In this Chapter:

- *A Mega-Widget Set*
- *[incr Widgets] Basics*
- *[incr Widgets] Tour*
- *Example Application*
- *Contributing to [incr Widgets]*
- *[incr Widgets] Distribution*
- *Acknowledgments*

6

[incr Widgets]

A Mega-Widget Set

Tcl/Tk GUI development is filled with patterns. Programmers usually find themselves creating the same portions of code over and over again. The problem is that the Tk widget set is fairly basic. Repetitiously, we add labels to our entry widgets and scrollbars to listboxes, canvases, and text widgets. In fact, those are relatively simple examples--consider the amount of code we replicate making various dialogs such as file selection and prompt dialogs. These are much more complicated and definitely more error prone. Seasoned coders try and bundle up these code segments, centralizing the functionality, creating composite widgets in a set of functional procedures. This strategy gets the job done, but's clunky and inflexible, lacking the encapsulation and cohesiveness of a standard Tk widget. Absent also is the Tk framework of commands such as *configure* and *cget*. The end result just doesn't seem like a Tk widget.

With the introduction of the [incr Tcl] and [incr Tk] extensions, this problem has been made much easier. They allow Tk widgets to be combined into patterns of higher level building blocks called "Mega-Widgets". [incr Tcl] allows us to move away from the object-based model of Tcl/Tk and into an object-oriented paradigm similar to that of C++ complete with inheritance. Data and commands can be encapsulated. Implementation can be hidden. The [incr Tk] extension goes even further, providing the Tk framework of configuration options and basic widget commands such as *configure* and *cget*. These are the kind of features that make prototyping really easy and allow you to create code quicker. In the simplest of terms, each mega-widget based on [incr Tcl] and [incr Tk] seamlessly blends with the standard Tk widgets. They look, act, and feel like Tk widgets.

[incr Widgets] is a set of mega-widgets based on [incr Tcl] and [incr Tk]. It delivers many general purpose widgets like option menus, selection boxes, and various dialogs whose counterparts are found in Motif and Windows. These mega-widgets replace many of most common patterns of widget combinations with higher level abstractions, blending with the standard Tk widgets, and making it easier to consistently develop well styled applications. [incr Widgets] is also distinguished by its consistent use of style, built-in intelligence, high degree of flexibility, ease of extension, and object-oriented implementation. Its use results in increased productivity, reliability, and style guide adherence. This chapter details the [incr Widgets] mega-widget set and illustrates its innovative concepts.

[incr Widgets] Programming Advantages

[incr Widgets] lets you create complex interfaces in substantially fewer lines of code. Not only can you work faster, but the resulting code is more readable and therefore easier to maintain. Let's take a look at a specific example, comparing the construction of a typical data entry interface using just Tcl/Tk and then again using [incr Widgets].

Using Tcl/Tk an entry form for name, address, and phone number would be constructed using combinations of frames, labels, and entry widgets. The frames would be needed to organize each entry field.

Example 6-1.

```
wm title . Form

frame .name
pack .name -fill x
label .name.label -text Name:
pack .name.label -side left
entry .name.entry
pack .name.entry -fill x

frame .address
pack .address -fill x
label .address.label -text Address:
pack .address.label -side left
entry .address.entry
pack .address.entry -fill x

frame .citystate
pack .citystate -fill x
label .citystate.label -text "City, State:"
pack .citystate.label -side left
entry .citystate.entry
pack .citystate.entry -fill x

frame .phone
```

Example 6-1.

```
pack .phone -fill x
label .phone.label -text Phone:
pack .phone