provides its own definition of **getValue()**.

The **setValue()** function allows you to programmatically set the internally-stored value of a preference item. The **setValue()** function automatically updates the displayed value to reflect the new internal value. As with the **getValue()** function, **setValue()** is not actually declared in the **VkPrefItem** base class; each preference item with an associated value provides its own definition of **setValue()**.

The **VkPrefItem::changed()** function checks to see whether or not the user has changed the value displayed on the screen so that it no longer matches the item's internally stored value:

```
virtual Boolean changed()
```

If the value has changed, **changed()** returns the Boolean value TRUE; otherwise, it returns FALSE. You should use **changed()** as a test to determine whether or not you need to call **getValue()** for a preference item.

Preference Item Access Functions

The **activate()** and **deactivate()** functions control whether or not a preference item is activated:

```
void activate()
void deactivate()
```

If the item is deactivated, the item is “grayed out” on the screen and the user cannot change the item’s value. Call **activate()** to activate an item and **deactivate()** to deactivate an item.

Occasionally you might want to achieve certain effects by manually setting the height of a preference item’s label or base widget. The **setLabelHeight()** and **setBaseHeight()** functions each accept as an argument an Xt Dimension value and respectively set the item’s label and base widget to the given height:

```
void setLabelHeight(Dimension h)
void setBaseHeight(Dimension h)
```

The **labelHeight()** function returns the current height of the item’s label widget, and the **baseHeight()** function returns the current height of the item’s base widget, each expressed as an Xt Dimension value:

```
Dimension labelHeight()
Dimension baseHeight()
```

The **labelWidget()** function returns the item’s label widget:

```
Widget labelWidget()
```

labelWidget() returns NULL if an item does not have a label widget.

The **type()** function returns an enumerated value of type VkPrefItemType that identifies an item’s type:

```
virtual VkPrefItemType type()
```

Valid return values are: PI_group, PI_list, PI_radio, PI_text, PI_toggle, PI_option, PI_empty, PI_label, PI_separator, PI_custom, and PI_none.

The **isContainer()** function returns TRUE if the preference item is one used to group (or contain) other items:

```
virtual Boolean isContainer()
```

Currently, **isContainer()** returns true for **VkPrefGroup**, **VkPrefRadio**, and **VkPrefList** items.

The ViewKit Preference Item Classes

The following sections describe the preference item classes provided by ViewKit. In addition to specific member functions listed, each class also supports all functions provided by the **VkPrefItem** class.

Text Fields

The **VkPrefText** class supports text field preference items, allowing users to enter text strings. Figure 8-3 shows a simple preference dialog containing a text field preference item.

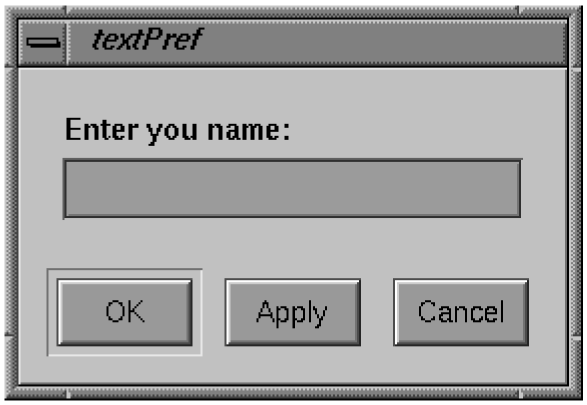


Figure 8-3 Example of a Text Field Preference Item

The **VkPrefText** constructor has the following form:

VkPrefText(const char **name*, int *columns* = 5)

The **VkPrefText** constructor expects as its first argument the name of the preference item. You can optionally provide as a second argument an integer value specifying the default number of columns for the text field.

For example, creating the text field shown in Figure 8-3 requires only the line:

```
VkPrefText *name = new VkPrefText("name");
```

To set the label for the text field you must set the XmNlabelString resource of the preference item's label widget. Therefore, to set the label as shown in Figure 8-3, you must set the resource:

```
*nameLabel.labelString: Enter your name:
```

Refer to "Preference Item Labels" on page 223 for more information on setting the label of a preference item.

Use the **getValue()** function to retrieve the internally-stored value of the text field:

```
char *getValue()
```

getValue() duplicates the internal value and then returns a pointer to the duplicate string. (You should free this string when you no longer need it.) For example, the following line retrieves the value of the *name* text field shown above:

```
userName = name->getValue();
```

Use the **setValue()** function to programmatically set the value of the text field:

```
void setValue(const char *str)
```

setValue() copies the string that you pass as an argument, sets the internally-stored value to that string, and updates the value displayed by the text field. For example, the following line sets the value of the *name* text field shown above to "John Doe:"

```
name->setValue("John Doe");
```

Toggle Buttons

The **VkPrefToggle** class supports a single toggle button preference item. You can group multiple toggle buttons using a **VkPrefGroup** or **VkPrefList** item, and you can enforce radio-style behavior on a group of toggles by grouping them in a **VkPrefRadio** item. These classes are discussed later in this chapter.

Figure 8-4 shows a simple preference dialog containing a single toggle button preference item.

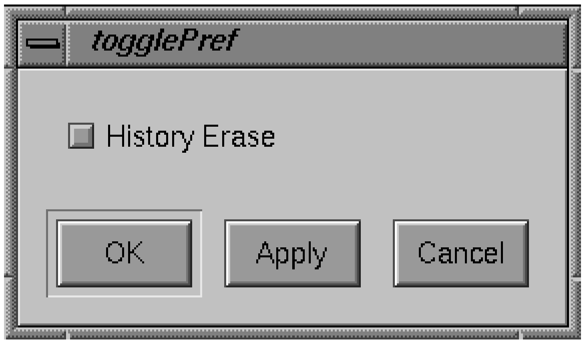


Figure 8-4 Example of a Toggle Button Preference Item

The **VkPrefToggle** constructor has the following form:

`VkPrefToggle(const char *`*name*`, Boolean` *forceLabelFormat* `= FALSE)`

The first argument the **VkPrefToggle** constructor expects is the name of the preference item. For example, creating the toggle button shown in Figure 8-4 requires only the line:

```
VkPrefToggle *erase = new VkPrefToggle("erase");
```

You can provide an optional Boolean value as a second argument to the **VkPrefToggle** constructor. A TRUE value forces the **VkPrefToggle** object to create and use a label widget as described in “Preference Item Labels” on page 223. Otherwise, if the value is FALSE, the behavior of the label is determined as described below in “Setting Toggle Preference Item Labels.” The default value is FALSE.

Setting Toggle Preference Item Labels

Setting the label for a toggle preference item is more complex than with other preference items. Unlike many of the other preference items, the ToggleButton widget that is the base widget of the **VkPrefToggle** item includes a text label. Therefore, to set that label, you must set the XmNlabelString resource of the preference item’s base widget instead of its label widget. For example, to set the label as shown in Figure 8-4, you must set the resource:

```
*eraseBase.labelString: History Erase
```

This works for all cases except for when a toggle is an item in a vertical **VkPrefGroup** or **VkPrefRadio** item that contains items other than toggles. (A group that contains more than one type of preference item is a *non-homogenous group*; a group that contains only one type of preference item is a *homogenous group*.) To understand why this is done, consider first a simple vertical **VkPrefGroup** containing only two toggles, as shown in Figure 8-5. In this case, the labels appear to the right side of the buttons as they normally do.

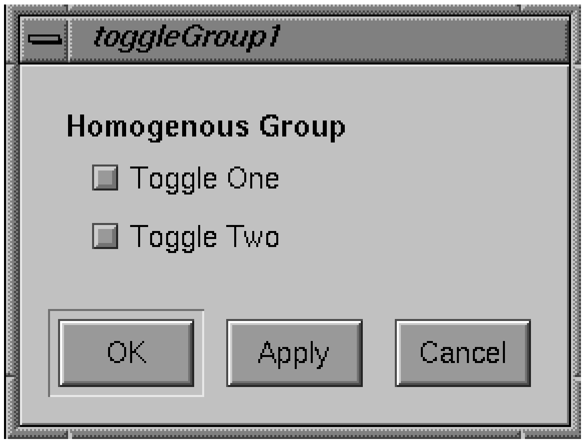


Figure 8-5 Toggle Preference Items in a Homogenous Vertical Group

When toggle items appear in a homogenous group like the one shown in Figure 8-5, you should set the XmNlabelString resources for the base widgets of the toggle items. For example:

```
*firstToggleBase.labelString:   Toggle One
*secondToggleBase.labelString:  Toggle Two
```

However, the labels for most other preference items appear to the left of the items. Left uncorrected, if a vertical, non-homogenous **VkPrefGroup** or **VkPrefRadio** contained a toggle item, the label for the toggle would not align with the other labels.

Therefore, in the case of a non-homogenous vertical **VkPrefGroup** or **VkPrefRadio**, ViewKit sets the XmNlabelString resource of all toggle items' base widgets to NULL and instead displays their label widgets. The result is that all of the preference items' labels correctly align, as shown in Figure 8-6.

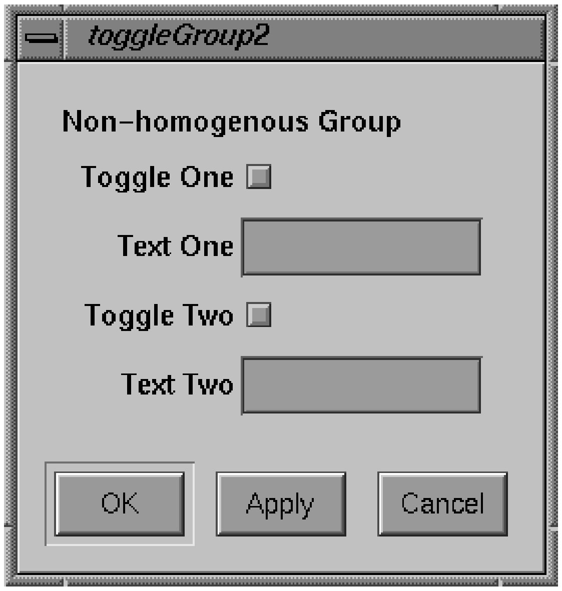


Figure 8-6 Toggle Preference Items in a Non-Homogenous Vertical Group

When toggle items appear in a non-homogenous, vertical group like the one shown in Figure 8-6, you should set the XmNlabelString resources for the label widgets of the toggle items rather than the base widgets. For example:

```
*firstToggleLabel.labelString:   Toggle One
*secondToggleLabel.labelString:  Toggle Two
```

Note that if you provide the Boolean value TRUE as a second argument to the **VkPrefToggle** constructor, the **VkPrefToggle** object always creates and uses a label widget instead of using the base widget's text label.

Refer to "Preference Item Labels" on page 223 for more information on setting the label of a preference item.

Getting and Setting Toggle Preference Item Values

Use the **getValue()** function to retrieve the Boolean value of the toggle:

```
Boolean getValue()
```

For example, the following line retrieves the value of the *firstToggle* toggle shown above:

```
toggleSet = firstToggle->getValue();
```

Use the **setValue()** function to programmatically set the value of the toggle:

```
void setValue(Boolean value)
```

setValue() sets the internally-stored value to the Boolean value you pass as an argument, and updates the value displayed by the toggle. For example, the following line sets the value of the *secondToggle* toggle shown above to TRUE:

```
secondToggle->setValue(TRUE);
```

Option Menus

The **VkPrefOption** class supports option menu preference items, allowing users to choose an option from a menu. Figure 8-7 shows a simple preference dialog containing an option menu preference item.

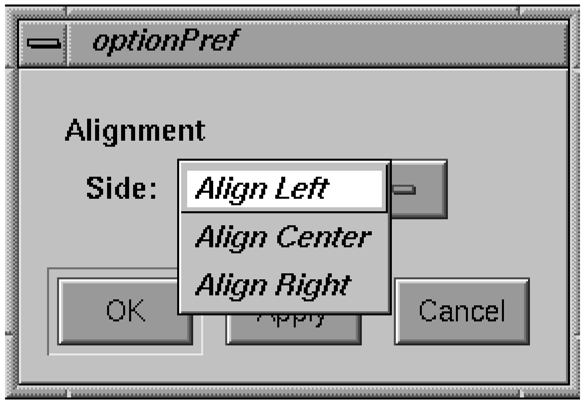


Figure 8-7 Example of an Option Menu Preference Item

The **VkPrefOption** constructor has the following form:

```
VkPrefOption(const char *name, int numEntries)
```

The **VkPrefOption** constructor expects as its first argument the name of the preference item. The second argument is an integer value specifying the number of entries in the option menu.

For example, you can create the option menu shown in Figure 8-7 with the line:

```
VkPrefOption *align = new VkPrefOption("align", 3);
```

Setting Option Menu Preference Item Labels

To set the label for the option menu you must set the XmNlabelString resource of the preference item's label widget. Therefore, to set the label as shown in Figure 8-7, you must set the resource:

```
*alignLabel.labelString: Alignment
```

Refer to "Preference Item Labels" on page 223 for more information on setting the label of a preference item.

To set the labels for the individual items in the option menu, use the **setLabel()** function:

void setLabel(int *index*, const char **label*)

setLabel() expects two arguments. The first is an integer value specifying the index of the of the menu item. Menu items are numbered starting with 0.

The second **setLabel()** argument is a character string. This string is first treated as a resource name which is looked up relative to the menu item's widget. If the resource value exists, it is used as the label. If no resource is found, or if the string contains spaces or newline characters, the string itself is used as the label.

For example, the following lines set the labels for the option menu items shown in Figure 8-7 directly:

```
align->setLabel(0, "Align Left");
align->setLabel(1, "Align Center");
align->setLabel(2, "Align Right");
```

On the other hand, the following lines set the labels using resource values:

```
align->setLabel(0, "alignLeft");
align->setLabel(1, "alignCenter");
align->setLabel(2, "alignRight");
```

In the second case, you would also have to set the appropriate resource values. You could do so using the **setDefaultResources()** function, or you could include the following lines in a resource file:

```
*align*alignLeft:    Align Left
*align*alignCenter:  Align Center
*align*alignRight:   Align Right
```

You can retrieve the label for a given item using the **getLabel()** function:

```
char *getLabel(int index)
```

index is the index of the menu item.

Note: **getLabel()** returns the same string that you passed to **setLabel()** when setting the item's label. Therefore, if you set the item's label by specifying a resource name, **getLabel()** returns the resource name, not the value of the resource.

Dynamically Changing the Number of Option Menu Items

In the **VkPrefOption** constructor, you must provide an argument specifying the number of elements in the option menu. However, after creating an option menu preference item, you can resize it as needed using the **setSize()** function:

```
void setSize(int numEntries)
```

setSize() accepts an integer argument specifying the new size of the option menu. If the new size is smaller than the old size, **setSize()** automatically deletes all unneeded widgets. If the new size is larger, **setSize()** automatically creates and manages any additional widgets needed.

You can determine the current size of an option menu preference item using the **size()** function:

```
int size()
```

You can access any of the button widgets contained in the option menu with the **getButton()** function:

```
Widget getButton(int index)
```

Simply specify the index of the button you want and **getButton()** returns the appropriate widget.

Getting and Setting Option Menu Preference Item Values

Use the **getValue()** function to retrieve the internally-stored value of the option menu:

```
int getValue()
```

getValue() returns an integer value specifying the index of the chosen menu entry. For example, the following line retrieves the value of the *align* text field shown above:

```
alignment = align->getValue();
```

Use the **setValue()** function to programmatically set the value of the option menu:

```
void setValue(int index)
```

setValue() sets the internally-stored value to the index value you pass as an argument, and updates the value displayed by the option menu. For example, the following line sets the value of the *alignment* text field shown above to 1, corresponding to the "Align Center" option:

```
align->setValue(1);
```

Labels

The **VkPrefLabel** class supports text labels for preference dialogs.

Note: **VkPrefLabel** is useful only in conjunction with **VkPrefList**. You should not use **VkPrefLabel** with either **VkPrefGroup** or **VkPrefRadio**; **VkPrefLabel** does not create a label widget and therefore it does not align properly with other items contained in a **VkPrefGroup** or **VkPrefRadio** item.

Figure 8-8 shows a simple preference dialog containing a label preference item.

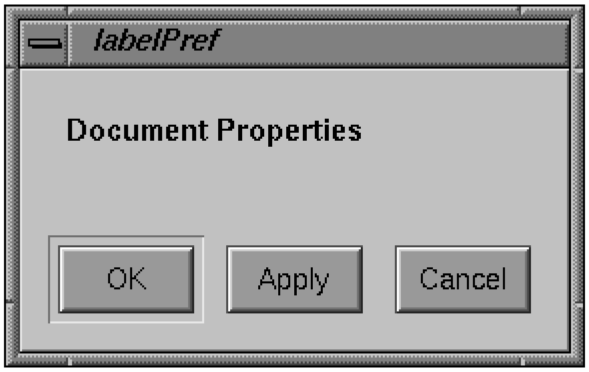


Figure 8-8 Example of a Label Preference Item

The only argument the **VkPrefLabel** constructor expects is the name of the preference item:

```
VkPrefLabel(const char *name)
```

For example, creating the label shown in Figure 8-8 requires only the line:

```
VkPrefLabel *dialogName = new
VkPrefLabel("dialogName");
```

Many other ViewKit preference items include label widgets in addition to their base widget; however, in the case of the **VkPrefLabel** item, the label *is* the base widget. Therefore, in preference item groups, a **VkPrefLabel** item aligns with other base widgets, not with other label widgets.

Because the label that is displayed for a **VkPrefLabel** item is the base widget, you set the label's text by setting the XmNlabelString resource of the item's base widget. Therefore, to set the label as shown in Figure 8-8, you must set the resource:

```
*dialogNameBase.labelString: Document Properties
```

Refer to "Preference Item Labels" on page 223 for more information on setting the label of a preference item.

Separators

The **VkPrefSeparator** class supports a simple separator for use in preference dialogs.

Note: **VkPrefSeparator** is useful only in conjunction with **VkPrefList**. You should not use **VkPrefSeparator** with either **VkPrefGroup** or **VkPrefRadio**; **VkPrefSeparator** does not create a label widget and therefore it does not align properly with other items contained in a **VkPrefGroup** or **VkPrefRadio** item.

The only argument the **VkPrefSeparator** constructor expects is the name of the preference item:

```
VkPrefSeparator(const char *name)
```

For example:

```
VkPrefSeparator *sep = new
VkPrefSeparator("sep");
```

“Empty” Space Preference Items

The **VkPrefEmpty** class provides a “null” item that you can use to add extra space between other items. This preference item is useful only in conjunction with one of the grouping preference items: **VkPrefGroup**, **VkPrefRadio**, or **VkPrefList**.

The **VkPrefEmpty** constructor accepts no arguments:

```
VkPrefEmpty()
```

For example:

```
VkPrefEmpty *space = new VkPrefEmpty();
```

Groups of Preference Items

ViewKit provides three classes for creating groups of items: **VkPrefGroup**, **VkPrefRadio**, and **VkPrefList**. Both **VkPrefRadio** and **VkPrefList** are implemented as subclasses of **VkPrefGroup**.

Comparison of Group Preference Items

VkPrefGroup defines a group of related items. You can specify either vertical or horizontal layout; the default is vertical. With a vertical layout, **VkPrefGroup** pads items so that they take equal space. You have the option of displaying a label for the group.

Figure 8-9 shows an example of a vertical **VkPrefGroup** item with a label. The label is the group item's label widget, not a **VkPrefLabel** item. The **VkPrefGroup** item right-aligns the labels for all of the items it contains. (Because the **VkPrefToggle** items are part of a non-homogenous **VkPrefGroup** item, you must set the XmNlabelString resources of their label widgets instead of their base widgets, as described in "Setting Toggle Preference Item Labels" on page 229.) Also, all items are allocated the same amount of vertical space. If you were to add a larger item to this group, the group item would allocate for each item the same amount of vertical space.

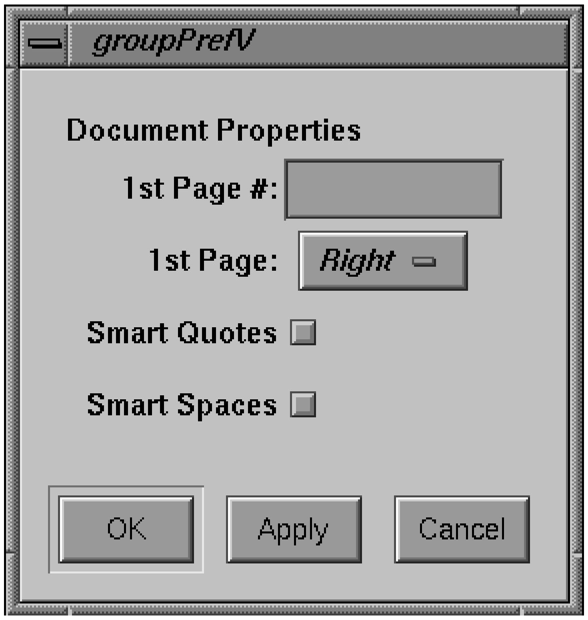


Figure 8-9 Example of a Vertical **VkPrefGroup** Item with Label

Figure 8-10 shows the same preference items grouped by a horizontal **VkPrefGroup** item with a label.

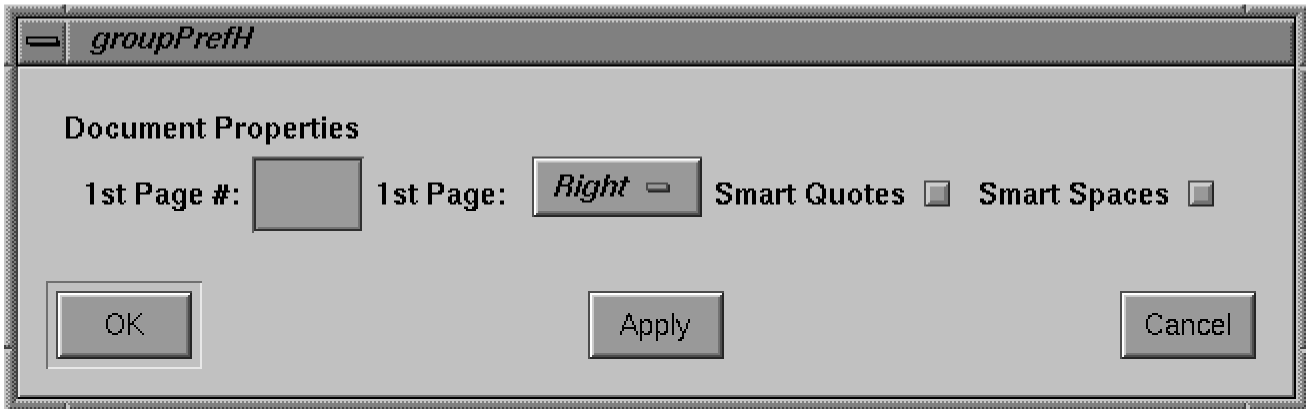


Figure 8-10 Example of a Horizontal **VkPrefGroup** Item with Label

VkPrefList is similar to **VkPrefGroup**; however, it supports only a vertical orientation and it does not support displaying a group label. Unlike **VkPrefGroup**, **VkPrefList** does not pad its items so that they take equal space; instead, each item takes only as much space as it needs. Typically, you use a **VkPrefList** item to group other group items. For example, in Example 8-1, the top-level **VkPrefList** item contained a **VkPrefLabel** item and two **VkPrefGroup** items—one vertical and one horizontal—separated by two **VkPrefSeparator** items.

VkPrefList is also the only grouping item to which you should add **VkPrefLabel** or **VkPrefSeparator** items. You should not use **VkPrefLabel** or **VkPrefSeparator** with either **VkPrefGroup** or **VkPrefRadio**; they do not create label widgets and therefore do not align properly with other items contained in a **VkPrefGroup** or **VkPrefRadio** item.

Figure 8-11 shows an example of a **VkPrefList**. Note that the **VkPrefList** item does not contain a group label; if you want to provide a label for a **VkPrefList** item, you can include a **VkPrefLabel** item in it. Also note that the **VkPrefList** item does not align the labels of the items it contains. (Because the **VkPrefToggle** items are part of a **VkPrefList** item, you must set the XmNlabelString resources of their base widgets instead of their label widgets, as described in "Setting Toggle Preference Item Labels" on page 229.) Each item is allocated only as much vertical space as it needs. If you were to add a larger item to this group, it would not affect the vertical spacing of the other items.

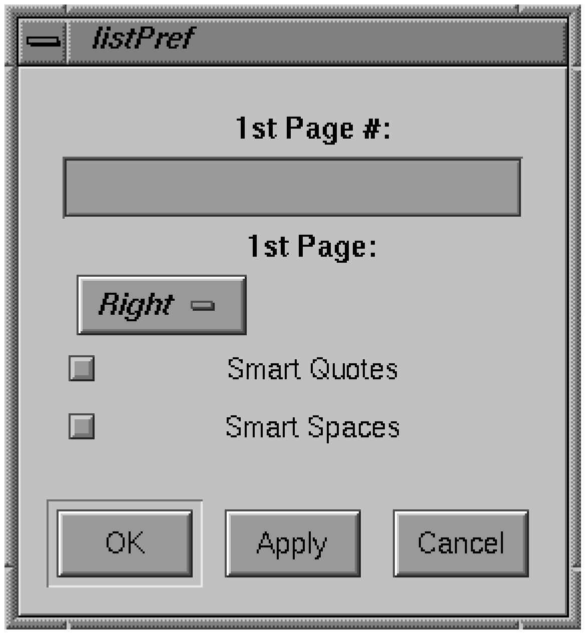


Figure 8-11 Example of a **VkPrefList** Item

VkPrefRadio is almost identical to **VkPrefGroup** except that you use it only for enforcing radio-style behavior on the **VkPrefToggle** items that it contains. You should add only **VkPrefToggle** items to a **VkPrefRadio** item. Otherwise, **VkPrefRadio** supports the same functionality as **VkPrefGroup**.

Figure 8-12 shows an example of a vertical **VkPrefRadio** item with a label. The label is the group item's label widget, not a **VkPrefLabel** item. Because the **VkPrefToggle** items are part of a homogenous **VkPrefRadio** item, you must set the XmNlabelString resources of their base widgets instead of their label widgets, as described in "Setting Toggle Preference Item Labels" on page 229.

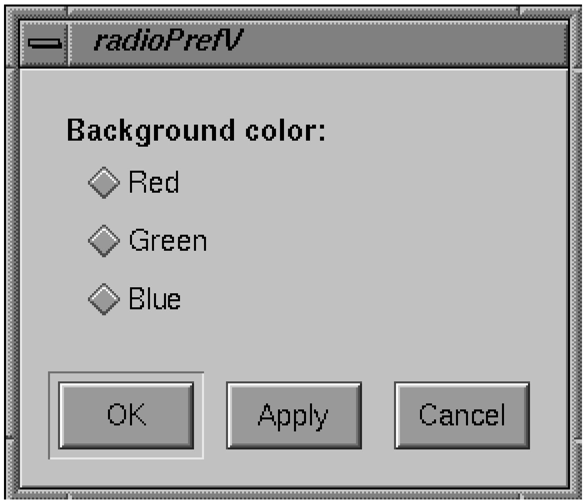


Figure 8-12 Example of a Vertical **VkPrefRadio** Item with Label

Creating Group Preference Items

The **VkPrefGroup** constructor has the following form:

```
VkPrefGroup(const char *name,
            Boolean horizOrientation = FALSE,
            Boolean noLabel = FALSE)
```

The **VkPrefGroup** constructor expects as its first argument the name of the preference item. The second argument is an optional Boolean value that determines the orientation of the group; FALSE, the default value, specifies a vertical orientation and TRUE specifies a horizontal orientation. The third argument is an optional Boolean value that determines whether or not to display a label for the group; FALSE, the default value, specifies that the group *should* display the label and TRUE specifies that the group *should not* display the label.

For instance, Example 8-1 contained the following constructor:

```
VkPrefGroup *numberGroup = new VkPrefGroup("numberGroup");
```

This created a new **VkPrefGroup** item named “numberGroup” with a vertical orientation and a visible label. Example 8-1 also contained the following constructor:

```
VkPrefGroup *horizGroup = new VkPrefGroup("horizGroup",
                                          TRUE, TRUE);
```

This created a new **VkPrefGroup** item named “horizGroup” with a horizontal orientation and no visible label.

The **VkPrefRadio** constructor accepts the same arguments as the **VkPrefGroup** constructor:

```
VkPrefRadio(const char *name,
            Boolean horizOrientation = FALSE,
            Boolean noLabel = FALSE)
```

For instance, Example 8-1 contained the following constructor:

```
VkPrefRadio *paginationGroup = new VkPrefRadio("paginationGroup");
```

This created a new **VkPrefRadio** item named “paginationGroup” with a vertical orientation and a visible label.

VkPrefList accepts only one argument, a character string specifying the name of the item:

```
VkPrefList(const char *name)
```

As noted earlier, all **VkPrefList** items have a vertical orientation and do not display a label. Example 8-1 created a **VkPrefList** item as the top-level preference item to contain all other preference items:

```
VkPrefList *docList = new VkPrefList("docList");
```

Adding and Deleting Preference Items from a Group Item

After creating a group item, you can add other items to it with the **addItem()** function:

```
void addItem(VkPrefItem *item)
```

Preference items appear in the order in which you add them. Example 8-1 added five preference items to the *docList* preference item:

```
docList->addItem(dialogName);
docList->addItem(sep1);
docList->addItem(numberGroup);
docList->addItem(sep2);
docList->addItem(horizGroup);
```

Once you have added items to a group item, you can access an individual child item with the **item()** function:

```
VkPrefItem *item(int item)
```

Simply provide an integer index value as an argument and **item()** returns a pointer to the desired preference item. The numbering of preference items within a group begins with 0, so to retrieve a pointer to the *numberGroup* item added above to *docList*, you could use the line:

```
item = docList->index(2);
```

The **size()** function returns the number of preference items currently associated with a group item:

```
int size()
```

The **deleteChildren()** function deletes all the items contained by a group item:

```
virtual void deleteChildren()
```

Note that this function does not just disassociate the items from the parent group item, it actually deletes the items. This is useful for freeing memory in a destructor. ViewKit does not provide any means of disassociating preference items without deleting them or of deleting individual items in a group. This should not pose a problem as most applications create preference dialogs at startup and almost never need to modify them afterwards.

Monitoring the Values of Preference Items Associated with a Group Item

The group preference items provide a **changed()** function just like all other preference items; however **changed()** operates differently with group items than it does with individual preference items. In group items, **changed()** calls the **changed()** functions of all child items in the group and returns TRUE if any of the child items have changed.

Setting Group Item Labels

To set the label for a **VkPrefGroup** or **VkPrefRadio** item, you must set the XmNlabelString resource of the preference item's label widget. (Remember that **VkPrefList** items do not display labels.) Example 8-1 illustrated this by setting the labels for numerous group items:

```
*numberGroupLabel.labelString:      Numbering:
*paginationGroupLabel.labelString:  Pagination:
*textGroupLabel.labelString:        Text:
```

Refer to "Preference Item Labels" on page 223 for more information on setting the label of a preference item.

The ViewKit Preference Dialog Class

The base preference dialog class, **VkPrefDialog**, is a subclass of **VkGenericDialog**, which is in turn a subclass of **VkDialogManager**. Thus, the **VkPrefDialog** class inherits all of the functions and data members provided by these base classes. For example, you post preference dialogs using the various **post()** variants, you set a preference dialog's title using the **setTitle()** function, and you set its button labels using the **setButtonLabels()** function.

Creating a Preference Dialog

Unlike the other dialog classes, **VkPrefDialog** does not create a global instance of a preference dialog. Instead, you must create a separate instance of **VkPrefDialog** for each preference dialog that you want to display in your program. For very simple preference dialogs (for example, just a few toggle buttons), you might be able to directly instantiate a **VkPrefDialog** object; however, in most cases you should create a separate subclass of **VkPrefDialog** for each preference dialog in your application. This is described in "Creating Preference Dialog Subclasses" on page 249.

The form of the **VkPrefDialog** constructor is:

VkPrefDialog(const char **name*, VkPrefItem **item* = NULL)

The **VkPrefDialog** constructor expects as its first argument the name of the preference dialog. The second argument is an optional pointer to a preference item that the dialog should use as the top-level preference item. See “Setting the Preference Items for a Preference Dialog” on page 245 for more information on setting the top-level preference item.

For example, the following line creates a preference dialog named “simplePref”:

```
VkPrefDialog *simplePref = new
VkPrefDialog("simplePref");
```

Setting the Preference Items for a Preference Dialog

A preference dialog can have only one top-level preference item. In most cases, you use a group item such as **VkPrefList** as the top-level item.

As described in “Creating a Preference Dialog” on page 244, you can set the top-level preference item in the **VkPrefDialog** constructor. You can also set the top-level item with the **setItem()** function:

```
void setItem(VkPrefItem *item)
```

Note: If the preference dialog already has a top-level preference item associated with it, **setItem()** replaces that item with the new item, but does not delete the old item. This allows you to reuse the old preference item later.

For example, the following line sets the item *docList* as the top-level item of the preference dialog *simplePref*:

```
simplePref->setItem(docList);
```

The **item()** function returns a pointer to the top-level item associated with a preference dialog:

```
VkPrefItem *item()
```

Posting and Dismissing Preference Dialogs

You post preference dialogs using any of the various **post()** variants provided by the base ViewKit dialog classes. You should not pass a message string argument to the **post()** function when posting a preference dialog.

For example, the following line posts the *simplePref* dialog as a non-modal, non-blocking dialog:

```
simplePref->post();
```

You should rarely have to unpost a preference dialog programmatically. ViewKit automatically dismisses a preference dialog when the user clicks either the *OK* or *Cancel* button. If for some reason you do need to unpost a preference dialog from your program, use the **unpost()** function.

Responding When the User Clicks a Preference Dialog Button

When the user clicks the *OK* or *Apply* button on a preference dialog, the dialog automatically applies any change of values to the preference dialog's items by setting the items's internally-stored values so that they match whatever is currently displayed on the screen. If the user clicks the *OK* button, the preference dialog calls its **hide()** function to remove itself from the screen. If the user clicks on the *Apply* button, the preference dialog remains visible on the screen.

When the user clicks the *Cancel* button on a preference dialog, the dialog automatically resets all of the dialog's preference items's on-screen values so that they match the items's internally-stored values. Additionally, the preference dialog calls its **hide()** function to remove itself from the screen.

The **VkPrefDialog** class also supplies a ViewKit member function callback named *prefCallback*. The preference dialog activates this callback whenever the user clicks the dialog's *Apply*, *OK*, or *Cancel* button. The callback passes as call data an enumerated value of type VkDialogReason, which is defined in **VkDialogManager**. The value can be any of VkDialogManager::OK, VkDialogManager::APPLY, or VkDialogManager::CANCEL, corresponding to the button that the user clicked. To notify components in your application when the user changes preferences associated with a preference dialog, register member functions with this ViewKit callback.

Note: When the user clicks the *OK* button, ViewKit first updates the preference items's internally stored values and activates the *prefCallback* callback with VkDialogManager::APPLY as the call data. Then, ViewKit activates the *prefCallback* callback with VkDialogManager::OK as the call data. In some ways, this is analogous to a IRIS IM pushbutton performing an **activate()** action followed by a **disarm()** action when a user clicks it. You can use this feature to perform certain actions whenever the user updates preference values by clicking either the *Apply* or *OK* button, and a separate set of actions when the user dismisses the preference dialog by clicking the *OK* button.

For example, consider a window, *myWindow*, that is a member of the subclass **MyWindow**, derived from **VkWindow**. In this example, assume that there is a preference dialog, *displayPrefs*, that is a member of the subclass **DisplayPrefDialog**, derived from **VkPrefDialog**, that allows the user to specify certain display parameters such as the font. *myWindow* could register its member function **MyWindow::fontChanged()** to be called whenever the user clicks a button in the preference dialog *displayPrefs*, by using the following line of code:

```
displayPrefs->addCallback(VkPrefDialog::prefCallback,
                  this,
                  (VkCallbackMethod) &MyWindow::fontChanged);
```

When **MyWindow::fontChanged()** is called, it checks to see if any of the parameters in which it is interested have changed and, if so, performs whatever processing is needed. For example:

```
void MyWindow::fontChanged(VkComponent *obj,
                           void *clientData,
                           void *callData)
{
  DisplayPrefDialog *dialog = (DisplayPrefDialog*) obj;
  MyWindow *win = (MyWindow*) clientdata;
  VkDialogManager::VkDialogReason reason =
                  (VkDialogManager::VkDialogReason) callData;
  // If the user clicked Cancel, nothing changed
  if (reason == VkDialogManager::CANCEL)
    return;
  // Now process new preference values as needed ...
}
```

Using Values Set in a Preference Dialog

To retrieve the value of a preference item, simply call that item's **getValue()** function.

This implies that preference items must be accessible to all components that need to use the preference values. For example, if you create a subclass for a preference dialog, declare as "public" those preference items that you want to access outside of the dialog.

Example 8-3 shows the header for a **NamePref** subclass in which two preference items, *firstName* and *lastName*, are declared "public." These two preference items can be accessed by other components in the applications.

Example 8-3 Declaring Preference Items as Publicly Accessible

```
class NamePref: public VkPrefDialog {

  protected:
    VkPrefGroup *nameGroup;

    static String _defaultResources[];
    virtual Widget createDialog(Widget parent);

  public:
    VkPrefText *firstName;
    VkPrefText *lastName;

    NamePref ( const char *name );
    ~NamePref();
    virtual const char* className();
};
```

The **NamePref** subclass also contains a group, *nameGroup*, which is declared "protected." In most cases, outside components would not need to access a group item. One case in which it could be useful to make a group item publicly accessible is if you want other components to be able to activate and deactivate a group of preference items by calling the **activate()** and **deactivate()** functions on that group item.

Creating Preference Dialog Subclasses

The preferred method of handling preference dialogs in ViewKit applications is to create a separate subclass for each preference dialog in the application. Properly designed, a preference dialog can serve as a self-contained component that you can use in multiple applications.

The first step in creating a preference dialog subclass is to decide what preference items to include. List all of the information you want to be able to set with the preference dialog and determine which preference item class is appropriate for each item. For example, an item requiring text input is an obvious candidate for a **VkPrefText** item. However, an item allowing the user to choose one of several options can be handled by either a single **VkPrefOption** item or a number of **VkPrefToggle** items grouped with a **VkPrefRadio** item. Presumably, you want all of these preference items to be accessible outside of the preference dialog, so you want to declare these items in the "public" section of your class declaration.

Then determine the layout you want for the preference dialog. You should group similar items together so that a user can easily find and set related items. The layout determines what group items you need. Usually, you can define these items in the "private" or "protected" section of your class declaration; however, in some cases, you might want to declare some groups as "public." For example, you might want to be able to activate and deactivate a group of preference items by calling the **activate()** and **deactivate()** functions on that group item.

Then determine how you want to "publicize" changes in preference items to other components in your application. In many cases, those components can simply call the **getValue()** functions for appropriate items as needed. However, some components need to be notified immediately whenever certain preference items change. In most cases, these components can register ViewKit member function callbacks with the preference dialog that are called whenever the user clicks one of the dialog's buttons. The components can then test for changes in preference item values in their callback functions and react accordingly.

In some cases, you might need to perform special processing when the user clicks one of the preference dialog's buttons. In that case, you can override the default **ok()**, **apply()**, or **cancel()** function for the dialog. These functions are called whenever the user clicks the corresponding button. In your override definition, you should perform whatever processing is needed and then call the base **VkPrefDialog::ok()**, **VkPrefDialog::apply()**, of **VkPrefDialog::cancel()** function as appropriate.

Usually you should also provide a set of default resource values to serve as labels for all the dialog's preference items. To do so, you must override the **createDialog()** function, which creates and manages all of the widgets in a preference dialog. Your preference dialog's **createDialog()** function must perform the following tasks in order:

1. Call **setDefaultResources()** to set the dialog's default resources.
2. Create all preference items for the dialog.
3. Set the dialog's top-level item using the **setItem()** function.
4. Call the base **VkPrefDialog::createDialog()** function to create the dialog.
5. Pass the dialog's base widget, returned by **VkPrefDialog::createDialog()**, as the return value of **createDialog()**.

Example 8-1 shows a complete example of a preference dialog subclass. You could include **DocPrefDialog** dialogs in any application that needed to set various document parameters.

Chapter 9

The ViewKit Graph Component

ViewKit provides a high-level component, **VkGraph**, for displaying and manipulating complex arc-and-node graphs. Figure 9-1 shows the inheritance graph for **VkGraph** and an auxiliary class, **VkNode**.

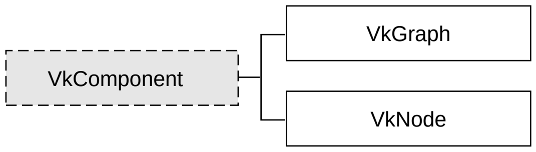


Figure 9-1 The Inheritance Graph for the ViewKit Graph Classes

Overview of ViewKit Graphs

VkGraph is a self-contained ViewKit component for displaying and manipulating complex arc-and-node graphs. The graph can be disconnected and can contain cycles. **VkGraph** can arrange the nodes horizontally or vertically and change the orientation interactively. **VkGraph** also provides controls for interactive zooming, node repositioning, and node alignment. Figure 9-2 shows an example of a graph created using the **VkGraph** component.

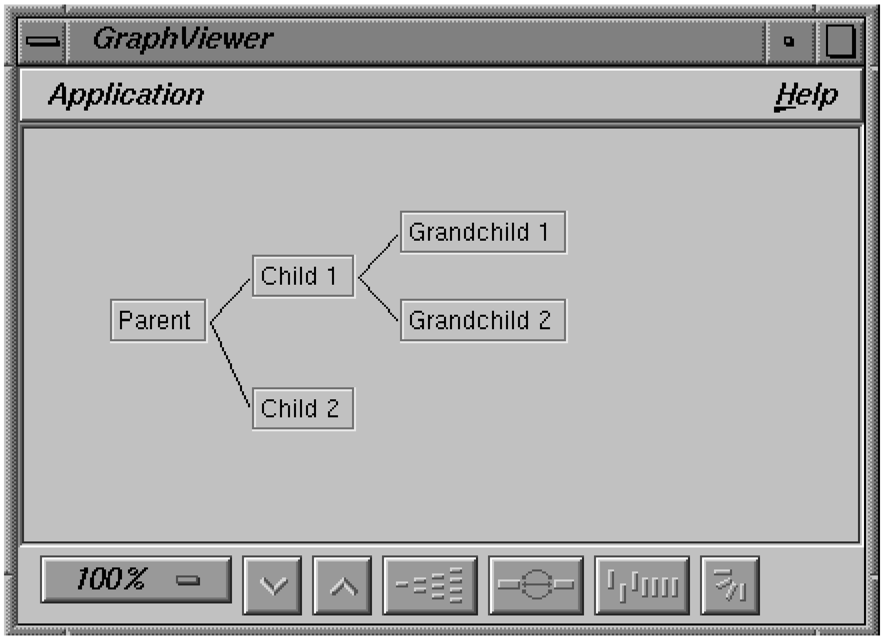


Figure 9-2 Example of a Graph Created with **VkGraph**

All nodes displayed by a **VkGraph** component must be instances of the **VkNode** class or subclasses that you derive from it. By default, **VkNode** creates a SgIconGadget(3x), but if you create a subclass **VkNode**, you can use any widget for a node.

The Graph Widget

The basis of the **VkGraph** class is the SgGraph widget, which manages and displays the graph. This section provides an overview of the SgGraph widget. For in-depth information on interacting with the graph widget, consult the SgGraph(3x) man page.

A primary responsibility of the SgGraph widget is to clearly and systematically lay out the nodes. The graph layout algorithm is a simple and efficient tree layout algorithm designed to handle forests of nodes. It lays out nodes as a multi-rooted tree.

By default, the graph widget created by the **VkGraph** class operates in a read-only mode in which the graph widget is used primarily as a layout manager for arranging the node widgets. By modifying certain SgGraph resources, you can also interactively edit the displayed graph, creating and moving arcs and nodes. However, to support most of the functionality of the edit mode, you must provide callback functions and other information to the graph widget so that you can interpret the edit operations and use them in your program.

Refer to the SgGraph(3x) man page for details on the resources and callbacks used for edit mode. Also refer to the example in */usr/share/src/ViewKit/ComponentDemos/graph*.

Example of Building a Graph

The process of building and displaying a graph using the **VkGraph** component consists of the following steps:

1. Creating the nodes
2. Specifying node connectivity
3. Indicating which nodes to display
4. Laying out the graph

Example 9-1 illustrates this process by showing the code used to create the graph shown in Figure 9-2.

Example 9-1 Example of Creating a Graph Using **VkGraph**

```
#include <Vk/VkApp.h>
#include <Vk/VkWindow.h>
#include <Vk/VkNode.h>
#include <Vk/VkGraph.h>
#include <Vk/VkMenu.h>
```

```
class GraphWindow: public VkWindow {

  public:
    GraphWindow( const char *);
    ~GraphWindow();
    virtual const char* className();
  protected:
    VkGraph *graph;
    VkNode *p_node, *c1_node, *c2_node, *gc1_node, *gc2_node;

  private:
    static void quitCallback (Widget, XtPointer, XtPointer);
    static VkMenuDesc appMenuPane[];
};


VkMenuDesc GraphWindow::appMenuPane[] = {
  { ACTION,   "Quit",      &GraphWindow::quitCallback },
  { END }
};

GraphWindow::GraphWindow(const char *name) : VkWindow( name )
{
    // Create nodes

    p_node   = new VkNode("parentNode", "Parent");
    c1_node  = new VkNode("childNode1", "Child 1");
    c2_node  = new VkNode("childNode2", "Child 2");
    gc1_node = new VkNode("grandChildNode1", "Grandchild 1");
    gc2_node = new VkNode("grandChildNode2", "Grandchild 2");

    // Create graph

    graph   = new VkGraph( "graph", mainWindowWidget() );

    // Add nodes to graph

    graph->add(p_node, c1_node);     // p_node is parent to c1_node
    graph->add(p_node, c2_node);     // p_node is parent to c2_node
    graph->add(c1_node, gc1_node);   // c1_node is parent to gc1_node
    graph->add(c1_node, gc2_node);   // c1_node is parent to gc2_node
    graph->displayAll();             // Display all nodes in graph
    graph->doLayout();               // Layout the graph

    addView(graph);                  // Set graph to be window's view
    addMenuPane("Application", appMenuPane);         // Create menu bar
}
GraphWindow::~GraphWindow()
{
    delete graph;
    delete p_node;
    delete c1_node;
    delete c2_node;
    delete gc1_node;
    delete gc2_node;
}

const char* GraphWindow::className()
{
    return "GraphWindow";
}

void GraphWindow::quitCallback ( Widget, XtPointer, XtPointer )
{
    theApplication->quitYourself();
}

void main(int argc, char **argv)
{
  VkApp       *myApp    = new VkApp("GraphViewer", &argc, argv);
  GraphWindow *graphWin = new GraphWindow("GraphViewer");

  graphWin->show();
  myApp->run();
}
```

This example creates a **VkWindow** subclass to contain the graph. The graph itself is created in the **GraphWindow** constructor:

1. The program creates five nodes. These nodes are instances of the **VkNode** class, which is described in "The ViewKit Node Class" on page 261. The version of the **VkNode** constructor used in this example accepts a name that is used for internal reference and a label that is displayed.
2. The program creates a **VkGraph** object. The **VkGraph** constructor accepts as arguments a name and a parent widget, in this case, the main window widget obtained by **mainWindowWidget()**.
3. The program adds the nodes to the graph using **VkGraph::add()**. When called with pointers to two nodes, this function associates the nodes with the graph, and marks the first node as being the parent of the second node. In this way, the program specifies the structure of the graph.
4. The program calls **VkGraph::displayAll()**, which indicates that the graph should display all nodes.
5. The program calls **VkGraph::doLayout()**, which lays out the graph according to the layout algorithm and manages all widgets associated with the graph.

Interactive Viewing Features Provided by VkGraph

In addition to displaying a graph, **VkGraph** automatically provides controls for interactively manipulating the graph. One set of controls is contained in the control panel, shown in Figure 9-3, which appears along the bottom of the graph.

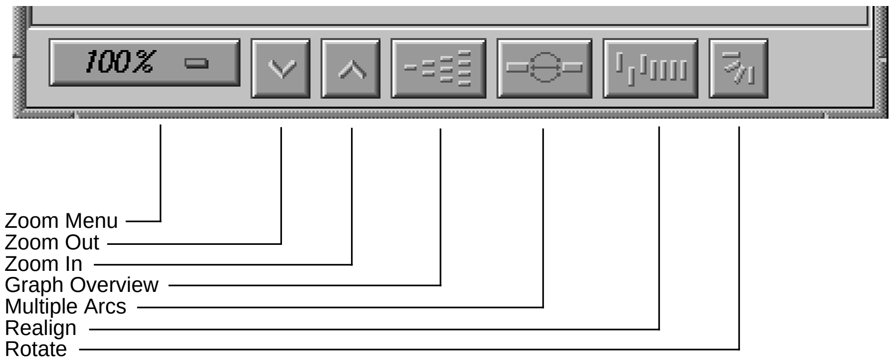


Figure 9-3 The Graph Command Panel

The control panel contains buttons and a menu that allow the user to interactively control various characteristics of the graph's display. Using the control panel the user can:

- zoom in or out
- display a graph overview
- toggle between displaying and hiding duplicate arcs connecting nodes
- align nodes
- toggle between horizontal and vertical orientation

Additionally, **VkGraph** automatically creates popup menus that contain commands that allow the user to hide and display nodes in the graph.

Zooming

VkGraph provides eight preset zoom settings that allow the user to shrink or enlarge the size of the graph. The user can directly set the zoom value using the Zoom menu shown in Figure 9-4.

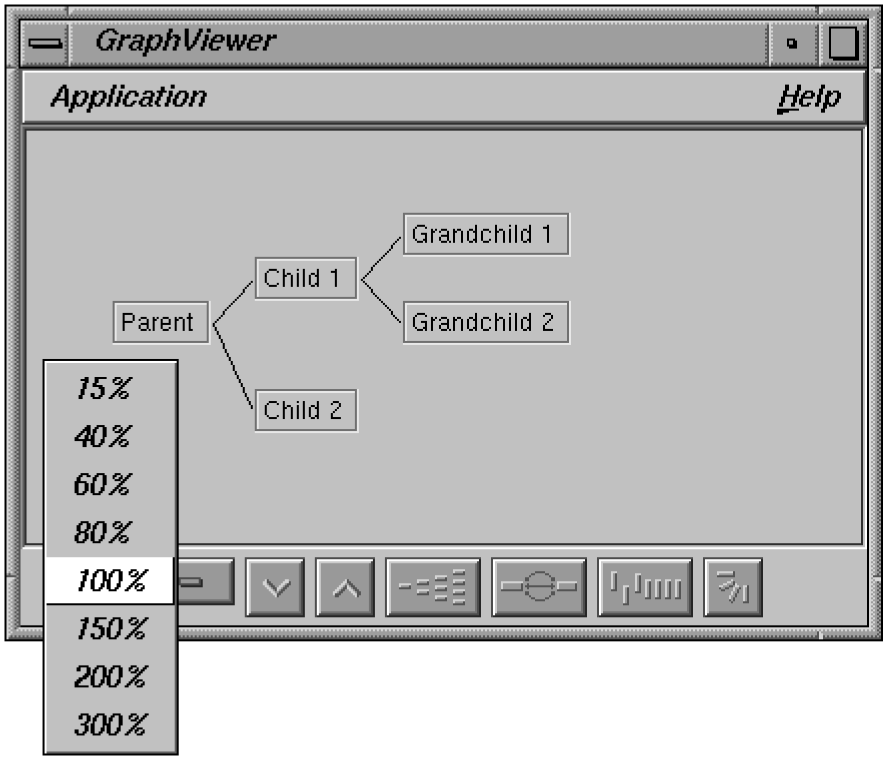


Figure 9-4 Interactively Changing the Graph Zoom Value

Clicking the *Zoom Out* button (the down-arrow button immediately to the right of the Zoom menu) changes the zoom setting to the next lower value, and clicking the *Zoom In* button (the up-arrow button to the right of the *Zoom Out* button) changes the zoom setting to the next higher value.

Graph Overview

The user can display an overview of all a graph's visible nodes by clicking the *Graph Overview* button.

Within the overview window is a viewport that represents the boundaries of the graph visible in the main graph window. The user can click the viewport and drag it to a new location to change the area visible in the main graph window. As the user drags the viewport, the main graph window scrolls to match the viewport's location in the overview.

The overview window also contains an Admin menu with these commands:

"Scale to Fit"	Scales the graph in the window to match the aspect ratio of the window.
"Show Arcs"	Shows the arcs between nodes. This option is turned on by default; if the arcs clutter the window, the user can turn off the option, which removes the arcs from the window.
"Close"	Closes the overview window.

Displaying Duplicate Arcs

By default, the graph displays only a single arc between nodes, even if you define multiple connections between the nodes. The user can click the *Multiple Arcs* button to display multiple arcs between nodes; the graph displays an arc for each connection you defined. The user can turn off multiple-arc display by clicking the *Multiple Arcs* button again.

Realigning Nodes

Occasionally, as a result of moving or displaying nodes, your graph display might become cluttered. The *Realign* button "cleans up" the graph display by laying out all visible nodes again.

Toggling Between Horizontal and Vertical Orientation

The default graph orientation is horizontal. The user can change to a vertical orientation by clicking the *Rotate Graph* button. The user can return to the horizontal orientation by clicking the *Rotate Graph* button again.

Hiding and Displaying Nodes

VkGraph provides controls that allow the user to hide a single node, reveal a node's parents or children, or collapse the part of the graph that branches from a node. To perform any of these actions, the user moves the pointer onto the node and presses the right mouse button to open the popup Node menu. The Node menu contains four commands; only commands applicable to that node are made available. Nonapplicable commands are grayed. The commands are:

"Hide Node"	Hides the node and connecting arcs from the graph.
"Collapse Subgraph"	Hides all descendent nodes and connecting arcs.
"Show Immediate Children"	Displays the node's immediate child nodes and connecting arcs. This command does not display more than the first subordinate level of nodes.
"Show Parents"	Displays the node's immediate parent nodes and connecting arcs.

Edit Mode Operations

There are additional operations that a user can perform if you set the graph to edit mode, as described in "The Graph Widget" on page 252. By default, the graph widget created by the **VkGraph** class operates in read-only mode. You can set the graph widget to edit mode in a **VkGraph** subclass.

Note: To support much of the functionality of the edit mode, you must provide callback functions and other information to the graph widget so that you can interpret the edit operations and use them in your program. Refer to the SgGraph(3x) man page for details on the resources and callbacks used for edit mode.

You must select one or more nodes before you can perform an operation on it. You can select nodes only if the graph is in edit mode. By default, the graph is created in display-only mode.

To perform most operations in edit mode, the user must first select one or more nodes. The user can select a single node by clicking it with the left mouse button. The graph highlights the selected node. The user can select additional nodes by holding down the **`<Control>`** key as he or she clicks additional nodes with the left mouse button. The user can also select multiple nodes with a bounding box by moving the pointer to a spot on the graph where there is no node or arc, then holding down the left mouse button and dragging out a bounding box. When the button is released, all nodes fully enclosed by the box are selected. (Partially enclosed nodes aren't selected.)

The user can deselect nodes by clicking the left mouse button on a blank section of the graph.

The user can move a node by clicking that node with the middle mouse button and then dragging the node anywhere in the graph window. The user can move several nodes at once by first selecting the nodes and then clicking any one of the nodes with the middle mouse button and dragging the nodes to their new position.

The popup Selected Nodes menu allows the user to perform an operation on all selected nodes. To open the Selected Nodes menu, the user moves the pointer to any blank area of the graph, and then presses the right mouse button. The menu has three commands:

"Hide Selected Nodes"
Hides all selected nodes and their connecting arcs.

"Collapse Selected Nodes"
Hides all descendent nodes and connecting arcs of the selected nodes.

"Expand Selected Nodes"
Displays the immediate children of all the selected nodes.

The ViewKit Node Class

VkGraph requires that all nodes that it contains be instances of either the **VkNode** class or a subclass of **VkNode**. The **VkNode** class is responsible for tracking the connectivity, display characteristics, and other features of the nodes. **VkNode** is a subclass of **VkComponent**.

The **VkNode** class provides only basic support for interacting with the node widget. In particular, you can set the string displayed as a label through the **VkNode** constructor; however, you can create subclasses of **VkNode** that support any widget type, as discussed in "Creating Node Subclasses" on page 265.

Basic Node Functionality

This section describes the basic functionality provided by the **VkNode** class. Most **VkNode** functions other than the constructor are for use by **VkGraph**; however, you might occasionally find some of the utility and access functions useful.

Node Constructor and Destructor

The **VkNode** constructor has two forms:

```
VkNode(const char *name, const char *label = NULL)

VkNode(const char *name, VkNode *parent,
       const char *label = NULL)
```

name is the node’s component name. You should provide unique names for all nodes. *label* is the label that the node displays when visible in a graph. If you do not provide a label, the node uses the component name as the label. You can optionally provide a pointer to an existing node, which the constructor uses as a parent node for the new node.

As an example, the following line of code creates the node *state19* with the internal name “state19” and the label “Indiana”:

```
VkNode state19 = new VkNode("state19", "Indiana");
```

The following line of code creates a new node, *city41*, as a child of *state19*. The name of the new node is “city41” and the label is “Terre Haute”:

```
VkNode city41 = new VkNode("city41", state19, "Terre Haute");
```

Note: The **VkNode** constructor merely initializes internal variables; it does not create any widgets. The **VkGraph** object of which a **VkNode** object is a member can create and destroy node widgets as needed. The **VkGraph** object calls a protected member function, **VkNode::build()**, whenever it needs to create a node’s widget. “Creating Node Subclasses” on page 265 discusses **build()** in more detail.

The **VkNode** destructor destroys the node’s widget if it exists and deallocates all other internal storage.

Node Utility Functions

VkNode maintains a list of child nodes that you can access using the access functions described in “Node Access Functions” on page 263. By default, the order of the child nodes in this list depends on the order in which you specified the child relationships. The first child node you specify has an index of 0, the second 1, and so on.You can use the **VkNode::sortChildren()** to sort the immediate child nodes of a node:

```
void sortChildren()
```

The default algorithm used by **sortChildren()** sorts nodes alphabetically by their internal node names (not their labels).

You can direct **VkNode** to use a different sort comparison function with **VkNode::setSortFunction()**:

```
static void setSortFunction(VkNodeSortFunction func)
```

The type definition of VkNodeSortFunction is:

```
typedef int (*VkNodeSortFunction)(VkNode *, VkNode *)
```

The function you provide must be a static function that accepts as arguments two nodes, and returns an integer value less than zero if the first node comes before the second node, zero if the two nodes are equal, and greater than zero if the second node comes before the first node. For example, the following function sorts nodes by their label strings:

```
static int sortNodesByLabel(VkNode *one, VkNode *two)
{
    int value = strcmp(one->label(), two->label());
    return value;
}
```

(“Node Access Functions” on page 263 describes **VkNode::label()**.)

Node Access Functions

VkNode provides a number of access functions for obtaining values associated with a node.

You can retrieve the node’s component name using **VkNode::name()**:

```
char *name() const
```

You can retrieve the node's label string with **VkNode::label()**:

```
virtual char *label()
```

If you did not provide a label string in the node constructor, the value of the label string is the same as the component's name.

You can determine the number of parent and child nodes with **VkNode::nParents()** and **VkNode::nChildren()** respectively:

```
int nParents() const
int nChildren() const
```

You can retrieve a specific parent or child node using **VkNode::parent()** and **VkNode::child()** respectively:

```
VkNode *parent(int index) const
VkNode *child(int index) const
```

By default, the order of the parent and child nodes depends on the order in which you specified the parent or child relationships. The first parent node you specify has an index of 0, the second 1, and so on. Initially, the child nodes are numbered similarly; however, if you sort the child nodes using the **sortChildren()** function, the nodes are reordered according to the sort function you used. For example, if you sorted the child nodes alphabetically by component name, the first child node alphabetically has an index of 0, the second 1, and so on.

You can find a particular parent or child node by component name using **VkNode::findParent()** and **VkNode::findChild()** respectively:

```
VkNode *findParent(char *name)
VkNode *findChild(char *name)
```

These functions return a pointer to the node if found, and NULL if they do not find the node. These functions search only immediate parent or child nodes, not all ancestor or descendent nodes.

Creating Node Subclasses

You can create subclasses of **VkNode** to extend its features in a variety of ways to maintain additional data or to change the way the node displays itself in a graph. Some possibilities include:

- providing access functions for setting and retrieving resources of the default SgIconGadget(3x) widget provided by the **VkNode** base class
- using widgets other than the default SgIconGadget(3x) widget
- creating additional data members and member functions to store application-specific node information

You have a great deal of flexibility in deciding how to extend the **VkNode** class. The important restriction that you must keep in mind is that the **VkGraph** object of which a **VkNode** object is a member can create and destroy node widgets as needed. Therefore, in your subclass function definitions you cannot assume that your node's widget exists.

The **VkGraph** object calls a protected member function, **VkNode::build()**, whenever it needs to create a node's widget. If you want to use the additional features of the default SgIconGadget widget or if you want to use a different widget in you subclass, you must override **build()**:

```
virtual void build(Widget parent)
```

If you simply want to use the additional features of the default SgIconGadget widget, you can call **VkNode::build()** from within your subclass's **build()** function to create the SgIconGadget widget and set the widget's label. Then, you can perform any additional operations you want. (Consult the SgIconGadget(3x) reference page for more information on using this widget.) For example:

```
void MyNode::build(Widget parent)
{
    VkNode::build(parent);
    // Additional setup...
}
```

If you want to use your own widget or widget hierarchy, create the widget(s) using *parent* as the parent widget, and assign the widget or root of a widget hierarchy to the *_baseWidget* data member. After creating the *_baseWidget*, call **installDestroyHandler()**, as described in "Handling Component Widget Destruction" on page 24.

From within a **VkNode** subclass you can also access the *_label* data member:

`char *`*_label*

_label contains the node's label string as set by the **VkNode** constructor.

The ViewKit Graph Class

This section describes how to build and manipulate graphs using the **VkGraph** class. Minimally, you must perform the following actions to build and display a ViewKit graph:

1. Create the **VkGraph** object
2. Create the nodes as instances of **VkNode** or a subclass
3. Add the nodes to the graph and specify the node connectivity
4. Indicate which nodes to display
5. Lay out the graph

The Graph Constructor and Destructor

The **VkGraph** constructor is simple with few arguments. You must provide a name and the parent widget for the graph:

`VkGraph(char *`*name*`, Widget `*parent*`)`

The **VkGraph** destructor destroys the graph. It does not destroy any **VkNode** objects that are part of the graph.

Adding Nodes and Specifying Node Connectivity

After you create nodes, you must add them to the graph object you created. Also, if you didn't specify the parent-child relationship for the nodes when you created them, you should supply the remaining connectivity information when adding the nodes to the graph. (See "The ViewKit Node Class" on page 261 for information on creating nodes.)

The **VkGraph::add()** function adds nodes to a graph object:

```
virtual int add(VkNode *node)

virtual void add(VkNode *parent, VkNode *child,
                 char *attribute = NULL)
```

If you supply only one node pointer as an argument, **add()** simply adds the node to the graph. If you have already added the node to the graph, **add()** does nothing.

If you supply two node pointers as arguments, **add()** adds both nodes to the graph and establishes the first node as the parent of the second node. If you have already added either node to the graph, **add()** does not add the node again, but it does establish the parent-child relationship between the nodes.

Note: The second form of **add()** establishes the parent-child relationship between nodes even if one already exists. Thus, it is possible to have more than one connection between nodes. By default, the graph displays only a single arc between connected nodes, even if you define multiple connections between the nodes. However, as described in “Displaying Duplicate Arcs” on page 259, by clicking the graph’s Multiple Arcs button the user can force the graph to an arc for each connection you defined. To turn off multiple-arc display, the user can click the Multiple Arcs button again.

When specifying a parent/child connection using **add()**, you can specify an *attribute* for that connection. An attribute is an arbitrary name that you can use to control the appearance of the arc widget that connects the two nodes. For example, assume that you add two nodes to a graph as follows:

```
graph->add(parent, child, "primary");
graph->add(parent, child, "secondary");
```

The resulting graph displays two connecting arcs between the two nodes. You can now specify X resources to control various aspects of the arc. For example:

```
*primary*foreground:      red
*primary*arcDirection:    bidirected
*secondary*foreground:    blue
*secondary*arcDirection:  undirected
*secondary*style:         LineOnOffDash
```

You can use this method to set many of the resources supported by the SgArc widget. The resources you can specify are: XmNforeground, XmNtoSide, XmNfromSide, XmNfromPosition, XmNtoPosition, XmNarcDirection, XmNfontList, XmNarcWidth, XmNstyle, and XmNdashes. See the SgArc(3x) man page for details on these resources.

The following code fragment creates a graph, creates two nodes, establishes a parent-child relationship between the nodes, and adds the nodes to the graph:

```
graph   = new VkGraph("graph", parent);
p_node  = new VkNode("parentNode", "Parent");
c1_node = new VkNode("childNode1", p_node, "Child 1");
graph->add(p_node);
graph->add(c1_node);
```

Note that in this example, the connection between the two nodes is established when you create *c1_node*. Therefore, you must add the nodes to the graph using separate calls to **add()**. If, instead of the two separate calls, you execute:

```
graph->add(p_node, c1_node);
```

then you not only add the two nodes to the graph, but you establish a second connection between the nodes.

You can accomplish the same result as above by creating the nodes without providing the parent-child relationship, and then specifying the connection when you add the nodes to the graph. The following code fragment is functionally equivalent to that shown above:

```
graph   = new VkGraph("graph", parent);
p_node  = new VkNode("parentNode", "Parent");
c1_node = new VkNode("childNode1", "Child 1");
graph->add(p_node, c1_node);
```

Removing Nodes

You can remove nodes from a graph using **VkGraph::remove()**:

```
virtual void remove(VkNode *node, Boolean deleteNode = FALSE)
```

By default, **remove()** removes the node from the graph but does not delete it. If you set the *deleteNode* argument to TRUE, **remove()** deletes the node when it removes it.

Indicating Which Nodes to Display

Once you have added all nodes to a graph and specified their connectivity, you must indicate which nodes the graph should display. **VkGraph** provides many functions that allow you to display or hide all of the graph, individual nodes, and portions of node subtrees.

After displaying nodes, you should call one of the graph layout member functions as described in “Laying Out the Graph” on page 272. Otherwise, the nodes might not display in desired locations.

The basic display functions are **VkGraph::displayAll()** and **VkGraph::clearAll()**:

```
virtual void displayAll()
void clearAll()
```

displayAll() displays all nodes and **clearAll()** hides all nodes. Typically, after creating your graph, you execute **displayAll()** to display all of the nodes. For example:

```
graph->displayAll();
```

Sometimes you might want to display only portions of your graph. **VkGraph** provides functions to operate on either single nodes or subtrees of nodes.

The **VkGraph::display()** function displays a single node:

```
virtual void display(VkNode *child)
virtual VkNode *display(char *name)
```

You can provide **display()** with either a pointer to the node or the component name of the node. If you provide the node’s name, this function returns a pointer to the node.

VkGraph::undisplay() hides a single node:

```
virtual void undisplay(VkNode *node)
virtual void hideNode(VkNode *node)
```

VkGraph::hideNode() is equivalent to **undisplay()**.

VkGraph also provides a large number of functions that display or hide portions of the graph:

- **displayWithChildren()** displays a node and all of its immediate child nodes (not all descendent nodes). If you provide the node’s name, this function returns a pointer to the node.

  ```
  virtual void displayWithChildren(VkNode *node)
  virtual VkNode *displayWithChildren(char *name)
  ```

- **expandNode()** is functionally equivalent to **displayWithChildren()** except that it also calls **VkGraph::doSubtreeLayout()** to lay out the child nodes according to the graph's layout algorithm. See "Laying Out the Graph" on page 272 for more information on **doSubtreeLayout().**

  ```
  virtual void expandNode(VkNode *node)
  ```

- **displayWithAllChildren()** displays a node and all of its descendent nodes. If you provide the node's name, this function returns a pointer to the node.

  ```
  virtual void displayWithAllChildren(VkNode *node)
  virtual VkNode *displayWithAllChildren(char *name)
  ```

- **expandSubgraph()** is functionally equivalent to **displayWithAllChildren()** except that it also calls **VkGraph::doSubtreeLayout()** to lay out the child nodes according to the graph's layout algorithm. See "Laying Out the Graph" on page 272 for more information on **doSubtreeLayout().**

  ```
  virtual void expandSubgraph(VkNode *node)
  ```

- **hideAllChildren()** hides all of a node's descendent nodes. Note that this function does not hide *node* itself.

  ```
  virtual void hideAllChildren(VkNode *node)
  ```

- **hideWithAllChildren()** hides a node and all of its descendent nodes.

  ```
  virtual void hideWithAllChildren(VkNode *node)
  ```

- **displayWithParents()** displays a node and all of its immediate parent nodes (not all ancestor nodes). If you provide the node's name, this function returns a pointer to the node.

  ```
  virtual void displayWithParents(VkNode *node)
  virtual VkNode *displayWithParents(char *name)
  ```

- **displayWithAllParents()** displays a node and all of its ancestor nodes. If you provide the node's name, this function returns a pointer to the node.

  ```
  virtual void displayWithAllParents(VkNode *node)
  virtual VkNode *displayWithAllParents(char *name)
  ```

- **hideParents()** hides all of a node's immediate parent nodes (not all ancestor nodes). Note that this function does not hide *node* itself.

  ```
  virtual void hideParents(VkNode *node)
  ```

- **displayParentsAndChildren()** displays a node and all of its immediate parent and child nodes (not all ancestor and descendent nodes). If you provide the node's name, this function returns a pointer to the node. Note that this function *does* display *node* itself.

  ```
  virtual void displayParentsAndChildren(VkNode *node)
  virtual VkNode *displayParentsAndChildren(char *name)
  ```

- **hideParentsAndChildren()** hides all of a node's immediate parent and child nodes (not all ancestor and descendent nodes). Note that this function *does not* hide *node* itself.

  ```
  virtual void hideParentsAndChildren(VkNode *node)
  ```

You can also create your own functions for determining whether or not nodes are displayed and then use the **VkGraph::displayIf()** function to apply those functions:

```
virtual void displayIf(VkGraphFilterProc)
```

The type definition of VkGraphFilterProc is:

```
typedef Boolean (*VkGraphFilterProc) (VkNode *)
```

The function you provide must be a static function that accepts a node as an arguments and returns TRUE if the node should be displayed.

Note: **displayIf()** does *not* hide (that is, call **undisplay()**) if the filter function returns FALSE for a node. Therefore, if you want to display only those nodes for which the filter function returns TRUE, you must first call **clearAll()**.

For example, the following function displays only those nodes whose names begin with the string "state":

```
static Boolean displayState(VkNode *node)
{
    if ( strcmp("state", node->name(), 5)
        return TRUE;
    else
        return FALSE;
}
```

Laying Out the Graph

The final step in displaying a graph is to lay it out. Laying out the graph arranges the widgets in a logical manner and then manages the widgets.

To lay out the entire graph, call the **VkGraph::doLayout()** function, which applies the layout algorithm to the entire graph and then manages all widgets associated with the graph:

```
void doLayout()
```

If you modify the graph after displaying it, or if you allow the user to edit the graph interactively, the graph might become cluttered and you might want to lay out the graph again. To do so you can call **doLayout()** again to force the graph to reapply the layout algorithm to the graph to clean up the display. As an example, the Realign button provided on the graph command panel simply calls **doLayout()** whenever the user clicks the button.

If, after displaying the graph, you display any additional nodes (for example, using the **VkGraph::display()** function), you must force a layout of the graph to manage all the widgets you created. You can call **doLayout()** again to do so, but this applies the layout algorithm to the entire graph. Doing so could produce major changes in the layout of the entire graph, which could be disruptive and undesired if the user has previously moved nodes. Also, it could take considerable time if the graph is large. In this case, you can instead call the **VkGraph::doSubtreeLayout()** function which, given a root node, applies the layout algorithm to just a subtree of the graph:

void doSubtreeLayout(VkNode **node*)

For example, the following code fragment illustrates displaying a graph, *graph*, and then displaying another node, *newNode*:

```
// At this point, all nodes are created, the connectivity is
// specified, and certain nodes selected to be displayed

// Lay out and display the graph

graph->doLayout();
```

```
// Mark newNode to be displayed

graph->display(newNode);

// Display newNode, re-laying out only the subtree
// under newNode

graph->doSubtreeLayout(newNode);
```

VkGraph::doSparseLayout() is a special-purpose build and layout function that displays the relationship between a node and its grandparent nodes even if the node's parents are not displayed:

```
void doSparseLayout()
```

doSparseLayout() performs a special build of the graph and whenever it finds a node with an undisplayed parent node, it checks to see whether there are any displayed grandparent nodes. If **doSparseLayout()** finds such grandparent nodes, it creates a dashed-line arc (instead of a solid-line arc) to connect the node and its grandparent nodes. After finishing the build process, **doSparseLayout()** performs a layout of the entire graph and manages all widgets associated with the graph.

Butterfly Graphs

So far, this chapter has discussed creating tree graphs using the **VkGraph** class. However, **VkGraph** also supports *butterfly graphs*, which display only a central node and its immediate parent and child nodes. The central node of a butterfly graph is called the *butterfly node*.

VkGraph can construct a butterfly graph from any graph specification. All you need to do is call **VkGraph::displayButterfly()** to specify one node as the butterfly node; **VkGraph** automatically determines which nodes to display:

```
virtual void displayButterfly(VkNode *node)
virtual VkNode *displayButterfly(char *name)
```

Then call **VkGraph::doLayout()** to lay out the graph as you normally would. For example, assuming that you have already defined a graph specification for a graph called *graph*, the following code fragment would instruct the graph object to display a butterfly graph centered on the node *centerNode*:

```
graph->displayButterfly( centerNode );
graph->doLayout();
```

After displaying a butterfly graph, you can use **displayButterfly()** to specify a new butterfly node and display a different butterfly graph given the same graph specification. For example, the following code fragment illustrates setting a new butterfly node, *newCenter*, after displaying the butterfly graph in the example above:

```
graph->displayButterfly( newCenter );
graph->doLayout();
```

After displaying a butterfly graph, you can return to displaying a normal tree graph by setting the layout style to XmGRAPH using the **VkGraph::setLayoutStyle()** function:

```
virtual void setLayoutStyle(char type)
```

For example, the following code fragment illustrates displaying the entire graph specified by *graph* after displaying the butterfly graphs above:

```
graph->setLayoutStyle( XmGRAPH );
graph->displayAll();
graph->doLayout();
```

Displaying a Graph Overview

As discussed in “Graph Overview” on page 258, by clicking the *Graph Overview* button in the graph command panel, a user can display an overview of all a graph’s visible nodes.

You can also display the overview window programmatically using **VkGraph::showOverview()**:

```
void showOverview()
```

Call **VkGraph::hideOverview()** to programmatically hide the overview window:

```
void hideOverview()
```

You can obtain a pointer to the overview window’s **VkWindow** object using **VkGraph::overviewWindow()**:

```
VkWindow *overviewWindow()
```

Graph Utility Functions

VkGraph provides the following utility functions:

- **VkGraph::setZoomOption()** sets the zoom value for the graph. Pass to this function the integer index corresponding to the index in the Zoom Menu of the magnification that you want. ("Zooming" on page 257 describes the Zoom Menu and its default values.)

  ```
  virtual void setZoomOption(int index)
  ```

- **VkGraph::sortAll()** sorts all nodes associated with the graph by calling **VkNode::sortChildren()** on all nodes. ("Node Utility Functions" on page 263 describes **VkNode::sortChildren()**.)

  ```
  void sortAll()
  ```

- **VkGraph::forAllNodesDo()** allows you to perform some action on all nodes registered with a graph. The type definition of VkGraphNodeProc is:

  ```
  typedef void (*VkGraphNodeProc) (VkNode *)
  ```

  The function you provide must be a static function that accepts a node as an arguments and has a void return value.

  ```
  virtual void forAllNodesDo(VkGraphNodeProc function)
  ```

- **VkGraph::makeNodeVisible()** ensures that a particular node is in the visible portion of the graph's window. If the node you specify is not currently visible, **makeNodeVisible()** scrolls the graph until the specified node appears in the visible portion of the window.

  ```
  void makeNodeVisible(VkNode *node)
  ```

- **VkGraph::saveToFile()** prompts the users for a file name and saves a PostScript® version of the graph to that file.

  ```
  void saveToFile()
  ```

- **VkGraph::setSize()** allows you to pre-allocate space in your graph's internal tables for the number of nodes you specify. If you know how many nodes you plan to add to your graph, calling **setSize()** before adding nodes to your graph can save time because the graph can allocate all memory needed in one operation instead of expanding the tables dynamically as you add nodes. Your graph can still allocate additional space if you actually add more nodes than you reserved space for using **setSize()**.

  ```
  void setSize(int entries)
  ```

Graph Access Functions

VkGraph provides the following access functions for obtaining values associated with the graph:

- **VkGraph::numNodes()** returns the number of nodes in the graph.

  ```
  int numNodes()
  ```

- **VkGraph::find()** returns the first **VkNode** object registered with the **VkGraph** object that has the given name.

  `VkNode *find(char *`*name*`)`

- **VkGraph::graphWidget()** returns the SgGraph widget instantiated by the **VkGraph** component. Not all the functionality of the SgGraph widget is encapsulated in the **VkGraph** class, and it is sometimes useful to set various resources directly on the graph widget.

  ```
  Widget graphWidget()
  ```

- **VkGraph::workArea()** returns the XmForm widget at the bottom of the **VkGraph** component, which contains the graph controls. You can use this area to add additional controls.

  ```
  Widget workArea()
  ```

- **VkGraph::twinsButton()** returns the Multiple Arcs button widget used to control whether sibling arcs are shown.

  ```
  Widget twinsButton()
  ```

- **VkGraph::relayButton()** returns the Realign button widget used to relay the graph.

  ```
  Widget relayButton()
  ```

- **VkGraph::reorientButton()** returns the Rotate button widget used to reorient the graph.

  ```
  Widget reorientButton()
  ```

Reusing a Graph Object

Occasionally, after displaying one graph, you might want to display an entirely different graph. The simplest method of accomplishing this is to create another **VkGraph** object for the new graph.

However, creating a new graph object entails the overhead of creating many new widgets and data structures. Sometimes it is simpler, faster, and more appropriate to re-use the existing graph object. For example, consider a window in which you are displaying a graph of C++ class hierarchies associated with a program. The window might contain controls that allow the user to select other programs to examine. If the user selects a new program to examine, the most convenient thing to do would be to keep the existing graph object but “clear it” of all existing information.

The **VkGraph::tearDownGraph()** function provides this ability:

```
virtual void tearDownGraph()
```

It tears down the graph by destroying all arc and node widgets and deleting all **VkNode** objects associated with the graph. This function is equivalent to deleting all **VkNode** objects associated with the graph, deleting the graph object, and creating a new graph object with the same name, but entails less overhead processing than if you were to explicitly perform these actions separately.

ViewKit Callbacks Associated with VkGraph

The **VkGraph** class declares two ViewKit member function callbacks.

VkGraph activates the *VkGraph::arcCreatedCallback* whenever the graph creates a SgArc widget to connect two nodes. The *arcCreatedCallback* callback includes as call data the newly created SgArc widget. See the SgArc(3x) reference pages for information on the SgArc widget.

VkGraph activates the *VkGraph::arcDestroyedCallback* whenever the graph destroys all arc widgets as a result of a call to **VkGraph::clearAll()** (see “Indicating Which Nodes to Display” on page 268). **VkGraph** activates the *arcDestroyedCallback* callback once for every arc destroyed, including as call data the SgArc widget destroyed. See the SgArc(3x) reference pages for information on the SgArc widget.

X Resources Associated with VkGraph

VkGraph sets several X resources that specify the labels of its popup menus. You can override these values in an app-defaults file if you want to provide your own labels. The resources and their default values are:

```
*graph*popupMenu*hideNode*labelString:              Hide Node
*graph*popupMenu*collapseSubgraph*labelString:      Collapse Subgraph
*graph*popupMenu*expandOneLevel*labelString:        Show Immediate Children
*graph*popupMenu*expandSubgraph*labelString:        Expand Subgraph
*graph*popupMenu*hideParents.labelString:           Hide Parents
*graph*popupMenu*expandParents.labelString:         Show Parents
*graph*popupMenu*selectedNodes.labelString:         Selected Nodes
*graph*popupMenu*hideSelectedNodes.labelString:     Hide
*graph*popupMenu*collapseSelectedNodes.labelString: Collapse
*graph*popupMenu*expandSelectedNodes.labelString:   Expand
```

Subclassing VkGraph

VkGraph provides much of the functionality that you should require for displaying and manipulating graphs. In most other cases, you can obtain a pointer to the SgGraph widget using the **graphWidget()** access function and operate directly on the widget.

However, sometimes you might want to perform additional processing when certain actions occur. In a case like this, you can create a subclass of **VkGraph**. **VkGraph** provides a number of virtual "hook" functions that you can override and implement additional functionality:

- **VkGraph::buildCmdPanel()** builds the command panel at the bottom of the graph. You can override this function to create your own custom command panel for your graph.

  ```
  virtual void buildCmdPanel(Widget parent)
  ```

- **VkGraph::buildZoomMenu()** builds the Zoom menu, the *Zoom Out* button, and the *Zoom In* button as part of the command panel. You can override this function to provide your own custom zoom controls for your graph.

  ```
  virtual void buildZoomMenu(Widget parent)
  ```

- **VkGraph::addMenuItems()** allows you to modify the Node popup menu described in “Hiding and Displaying Nodes” on page 260. You can override this function and use the various functions provided by the **VkMenu** class to add new menu item or delete default menu items. “The ViewKit Menu Base Class” on page 127 describes the functions provided by **VkMenu**.

  `virtual void addMenuItems(VkPopupMenu *`*menu*`)`

- **VkGraph::popupMenu()** posts the Node popup menu described in “Hiding and Displaying Nodes.” The function receives two arguments: a pointer to the node on which the user clicked the right mouse button, and the X ButtonPress event. By default, the function: 1) activates and deactivates menu items to reflect the valid options for the node; 2) sets the label of the popup menu to be the same as the label of the node; and 3) calls the popup menu’s **show()** function, passing *event* as an argument.

  You can override this function if you want to change its behavior or support any additional menu items that you added by overriding **addMenuItems()**.

  `virtual void popupMenu(VkNode *`*node*`, XEvent *`*event*`)`

- **VkGraph::addDesktopMenuItems()** allows you to modify the Selected Nodes popup menu described in “Edit Mode Operations” on page 260. You can override this function and use the various functions provided by the **VkMenu** class to add new menu items or delete default menu items. “The ViewKit Menu Base Class” describes the functions provided by **VkMenu**.

  `virtual void addDesktopMenuItems(VkPopupMenu *`*menu*`)`

- **VkGraph::twinsVisibleHook()** is called when the user toggles the *Multiple Arcs* or “twins” button. The new state of the twins buttons is passed as an argument to this function. By default, the function is empty. You can override this function to perform additional operations when the graph changes its display mode.

  `virtual void twinsVisibleHook(Boolean `*state*`)`

Chapter 10

Miscellaneous ViewKit Display Classes

This chapter contains descriptions of miscellaneous ViewKit classes that you use primarily to display information or to manage display items. Figure 10-1 shows the inheritance graph for these classes.

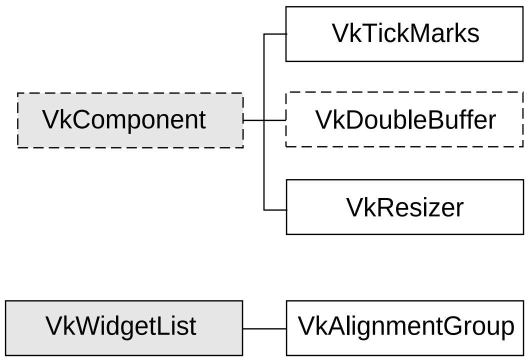


Figure 10-1 The Inheritance Graph for the Miscellaneous ViewKit Display Classes

ViewKit Support for Double-Buffered Graphics

VkDoubleBuffer is an abstract class that provides support for components that need to display double-buffered graphics.

Note: **VkDoubleBuffer** provides software double-buffering only; it does not use the hardware double-buffering available on many Silicon Graphics workstations. As a result, you might notice some flickering in your **VkDoubleBuffer** animations.

You must create a separate subclass of **VkDoubleBuffer** for each double-buffered display component in your application. In each subclass, you include the Xlib calls to create the text or graphics that the component displays. You do not have to worry about handling Expose events or resize requests as **VkDoubleBuffer** handles these automatically.

The public interface to **VkDoubleBuffer** consists simply of a function that your application calls whenever it needs to update the component's display. For example, to drive an animation, you could set a timer to update a component at a desired interval.

Double Buffer Constructor and Destructor

The **VkDoubleBuffer** constructor accepts the standard ViewKit component constructor arguments, a component name and a parent widget:

```
VkDoubleBuffer(const char *name, Widget parent)
```

The constructor creates the various widgets and Pixmaps used by the component and installs callbacks to handle Expose events and resize requests. In your subclass constructor, you can initialize any graphics contexts and other data that your component requires.

The **VkDoubleBuffer** destructor frees the widgets and Pixmaps allocated by the **VkDoubleBuffer** constructor:

```
~VkDoubleBuffer()
```

In your subclass destructor you should free any graphics contexts and other data allocated by your component.

Drawing in the Double Buffer Component

The **VkDoubleBuffer** class calls your component's **draw()** function when your component needs to draw a new frame:

```
virtual void draw()
```

draw() is declared by **VkDoubleBuffer** as a pure virtual function, and it is the only function you must override when creating a derived class of **VkDoubleBuffer**. The **draw()** function should use Xlib calls to display text or graphics by drawing to the *_canvas* data member:

```
Pixmap _canvas
```

The derived class always draws to the back buffer, although the derived class does not need to be aware of this. The **VkDoubleBuffer** class copies the contents of this Pixmap to the front buffer as needed.

Switching Buffers in the Double Buffer Component

VkDoubleBuffer::update() is the public member function that the application calls to update the component's display:

```
virtual void update()
```

update() calls your component's **draw()** function to obtain a new frame. Then it swaps buffers, and if the component is currently displayed, updates the screen with the contents of the front buffer. Finally, **update()** clears the back buffer by filling it with the component's background color.

Handling Double Buffer Component Resize Requests

VkDoubleBuffer automatically handles window resize requests, resizing the front and back buffers and filling them with the component's background color. If you need to perform additional operations in your derived class, you can override the virtual function **VkDoubleBuffer::resize()**:

```
virtual void resize()
```

VkDoubleBuffer calls **resize()** after resizing and reinitializing the buffers. The new height and width of the drawing area are contained in the *_width* and *_height* data members:

```
Dimension _width
Dimension _height
```

Tick Marks for Scales

The **VkTickMarks** class, derived from **VkComponent**, displays a vertical set of tick marks. Most frequently, you would use this component next to a vertical IRIS IM XmScale(3Xm) widget. By default, a **VkTickMarks** component right-justifies its tick marks and displays its labels to the left, which is appropriate if you display the component to the left of a scale. You can also configure a **VkTickMarks** component to left-justify its tick marks and display its labels to the right, which is appropriate if you display the component to the right of a scale. Figure 10-2 shows an example of each version of the tick marks.

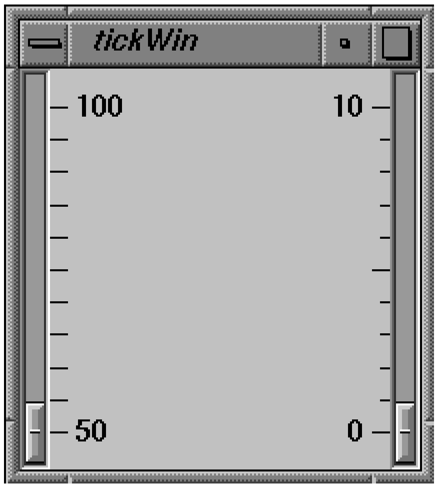


Figure 10-2 Examples of the **VkTickMarks** Component

Tick Marks Component Constructor

The **VkTickMarks** constructor accepts five arguments:

```
VkTickMarks(char* name, Widget parent, Boolean labelsToLeft = TRUE,
            Boolean noLabels = FALSE, Boolean centerLabels = FALSE)
```

The first two arguments are the standard ViewKit component constructor arguments, a component name and a parent widget. If *labelsToLeft* is TRUE, the tick marks are right-justified and the labels appear to the left; if *labelsToLeft* is FALSE, the tick marks are left-justified and the labels appear to the right. If you set *noLabels* to TRUE, the **VkTickMarks** component does not display any labels. If you set *centerLabels* to TRUE, the **VkTickMarks** component centers the labels. This is useful if you want to center a **VkTickMarks** object between two XmScale widgets.

Configuring the Tick Marks

You can set the scale of the tick marks with the **VkTickMarks::setScale()** function:

```
void setScale(int min, int max,
              int majorInterval, int minorInterval)
```

min and *max* specify the minimum and maximum values for the tick mark component. If you set the **VkTickMarks** component to display labels, it displays these minimum and maximum values next to the bottom and top tick marks respectively.

majorInterval and *minorInterval* specify the tick mark spacing. You can specify the number of units (not pixels) between each major and minor tick mark.

For example, the following sets the minimum value of the *ticks* **VkTickMarks** object to 0, the maximum to 1000, the major interval to 100, and the minor interval to 50:

```
ticks->setScale( 0, 1000, 100, 50 );
```

Figure 10-3 shows the resulting display of the **VkTickMarks** object.

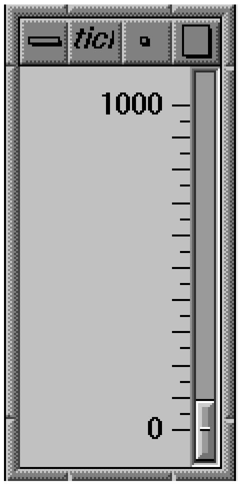


Figure 10-3 Example of Setting Tick Mark Scale and Spacing

If you do not use **setScale()** to set the scale of the tick marks, **VkTickMarks** uses the values of the resources “minimum,” “maximum,” “majorInterval,” and “minorInterval” to set the respective scale values.

You can add additional labels to the scale with **VkTickMarks::addLabel()**:

```
void addLabel(int value)
```

The **VkTickMarks** object displays a label at the value you indicate. You can call **addLabel()** multiple times to add multiple labels.

The **VkTickMarks::setMargin()** function controls the **VkTickMarks** margins:

```
void setMargin(int marginTop, int marginBottom);
```

setMargin() allows you to specify the spacing between the top of the **VkTickMarks** component and the first tick mark, and the bottom of the component and the last tick mark. The default settings are designed for use next to an XmScale widget: the first and last tick marks align horizontally with the mark in the middle of the scale's slider.

X Resources Associated with the Tick Marks Component

The **VkTickMarks** class provides several X resources that determine display characteristics of the component:

minimum	The initial minimum value (default value 0)
maximum	The initial maximum value (default value 10)
majorInterval	The major tick interval (default value 5)
minorInterval	The minor tick interval (default value 1)
majorSize	The width in pixels of the major tick marks (default value 10)
minorSize	The width in pixels of the minor tick mark width (default value 6)
labelSpacing	The spacing in pixels between tick marks and labels (default value 3)
marginTop	The margin in pixels between the top of the component and the top tick mark (default value 19)
marginBottom	The margin in pixels between the bottom of the component and the bottom tick mark (default value 19)
lineThickness	The thickness in pixels of the tick marks thickness (default value 1)
label.foreground	The foreground color used for labels and tick marks
label.background	The background color used for labels and tick marks
label.fontList	The font used for labels

Management Classes for Controlling Component and Widget Display Characteristics

ViewKit provides some management classes that control the display of components and widgets. These classes function as attachments: you attach them to one or more existing widgets or components. Then you can use the management class to control some aspect of displaying the widgets and components to which the class is attached.

ViewKit Support for Aligning Widgets

The **VkAlignmentGroup** class provides support for aligning collections of widgets with each other in various ways. **VkAlignmentGroup** is derived from the convenience class **VkWidgetList**. Consult the **VkWidgetList**(3x) reference page for more information on that class.

To use the **VkAlignmentGroup** class, you create a **VkAlignmentGroup** object, add widgets or components to the group, and then call one of the alignment functions provided by **VkAlignmentGroup**.

The Alignment Group Constructor and Destructor

The **VkAlignmentGroup** constructor does not take any arguments:

```
VkAlignmentGroup()
```

VkAlignmentGroup objects do not require names because they are not components; ViewKit uses names to uniquely identify the widget trees of components, and the **VkAlignmentGroup** class does not create any widgets.

The **VkAlignmentGroup** destructor destroys only the **VkAlignmentGroup** object. If you have widgets managed by the object, they are unaffected by the **VkAlignmentGroup** destructor.

Adding Widgets and Components to an Alignment Group

Use the **add()** function to add widgets or components to a **VkAlignmentGroup** object:

```
virtual void add(Widget w)
virtual void add(VkComponent *obj)
virtual void add(VkOptionMenu *menu)
```

If you provide a widget, **add()** adds that widget to the alignment group. If you provide a pointer to a component, **add()** adds the component's base widget to the alignment group. If you provide a pointer to a **VkOptionMenu** object, **add()** adds all menu items individually to the **VkAlignmentGroup** object rather than adding the **VkOptionMenu** object as an entity.

Removing Widgets and Components from an Alignment Group

You can remove widgets or components from a **VkAlignmentGroup** object with the **remove()** function inherited from **VkWidgetList**:

```
virtual void remove(Widget w)
virtual void remove(VkComponent *obj)
```

Provide the widget ID or component pointer that you used to add the widget or component to the alignment group.

Aligning Widgets and Components in an Alignment Group

To align or distribute the elements in a **VkAlignmentGroup** object, call one of the following functions (all of which take no arguments and have a void return type):

alignLeft()	Aligns the left edges of all widgets by repositioning all widgets so that the left side of each widget is moved to the right-most left edge of any widget in the group.
alignRight()	Aligns the right edges of all widgets by repositioning all widgets so that the right side of each widget is moved to the right-most position occupied by any widget in the group.
alignTop()	Aligns the top edges of all widgets by repositioning all widgets so that the top of each widget is moved to the bottom-most top edge of any widget in the group.
alignBottom()	Aligns the bottom edges of all widgets by repositioning all widgets so that the bottom of each widget is moved to the bottom-most position occupied by any widget in the group.
alignWidth()	Resizes all widgets to the width of the largest widget in the group.
alignHeight()	Resizes all widgets to the height of the largest widget in the group.
makeNormal()	Returns all widgets to their desired widths and heights.

distributeVertical()
Repositions all widgets so that they are positioned evenly in the vertical direction, according to the spacing between widgets, between the position of the first and last widgets in the group.

distributeHorizontal()
Repositions all widgets so that they are positioned evenly in the horizontal direction, according to the spacing between widgets, between the position of the first and last widgets in the group.

Alignment Group Access Functions

VkAlignmentGroup provides the following access functions:

- **VkAlignmentGroup::width()** returns the maximum width of all widgets in the group. This value is not set until after you have called **alignWidth()**.

  ```
  Dimension width()
  ```

- **VkAlignmentGroup::height()** returns the maximum height of all widgets in the group. This value is not set until after you have called **alignHeight()**.

  ```
  Dimension height()
  ```

- **VkAlignmentGroup::x()** returns the minimum *x* position of all widgets in the group. This value is not set until after you have called either **alignLeft()** or **alignRight()**.

  ```
  Position x()
  ```

- **VkAlignmentGroup::y()** returns the minimum *y* position of all widgets in the group. This value is not set until after you have called either **alignTop()** or **alignBottom()**.

  ```
  Position y()
  ```

VkAlignmentGroup also inherits all of the access and utility functions provided by **VkWidgetList**. Consult the **VkWidgetList**(3x) reference page for more information on that class.

ViewKit Support for Resizing and Moving Widgets

The **VkResizer** class provides controls for moving and resizing an existing widget. Figure 10-4 shows a simple example of a push button with a **VkResizer** attachment.

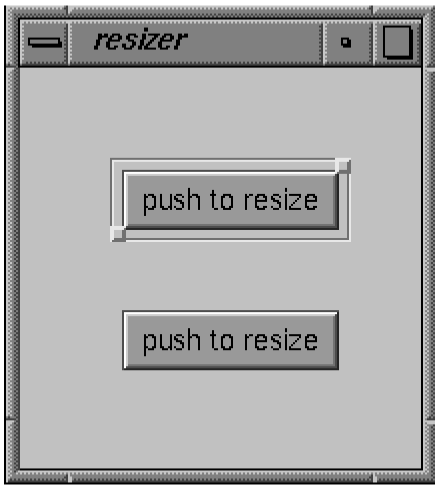


Figure 10-4 A Widget With a **VkResizer** Attachment

If you use the left mouse button to click either of the square handles provided by the **VkResizer** object, you can drag the handle to a new location. When you release the handle, the **VkResizer** object resizes the widget to which it is attached so that the widget matches the new size of the **VkResizer** object. Figure 10-5 shows an example of resizing the push button shown in Figure 10-4.

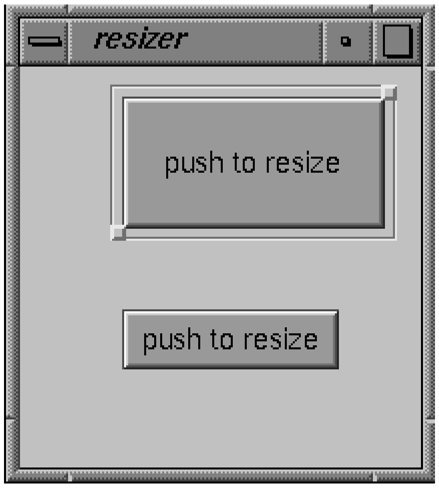


Figure 10-5 Effect of Resizing a Widget With a **VkResizer** Attachment

If you use the middle mouse button to click either of the square handles provided by the **VkResizer** object, you can drag the entire widget to a new location. When you release the handle, the **VkResizer** object moves the widget to which it is attached to the new location of the **VkResizer** object. Figure 10-6 shows an example of moving the push button shown in Figure 10-5.

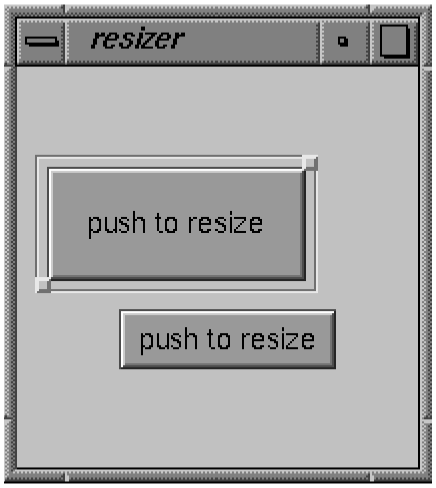


Figure 10-6 Effect of Moving a Widget With a **VkResizer** Attachment

To use the **VkResizer** class, you create a **VkResizer** object, associate an existing widget with the object, and then display the resizer's geometry controls.

The Resizer Constructor and Destructor

The **VkResizer** constructor accepts two Boolean arguments:

```
VkResizer(Boolean autoAdjust = FALSE, Boolean liveResize = FALSE)
```

autoAdjust controls whether the **VkResizer** object automatically tracks outside geometry changes of its attached widget. If you set this value to TRUE, the **VkResizer** object automatically adjusts its geometry controls whenever its attached widget changes geometry. If you set this value to FALSE, you must call the **VkResizer::adjustGeometry()** function whenever you want the **VkResizer** object to adjust its geometry controls to the geometry of its attached widget. The default value of this argument is FALSE.

liveResize controls whether the widget itself or a rectangle representing the widget area is displayed during geometry changes. Setting the second parameter to TRUE causes intermediate geometry changes in the attached widget, which may affect performance. The default value is FALSE.

VkResizer objects do not require names because they are not components; ViewKit uses names to uniquely identify the widget trees of components, and the **VkResizer** class does not create any widgets.

The **VkResizer** destructor destroys only the **VkResizer** object. If you have a widget attached to the object, it is unaffected by the **VkResizer** destructor.

Attaching and Detaching a Resizer Object to and from a Widget

Once you have created a **VkResizer** object, use the **VkResizer::attach()** function to attach it to an existing widget:

```
void attach(Widget w)
```

You can also attach a **VkResizer** object to a component by attaching it to the component's base widget. For example, if *resizer* is a **VkResizer** object and *obj* is a component, you can attach the resizer to the component as follows:

```
resizer->attach( obj->baseWidget() );
```

If the **VkResizer** object is already attached to a widget, it detaches from the old widget before attaching to the new one. You can use the **VkResizer::detach()** function to detach a **VkResizer** object from a widget without immediately attaching it to another:

```
void detach()
```

Displaying the Resizer Object's Geometry Controls

After attaching a **VkResizer** object to a widget, you must call the **VkResizer** object's **VkResizer::show()** function to display its geometry controls:

```
void show()
```

You can hide the geometry controls by calling the **VkResizer** object's **VkResizer::hide()** function:

```
void hide()
```

The **VkResizer::shown()** function returns a Boolean value indicating whether the **VkResizer** object is visible and displaying its geometry controls:

```
Boolean shown()
```

Resizer Utility Functions

You can configure the **VkResizer** object's geometry manipulations with the **VkResizer::setIncrements()** function:

```
void setIncrements(int resizeWidth, int resizeHeight,
                   int moveX, int moveY)
```

setIncrements() accepts four integer arguments. The first two arguments specify the resize increments in the horizontal and vertical dimension, respectively. The last two arguments specify the move increments in the horizontal and vertical dimension, respectively. Setting an increment to zero prohibits resizing or moving in that dimension.

ViewKit Callbacks Associated with the Resizer

The **VkResizer** class also provides a ViewKit member function callback named *VkResizer::stateChangedCallback*:

```
static const char *const stateChangedCallback
```

This callback informs the application when **VkResizer** has modified the geometry of its attached widget. The callback supplies as call data a value of the enumerated type VkResizerReason (defined in <*Vk/VkResizer.h*>). The value can be any of VR_resizing, VR_moving, VR_resized, or VR_moved. VR_resizing and VR_moving indicate that resizing or moving are in progress, and are sent repeatedly as the user adjusts the geometry. VR_resized and VR_moved indicate that the resizing or moving is complete, and are sent when the user releases the **VkResizer** geometry controls.

Chapter 11

Miscellaneous ViewKit Data Input Classes

This chapter contains descriptions of miscellaneous ViewKit classes that you would use primarily for data input. Figure 11-1 shows the inheritance graph for these classes.

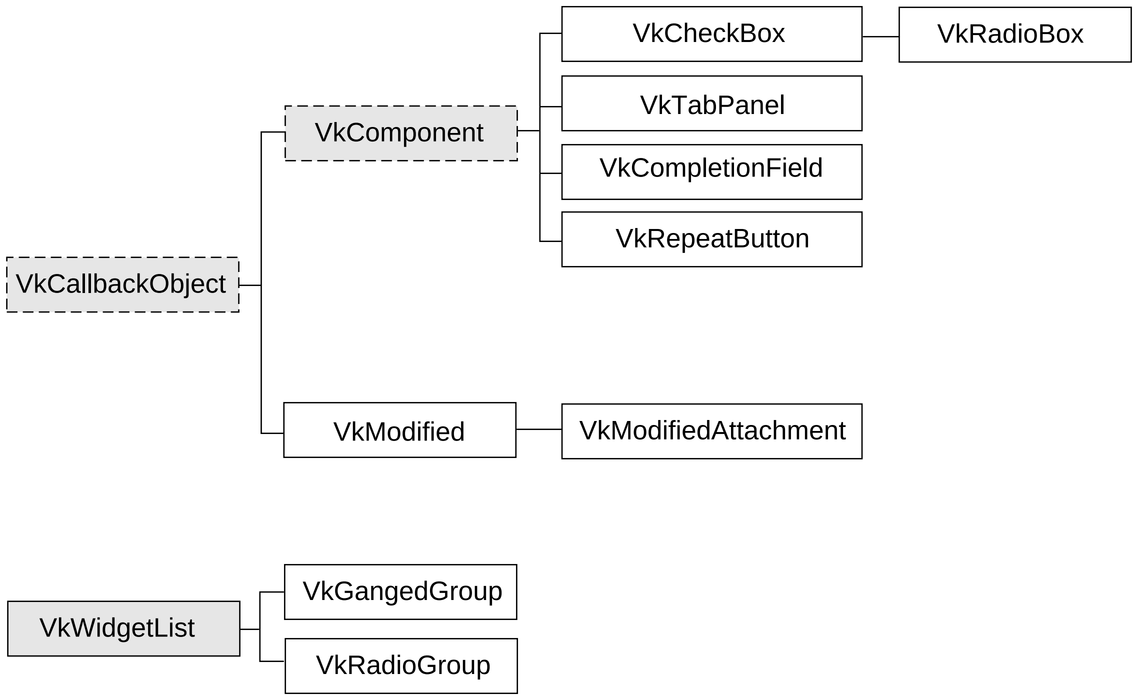


Figure 11-1 The Inheritance Graph for the Miscellaneous ViewKit Input Classes

The Check Box Component

The **VkCheckBox** class, derived from **VkComponent**, provides a simple method for creating check boxes. Instantiating the component creates an empty, labeled component to which you can add individual toggle buttons. **VkCheckBox** provides a variety of methods for determining when the user changes the state of a toggle; you can use the method most convenient for your applications. You can also programmatically change the values of the toggles.

Creating a Check Box

The **VkCheckBox** constructor accepts the standard ViewKit component name and parent widget arguments:

```
VkCheckBox(const char *name, Widget parent)
```

The constructor creates an empty, labeled component.

Adding Toggles to the Check Box

You add toggles to the check box using the **VkCheckBox::addItem()** function:

```
Widget addItem(char *name, Boolean state = FALSE,
               XtCallbackProc proc = NULL,
               XtPointer clientData = NULL)
```

name is the name of the toggle item. You can specify its initial state by providing a *state* argument; TRUE sets the toggle and FALSE clears it.

You can also provide an Xt-style callback function, *proc*, that **VkCheckBox** activates whenever the user changes the value of the toggle; and *clientData*, which **VkCheckBox** passes as client data to the callback function. Following ViewKit conventions as described in “Using Xt Callbacks with Components” on page 21, if you provide a callback function, you should pass the *this* pointer as client data so that the callback functions can retrieve the pointer, cast it to the expected component type, and call a corresponding member function. “Using Xt-Style Callbacks to Handle Changes in Check Box Toggle Values” on page 300 further discusses how to use the callback function.

Setting Check Box and Toggle Labels

The **VkCheckBox** component creates a LabelGadget named "label" to display a label. Each toggle in the check box is implemented as a ToggleButtonGadget. The name of the gadget is the *name* string that you provide to **addItem()** when you add the toggle.

Set the XmNlabelString resource of the check box label and its toggles to set their labels:

- Use the **VkComponent::setDefaultResources()** function to provide default resource values as described in "Setting Default Resource Values for a Component" on page 30.
- Set resource values in an external app-defaults resource file. Any values you provide in an external file will override values that you set using the **VkComponent::setDefaultResources()** function. This is useful when your application must support multiple languages; you can provide a separate resource file for each language supported.
- Set the resource values directly using the **XtSetValues()** function. Values you set using this method override any values set using either of the above two methods. You should avoid using this method as it "hard codes" the resource values into the code, making them more difficult to change.

For example, consider a simple window that contains only a check box with four toggles, as shown in Figure 11-2.

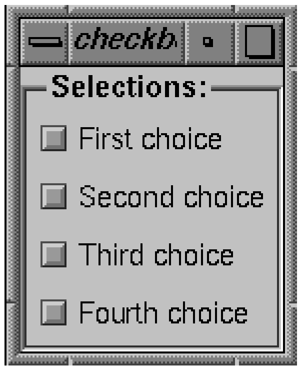


Figure 11-2 Example Check Box

Example 11-1 shows the code used to create this check box.

Example 11-1 Code to Create Example Check Box

```
#include <Vk/VkApp.h>
#include <Vk/VkSimpleWindow.h>
#include <Vk/VkCheckBox.h>

class CheckBoxWindow: public VkSimpleWindow {
  protected:
    virtual Widget setUpInterface ( Widget parent );
    static String _defaultResources[];

  public:
    CheckBoxWindow ( const char *name ) : VkSimpleWindow ( name ) { }
    ~CheckBoxWindow();
    virtual const char* className();
};

CheckBoxWindow:: ~CheckBoxWindow()
{ }

const char* CheckBoxWindow::className() { return "CheckBoxWindow"; }

String CheckBoxWindow::_defaultResources[] = {
  "*check*label.labelString: Selections:",
  "*check*one*labelString: First choice",
  "*check*two*labelString: Second choice",
  "*check*three*labelString: Third choice",
  "*check*four*labelString: Fourth choice",
  NULL
};

Widget CheckBoxWindow::setUpInterface ( Widget parent )
{
  setDefaultResources(parent, _defaultResources);

  VkCheckBox *cb = new VkCheckBox("check", parent);
  cb->addItem("one");
  cb->addItem("two");
  cb->addItem("three");
  cb->addItem("four");
  cb->show();
  return cb->baseWidget();
}

void main ( int argc, char **argv )
{
VkApp *cbApp = new VkApp("checkBoxApp", &argc, argv);
CheckBoxWindow *cbWin = new CheckBoxWindow("checkbox");

cbWin->show();
cbApp->run();}
```

Setting and Getting Check Box Toggle Values

After creation, you can programmatically set the state of any toggle with the **VkCheckBox::setValue()** function:

`void setValue(int` *index*`, Boolean` *newValue*`)`

index is the position of the toggle in the check box; the first toggle in the check box has an index of 0. *newValue* is the new state for the toggle; TRUE sets the toggle and FALSE clears it.

Note: Setting a toggle using **setValue()** activates the toggle's valueChanged callback. This in turn activates all of the **VkCheckBox** object's methods for detecting changes in toggle values as described in "Recognizing Changes in Check Box Toggle Values" on page 300.

You can set the values of multiple toggles using the **VkCheckBox::setValues()** function:

`void setValues(Boolean *`*values*`, int` *numValues*`)`

The Boolean array *values* specifies the new values for a group of toggles in the check box beginning with the first toggle. *numValues* specifies the number of values the *values* array contains.

Note: Setting toggles using **setValues()** activates each toggle's valueChanged callback. This, in turn, activates all of the **VkCheckBox** object's methods for detecting changes in toggle values, as described in "Recognizing Changes in Check Box Toggle Values," once for each toggle changed.

You can retrieve the value of a specific toggle with the **VkCheckBox::getValue()** function:

```
int getValue(int index)
```

index is the position of the toggle in the check box; the first toggle in the check box has an index of 0. The function returns TRUE if the toggle is set and FALSE if the toggle is not set.

Recognizing Changes in Check Box Toggle Values

VkCheckBox provides three different methods that you can use to determine when the user changes the value of a toggle: Xt-style callbacks, ViewKit callbacks, and subclassing. You can use whichever method is most convenient.

Using Xt-Style Callbacks to Handle Changes in Check Box Toggle Values

The first method of determining when the user changes a toggle value is to register an Xt-style callback for each toggle button. When you create a toggle with the **addItem()** function, you can optionally specify a callback function and client data. When the value of the toggle changes, the callback function is called with the client data you provided, and a pointer to a XmToggleButtonCallbackStruct structure as call data.

For example, the following adds a toggle named “lineNumbers” to the *parametersBox* check box and registers a callback function:

```
MyComponent::MyComponent(const char *name, Widget parent) : VkComponent (name)
{
  // ...
  parametersBox->addItem("lineNumbers", FALSE,
                         &MyComponent::toggleLineNumbersCallback(),
                         (XtPointer) this );
  // ...
}
```

MyComponent::toggleLineNumbersCallback(), which must be declared as a static member function of the class **MyComponent**, is registered as a callback function for this toggle, and the *this* pointer is used as the client data. The definition of **toggleLineNumbersCallback()** could look like this:

```
void MyComponent::toggleLineNumbersCallback(Widget,
                                          XtPointer clientData,
                                          XtPointer callData )
{
  MyComponent *obj = (MyComponent) clientData;
  XmToggleButtonCallbackStruct *cb =
                               (XmToggleButtonCallbackStruct) callData;

  // Call MyComponent::toggleLineNumbers(), a regular member function to either
  // display or hide line numbers based on the value of the toggle.

  obj->toggleLineNumbers(cb->set);
}
```

Using ViewKit Callbacks to Handle Changes in Check Box Toggle Values

The second method of determining when the user changes a toggle value is to use a ViewKit callback. The **VkCheckBox** component provides the *VkCheckBox::itemChanged* callback. Any ViewKit component can register a member function to be called when the user changes a check box toggle. The **VkCheckBox** object provides the integer index of the toggle as client data to the callback functions.

Note: The *itemChanged* callback is activated whenever the user changes any of the toggles; you cannot register a ViewKit callback for an individual toggle.

For example, the following line registers the member function **MyComponent::parameterChanged()** as a ViewKit callback function to be called whenever the user changes a toggle in the *parametersBox* check box:

```
MyComponent::MyComponent(const char *name, Widget parent) : VkComponent (name)
{
  // ...
  parametersBox->addCallback(VkCheckBox::itemChanged, this,
                            (VkCallbackMethod) &MyComponent::parameterChanged );
  // ...
}
```

Note that in this example, no client data is provided.

The definition of **parameterChanged()** could look like this:

```
void MyComponent::parameterChanged(VkComponent *obj, void *,
                                   void *callData )
{
  VkCheckBox *checkBox = (VkCheckBox) obj;
  int index = (int) callData;
  switch (index) {
    // ...

    // Assume that the constant LINE_NUMBER_INDEX is set to the index of
    // the "lineNumber" toggle. If the "lineNumber" toggle value changed,
    // Call MyComponent::toggleLineNumbers(), a regular member function to
    // either display or hide line numbers based on the value of the toggle
    case LINE_NUMBER_INDEX:
      toggleLineNumbers( checkBox->getValue(index) );

    // ...
  }
}
```

Using SubClassing to Handle Changes in Check Box Toggle Values

The third method of determining when the user changes a toggle value is to create a subclass of **VkCheckBox**. Whenever the user changes a toggle, **VkCheckBox** calls the virtual function **VkCheckBox::valueChanged()**:

`virtual void valueChanged(int` *index*`, Boolean` *newValue*`)`

index is the index of the item that changed and *newValue* is the current (new) value of that item. By default, **valueChanged()** is empty. You can override its definition in a subclass and perform whatever processing you need.

Derived classes have access to the following protected data members of the **VkCheckBox** class:

- An instance of the ViewKit WidgetList(3x) class that contains all toggle buttons added to the check box

  `VkWidgetList *`*_widgetList*

- The RowColumn widget that contains the toggle buttons

  `Widget `*_rc*

- The label widget for the check box

  `Widget `*_label*

The Radio Check Box Component

The **VkRadioBox** class provides a simple method for creating radio check boxes (that is, check boxes in which only one toggle at a time can be selected). **VkRadioBox** is a subclass of **VkCheckBox**. The only difference between the two classes is that **VkRadioBox** enforces radio behavior on the toggles it contains.

VkRadioBox provides all of the same functions and data members as **VkCheckBox** does. You use the **VkRadioBox** class in the same way that you do the **VkCheckBox** class.

For example, consider a simple window that contains only a check box with four toggles as shown in Figure 11-3.

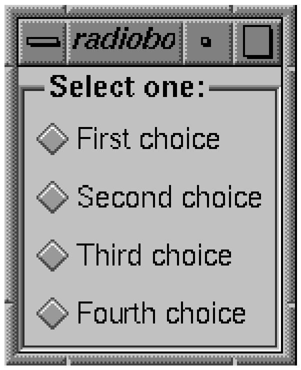


Figure 11-3 Example Radio Box

Example 11-2 contains the code used to create this check box.

Example 11-2 Code to Create Example Radio Box

```
#include <Vk/VkApp.h>
#include <Vk/VkSimpleWindow.h>
#include <Vk/VkRadioBox.h>

class RadioBoxWindow: public VkSimpleWindow {

  protected:
    virtual Widget setUpInterface ( Widget parent );
    static String _defaultResources[];

  public:
    RadioBoxWindow ( const char *name ) : VkSimpleWindow ( name ) { }
    ~RadioBoxWindow();
    virtual const char* className();
};
RadioBoxWindow:: ~RadioBoxWindow()
{ }

const char* RadioBoxWindow::className() { return "RadioBoxWindow"; }

String RadioBoxWindow::_defaultResources[] = {
  "*radio*label.labelString: Select one:",
  "*radio*one*labelString: First choice",
  "*radio*two*labelString: Second choice",
  "*radio*three*labelString: Third choice",
  "*radio*four*labelString: Fourth choice",
 NULL
};

Widget RadioBoxWindow::setUpInterface ( Widget parent )
{
  setDefaultResources(parent, _defaultResources);

  VkRadioBox *rb = new VkRadioBox("radio", parent);
  rb->addItem("one");
  rb->addItem("two");
  rb->addItem("three");
  rb->addItem("four");
  rb->show();

  return rb->baseWidget();
}

void main ( int argc, char **argv )
{
  VkApp *rbApp = new VkApp("radioBoxApp", &argc, argv);
  RadioBoxWindow *rbWin = new RadioBoxWindow("radiobox");

  rbWin->show();
  rbApp->run();
}
```

The Tab Panel Component

The **VkTabPanel** class, derived from **VkComponent**, displays a row or column of overlaid tabs. A tab can contain text, a pixmap, or both. The user can click a tab with the left mouse button to select it. One tab is always selected, and appears on top of all the others. When the user selects a tab, **VkTabPanel** activates a ViewKit member function callback indicating which tab the user selected. You can register callback functions to perform actions based on the tabs selected.

Figure 11-4 shows an example of a horizontal **VkTabPanel** component.

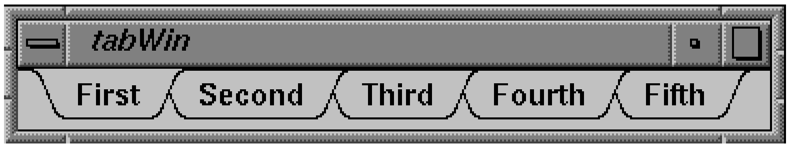


Figure 11-4 Example of a Horizontal **VkTabPanel** Component

Figure 11-5 shows an example of a vertical **VkTabPanel** component.

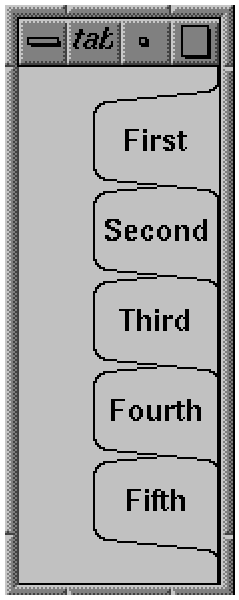


Figure 11-5 Example of a Vertical **VkTabPanel** Component

When the tabs do not fit within the provided space, the **VkTabPanel** object "collapses" tabs on the left and right ends of the component (or top and bottom if the **VkTabPanel** object is vertical). The example in Figure 11-6 shows these collapsed tabs.

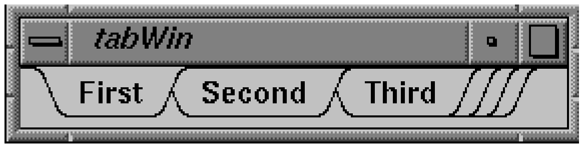


Figure 11-6 Example of Collapsed Tabs in a **VkTabPanel** Component

The user can click the collapsed tabs with either the left or right mouse button to display a popup menu listing all the tabs, as shown in Figure 11-7. The user can then select a tab by choosing the corresponding menu item.

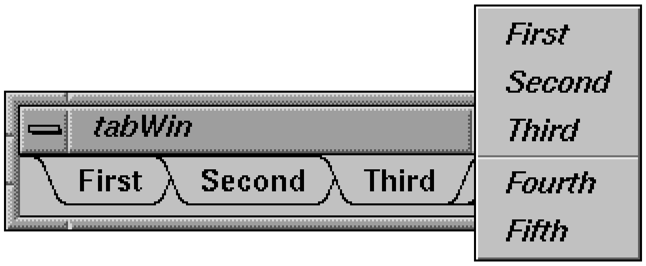


Figure 11-7 Using the Popup Menu to Select a Collapsed Tab in a **VkTabPanel** Component

The **VkTabPanel** class also provides work areas implemented as IRIS IM Form widgets to the left and right of the tab display (or top and bottom if the **VkTabPanel** object is vertical). By default, these work areas are empty. You can access these work area widgets and implement additional displays or controls if you desire. "Tab Panel Access Functions" on page 312 describes the work area access functions.

The Tab Panel Constructor

The **VkTabPanel** constructor initializes the tab panel and allocates all resources required by the component:

```
VkTabPanel(char* name, Widget parent,
           Boolean horizOrientation = TRUE, int tabHeight = 0)
```

name and *parent* are the standard component name and parent widget arguments.

The optional *horizOrientation* argument determines the orientation of the tab panel. If *horizOrientation* is TRUE, the tab panel is horizontal; if it is FALSE, the tab panel is vertical.

The optional *tabHeight* argument determines the height of the tab display area. The default value, 0, indicates that tab height is determined by the default label height. If you plan to include pixmaps in your tabs, you should specify a height sufficient to contain your largest pixmap. You can also set the tab height by setting the value of the **VkTabPanel** object's "tabHeight" resource. For example, to set the tab height of the **VkTabPanel** object *tabs* to 30, you could include the following line in an app-default file:

```
*tabs*tabHeight:    30
```

Note: In most cases when you display a vertical tab panel, you must explicitly set the height of the tab display area. As described above, the default tab display area height is determined by the tab label's font height rather than the width of the label. As a result, the tabs might not be large enough to display all of the label text.

Adding Tabs to a Tab Panel

Once you have created a tab panel, you can add a tab to it using **VkTabPanel::addTab()**:

```
int addTab(char *label, void *clientData, Boolean sorted = FALSE)
```

label specifies the label displayed by the tab. You should use a distinct label for each tab. **addTab()** first treats this argument as a resource name which is looked up relative to the tab panel's name. If the resource exists, its value is used as the tab label. If no resource is found, or if the string contains spaces or newline characters, the string itself is used as the tab label.

When the user selects this tab, the **VkTabPanel** object activates either *VkTabPanel::tabSelectCallback* or *VkTabPanel::tabPopupCallback* (depending on how the user selected the tab). If you provide a pointer to some data as the *clientData* argument to **addTab()**, the tab panel includes that data as part of the VkTabCallbackStruct returned as call data by the callbacks. "Responding to Tab Selection" on page 311 describes in depth these callbacks and how to use them.

The *sorted* flag determines where the new tab is added in relation to existing tabs. If *sorted* is FALSE, **addTab()** adds the tab after all existing tabs; if *sorted* is TRUE, **addTab()** inserts the tab before the first tab whose label alphabetically succeeds the new tab's label.

Note: **addTab()** compares the labels actually displayed in the tabs, so if you use resources to specify tab labels, **addTab()** correctly uses the labels specified by the resource values.

The return value of **addTab()** is the position of the newly-added tab in the tab panel. Tabs are numbered sequentially, with 0 representing the left-most tab in a horizontal tab panel or the top-most tab in a vertical tab panel.

New tabs initially have a NULL pixmap. If you want to add a pixmap to a label, see "Adding a Pixmap to a Tab" on page 310.

If the new tab is the first tab in the group, **addTab()** automatically selects the tab by calling **VkTabPanel::selectTab()**. Note that **selectTab()** activates *VkTabPanel::tabSelectCallback*, so if you register a callback function before adding a tab, you activate that callback function when you add your first tab. See "Responding to Tab Selection" on page 311 for more information on **selectTab()** and *VkTabPanel::tabSelectCallback*.

You can add more than one tab at a time using **VkTabPanel::addTabs()**:

```
void addTabs(char **labels, void **clientDatas, int numTabs,
             Boolean sorted = FALSE)
```

labels is an array of tab label strings. As with **addTab()**, these label strings are first treated as resource names which are looked up relative to the tab panel's name. If the resources exist, their values are used as the tab labels. If a particular resource name is not found, or if the string contains spaces or newline characters, the label string itself is used as the tab label. *clientDatas* is an array of client data; the data for a particular tab is included as part of the VkTabCallbackStruct returned as call data by the selection callbacks. *numLabels* specifies the number of tabs to be added by **addTabs()**. *sorted* determines whether or not the tabs are sorted as **addTabs()** adds them.

Removing a Tab from a Tab Panel

You can remove a tab from a tab panel using **VkTabPanel::removeTab()**:

```
Boolean removeTab(int index)
Boolean removeTab(char *label)
```

You can specify the tab to remove using either its position index or its label. If **removeTab()** successfully removes the tab, it returns TRUE; otherwise, if the position index was out of range or it couldn't find a tab with the label string you specified, it returns FALSE.

Note: If you use the same label for two or more tabs and provide a label string to **removeTab()**, it removes the first tab (that is, the one with the lowest index) that matches the label string. In general, you should avoid using duplicate label strings.

Adding a Pixmap to a Tab

You can set or change the pixmap associated with a tab using **VkTabPanel::setTabPixmap()**:

```
Boolean setTabPixmap(int index, Pixmap pixmap)
Boolean setTabPixmap(char *label, Pixmap pixmap)
```

You can specify the tab using either its position index or its label. If **setTabPixmap()** successfully sets the tab, it redraws the tabs and returns TRUE; otherwise, if the position index was out of range or it couldn't find a tab with the label string you specified, it returns FALSE.

The Pixmap can be either a bitmap (Pixmap of depth 1) or a full color Pixmap.

Note: If you use the same label for two or more tabs and provide a label string to **setTabPixmap()**, it sets the pixmap for the first tab (that is, the one with the lowest index) that matches the label string. In general, you should avoid using duplicate label strings

To remove an existing pixmap from a tab, call **setTabPixmap()** with a NULL pixmap.

You can retrieve the pixmap currently installed in a tab using **VkTabPanel::tabPixmap()**:

```
Boolean tabPixmap(int index, Pixmap *pixmap_return)
Boolean tabPixmap(char *label, Pixmap *pixmap_return)
```

You can specify the tab using either its position index or its label. If **tabPixmap()** is successful, the function returns TRUE and the value of the *pixmap_return* argument is set to point to the tab's pixmap; otherwise, if the position index was out of range or the function couldn't find a tab with the label string you specified, **tabPixmap()** returns FALSE.

Responding to Tab Selection

The user can select a tab two different ways: 1) by clicking a tab with the left mouse button; or 2) by clicking a group of collapsed tabs with either the left or right mouse button and choosing a menu item corresponding to a tab. When the user selects a tab by either method, the **VkTabPanel** object activates its *VkTabPanel::tabSelectCallback*. You can register callback functions to perform actions based on the tabs selected.

When activated, *tabSelectCallback* provides a pointer to a VkTabCallbackStruct as call data. The format of VkTabCallbackStruct is:

```
typedef struct {
    char *label;
    void *clientData;
    int tabIndex;
    XEvent *event;
} VkTabCallbackStruct
```

label is the label displayed by the tab. Note that if you set the label by specifying a resource name when you added this tab, the value of *label* is the value of the resource you specified.

clientData is the client data you provided when you added this tab to the tab panel.

tabIndex is the position index of the tab. Tabs are numbered sequentially, with 0 representing the left-most tab in a horizontal tab panel or the top-most tab in a vertical tab panel.

If the user selected the tab directly (that is, not through the popup menu), *event* is the ButtonPress event that triggered the selection. Otherwise, *event* is NULL.

In your callback function, you should cast the call data to (VkTabCallbackStruct *), determine which tab the user selected, and perform whatever action is appropriate.

The **VkTabPanel** object also detects when the user clicks the right mouse button on one of the tabs. Doing so does not select the tab, but it does cause **VkTabPanel** to activate its *VkTabPanel::tabPopupCallback*. When activated, *tabPopupCallback* provides a pointer to a VkTabCallbackStruct as call data. You can register callback functions to handle this event and perform any actions that you want.

You can programmatically select a tab using **VkTabPanel::selectTab()**:

```
Boolean selectTab(int index, XEvent *event = NULL);
Boolean selectTab(char *label, XEvent *event = NULL);
```

You can specify the tab to select using either its position index or its label. If **selectTab()** successfully selects the tab, it returns TRUE; otherwise, if the position index is out of range or it can't find a tab with the label string you specified, it returns FALSE.

Note: If you use the same label for two or more tabs and provide a label string to **selectTab()**, it selects the first tab (that is, the one with the lowest index) that matches the label string. In general, you should avoid using duplicate label strings.

You can optionally provide an *event* argument that **selectTab()** places in a VkTabCallbackStruct structure, which is then passed as call data to *tabSelectCallback*.

You can also determine the currently selected tab with **VkTabPanel::selectedTab()**:

```
int selectedTab()
```

selectedTab() returns the index of the currently selected tab. Tabs are numbered sequentially, with 0 representing the left-most tab in a horizontal tab panel or the top-most tab in a vertical tab panel.

Tab Panel Access Functions

VkTabPanel provides several functions for accessing information about a tab panel and its tabs:

- **VkTabPanel::getTab()** retrieves information about a specific tab. Specify the position index of the tab with the *index* argument. **getTab()** sets the value of the *label_return* argument to point to the tab's label. Note that if you set the label by specifying a resource name when you added this tab, the value of *label_return* is the value of the resource you specified. **getTab()** sets the value of the *clientData_return* argument to point to the client data you provided when you added the tab.

  getTab() returns TRUE if it is successful, and FALSE if the position index was out of range.

  ```
  Boolean getTab(int index, char **label_return,
                 void **clientData_return)
  ```

- **VkTabPanel::horiz()** returns TRUE if the tab component is horizontally oriented, and FALSE if it is vertically oriented.

  ```
  Boolean horiz()
  ```

- **VkTabPanel::size()** returns the number of tabs in the tab panel.

  ```
  int size()
  ```

- **VkTabPanel::tabHeight()** returns the height of the tab display area. This is the maximum display height for pixmaps. Larger pixmaps are truncated, and smaller pixmaps are centered. The height of the tab display area is determined by: 1) the value you specify in the **VkTabPanel** constructor; 2) the value of the **VkTabPanel** component's "tabHeight" resource; 3) the value of the "height" resource of the "tabLabel" widget created by **VkTabPanel**; or 4) the height of the tab label's font as specified by the "fontList" resource of the "tabLabel" widget created by **VkTabPanel**. If you attempt to set the tab height through multiple methods, method 1 has the highest precedence and method 4 has the lowest.

  Note: In most cases when you display a vertical tab panel, you must explicitly set the height of the tab display area. As described above, the default tab display area height is determined by the tab label's font height rather than the width of the label. As a result, the tabs might not be large enough to display all of the label text.

  The total height of a tab, including decoration, is:

  1. the height of the tab display area as returned by **tabHeight()**; plus
  2. the tab's top and bottom margin, determined by the value of the "marginHeight" resource of the "tabLabel" widget created by **VkTabPanel**; plus
  3. the value of the **VkTabPanel** component's "additionalMarginHeight" resource.

  The total height of the **VkTabPanel** component (or width, if the tab panel is horizontal) is the total height of the tab as described above, plus the value of the **VkTabPanel** component's "margin" resource.

  ```
  int tabHeight()
  ```

- **VkTabPanel::uniformTabs()** returns TRUE if the tabs have a uniform width (or height, if the tab panel is vertical). By default, tabs take on the width necessary to display their label and pixmap. You can force all tabs to take the width of the largest tab in the group by setting the **VkTabPanel** component's "uniformTabs" resource to TRUE.

  The total width of a tab, including decoration, is: 1) the width of the tab label; plus 2) if the tab has a pixmap installed, the width of the pixmap plus the pixmap spacing, determined by the value of the **VkTabPanel** component's "pixmapSpacing" resource; plus 3) the tab's left and right margin, determined by the value of the "marginWidth" resource of the "tabLabel" widget created by **VkTabPanel** plus the value of the **VkTabPanel** component's "additionalMarginWidth" resource.

  ```
  Boolean uniformTabs()
  ```

- **VkTabPanel::lineThickness()** returns the line thickness used when drawing the tab edges. The line thickness defaults to 1. You can set this value through the "lineThickness" resource of the **VkTabPanel** component, but a line thickness other than 1 might not render properly.

  ```
  int lineThickness()
  ```

- **VkTabPanel::tabBg()** returns the color used for the background area around the tabs. This color is set by the "background" resource of the **VkTabPanel** component.

  ```
  Pixel tabBg()
  ```

- **VkTabPanel::labelFg()** returns the color used for tab foregrounds (that is, the tab lettering and the foreground bits if the pixmap you supply is a bitmap). This color is set by the "foreground" resource for the "tabLabel" widget created by **VkTabPanel**.

  ```
  Pixel labelFg()
  ```

- **VkTabPanel::labelBg()** returns the color used for tab backgrounds. This color is set by the "background" resource for the "tabLabel" widget created by **VkTabPanel**. When a bitmap is supplied as the pixmap, this color is used for the background bits.

  ```
  Pixel labelBg()
  ```

- **VkTabPanel::gc()** returns the X graphics context used for drawing the tabs. This might be useful if you create pixmaps and want to use the same foreground and background colors as the tabs.

  ```
  GC gc()
  ```

- **VkTabPanel::area1()** returns the work area widget to the left of the tab display (or top if the tab panel is vertical), and **VkTabPanel::area2()** returns the work area widget to the right of the tab display (or bottom if the tab panel is vertical). Both work areas are implemented as IRIS IM Form widgets. By default, these work areas are empty. You can access these work area widgets and implement additional displays or controls if you desire.

  ```
  Widget area1()
  Widget area2()
  ```

X Resources Associated with the Tab Panel Component

The **VkTabPanel** class provides several X resources that determine display characteristics of the component:

additionalMarginHeight	Additional height, expressed in pixels, added to the margin between the top and bottom of the tab border and the tab display area (default value 2)
additionalMarginWidth	Additional width, expressed in pixels, added to the margin between the sides of the tab border and the tab display area (default value 4)
background	The background color of the **VkTabPanel** component, shown in the space around the tabs
endMultiplier	The number of overlapped tab symbols displayed as an “end indicator” when there are more tabs in the panel than can be displayed at one time (default value 3)
endSpacing	The space, expressed in pixels, between overlapped tab symbols in the “end indicator” (default value 9)
lineThickness	The line thickness used when drawing the tab edges. The default value is 1. You can provide another value, but line thickness other than 1 might not render properly.
margin	The margin, expressed in pixels, between the tab edges and the component edge (default value 5)
margin1	The margin, expressed in pixels, between the left or top work area widget and the tabs (default value 5)

margin2	The margin, expressed in pixels, between the right or bottom work area widget and the tabs (default value 5)
pixmapSpacing	If the tab contains a pixmap, the space, expressed in pixels, between the tab label and the pixmap (default value 3)
selectedTabBackground	The background color of the selected tab
sideOffset	The amount of tab overlap, expressed in pixels (default value 17)
tabHeight	The height of the tab display area is determined by: 1) the value you specify in the **VkTabPanel** constructor; 2) the value of the **VkTabPanel** component's "tabHeight" resource; 3) the value of the "height" resource of the "tabLabel" widget created by **VkTabPanel**; or 4) the height of the tab label's font as specified by the "fontList" resource of the "tabLabel" widget created by **VkTabPanel**. If you attempt to set the tab height through multiple methods, method 1 has the highest precedence and method 4 has the lowest precedence. The default value of "tabHeight" is 0.
uniformTabs	Determines whether all tabs have the same width. The default value, FALSE, allows tabs to be wide enough to display their label and pixmap. You can force all tabs to take the width of the largest tab in the group by setting this resource to TRUE.

The **VkTabPanel** class creates a widget called "tabLabel" to manage the tabs in a tab panel. **VkTabPanel** provides several X resources that determine display characteristics of the "tabLabel" widget:

tabLabel.background	The color used for tab backgrounds. When a bitmap is supplied as the pixmap, this color is used for the background bits.
tabLabel.fontList	The font used for tab labels. If the values of the "tabLabel.height" and "tabHeight" resources are 0, and you do not specify a tab height in the **VkTabPanel** constructor, the height of the font is also used as the height of the tab display area.
tabLabel.foreground	The color used for tab foregrounds (that is, the tab lettering and the foreground bits if the pixmap you supply is a bitmap)

tabLabel.height
: The height of the tab display area is determined by: 1) the value you specify in the **VkTabPanel** constructor; 2) the value of the **VkTabPanel** component's "tabHeight" resource; 3) the value of the "height" resource of the "tabLabel" widget created by **VkTabPanel**; or 4) the height of the tab label's font as specified by the "fontList" resource of the "tabLabel" widget created by **VkTabPanel**. If you attempt to set the tab height through multiple methods, method 1 has the highest precedence and method 4 has the lowest precedence. The default value of "tabLabel.height" is 0.

tabLabel.marginHeight
: The margin, expressed in pixels, between the top and bottom of the tab border and the tab display area

tabLabel.marginWidth
: The margin, expressed in pixels, between the sides of the tab border and the tab display area

The Text Completion Field Component

The **VkCompletionField** class, derived from **VkComponent**, provides a text input field component that supports name expansion. While typing in the field, if the user types a space, then the component attempts to complete the current contents of the field based on a list of possible expansions provided by the application. For example, in a field where the user is expected to enter a file name, the application could provide a list of all files in the current working directory.

The Text Completion Field Constructor and Destructor

The **VkCompletionField** constructor accepts the standard ViewKit component name and parent widget arguments:

```
VkCompletionField(const char *name, Widget parent)
```

The constructor creates an IRIS IM TextField widget as the component's base widget. You can access this widget using the **baseWidget()** function provided by **VkComponent**.

The **VkCompletionField** destructor destroys the component's widget and associated data, including the **VkNameList** object that stores the list of possible expansions. You should be aware of this in case you provide an existing **VkNameList** object as an argument to the **VkCompletionField::clear()** function, described in "Setting and Clearing the Text Completion Field Expansion List." Consult the **VkNameList**(3x) reference page for more information on that class.

Setting and Clearing the Text Completion Field Expansion List

You can add individual strings to the completion list by passing them as arguments to the **VkCompletionField::add()** function:

```
void add(char *name)
```

You can clear the completion list by calling the **VkCompletionField::clear()** function:

```
void clear(VkNameList *nameList = NULL)
```

If you provide a **VkNameList** object, **clear()** deletes the current completion list and uses the **VkNameList** object that you provide as the new completion list for the completion field. Consult the **VkNameList**(3x) reference page for more information on that class.

Retrieving the Text Completion Field Contents

The **VkCompletionField::getText()** function duplicates the contents of the text field and then returns a pointer to the duplicate string:

```
char *getText()
```

Note: Because **getText()** creates a copy of the text field's contents, you can safely change or delete the returned string.

For example, the following line retrieves the contents of a **VkCompletionField** object called *fileName* and assigns the string to the variable *openFile*:

```
openFile = fileName->getText();
```

Responding to Text Completion Field Activation

The **VkCompletionField** class supplies a ViewKit member function callback named *VkCompletionField::enterCallback*. This callback is activated whenever the user presses the **<Enter>** key while typing in the text field. The callback does not pass any call data. If you want to notify a ViewKit component whenever the user presses the **`<Enter>`** key while typing in a **VkCompletionField** object, register a member function of that component as an *enterCallback* function.

Deriving Text Completion Field Subclasses

The **VkCompletionField** class should be sufficient for most applications; however, if you want to have more control over the expansion process you can create a subclass of **VkCompletionField**.

The protected member function **VkCompletionField::expand()** is called whenever the user types in the text field:

```
virtual void expand(struct XmTextVerifyCallbackStruct *cb)
```

By default, **expand()** checks whether the user has typed a space, and if so, tries to expand the current contents of the text field; if the user types any other character, **expand()** simply adds that character to the text field. At any point after an expansion, the **VkNameList** object pointed to by the protected data member *_currentMatchList* contains a list of all possible expansions:

```
VkNameList *_currentMatchList
```

You can override the **expand()** function to install your own expansion algorithm. You have access to the **VkNameList** object pointed to by the protected data member *_nameList*, which contains all possible expansions registered with the component:

```
VkNameList *_nameList
```

You can also override the protected member function **VkCompletionField::activate()**, which is called whenever the user presses the **`<Enter>`** key while typing in the text field:

```
virtual void activate(struct XmTextVerifyCallbackStruct *cb)
```

activate() is called after expanding the current contents of the text field and after invoking all member functions registered with the *enterCallback* callback. By default, this function is empty.

The Repeating Button Component

The **VkRepeatButton** class, derived from **VkComponent**, provides an auto-repeating pushbutton. A regular pushbutton activates only once when the user clicks it and releases it. A **VkRepeatButton** behaves more like a scrollbar button: it activates when the user clicks it; after a given delay it begins repeating at a given interval; and it stops activating when the user releases it.

The Repeating Button Constructor

The **VkRepeatButton** constructor takes three arguments:

```
VkRepeatButton(char *name, Widget parent,
               VkRepeatButtonType type)
```

name is a character string specifying the component name. *parent* is the parent widget of the component. *type* is a VkRepeatButtonType enumerated value specifying the type of button to create. This value can be any of RB_pushButton, RB_pushButtonGadget, RB_arrowButton, or RB_arrowButtonGadget. These create PushButton, PushButtonGadget, ArrowButton, and ArrowButtonGadget widgets, respectively.

Responding to Repeat Button Activation

A **VkRepeatButton** object triggers a *VkRepeatButton::buttonCallback* ViewKit callback whenever the button activates. Any ViewKit object can register a member function with the callback to be invoked when the button activates.

The callback provides an XmAnyCallbackStruct pointer as call data; the XmAnyCallbackStruct.reason contains the reason for the callback, and the XmAnyCallbackStruct.event field contains the event that triggered the callback.

Repeating Button Utility and Access Functions

The **VkRepeatButton::setParameters()** function changes the delay parameters for the button:

```
void setParameters(long initial, long repeat)
```

initial controls how long, in milliseconds, the user has to hold the button down before it begins to repeat. *repeat* controls the interval between auto-repeat activations, in milliseconds.

If you need to determine the type of a **VkRepeatButton** after creation, you can call the **VkRepeatButton::type()** function:

```
VkRepeatButtonType type()
```

The return value is a VkRepeatButtonType enumerated value specifying the type of button. This value can be any of RB_pushButton, RB_pushButtonGadget, RB_arrowButton, or RB_arrowButtonGadget, which indicates PushButton, PushButtonGadget, ArrowButton, and ArrowButtonGadget widgets, respectively.

X Resources Associated with the Repeating Button Component

The **VkRepeatButton** class provides the following X resources that determine operating characteristics of the component:

initialDelay	The initial delay in milliseconds before auto-repeat begins (default value 1000)
repeatDelay	The auto-repeat interval in milliseconds (default value 200)

Management Classes for Controlling Component and Widget Operation

ViewKit provides some management classes that control the operation of components and widgets. These classes function as *attachments*: you attach them to one or more existing widgets or components. Then, you can use the management class to control some aspect of operation of the widgets and components to which the class is attached.

Supporting “Ganged” Scrollbar Operation

The **VkGangedGroup** class provides support for “ganging” together IRIS IM ScrollBar or Scale widgets so that all of them move together; when the value of one of the ScrollBar or Scale widgets changes, all other widgets in the group are updated with that value. **VkGangedGroup** is derived from the convenience class **VkWidgetList**. Consult the **VkWidgetList**(3x) reference page for more information on that class.

To use the **VkGangedGroup** class, you create a **VkGangedGroup** object and add widgets or components to the group. Thereafter, the **VkGangedGroup** object automatically updates all of the scales and scrollbars in the group whenever the value of one of them changes.

The Ganged Scrollbar Group Constructor and Destructor

The **VkGangedGroup** constructor does not take any arguments:

```
VkGangedGroup()
```

VkGangedGroup objects do not require names because they are not components; ViewKit uses names to uniquely identify the widget trees of components, and the **VkGangedGroup** class does not create any widgets.

The **VkGangedGroup** destructor destroys only the **VkGangedGroup** object. If you have widgets or components managed by the object, they are unaffected by the **VkGangedGroup** destructor.

Adding Scales and Scrollbars to a Ganged Group

Use the **VkGangedGroup::add()** function to add widgets or components to a **VkGangedGroup** object:

```
virtual void add(Widget w)
virtual void add(VkComponent *obj)
```

If you provide a widget, **add()** adds that widget to the alignment group. If you provide a pointer to a component, **add()** adds the component's base widget to the alignment group.

Note: If you add a component to a **VkGangedGroup** object, the base widget of that component must be an IRIS IM ScrollBar or Scale
widget.

Removing Scales and Scrollbars from a Ganged Group

You can remove widgets or components from a **VkGangedGroup** object with the **remove()** function inherited from **VkWidgetList**:

```
virtual void remove(Widget w)
virtual void remove(VkComponent *obj)
```

Provide the widget ID or component pointer that you used to add the widget or component to the ganged group.

You can also use the **removeFirst()** and **removeLast()** functions inherited from **VkWidgetList** to remove the first or last item respectively in the ganged group:

```
virtual void removeFirst()
virtual void removeLast()
```

Enforcing Radio-Style Behavior on Toggle Buttons

IRIS IM supports collections of toggle buttons that exhibit one-of-many or "radio-style" behavior by placing all related buttons in a RadioBox widget. This is adequate in many cases, but in some cases it is useful to enforce radio-style behavior on a collection of buttons dispersed throughout an application.

The **VkRadioGroup** class provides support for enforcing radio-style behavior on an arbitrary group of toggle buttons, no matter where they appear in your application's widget hierarchy. The **VkRadioGroup** class supports both IRIS IM ToggleButton and ToggleButtonGadget widgets. Furthermore, you can add IRIS IM PushButton and PushButtonGadget widgets to a **VkRadioGroup** object; the **VkRadioGroup** object simulates radio-style behavior on these buttons by displaying them as armed when the user selects them (using the XmNarmColor color resource as the button's background color and displaying the XmNarmPixmap if the button contains a pixmap).

VkRadioGroup is derived from the convenience class **VkWidgetList**. Consult the **VkWidgetList**(3x) reference page for more information on that class.

To use the **VkRadioGroup** class, create a **VkRadioGroup** object and add widgets or components to the group. Thereafter, the **VkRadioGroup** object automatically updates all buttons contained in the group whenever the user selects one of the buttons.

Note: Membership in a **VkRadioGroup** object is not exclusive; a widget can potentially belong to multiple groups at once.

The Radio Group Constructor and Destructor

The **VkRadioGroup** constructor does not take any arguments:

```
VkGangedGroup()
```

VkRadioGroup objects do not require names because they are not components; ViewKit uses names to uniquely identify the widget trees of components, and the **VkRadioGroup** class does not create any widgets.

The **VkRadioGroup** destructor destroys only the **VkRadioGroup** object. If you have widgets or components managed by the object, they are unaffected by the **VkRadioGroup** destructor.

Adding Toggles and Buttons to a Radio Group

Use the **VkRadioGroup::add()** function to add widgets or components to a **VkRadioGroup** object:

```
virtual void add(Widget w)
virtual void add(VkComponent *obj)
```

If you provide a widget, **add()** adds that widget to the radio group. If you provide a pointer to a component, **add()** adds the component's base widget to the alignment group.

Note: If you add a component to a **VkRadioGroup** object, the base widget of that component must be an IRIS IM ToggleButton, ToggleButtonGadget, PushButton, or PushButtonGadget widget.

Removing Toggles and Buttons from a Radio Group

You can remove widgets or components from a **VkRadioGroup** object with the **remove()** function inherited from **VkWidgetList**:

```
virtual void remove(Widget w)
virtual void remove(VkComponent *obj)
```

Provide the widget ID or component pointer that you used to add the widget or component to the radio group.

You can also use the **removeFirst()** and **removeLast()** functions inherited from **VkWidgetList** to remove the first or last item, respectively, in the radio group:

```
virtual void removeFirst()
virtual void removeLast()
```

Deriving Radio Group Subclasses

If you use a direct instantiation of **VkRadioGroup**, you must rely on Xt callback functions registered directly with the toggle buttons to detect and handle state changes in the group. Another approach is to derive a subclass of **VkRadioGroup** and override the protected **VkRadioGroup::valueChanged()** function:

```
virtual void valueChanged (Widget w, XtPointer callData)
```

valueChanged() is called whenever any member of the radio group changes state. The first argument is the selected widget. The second argument is the call data from the XmNvalueChangedCallback (in the case of a ToggleButton or ToggleButtonGadget widget) or the XmNactivateCallback (in the case of a PushButton or PushButtonGadget widget).

You can override **valueChanged()** to receive notification of state changes and perform any actions you want. If you override **valueChanged()**, you should call **VkRadioGroup::valueChanged()** to update the states of all members of the radio group before performing any other actions.

Modified Text Attachment

The **VkModifiedAttachment** class provides support for tracking the previous and current values in an IRIS IM Text or TextField widget. The **VkModifiedAttachment** class automatically displays a dogear (a “folded corner”) in the upper-right corner of the text widget when the user changes the text value. Figure 11-8 shows an example of a text widget with a **VkModifiedAttachment** dogear.

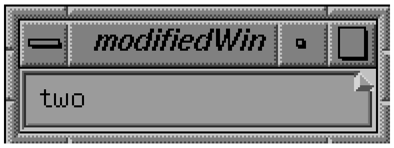


Figure 11-8 Example of a **VkModifiedAttachment** Dogear

The user can “flip” between the previous and current text values by clicking the dogear. Figure 11-9 demonstrates the results of flipping to a previous text value by clicking the dogear.

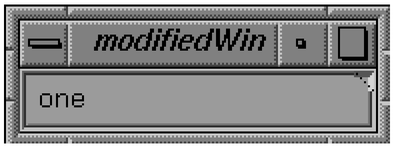


Figure 11-9 Example of “Flipping” to a Previous Text Widget Value Using a **VkModifiedAttachment** Dogear

When the user presses the **`<Return>`** key in the text field, the text displayed becomes the current value of the text field and the previously-displayed text becomes the previous value. If the current and previous values are the same, the **VkModifiedAttachment** object does not display the dogear; the **VkModifiedAttachment** object redisplays the dogear when the current and previous values are different.

Note: If the user clicks the dogear before pressing the **`<Return>`** key, any changes the user made are discarded.

To use the **VkModifiedAttachment** class, you must: 1) create an IRIS IM Text or TextField widget; 2) create a **VkModifiedAttachment** object; 3) attach the **VkModifiedAttachment** object to the widget; and 4) display the **VkModifiedAttachment** object (to display its dogear).

The **VkModifiedAttachment** class also provides several functions for retrieving the previous and current values of the text field, setting the value of the text field, and managing the display of the object.

Note: Because the **VkModifiedAttachment** class adds callback functions to handle the changes in value of the text widget, you should not register your own XmNactivateCallback or XmNvalueChangedCallback functions with the text widget. Instead, you should use the *VkModifiedAttachment::modifiedCallback* ViewKit callback to determine when the text widget changes its value, and use the **VkModifiedAttachment** access functions to obtain the current or previous value of the text widget.

VkModifiedAttachment is derived from the **VkModified** base class, which tracks previous and current text values not necessarily associated with a text widget. In most cases, you will use the **VkModifiedAttachment** class; therefore, this section describes the functions inherited from **VkModified** along with the functions implemented by **VkModifiedAttachment**. For more information on the **VkModified** class, consult the **VkModified**(3Vk) reference page.

Note: The **VkModified** and **VkModifiedAttachment** classes are both declared in the <*Vk/VkModified.h*> header file.

The Modified Text Attachment Constructor and Destructor

The **VkModifiedAttachment** constructor accepts three Boolean values:

```
VkModifiedAttachment(Boolean blankIsValue = FALSE,
                     Boolean autoAdjust = TRUE,
                     Boolean incrementalChange = FALSE)
```

blankIsValue determines whether the **VkModifiedAttachment** object accepts a null string (a blank) as a valid previous value when displaying the dogear. If *blankIsValue* is FALSE, the **VkModifiedAttachment** object does not display the dogear if the previous value is blank.

autoAdjust determines whether the **VkModifiedAttachment** object automatically watches its attached text widget for geometry changes and adjusts its own area accordingly. If you set this value to FALSE, you must explicitly call **VkModifiedAttachment::adjustGeometry()** after changing the geometry of the text widget.

If *incrementalChange* is TRUE, each incremental change to the text value updates the current and previous values. In this mode, activation of the text widget's XmNvalueChangedCallback callback is considered an incremental change. Examples of incremental changes are: each character added or deleted, each deletion of selected characters, and each text insertion by pasting selected text. If *incrementalChange* is FALSE, the **VkModifiedAttachment** object updates the current and previous values only when the user presses the **`<Return>`** key in the text field.

The **VkModifiedAttachment** destructor destroys only the **VkModifiedAttachment** object. If you have a widget attached to the object, it is unaffected by the **VkModifiedAttachment** destructor.

Attaching and Detaching the Modified Text Attachment to and from a Widget

Once you have created a **VkModifiedAttachment** object, use the **VkModifiedAttachment::attach()** function to attach it to an existing widget:

```
void attach(Widget w)
```

If the **VkModifiedAttachment** object is already attached to a widget, it detaches from the old widget before attaching to the new widget. You can use the **VkModifiedAttachment::detach()** function to detach a **VkModifiedAttachment** object from a widget without immediately attaching it to another widget:

```
void detach()
```

Displaying and Hiding the Modified Text Attachment

Once you have attached a **VkModifiedAttachment** object to a text widget, you must call **VkModifiedAttachment::show()** to display the attachment:

```
void show()
```

You can hide a **VkModifiedAttachment** object by calling **VkModifiedAttachment::hide()**:

```
void hide()
```

When a **VkModifiedAttachment** object is hidden, it still tracks the current and previous values of the text widget to which it is attached; the user simply cannot toggle between the values. You can still use the **VkModifiedAttachment** class's access functions to retrieve the previous and current values of the text field.

VkModifiedAttachment::expose() forces a redraw of the attachment's dogear:

```
void expose()
```

expose() is called whenever the dogear widget receives an Expose event. Normally, you should not need to call this function.

Retrieving the Current and Previous Values of the Text Widget

You can retrieve the current and previous values of the text widget with **value()** and **previousValue()** respectively:

```
char *value()
char *previousValue()
```

Note: Do not change or delete the character strings returned by **value()** and **previousValue()**.

Detecting Changes in the Text Widget

The **VkModifiedAttachment** class provides a ViewKit member function callback named *VkModifiedAttachment::modifiedCallback*:

```
static const char *const modifiedCallback
```

The **VkModifiedAttachment** object activates this callback whenever the text widget triggers its XmNactivateCallback or XmNvalueChangedCallback callback.
The *modifiedCallback* provides a pointer to a VkModifiedCallback structure as call data. VkModifiedCallback has the following structure:

```
typedef struct {
    VkModifiedReason reason;
    class VkModified *obj;
    XEvent *event;
} VkModifiedCallback
```

The VkModifiedCallback fields are:

reason	The reason for the callback. It can take one of two values: VM_activate, if the text widget triggered its XmNactivateCallback callback; or VM_valueChanged if the text widget triggered its XmNvalueChangedCallback callback.
obj	A pointer to the **VkModifiedAttachment** object
event	A pointer to the event that triggered the callback

Typically, your callback function should test the reason for the callback and perform an action if appropriate. For example, you can use one of the access functions to obtain the current or previous value of the text widget.

Note: Because the **VkModifiedAttachment** class adds callback functions to handle the changes in value of the text widget, you should not register your own XmNactivateCallback or XmNvalueChangedCallback callback functions with the text widget. Instead, always use the *modifiedCallback* ViewKit callback to determine when the text widget changes its value.

Controlling the Contents of the Text Widget

You can programmatically set the new current value of a **VkModifiedAttachment** object with **VkModifiedAttachment::setValue()**:

```
virtual void setValue(const char *value)
```

setValue() sets the object's new current value; the old current value becomes the previous value. **VkModifiedAttachment** forces the text widget to display the new current value.

VkModifiedAttachment::toggleDisplay() programmatically toggles the text widget display between the current value and the previous value:

```
virtual void toggleDisplay()
```

To determine which value the text widget is displaying, call **VkModifiedAttachment::latestDisplay()**:

```
Boolean latestDisplay()
```

latestDisplay() returns TRUE if the text widget is displaying the current value or FALSE if the text widget is displaying the previous value.

Finally, you can reset the contents of the text widget with **VkModifiedAttachment::displayValue()**

```
void displayValue()
```

displayValue() discards any changes the user may have made and updates the text widget with the current value (if the user has the current view selected) or the previous value (if the user has the previous view selected).

Adjusting the Modified Text Attachment's Geometry

By default, the **VkModifiedAttachment** object automatically watches its attached text widget for geometry changes and adjusts its own area accordingly. If you set the *autoAdjust* argument in the **VkModifiedAttachment** constructor to FALSE, you must explicitly call **VkModifiedAttachment::adjustGeometry()** after changing the geometry of the text widget to adjust the attachment's geometry:

```
void adjustGeometry()
```

You can also control the size of the **VkModifiedAttachment** dogear. By default, the dogear is 10 pixels wide by 10 pixels tall. You can set the width and height to different values with the **VkModifiedAttachment::setParameters()** function:

```
virtual void setParameters(Dimension width, Dimension height)
```

To retrieve the current width and height of the dogear, call **VkModifiedAttachment::getParameters()**:

```
void getParameters(Dimension *width, Dimension *height)
```

Other Modified Text Attachment Utility and Access Functions

The **VkModifiedAttachment** class provides several additional utility and access functions:

- **VkModifiedAttachment::fixPreviousValue()** allows you to specify a fixed value to use as the attachment's previous value. After setting a fixed previous value, the attachment does not update the previous value; this provides a "default" value that the user can always toggle to and use.

  If *setValueAlso* is TRUE, **fixPreviousValue()** also updates the attachment's current value to *fixedValue*; however, this does not permanently fix the current value.

  ```
  virtual void fixPreviousValue(char *fixedValue,
                                Boolean setValueAlso = TRUE)
  ```

- **VkModifiedAttachment::widget()** returns the text widget to which the **VkModifiedAttachment** object is currently attached.

  ```
  Widget widget()
  ```

- **VkModifiedAttachment::modified()** returns TRUE if the current value and the previous value are equal and FALSE if they are not equal.

  ```
  Boolean modified()
  ```

- **VkModifiedAttachment::setModified()** forces the value of the object's modified flag. If you set the value to TRUE, the **VkModifiedAttachment** object displays its dogear; otherwise, it hides its dogear.

  ```
  virtual void setModified(Boolean value)
  ```

X Resources Associated with the Modified Text Attachment

You can set the value of an XmNdisplayModified resource for a text widget to determine whether or not the attached **VkModifiedAttachment** object should display its dogear. If you set the text widget's XmNdisplayModified resource to TRUE or if you do not provide a value for the text widget's XmNdisplayModified resource, the attached **VkModifiedAttachment** object displays its dogear. This is the default behavior.

If you set the text widget's XmNdisplayModified resource to FALSE, the attached **VkModifiedAttachment** object does not display its dogear, but it does continue to track the text widget's current and previous values. You can still use the functions and callbacks provided by **VkModifiedAttachment** to manipulate the values and manage the text widget.

Appendix A

ViewKit Interprocess Message Facility

This appendix describes the ViewKit interprocess message facility, which consists of a set of classes that support the ToolTalk™ message service for interprocess communication.

Note: To develop applications that use ToolTalk, you must purchase the ToolTalk Development Option. In the United States and Canada, call SGI Direct at 800-800-SGI1 (7441) for more information about how to order the ToolTalk Development Option; outside the United States and Canada, please contact your local sales office or distributor. The ToolTalk development library provides low-level functions for setting up connections to the ToolTalk server and sending and receiving messages. The ViewKit message facility is built on top of the ToolTalk development library to provide a higher-level interface to creating and handling interprocess messages.

Figure A-1 shows the inheritance graph for the classes supporting the ViewKit message facility. The ViewKit message facility also provides several utility functions that are not class member functions.

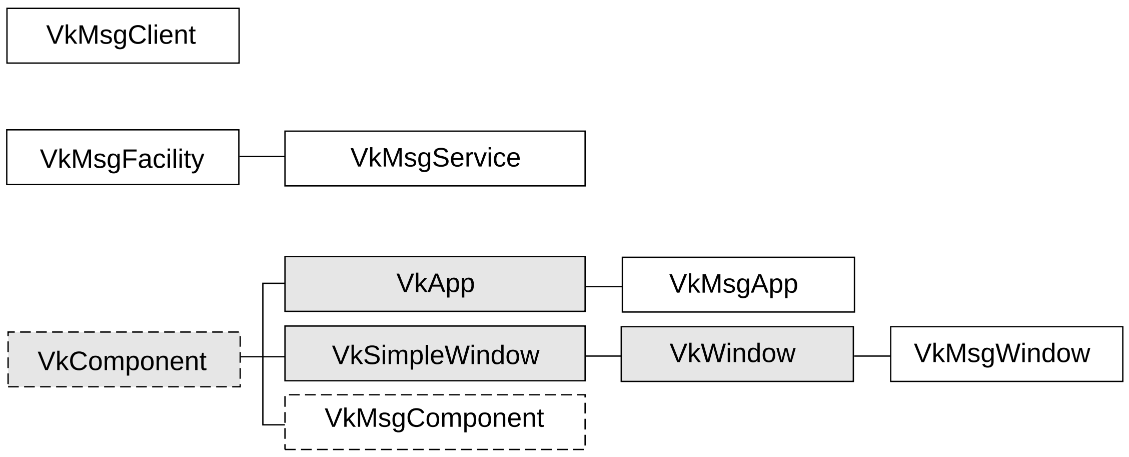


Figure A-1 The Inheritance Graph for the ViewKit Message Facility Classes

Review of ToolTalk Concepts

This section provides a brief review of some ToolTalk concepts and terms. For a complete description of the ToolTalk message service and directions for creating applications that interface with the ToolTalk service, consult the *ToolTalk Programmer's Guide*.

The ToolTalk message service allows independent applications to communicate with each other without having direct knowledge of each other. Applications exchange ToolTalk *messages* to communicate with other applications. *Sending applications* create, fill in, and send a message; the ToolTalk service determines the recipients and delivers the message to the *receiving applications*. Receiving applications retrieve messages, examine the information in the message, and then either discard the message or perform an operation and reply with the results.

Messages consist of a character string operator, followed by any number of arguments. The arguments can be integers, character strings, or binary strings. Also, a message can have attributes such as a filename. The receiver of a message is told the number and type of arguments, and is also given access to any attribute values.

For each type of message an application wants to receive, it must *register* a *message pattern* with the ToolTalk service. The message pattern describes the operator, arguments, and attributes that a message must have to be delivered to the application. When the application registers a pattern, it must indicate whether it wants to *observe* or *handle* messages of that type. Any number of applications can observe a message. The ToolTalk service forwards a copy of the message to each application with a matching message pattern. On the other hand, to ensure that a requested operation is performed only once, only one application can handle a message. If the ToolTalk service cannot find a handler for a message, it returns the message to the sending application indicating that delivery failed.

There are two types of messages: *notices* and *requests*. A sending application sends a notice to provide information to other applications; the sending application does not expect a reply to a notice. Receiving applications receive notices and perform whatever actions are needed to process the notice; the receiving applications do not send reply message in response to notices. A sending application sends a request to ask another application to perform an action; the sending application expects a single reply to a request. A receiving application receives the request, performs whatever actions are needed to process the request, and sends a reply message. The ToolTalk service handles the details of how requests are handled, and ensures that only one receiver gets to reply.

When the ToolTalk service determines that a message needs to be delivered to a specific process, but the process is not currently running, the ToolTalk service looks for instructions (provided by the application at installation time) on how to start the application.

Overview of ViewKit's ToolTalk Support

This section provides an overview of ViewKit's ToolTalk support. It discusses the classes you use to interface with ToolTalk and the policies enforced by those classes.

ViewKit Classes Supporting ToolTalk

The **VkMsgClient** class is the basis for interacting with ToolTalk in your application. **VkMsgClient** allows you to register and manage message patterns; declare actions for your application to perform when it receives messages; compose messages; and send notices and requests.

The **VkMsgApp** class, a subclass of **VkApp**, opens a ToolTalk connection and sets up all resources needed to send and receive ToolTalk messages.[1] You must instantiate a **VkMsgApp** object in your application instead of a **VkApp** object if you want ToolTalk support for your application. The **VkMsgApp** constructor creates an instance of **VkMsgClient** that you can use to manage messages in your application.

The **VkMsgWindow** class is a subclass of **VkWindow** that works with **VkMsgApp** to support the ViewKit interprocess message facility. To provide ToolTalk support for your application, you must use **VkMsgWindow** for your application's windows instead of **VkSimpleWindow** or **VkWindow**. The **VkMsgWindow** constructor creates an instance of **VkMsgClient** that you can use to manage messages in that window.

[1] **VkMsgApp** actually instantiates a **VkMsgService** object—which is in turn a subclass of **VkMsgFacility**—to perform ToolTalk initialization and support ToolTalk interaction. Although you should never need to use either the **VkMsgFacility** or **VkMsgService** classes directly, you might encounter them while debugging a ViewKit application that uses the ViewKit interprocess message facility. Consult the VkMsgFacility(3Vk) and the VkMsgService(3Vk) reference pages for more information on these classes.

The **VkMsgComponent** class is a subclass of **VkComponent** that works with **VkMsgApp** to support the ViewKit interprocess message facility. You should use **VkMsgComponent** to derive new components if those components must send or receive ToolTalk messages. You do not need to derive components from **VkMsgComponent** if those components do not interact with ToolTalk. The **VkMsgComponent** constructor creates an instance of **VkMsgClient** that you can use to manage messages in that component.

The ViewKit message facility also provides several utility functions for manipulating messages. These functions are implemented as normal functions rather than class member functions.

ViewKit Message Facility Policies

The ViewKit message facility provides mostly a mechanism for exchanging ToolTalk messages between applications, but it does impose some policies:

- Messages are always sent to all members of the session.
- Message actions receive all messages for a given operator, and cannot set patterns based on argument number or type.
- By default, applications connect to the default session when they're started.

These are the policies currently implemented, but they are subject to change in future releases of ViewKit; however, applications that adhere to these policies should not be affected by future changes.

Establishing a Connection to the ToolTalk Service

Creating an instance of the **VkMsgApp** class opens a ToolTalk connection and sets up all resources needed to send and receive ToolTalk messages. Remember to use the **VkMsgApp** class in your application instead of a **VkApp** object if you want ToolTalk support for your application. The syntax of the **VkMsgApp** constructor is:

```
VkMsgApp(char* appClassName, int* argc, char** argv,
         XrmOptionDescRec* optionList = NULL,
         int sizeOfOptionList = 0,
         const char* ptid = NULL,
         const char* sessid = NULL
         Boolean noProtocol = FALSE)
```

The first five arguments are the same as those that you can provide to the **VkApp** constructor. The *ptid* argument specifies a process type. It defaults to NULL which indicates no process type. You need to provide a *ptid* argument only if this application is autostarted (see “Registering Services for Autostart” on page 353). *sessid* specifies a session to join. If you don’t provide a value, the process joins the default session.

The *noProtocol* argument determines whether your application automatically provides support for handling “Lower,” “Raise,” and “Quit” messages. If this value is FALSE (the default), your application calls **VkApp::iconify()** upon receiving a “Lower” message, **VkApp::open()** and **VkApp::raise()** upon receiving a “Raise” message, and **VkApp::quitYourself()** upon receiving a “Quit” message.

You can also specify the session using command line arguments when you invoke your applications:

`-project` *sessid*
: Join the session specified by *sessid*

`-projectWindow` *windowid*
: Join the same session as the window specified by *windowid*

`-projectWindow`
: Allow the user to click a window and join the same session as the window specified

The **VkMsgApp** class also creates an instance of **VkMsgClient** that you can use to manage messages in your application. You can retrieve a pointer to this object with the **VkMsgApp::messageClient()** function:

```
VkMsgClient* messageClient()
```

Sending and Receiving ToolTalk Messages

This section describes how to register and manage message patterns, declare actions for your application to perform when it receives messages, composes messages, and sends notices and requests. You accomplish all of these tasks using the **VkMsgClient** class.

You do not explicitly instantiate **VkMsgClient** objects. Instead, the **VkMsgApp**, **VkMsgWindow**, and **VkMsgComponent** classes all automatically instantiate their own **VkMsgClient** objects and provide access functions to those objects.

Sending Notices and Requests

This section describes how to send notices and requests using **VkMsgClient**. Both notices and requests are types of messages.

A sending application sends a notice message to provide information to other applications; the sending application does not expect a reply to a notice. All of the parameters in a notice have a *mode* of “in”, which indicates that receiving applications should read only those parameters.

A sending application sends a request message to ask another application to perform an action; the sending application expects a single reply to a request. In a request, some of the message parameters have a mode of “out”, which indicates that the receiving application should fill in these parameters for the reply message, or “in / out”, which indicates that the receiving application should read those parameters and then fill them in with new values for the reply message.

Sending Simple Notices

If you’re sending a notice consisting of only string arguments or only integer arguments, you can use the **VkMsgClient::sendStringNotice()** or **VkMsgClient::sendIntNotice()** function, respectively:

```
void sendStringNotice(char *op ...)
void sendIntNotice(char *op ...)
```

For both functions, *op* is the message operator. **sendStringNotice()** expects a NULL-terminated list of pointers to character strings; **sendIntNotice()** expects a NULL-terminated list of integers. These functions create a message with the arguments you provide, send the message, and then automatically delete the message (that is, delete the storage space allocated by your application when the functions create the message).

Caution: Because **sendIntNotice()** expects a NULL-terminated list of arguments, be sure that you don’t provide a zero-valued argument or else you’ll prematurely terminate your argument list.

Composing and Sending Messages

To send requests or to send notices that contain a mix of argument types, you must compose the message before sending it. To begin composing a message, call **VkMsgClient::composeBegin()**:

```
void composeBegin()
```

Note: You can compose only one message at a time among all **VkMsgClient** objects.

You can then add arguments to your message one at a time, using **VkMsgClient::composeAdd()**:

```
void composeAdd(char *val, VkMsgMode mode = VK_MSG_IN)
void composeAdd(int ival, VkMsgMode mode = VK_MSG_IN)
void composeAdd(unsigned char *val, int len,
                VkMsgMode mode = VK_MSG_IN)
```

You can add as an argument: 1) a NULL-terminated character string; 2) an integer value; or 3) a binary string. If you provide a binary string, you must also specify the length of the string as the *len* argument.

The *mode* argument is an enumerated value of type VkMsgMode. VK_MSG_IN indicates that the argument is written by the sending application and can be read by the handling application and any observing applications. VK_MSG_OUT indicates that the argument is written by the handling applications and is read by the sending application. VK_MSG_INOUT indicates that the argument can be written by both the sending and handling applications, and can be read by the sending, handling, or any observing applications.

Once you finish composing the message, you send the message. If the message is a notice, use **VkMsgClient::sendNotice()**:

```
void sendNotice(char *op)
```

op is the message operator. **sendNotice()** sends the message you composed, and then automatically deletes the message (that is, deletes the storage space allocated by your application when you compose the message).

To send a request, use **VkMsgClient::sendRequest()**:

```
VkMsgMessage sendRequest(char *op)
```

sendRequest() sends the message you composed. *op* is the message operator. **sendRequest()** returns an opaque message handle. You can use this handle when calling the various utility functions provided by the ViewKit message facility as described in "Useful Functions When Handling Messages" on page 346.

When you send a request using **sendRequest()**, the function returns immediately. To obtain and handle the reply message to your request, your application must register a message action as described in "Receiving Notices and Handling Requests" on page 341.

Note: The ViewKit message facility automatically deletes the request message when your application receives the corresponding reply or failure message.

Sending Synchronous Requests

In some cases, your application might require a reply to a request before performing any other processing. In these cases, you can use **VkMsgClient::sendSyncRequest()** to send a *synchronous* request:

```
VkMsgMessage sendSyncRequest(char *op)
```

sendSyncRequest() uses a secondary X event loop to simulate a synchronous reply to a request. **sendSyncRequest()** blocks until it receives the reply message, which it passes as its return value. If the request fails, **sendSyncRequest()** returns NULL. Note that because **sendSyncRequest()** uses a secondary X event loop, you should beware of any problems with re-entrant code in any callbacks that could be invoked.

Once you receive the reply message, you can use the various utility functions provided by the ViewKit message facility, as described in "Useful Functions When Handling Messages" on page 346, to parse the reply. You must also use **VkMsgDestroy()** to destroy the message when you no longer need the reply:

```
VkMsgStatus VkMsgDestroy(VkMsgMessage msg)
```

VkMsgDestroy() is implemented as a normal function rather than a class member function. It is declared in the header file <*Vk/VkMsg.h*>. The VkMsgStatus return value is the same as the tt_status values used by ToolTalk; consult the *ToolTalk Programmer's Guide* for information on these values.

Specifying a Filename Message Attribute

The convention for passing a filename argument in a message is to specify the filename as a *message attribute* rather than a message argument. The **VkMsgClient** class provides the following functions for sending a message with a filename attribute:

```
void sendStringFileNotice(char *op, char *file ...)
void sendIntFileNotice(char *op, char *file ...)
void sendFileNotice(char *op, char *file)
void sendFileRequest(char *op, char *file)
void sendSyncFileRequest(char *op, char *file)
```

In these functions, *file* is the filename.

Receiving Notices and Handling Requests

This section describes how to receive notices and handle requests using the ViewKit message facility. It discusses message dispatch, writing message action callbacks, creating message patterns, associating message actions with the message patterns, and registering and unregistering message patterns.

Overview of Message Dispatch

For each type of message an application wants to receive, it must register a *message pattern* with the ToolTalk service. The message pattern describes the operator, arguments, and attributes that a message must have to be delivered to the application. Your application must also register *message actions*, callback functions that are called when your application receives messages with a particular message pattern.

When the ToolTalk service receives a message, it matches the message against all registered patterns. If the message is a notice, the ToolTalk service delivers a separate message to each application with a matching pattern; if the message is a request, the ToolTalk service selects the "best" pattern match and delivers a single message to the application with the matching pattern.

The ViewKit message facility then dispatches the message received by the application to *each* action registered for the matching operator, regardless of any other pattern information you provided when registering the action. Your action is responsible for testing the message arguments and determining whether to process the message or not.

As an example, consider an application that registers an action for a message pattern consisting of a “show” operator and an integer argument. If another process then sends a “show” message with an integer argument, the ToolTalk service sends the message to your application, and your application dispatches the message to the action. On the other hand, if another process sends a “show” message with a character string argument, the ToolTalk service does not send the message to your application because it does not match the registered pattern.

A complication arises if you have multiple actions registered for a particular message operator; for example, if in addition to the action described above, your application registered an action for a message pattern consisting of a “show” operator and a character string argument. In this case, the ToolTalk service sends to your application any “show” message with either an integer or a character string argument. The ViewKit message facility then dispatches those messages to *each* action in your application registered for the “show” operator. Each action is then responsible for testing the message arguments and determining whether to process the message or not.

Writing Message Action Callbacks

You implement message actions as callback functions that your application invokes when it receives a message matching a given message pattern. This section describes how to write the message action callback functions. “Creating and Registering Simple Message Patterns” on page 348 describes how to register these callback functions with the ViewKit message facility.

All message action callback functions must be of type VkMsgClientAction:

```
typedef Boolean (*VkMsgClientAction)(
  void*               /* clientData */,
  VkMsgFacilityReason /* reason */,
  VkMsgMessage        /* msg_in */,
  char*               /* op */,
  int                 /* argc */,
  VkMsgArg*           /* argv */
)
```

Note: The VkMsgClientAction function must be a regular function or a static member function; it cannot be a regular class member function.

The callback function arguments are:

clientData	The arbitrary client data you provided as the *clientData* argument when you registered this callback function using the **addAction()** function, as described in “Creating and Registering Simple Message Patterns” on page 348.
reason	The reason for calling the callback function, expressed as an enumerated value of type VkMsgFacilityReason. Possible values are: • VK_MSG_FACILITY_NOTICE—Notice • VK_MSG_FACILITY_REQUEST—Request • VK_MSG_FACILITY_REPLY—Reply to a previous request • VK_MSG_FACILITY_FAILED—Request failed (not handled) • VK_MSG_FACILITY_STARTED—Request caused autostart of the handler and was queued
msg_in	The incoming message in the form of an opaque message handle. You can use this handle when calling the various utility functions provided by the ViewKit message facility as described in “Useful Functions When Handling Messages” on page 346. For example, you might want to compare returned values against those you sent in your request.
op	The message operator
argc	The number of message arguments
argv	The message arguments, passed as an array of VkMsgArg structures.

The format of the VkMsgArg structure, used to pass the message arguments, is:

```
typedef struct {
  char *type;
  VkMsgValue value;
  VkMsgMode mode;
} VkMsgArg
```

The elements of the structure are:

type The argument type. This can take any of three pre-defined constant values:

- VK_MSG_INT_MSG_ARG_TYPE—integer value
- VK_MSG_STRING_MSG_ARG_TYPE—character string value
- VK_MSG_BSTRING_MSG_ARG_TYPE—binary string value

type The argument value, expressed as a VkMsgValue union. The definition of the VkMsgValue union is:

```
typedef union {
  int ival;
  char *val;
  VkMsgBValue bval;
} VkMsgValue
typedef struct {
  unsigned char *val;
  int len;
} VkMsgBValue
```

type The argument mode, expressed as an enumerated value of type VkMsgMode. Possible values are:

- VK_MSG_IN—the argument is written by the sending application and can be read by the handling application and any observing applications.
- VK_MSG_OUT—the argument is written by the handling applications and is read by the sending application.
- VK_MSG_INOUT—the argument can be written by both the sending and handling applications, and can be read by the sending, handling, or any observing applications.

Note: Remember that the ViewKit message facility dispatches the messages received by the application to *each* action registered for the matching operator, regardless of any other pattern information you provided when registering the actions. Your actions are responsible for testing the message arguments and determining whether or not to process the messages they receive.

Message action callbacks that process notices are relatively straightforward to write; the callback simply examines the message data and performs any actions required by the application. You can use the various utility functions provided by the ViewKit message facility, as described in “Useful Functions When Handling Messages” on page 346, to parse the message. Notice callbacks are not expected to send reply messages.

Message action callbacks that process requests must first decide whether or not to handle the request. If so, the callback should: 1) read any required data from the message’s “in” or “in / out” arguments; 2) perform any appropriate actions; 3) modify any “out” or “in / out” arguments; and 4) send the reply message using **VkMsgReply()**:

```
VkMsgStatus VkMsgReply(VkMsgMessage msg)
```

VkMsgReply() is implemented as a normal function rather than a class member function. It is declared in the header file <*Vk/VkMsg.h*>.

After performing any appropriate actions, your message action callback should return a Boolean value to indicate whether or not the ViewKit message facility should propagate the message to other callbacks registered for that action. A Boolean value of FALSE propagates the message; a value of TRUE does not propagate the message.

If a message action callback that processes a request decides to handle a request, it should return TRUE to prevent other message action callbacks from attempting to handle the request as well. If the callback decides not to handle a request, it should return FALSE to allow other callbacks to attempt to handle the message. If all of an application’s message actions reject a message, the ToolTalk service tries to dispatch the request to another application.

You should always return FALSE in message action callbacks that process notice messages.

The ViewKit message facility automatically destroys request messages when your application receives the corresponding reply or failure message. If for some reason you need to explicitly destroy a request message, call the **VkMsgDestroyRequest()** function:

```
VkMsgStatus VkMsgDestroyRequest(VkMsgMessage msg)
```

VkMsgDestroyRequest() is implemented as a normal function rather than a class member function. It is declared in the header file <*Vk/VkMsg.h*>. The VkMsgStatus return value is the same as the tt_status values used by ToolTalk; consult the *ToolTalk Programmer’s Guide* for information on these values.

Useful Functions When Handling Messages

The ViewKit message facility also provides several utility functions for manipulating messages. These functions are implemented as normal functions rather than class member functions. Most of these utility function are actually redefined ToolTalk functions. The ViewKit message facility provides this level of indirection to allow messaging services other than ToolTalk to be used. You should never directly call any of the ToolTalk routines in your application. Similarly, all of the ToolTalk constants (TT_*) have been replaced by ViewKit message facility equivalents (VK_MSG_*).

VkMsgTypeIsInt(), **VkMsgTypeIsString()**, and **VkMsgTypeIsBString()** check to see whether a given argument is an integer, a character string, or a binary string respectively:

```
Boolean VkMsgTypeIsInt(char *atype)
Boolean VkMsgTypeIsString(char *atype)
Boolean VkMsgTypeIsBString(char *atype)
```

The header file <*Vk/VkMsgUtils.h*> contains these declarations.

VkMsgSetIVal(), **VkMsgSetVal()**, and **VkMsgSetBVal()** change the integer, character string, or binary string value, respectively, of a given message argument:

```
VkMsgSetIVal(VkMsgMessage msg, int index, int value)
VkMsgSetVal(VkMsgMessage msg, int index, char *value)
VkMsgSetBVal(VkMsgMessage msg, int index,
             unsigned char *value, int len)
```

The header file <*Vk/VkMsg.h*> contains these declarations.

You can parse a message’s arguments into a VkMsgArg structure (as described in “Writing Message Action Callbacks” on page 342) with the **VkMsgParseArguments()** function:

```
void VkMsgParseArguments(VkMsgMessage msg, int *argc_return, VkMsgArg
**argv_return)
```

This function is declared in the header file <*Vk/VkMsgUtils.h*>. You must free the *argv* result when done.

You can retrieve the file attribute associated with a message using the **VkMsgFile()** function:

```
char *VkMsgFile(VkMsgMessage msg)
```

This function is declared in the header file <*Vk/VkMsg.h*>. You are responsible for freeing the returned value when you are finished using it. If there is no file attribute, **VkMsgFile()** returns NULL.

VkMsgIsErr() checks to see whether a VkMsgStatus value is an error status:

```
int VkMsgIsErr(VkMsgStatus status)
```

This function is declared in the header file <*Vk/VkMsg.h*>. If the VkMsgStatus value is a warning, **VkMsgIsErr()** returns 0; if it is an error **VkMsgIsErr()** returns 1.

VkMsgPtrError() determines whether a given opaque handle returned by a ViewKit message facility function is valid:

```
VkMsgStatus VkMsgPtrError(void *pointer)
```

This function is declared in the header file <*Vk/VkMsg.h*>. You can use this function to test for valid message handles or pattern handles. If the handle is valid, **VkMsgPtrError()** returns the constant VK_MSG_OK.

VkMsgFail() informs the ToolTalk service that your process cannot handle this request and that the message should not be offered to other processes of the same ptype as yours:

```
VkMsgStatus VkMsgFail(VkMsgMessage msg)
```

This function is declared in the header file <*Vk/VkMsg.h*>. The ToolTalk service sends the message back to the sender with state VK_MSG_FAILED.

VkMsgReject() informs the ToolTalk service that your process cannot handle this message:

```
VkMsgStatus VkMsgReject(VkMsgMesage msg)
```

This function is declared in the header file <*Vk/VkMsg.h*>. The ToolTalk service will try other handlers.

Creating and Registering Simple Message Patterns

To use your message actions, you must associate them with message patterns and then register those patterns with the ViewKit message facility.

Note: Be sure to register your application's message actions before executing **VkApp::run()** or posting any ViewKit dialog. You must register your message actions before entering any Xt event loop; otherwise your application might receive messages before registering message actions, and your application will not process the message as expected.

Use the **VkMsgClient::addAction()** function to create a message pattern and associate a message action with that pattern:

```
VkMsgPattern addAction(char *op, VkMsgClientAction proc,
                       void *clientData, VkMsgActionType type,
                       Boolean deleteMessage = TRUE)
```

addAction() creates and registers a simple message pattern consisting of only a message operator, *op*. You can create more detailed message patterns using the functions described in "Creating More Detailed Message Patterns" on page 349. After your application registers this pattern, it receives all messages sent that contain this operator. **addAction()** returns an opaque message pattern handle. You use this handle to remove this action with the **removeAction()** function described later in this section.

The *proc* argument is the callback function invoked when your application receives a message matching the message pattern. The function must be of type VkMsgClientAction. The ViewKit message facility passes the *clientData* argument to the function as client data. See "Writing Message Action Callbacks" on page 342 for information on writing message action callbacks.

The *type* argument specifies the type of message processing the message action implements. *type* is an enumerated value of type VkMsgActionType, which can take any of the following values:

VK_MSG_ACTION_OBSERVE
: Observe messages; use this value to process notices

VK_MSG_ACTION_HANDLE
: Handle messages; use this value to process requests

VK_MSG_ACTION_REPLY
    Process replies to a request

VK_MSG_ACTION_FAIL
    Process request failures

VK_MSG_ACTION_START
    Process notices of a message handler starting

The *deleteMessage* argument determines whether or not your application automatically deletes the message after all actions process it. By default, the ViewKit message facility automatically destroys the message and your message actions don't need to worry about it. However, if you want to save a copy of the message to deal with it later (for example, to send a reply), you must set *deleteMessage* to False.

After you've created your message patterns, you must call **VkMsgClient::updatePatterns()** to register those patterns:

```
void updatePatterns()
```

The **VkMsgClient::removeAction()** function removes an action:

```
void removeAction(VkMsgPattern pat)
```

pat is the message pattern returned by **addAction()**. **removeAction()** automatically unregisters the associated message pattern from the ViewKit message facility.

Creating More Detailed Message Patterns

The message pattern created by **addAction()** consists of only the message operator string. Any message received with that operator matches the particular message pattern and is dispatched to your action.

You can create more detailed message patterns, adding arguments and attributes, to restrict the messages that the ToolTalk service dispatches to your application. To create a more detailed pattern, you first create a basic message pattern using the **VkMsgClient::createAction()** function:

```
VkMsgPattern createAction(char *op, VkMsgClientAction proc,
                          void *clientData, VkMsgActionType type,
                          Boolean deleteMessage = TRUE)
```

createAction() accepts the same arguments as **addAction()**. Like **addAction()**, it creates a simple message pattern consisting of only a message operator, *op*, and associates the message action, *proc*, with that pattern. Unlike **addAction()**, **createAction()** does not automatically register the pattern with the ViewKit message facility. Instead, you can specify additional arguments or attributes to the message pattern using various pattern modifier functions.

All of the pattern modifier functions are implemented as normal functions rather than class member functions. They are actually redefined ToolTalk functions. The header file <*Vk/VkMsg.h*> contains these declarations. The ViewKit message facility provides this level of indirection to allow messaging services other than ToolTalk to be used. You should never directly call any of the ToolTalk routines in your application.

Consult the <*Vk/VkMsg.h*> header file for a list of pattern modifier functions you might find useful. Refer to the *ToolTalk Programmer's Guide* for information on these functions.

After modifying the message pattern, call **VkMsgClient::registerPattern()** to register the pattern with the ViewKit message facility:

```
VkMsgStatus registerPattern(VkMsgPattern pat)
```

pat is the opaque message pattern handle returned by **createAction()**. **registerPattern()** returns a VkMsgStatus value indicating its status. The VkMsgStatus values are the same as the tt_status values used by ToolTalk; consult the *ToolTalk Programmer's Guide* for information on these values.

After you've created your message patterns, you must call **VkMsgClient::updatePatterns()** to register those patterns:

```
void updatePatterns()
```

You can unregister a message pattern with the **VkMsgClient::unregisterPattern()** function:

```
VkMsgStatus unregisterPattern(VkMsgPattern pat)
```

pat is the opaque message pattern handle returned by **createAction()**. **unregisterPattern()** returns a VkMsgStatus value indicating its status. The VkMsgStatus values are the same as the tt_status values used by ToolTalk; consult the *ToolTalk Programmer's Guide* for information on these values.

As an example of creating a detailed message pattern, consider the following code:

```
pat = createAction("message_op", callback, this, VK_MSG_ACTION_HANDLE);
VkMsgPatternArg(pat, VK_MSG_IN, VK_MSG_INT_MSG_ARG_TYPE, NULL)
VkMsgPatternArg(pat, VK_MSG_OUT, VK_MSG_ALL_MSG_ARG_TYPE, NULL);
VkMsgPatternIArg(pat, VK_MSG_IN, VK_MSG_INT_MSG_ARG_TYPE, 5);
registerPattern(pat);
updatePatterns();
```

This example creates a message pattern that matches messages with the operator "message_op," and has three arguments. The first is an "in" integer argument. The second is an "out" argument of any type. The last is an "in" integer argument with value 5.

Detecting and Handling Errors in Handling Requests

There are two kinds of errors that can occur when a request is made and a reply is expected. Either no one handles the request (and ToolTalk could not autostart an appropriate service), or someone does handle the request and replies, but some error occurs while handling the request.

For instance, a request to "Raise the Mail Tool" would fail if there is no mail tool, since no reply could be received. In this case, if you had registered a VK_MSG_ACTION_FAIL type message action for the "Raise the Mail Tool" operator, your application would call that message action. If you were using **VkMsgClient::sendSyncRequest()**, it would return NULL.

If your request is successfully sent to a handler, but an error occurs while processing the request, most handlers send a reply but indicate that an error condition occurred. Many handlers return a status code indicating the return status.
To obtain the status code of a reply, call the **VkMsgStatusCode()** function:

```
int VkMsgStatusCode(VkMsgMessage msg)
```

Furthermore, some handlers provide useful error strings to display in the case of errors. To obtain the error string of a reply, call the **VkMsgStatusString()** function:

```
char *VkMsgStatusString(VkMsgMessage msg)
```

The meaning of status codes and the validity of status strings is dependent on the service handling the request. Both of these functions are implemented as a normal functions rather than class member functions. They are actually redefined ToolTalk functions. The header file <*Vk/VkMsg.h*> contains these declarations.

Supporting Messaging in Application Windows

The **VkMsgWindow** class is a subclass of **VkWindow** that works with **VkMsgApp** to support the ViewKit interprocess message facility. You should use **VkMsgWindow** for your application's windows instead of **VkSimpleWindow** or **VkWindow** if you want ToolTalk support for your application.

The **VkMsgWindow** constructor creates an instance of **VkMsgClient** that you can use to manage messages in that window. You can access the window's **VkMsgClient** object with the **VkMsgWindow::messageClient()** function:

```
VkMsgClient *messageClient()
```

VkMsgWindow also provides a variety of convenience functions for directly manipulating a window's **VkMsgClient** object. Consult the VkMsgWindow(3Vk) reference page for more information on these functions.

Supporting Messaging in Components

The **VkMsgComponent** class is a subclass of **VkComponent** that works with **VkMsgApp** to support the ViewKit interprocess message facility. You should use **VkMsgComponent** to derive new components if those components must send or receive ToolTalk messages. You do not need to derive components from **VkMsgComponent** if those components do not interact with ToolTalk. Furthermore, you usually should handle the ToolTalk interaction in an application through the application's **VkMsgApp** object or one of its **VkMsgWindow** objects.

The **VkMsgComponent** constructor creates an instance of **VkMsgClient** that you can use to manage messages in that component. You can access the window's **VkMsgClient** object with the **VkMsgWindow::messageClient()** function:

```
VkMsgClient *messageClient()
```

VkMsgComponent also provides a variety of convenience functions for directly manipulating a component's **VkMsgClient** object. Consult the VkMsgComponent(3Vk) reference page for more information on these functions.

Registering Services for Autostart

For some messages, you might want a service process to get autostarted as necessary. When the message is sent and the process is not available, the ToolTalk service starts the process and queues the message.

To set this up, you need to register your application class with what ToolTalk calls a "ptype," a process type. Refer to the *ToolTalk Programmer's Guide* for details, but in most situations the setup is similar. For example, you might have a file called *myapp.ptype* that contains:

```
ptype USR_MY_APP
{
  start "/usr/sbin/myApp";
  observe: session load_file() => start;
}
```

This indicates that the message "load_file" is observed by a process typed "USR_MY_APP" and that when a "load_file" message is sent, the ToolTalk service should execute "/usr/sbin/myApp."

Then, you instantiate the **VkMsgApp** object in the *myApp* application as follows:

```
VkMsgApp *myApp = new VkMsgApp("MyApp", &argc, argv, NULL, 0,
                               "USR_MY_APP");
```

Finally, when you install your application, you need to register this information in the static ToolTalk config file by executing:

```
/usr/sbin/tt_type_comp -dsystem myapp.ptype
/etc/killall -USR2 ttsession
```

This adds the contents of *myapp.ptype* to the system config file, and tells all existing ttsession processes to update their configurations.

When you express message patterns in a ptype, you should not register the pattern in your application. Otherwise, your message handler will be called twice. To register an action but not register the corresponding pattern, use **createAction()** instead of **addAction()**.

Troubleshooting Checklist

Here are some common mistakes to watch out for:

- When using ptype-registered patterns for autostart, make sure your executable has registered its ptype name in **VkMsgApp::VkMsgApp()** and that it has a non-specific action with no pattern (use **VkMsgClient::createAction()** instead of **addAction()**) to handle the request. Otherwise, ToolTalk autostarts your executable indefinitely.
- If you don't set the optional *deleteMessage* argument to TRUE in **VkMsgClient::addAction()**, you should never call **VkMsgDestroy()** or **VkMsgDestroyRequest()** in your message action.
- If you set the optional *deleteMessage* argument to FALSE in **VkMsgClient::addAction()**, you're responsible for destroying the message when you're through with it. If the message is a request, be sure to use **VkMsgDestroyRequest()** instead of **VkMsgDestroy()**. Otherwise, you may have memory heap corruption caused by double frees, if the requestor and handler are in the same process.
- You must call **VkMsgClient::updatePatterns()** after you use **VkMsgClient:::addAction()**. Otherwise, your new actions have no effect.
- When using the **VkMsgClient::composeAdd()** routines, be careful that you call **VkMsgClient::composeBegin()** first to reset the state of your composed arguments. Otherwise, random arguments will get added to your message.
- If you return TRUE from a message action where the reason was VK_MSG_ACTION_HANDLE, be sure that you call **VkMsgReply()**. Otherwise, ToolTalk assumes that your process is handling the message when in fact you never reply. The requestor waits forever for a response.

Also, it is useful to turn on *ttsession* (the ToolTalk server process) debugging output. You can toggle it on / off with:

```
/etc/killall -USR1 ttsession
```

Appendix B

Contributed ViewKit Classes

This appendix gives you an idea of how you can expand ViewKit by describing some ViewKit classes that users have contributed. These classes are not supported by Silicon Graphics and their interfaces might change in future ViewKit releases.

The ViewKit Meter Component

The **VkMeter** class supports simple compound bar charts, displayed in either vertical or horizontal mode. If you display multiple values, the data is presented in layers, with the bar representing the second value starting where the first value ends.

Meter Constructor and Destructor

The **VkMeter** accepts the standard ViewKit component constructor arguments: a component name and a parent widget:

```
VkMeter(const char *name, Widget parent)
```

You should rarely need to create subclasses of **VkMeter**.

The **VkMeter** destructor frees all space associated with the meter:

```
~VkMeter()
```

Resetting the Meter

Before adding any items for display to a **VkMeter** object, you must call **VkMeter::reset()** to reset the meter:

```
void reset(int peak = -1)
```

The first value, *peak*, sets the initial *peak value* displayed by the meter. All items displayed by the meter are scaled relative to the peak value. For example, if the peak value is 200 and one of your items is 40 units long, that item will be scaled to take 20% of the meter's total length. The default peak size is 100 units.

Note: To change meter values or otherwise update a meter object, you must call **reset()** and then add the items to the meter again.

Adding Items to a Meter

You add items for a **VkMeter** object to display with **VkMeter:add()**:

```
void add(int value, char *color)
void add(int value, Pixel pixel)
void add(int value, int width, char *color)
void add(int value, int width, Pixel pixel)
```

The *value* argument is the item's value. When displayed, the **VkMeter** class scales this value relative to the peak value set by **reset()**. For example, if the peak value is 500 and one of your items is 80 units long, that item will be scaled to take 16% of the meter's total length.

When you use these forms of the **add()** function, the **VkMeter** object displays the items sequentially. For example, if you have set the peak value to 100 and you add three items with values of 20, 10, and 30 in that order, the meter displays three bars: the first ranging from 0 to 20, the second from 20 to 30, and the third from 30 to 60.

All data items must have an associated color. You can specify the color as a Pixel value, *pixel*, or as a string, *color*. If you provide a string, **add()** first treats the string as the name of a resource that **add()** looks up relative to the component and converts to the desired color. If **add()** finds no such resource, it uses the string itself as the name of a color. For example, the following adds an item with the color "red":

```
add(10, "red");
```

The following adds an item with the color specified by the resource name "criticalColor":

```
add(20, "criticalColor");
```

You can specify the width of an item by providing a *width* argument, expressed in pixels. If you do not provide a width, the width of the item is the same as the width of the meter.

Two more complex forms of **add()** allow you to precisely control the position of bars in a meter, and even display bars side by side:

```
void add(int start, int size, int sideValue, int width, char *color)
void add(int start, int size, int sideValue, int width,
         Pixel color)
```

In these forms of **add()**, the first value, *start*, specifies the starting position of the bar, and the second value, *size*, specifies the size (length) of the bar. **VkMeter** scales these values relative to the peak value set by **reset()**. The third argument, *sideValue*, and the fourth argument, *width*, specify values in the opposite dimension. **VkMeter** does not scale these values relative to the meter's peak value.

For example, consider a meter with a peak value of 100. The following lines add four bars to the meter:

```
add(0, 20, 0, 10, "red");
add(0, 20, 10, 10, "blue");
add(0, 20, 20, 10, "green");
add(20, 20, 0, 30, "yellow");
```

If you display this meter vertically, it shows three vertical bars ranging from 0 to 20 side by side in red, blue, and green. Above them is a yellow bar spanning all of them and ranging from 20 to 40.

Updating the Meter Display

After adding all items to a meter, call the **VkMeter::update()** function to update the meter's display:

```
void update()
```

Note: Remember that if you want to change the meter display, you must first call **reset()** and then add each item in the new display.

Setting the Meter's Resize Policy

The meter you create can have either a fixed size or it can attempt to resize itself dynamically as it requires more or less room to display the items it contains. You can specify the meter's resize policy with **VkMeter::setResizePolicy()**:

```
void setResizePolicy( unsigned char policy )
```

You can provide any of the following values:

XmRESIZE_NONE
: The meter never attempts to resize itself. The application, or managing widget, is in complete control of the meter's size.

XmRESIZE_GROW
: The meter calls **XtSetValues()** on the widget used to display the meter to attempt to grow as needed. The success of the call to **XtSetValues()** depends on the parent widget's geometry management policy.

XmRESIZE_ANY
: The meter calls **XtSetValues()** on the widget used to display the meter to attempt to grow or shrink as needed. The success of the call to **XtSetValues()** depends on the parent widget's geometry management policy.

Determining the Desired Dimensions of the Meter

You can determine the dimensions that a meter needs to display itself completely by calling **VkMeter::neededWidth()** and **VkMeter::neededHeight()**:

```
Dimension neededWidth()
Dimension neededHeight()
```

X Resources Associated with the Meter Component

The following X resources are associated with the **VkMeter** class:

XmNorientation
: Determines the orientation of the meter. The default value is XmVERTICAL which specifies a vertical meter. Set the value of the resource to XmHORIZONTAL for a horizontal meter.

XmNresizePolicy
: Determines the resize policy of the meter, as described in "Setting the Meter's Resize Policy" on page 357. The default value is XmRESIZE_NONE.

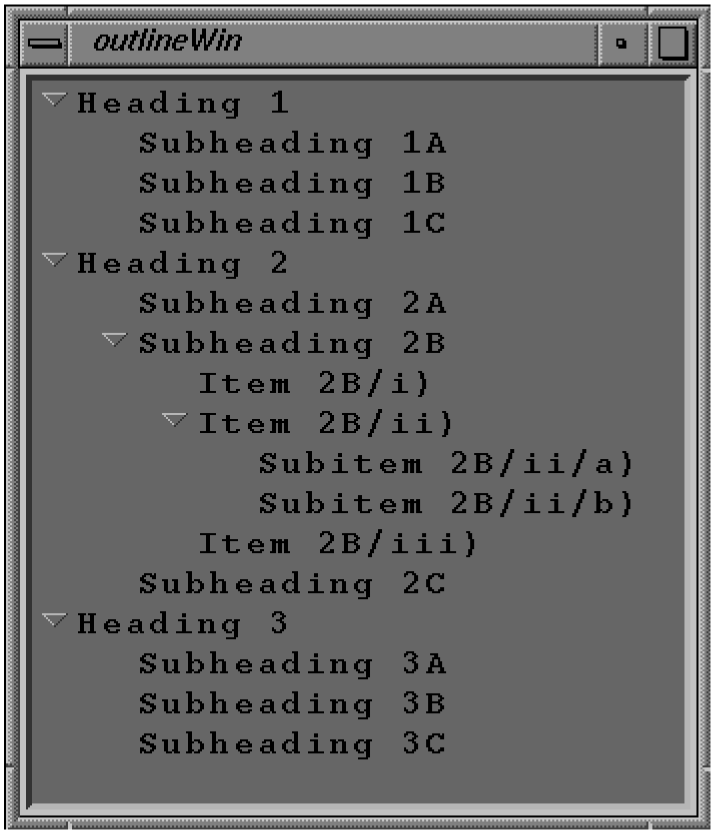
XmNdrawBorder
: Determines whether bars are drawn with borders. The default value is FALSE, in which case bars do not have borders. If you set the value to TRUE, bars have borders drawn in the color specified by the XmNborderColor resource.

The ViewKit Pie Chart Component

The **VkPie** class is derived from **VkMeter** and displays data in the same way as that class. However, rather than displaying the values as a bar chart, the **VkPie** class displays the data as a pie chart. See the description of **VkMeter** for details.

The ViewKit Outline Component

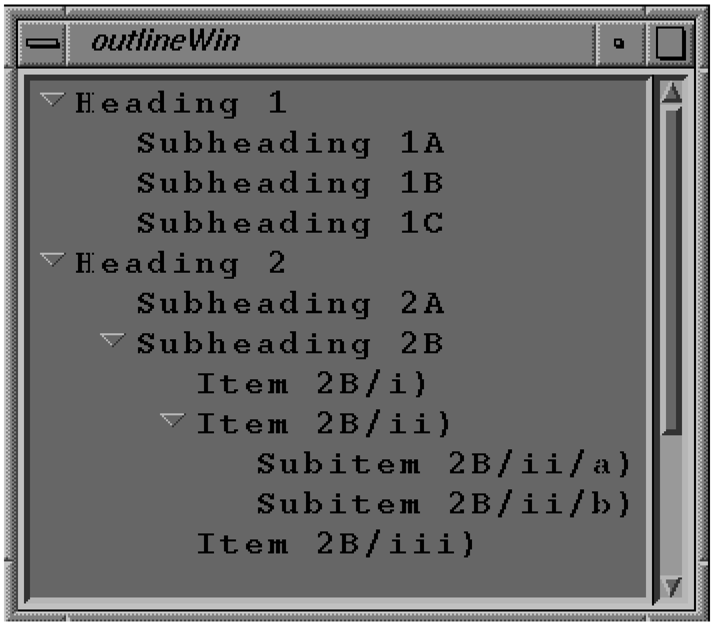
The **VkOutline** component, derived from **VkComponent**, displays a textual outline. **VkOutline** automatically indents items according to their depth in the outline. Figure B-1 shows an example of a **VkOutline** component containing three top-level items, each with several subitems.



```
▽Heading 1
    Subheading 1A
    Subheading 1B
    Subheading 1C
▽Heading 2
    Subheading 2A
  ▽ Subheading 2B
        Item 2B/i)
      ▽ Item 2B/ii)
            Subitem 2B/ii/a)
            Subitem 2B/ii/b)
        Item 2B/iii)
    Subheading 2C
▽Heading 3
    Subheading 3A
    Subheading 3B
    Subheading 3C
```

Figure B-1 Example of a **VkOutline** Component

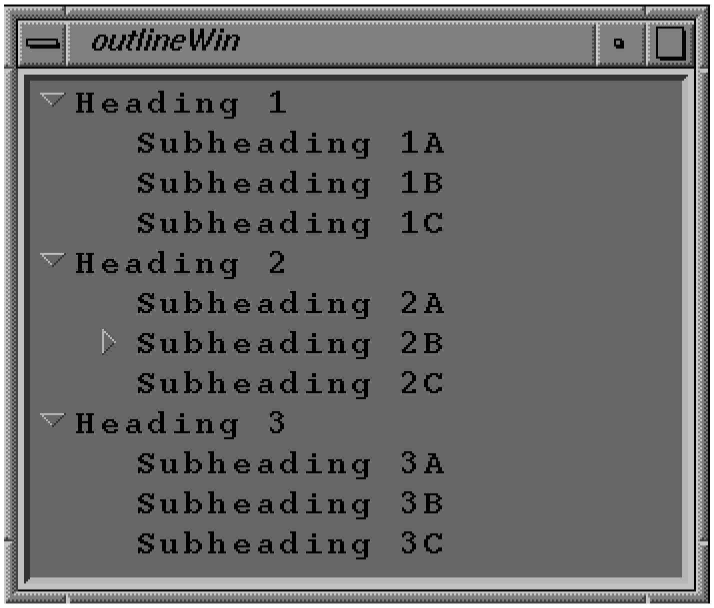
If there is not sufficient space to display the entire outline, the **VkOutline** component automatically displays a scrollbar, as shown in Figure B-2.



```
Heading 1
    Subheading 1A
    Subheading 1B
    Subheading 1C
Heading 2
    Subheading 2A
    Subheading 2B
        Item 2B/i)
        Item 2B/ii)
            Subitem 2B/ii/a)
            Subitem 2B/ii/b)
        Item 2B/iii)
```

Figure B-2 Example of a **VkOutline** Component with the Scrollbar Visible

The **VkOutline** component displays a control icon to the left of each outline item that contains subitems. The control icon denotes whether the sub-tree under the item is displayed (open) or not (closed). The user can click the left mouse button on the control icon to toggle between the open and closed states. Figure B-3 shows the results of closing the item “Subheading 2B,” shown in the previous figure.



Heading 1
    Subheading 1A
    Subheading 1B
    Subheading 1C
Heading 2
    Subheading 2A
    Subheading 2B
    Subheading 2C
Heading 3
    Subheading 3A
    Subheading 3B
    Subheading 3C

Figure B-3 Example of Closing a Heading in a **VkOutline** Component

Constructing an Outline Component

The **VkOutline** constructor accepts the standard ViewKit component constructor arguments: a component name and a parent widget:

`VkOutline (const char *`*name*`, Widget `*parent*`)`

Adding Items to an Outline

You can add items to the outline in a simple parent-child relation with **VkOutline::add()**:

`void add(char* `*parentName*`, char* `*childName*`)`

The actions performed by **add()** depend on whether either or both of the items already exist in the outline.

- If both items already exist in the outline, **add()** does nothing.
- If neither exists, **add()** creates *parentName* as a top-level item in the outline and then creates *childName* as a subitem of *parentName*.
- If *parentName* already exists but *childName* does not, **add()** creates *childName* as a subitem of *parentName*.
- If *childName* exists and *parentName* does not, and *childName* is a top-level item, **add()** "reparents" *childName* by adding *parentName* as a top-level item and moving *childName* in the outline so that it is a subitem of *parentName*.
- If *childName* exists and *parentName* does not, but *childName* is not a top-level item, **add()** does nothing.

parentName and *childName* are used both as item names and the text displayed in the outline. Note that you must use unique names for each item in the outline.

You can add multiple subitems to an existing item using **VkOutline::addChildren()**:

```
void addChildren(char** parentPath, char** childNames)

void addChildren(char** parentPath, char** childLabels,
                 char** childNames, void** childData)
```

The character string array *parentPath* specifies the complete path of the parent item through the outline. The first element of the *parentPath* array is the name of the top-most item of the outline containing the specified item, the second element is the name of the second-highest item, and so on, with the name of the item itself appearing last. You must NULL-terminate the array.

The character string array *childNames* contains the names of the subitems to add to the specified parent item. Note that you must use unique names for each item in the outline.

In the second form of **addChildren()**, you can provide *childLabels*, an array of character strings that provide display labels for the subitem you add. **VkOutline** displays these labels for the items instead of the item names.

In the second form of **addChildren()**, you can also provide *childData*, an array of pointers to arbitrary data. You can retrieve a pointer to the data associated with an item using **VkOutline::getHookAt()**, described in "Outline Utility and Access Functions" on page 367. Usually you need to use this data only if you create a subclass of **VkOutline**. In a subclass, you can add callbacks so that when the user selects an outline item, you can retrieve the data associated with that item and perform some action.

VkOutline::createPath() creates or extends a path in the outline:

```
void createPath(char** itemLabels, char** itemNames)
```

The character string array *itemNames* specifies a path through the outline. The first element of the *itemNames* array is the name of the top-most item of the outline containing the specified item, the second element is the name of the second-highest item, and so on, with the name of the item itself appearing last. You must NULL-terminate the array.

If path does not exist, then **createPath()** creates a new set of items with the first element in the path as the top-level item, the second element a subitem of the first, and so on. If **createPath()** finds a partial match in the existing outline, where the first element of *itemNames* matches the name of an existing top-level item and one or more lower-level items match succeeding elements of *itemNames*, **createPath()** adds those items needed to fully extend the path.

For those items that **createPath()** adds, it uses the corresponding elements from the *itemLabels* character string array as the display labels for those items. **VkOutline** displays these labels for the items instead of the item names.

Note: **createPath()** does not alter the labels for any existing items. **createPath()** uses the labels only when adding new items.

Whenever you add items to the outline, no matter which function you use to add them, you must call **VkOutline::displayAll()** to update the outline display:

```
void displayAll()
```

Setting Display Attributes for Outline Items

VkOutline allows you to designate items as “keywords” and display them in a different foreground color, background color, and / or font. You can also define up to four custom item highlights, each with its own foreground color, background color, and font attributes.

Use **VkOutline::setKeywordAttributes()** to define the keyword display attributes:

```
void setKeywordAttributes(Pixel fg, Pixel bg, XmFontList font)
```

fg is the foreground color for the item’s text. *bg* is the background color for the item. *font* is the font used to display the item’s text.

Use **VkOutline::displayAsKeyword()** to display an item with the keyword display attributes:

```
void displayAsKeyword(char** path)
```

You specify the complete path of the item through the outline as an array of character strings. The first element of the *path* array is the name of the top-most item of the outline containing the specified item, the second element is the name of the second-highest item, and so on, with the name of the item itself appearing last. You must NULL-terminate the array. Note that **displayAsKeyword()** requires the item names, not their display labels.

Use **VkOutline::setHighlightAttributes()** to define the display attributes of a custom highlight:

```
int setHighlightAttributes(Pixel fg, Pixel bg, XmFontList font)
```

fg is the foreground color for the item’s text. *bg* is the background color for the item. *font* is the font used to display the item’s text. **setHighlightAttributes()** returns an integer identifier for the highlight. You use this identifier to apply the highlight to outline items with the **highlight()** function described below. If **setHighlightAttributes()** could not allocate a custom highlight, it returns 0.

Use **VkOutline::highlight()** to display one or more items with display attributes of a custom highlight:

```
void highlight(int itemPos, int attribID)
void highlight(char** items, int attribID)
```

In the first form of **highlight()**, you specify the position index in the outline of the item you want to highlight. Items are numbered sequentially from the top of the outline starting with zero. *attribID* is the attribute identifier returned by **setHighlightAttributes()** of the custom highlight that you want to assign to the items.

In the second form of **highlight()**, *items* is an array of strings specifying the names of the items to highlight. Note that **highlight()** requires the item names, not their display labels. Again, *attribID* is the attribute identifier (returned by **setHighlightAttributes()**) of the custom highlight that you want to assign to the items.

You cannot remove a custom highlight from individual items; you can only remove the highlight from all items to which you have applied it. **VkOutline::unhighlight()** removes a custom highlight:

```
void unhighlight(int attribID)
```

attribID is the attribute identifier (returned by **setHighlightAttributes()**) of the custom highlight that you want to assign to the items.

Closing and Opening Outline Topics

You can programmatically toggle an outline item open or closed with **VkOutline::toggleChildren()**:

```
virtual void toggleChildren(int position)
```

position is the item's position in the SgList widget. Items are numbered sequentially from the top of the outline starting with zero.

You can determine the effects of the last toggle operation, whether a result of user interaction or a call to **toggleChildren()**, by calling **VkOutline::effectOfLastToggle()**:

```
int effectOfLastToggle(int& from, int& count)
```

If the last toggle operation opened an item (and therefore inserted items into the SgList widget), **effectOfLastToggle()** returns 1, sets the value of *from* to the position of the toggled item in the list, and sets the value of *count* to the number of items displayed by opening the item. If the last toggle operation closed an item (deleting items from the SgList widget), **effectOfLastToggle()** returns 0, sets the value of *from* to the position of the toggled item in the list, and sets the value of *count* to the number of items deleted from the list by closing the item.

You can determine whether a given item is closed with **VkOutline::isPathClosed()**:

```
int isPathClosed(char** path)
```

The character string array *path* specifies the complete path of the item through the outline. The first element of the *path* array is the name of the top-most item of the outline containing the specified item, the second element is the name of the second-highest item, and so on, with the name of the item itself appearing last. You must NULL-terminate the array.

isPathClosed() returns 1 if the item is closed, 0 if the item is open, and -1 if the item has no subitems.

Outline Utility and Access Functions

VkOutline provides the following utility and access functions:

```
void setIndentationWidth(int width)
```

VkOutline::setIndentationWidth() sets indentation width for future displays. The indentation width is the number of pixels to the right that the outline offsets a child item from its parent item.

```
void printTree()
```

VkOutline::printTree() prints the outline on the application's standard output.

```
void reset()
```

VkOutline::reset() re-initializes the outline, deleting all items. **reset()** retains any display attributes you created.

```
Widget listWidget()
```

VkOutline::listWidget() returns the widget ID of the SgList widget that the **VkOutline** uses to display the outline. Consult the SgList(3x) reference page for more information on the SgList widget.

```
void select(int position)
```

VkOutline::select() selects the string displayed at the given position of the SgList widget.

```
void getHookAt(int position)
```

VkOutline::getHookAt() retrieves the pointer to the data associated with an item given the item's position in the SgList widget. This is the data that you provided as the *childData* argument to **addChildren()** (see "Adding Items to an Outline" on page 362).

Usually, you need to use this data only if you create a subclass of **VkOutline**. In a subclass, you can add callbacks to the SgList widget so that when the user selects an outline item, you can retrieve the data associated with that item and perform some action.

VkOutlineASB

The **VkOutlineASB** class, a subclass of **VkOutline**, provides the same functionality as **VkOutline** except that it uses an annotated scrollbar. With **VkOutlineASB**, you can display colored bars in the scrollbar to indicate the positions of highlighted items in the outline.

All functions that **VkOutlineASB** inherits from **VkOutline** operate identically. **VkOutlineASB** provides one additional function, **VkOutlineASB::setAnnotation()**:

```
void setAnnotation(int attribID, Boolean state)
```

setAnnotation() determines whether or not the scrollbar displays annotations for a given display highlight. *attribID* is the attribute identifier returned by **setHighlightAttributes()** of a particular custom highlight. If *state* is TRUE, the scrollbar displays annotations for the given display highlight; if *state* is FALSE, the scrollbar does not display annotations for the given display highlight.

Appendix C

Using a Help System With ViewKit

This appendix describes how to use a help system with ViewKit applications. It covers:

- the user interfaces through which a user can obtain help
- ViewKit's programmatic interface (API) to external help libraries and instructions for creating ViewKit-compatible help libraries
- ViewKit's own simple help capability
- Additional support provided by ViewKit for creating help information for your applications

ViewKit also provides message line and popup QuickHelp. See "QuickHelp" on page 113 for more information.

User Interfaces to the Help System in a ViewKit Application

In a ViewKit application, the user can invoke the help system three different ways: 1) by pressing the **`<F1>`** key while the mouse pointer is over a widget, 2) by clicking the *Help* button in a dialog, or 3) by choosing an item from the Help menu.

The **`<F1>`** key is a standard IRIS IM method of invoking help. ViewKit applications provide an additional layer of interpretation to allow you to integrate this method of invoking help with the other methods provided by ViewKit. Rather than installing XmNhelpCallback functions on widgets, use ViewKit's programmatic interface to the help libraries as described in "ViewKit's Programmatic Interface to a Help Library" on page 370.

ViewKit dialogs also allow you to include a *Help* button as described in "Posting Dialogs" on page 188. The Help menu, implemented by the **VkHelpPane** class, also allows the user to invoke the help system. "ViewKit's Programmatic Interface to a Help Library" describes how to link these interfaces to a help system.

ViewKit’s Programmatic Interface to a Help Library

ViewKit applications interact with a help library through three C functions: **SGIHelpInit()**, **SGIHelpMsg()**, and **SGIHelpIndexMsg()**. To interface a ViewKit application to a help system, you need to implement only these three functions. ViewKit also provides a rudimentary help capability if you do not want to implement your own. “ViewKit Help” on page 372 describes this capability.

SGIHelpInit() initializes the help system:

```
int SGIHelpInit(Display *display, char *appClass, char *)
```

VkApp calls **SGIHelpInit()** from its constructor. *display* is the application’s Display structure, and *appClass* is the application’s class name. The third argument to **SGIHelpInit()** is reserved for future Silicon Graphics use. A return value of 0 indicates failure.

A ViewKit application calls **SGIHelpMsg()** when it needs to request help:

```
int SGIHelpMsg(char *in_key, char *, char *)
```

in_key is a character token that **SGIHelpMsg()** uses to look up help material. The value of *in_key* depends on how the user requested help. The subsections that follow describe how the value is determined. The other arguments to **SGIHelpMsg()** is reserved for future Silicon Graphics use. A return value of 0 indicates failure.

A ViewKit application calls **SGIHelpIndexMsg()** to display an index of help available:

```
int SGIHelpIndexMsg(char *in_key, char *)
```

in_key is a character token that **SGIHelpIndexMsg()** uses to look up a help index. The value of *in_key* depends on how the user requested help. The subsections that follow describe how the value is determined. The other argument to **SGIHelpIndexMsg()** are reserved for future Silicon Graphics use. A return value of 0 indicates failure.

Dialog Help Procedures

When you post a dialog as described in "Posting Dialogs" on page 188, you have the option of providing a *helpString* argument. If you provide a *helpString* argument, the dialog posted displays a *Help* button.

When the user clicks the *Help* button, your application calls **SGIHelpMsg()**, passing the *helpString* as the *in_key* character token.

Context-Sensitive Help Procedures

When the user presses the **`<F1>`** key while the mouse pointer is over a widget, as long as you have not provided XmNhelpCallback functions for widgets in your application, your application calls **SGIHelpMsg()**. The *in_key* character token that your application provides to **SGIHelpMsg()** is the fully qualified instance name hierarchy for the widget.

Help Menu Procedures

As noted in "The ViewKit Help Menu" on page 165, the Help menu implemented by the **VkHelpPane** class contains five selectable items: "Click for Help," "Overview," "Index," "Keys & Shortcuts," and "Product Information"

"Click for Help" provides another method of obtaining context-sensitive help. When the user chooses this item, the cursor changes into a question mark. The user can then click any widget in the application, which calls **SGIHelpMsg()**. The *in_key* character token that your application provides to **SGIHelpMsg()** is the fully qualified instance name hierarchy for the widget.

"Overview" calls **SGIHelpMsg()** to request overview help. If the **VkHelpPane** object is a descendant of a shell widget, the *in_key* character token that your application provides to **SGIHelpMsg()** is "*shellName*.overview" where *shellName* is the name of the shell widget. In most cases, the **VkHelpPane** object is a pane in a window's menu bar, so the *shellName* is the name of the window. If the **VkHelpPane** object is not a descendant of a shell widget, the *in_key* character token that your application provides to **SGIHelpMsg()** is simply "overview."

"Index" calls **SGIHelpIndexMsg()** to request an index of available help topics. If the **VkHelpPane** object is a descendant of a shell widget, the *in_key* character token that your application provides to **SGIHelpIndexMsg()** is "*shellName*.index" where *shellName* is the name of the shell widget. In most cases, the **VkHelpPane** object is a pane in a window's menu bar, so the *shellName* is the name of the window. If the **VkHelpPane** object is not a descendant of a shell widget, the *in_key* character token that your application provides to **SGIHelpIndexMsg()** is simply "index."

"Keys & Shortcuts" calls **SGIHelpMsg()** to request help on keys and shortcuts. If the **VkHelpPane** object is a descendant of a shell widget, the *in_key* character token that your application provides to **SGIHelpMsg()** is "*shellName*.keys" where *shellName* is the name of the shell widget. In most cases, the **VkHelpPane** object is a pane in a window's menu bar, so the *shellName* is the name of the window. If the **VkHelpPane** object is not a descendant of a shell widget, the *in_key* character token that your application provides to **SGIHelpMsg()** is simply "keys."

"Product Information" displays the Product Information dialog described in "Maintaining Product and Version Information" on page 74. The Product Information dialog has no connection to any help library.

ViewKit Help

The ViewKit library, *libvk*, includes a simple help capability that allows you to include help messages for your application by defining them in the X resource database. This may be sufficient for your needs.

ViewKit help is simple enough not to require any initialization, so **SGIHelpInit()** is defined to simply return the value 1.

Both **SGIHelpMsg()** and **SGIHelpIndexMsg()** are defined to accept the *in_key* character token argument and look up the resource "*in_key*.helpText" in the X resource library. They then display the help text retrieved from the resource database in an IRIS IM information dialog. If these functions cannot find an appropriate resource value, they display the message `Sorry, no help available on this topic` in the dialog.

The following lines show how you create the help message specifications for an application:

```
*helpText:             Application default help message
*row1*helpText:        Help message for the row1 widgets and its descendants
*row2*helpText:        Help message for the row2 widgets and its descendants
*row2*start*helpText:  Special help message for start, a child widget of row2
*overview*helpText:    Overview help message
```

In this example, the “*helpText” resource specification provides a default help message for the entire application. If a widget does not have a more specific help message resource specification, the application displays this default help message.

The “*row1*helpText” and “*row2*helpText” resource specifications provide help messages for these widgets and their descendants. For example, you could use a specification like this to provide a help message for a group of toggles or pushbuttons in a RowColumn widget.

The “*row2*start*helpText” specification provides a help message for a “start” widget, a descendant of the “row2” widget. It overrides the “*row2*helpText” message.

“*overview*helpText” provides a message that the application displays when the user chooses “Overview” from the Help menu. You can also implement your own help library. Since ViewKit predefines **SGIHelpInit()**, **SGIHelpIndexMsg()**, and **SGIHelpMsg** as weak symbols, they are overridden by your library, and no conflict ensues.

If you decide to write your own help library, you can examine the source for ViewKit’s help functions to get some ideas. The source is included in */usr/share/src/ViewKit/Utilities/VkHelpAPI.c++*. You can examine this source to get ideas for implementing your own help library.

ViewKit Support for Building Help

The default ViewKit help capability also provides support for determining the token strings passed to the help system. To use this feature, you must not link with any other help library. After you determine all of the token strings you need, you can then link with your chosen help library to provide the final help system for your application.

To determine the token strings, set the “*helpAuthorMode” resource for your application to TRUE. Then, when you run your application, whenever the help system would normally display a help message, it instead displays the token string passed to the help system.

Glossary

animated busy cursor

A cursor that is a sequence of pixmaps you can cycle through while in a busy state, giving the appearance of animation.

attachments

Management classes that control the operation of components and widgets.

base widget

The root of a widget subtree.

busy states

When you lock out user input during an operation.

butterfly node

The central node of a butterfly graph.

butterfly graphs

Tree graphs that display only a central node and its immediate parent and child nodes.

command classes

Classes that allow you to implement actions as objects.

components

A component encapsulates a collection of widgets, but also defines the behavior of the overall component.

fixed busy cursor

A cursor that retains the same appearance throughout a busy state.

homogenous group

A group that contains only one type of preference item.

main window

The first window created in every application is by default treated as the main window.

non-homogenous group

A group that contains more than one type of preference item.

peak value

The initial value in a meter object.

preference dialogs

A dialog box that allows the user to customize the behavior of an application.

view

A widget or ViewKit component that you use as your work area for the XmMainWindow widget.

ViewKit callbacks

A mechanism that allows a component to define conditions or events, the names of which are exported as public static string constants encapsulated by that component.

Index

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-2124-003.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 415-965-0964
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389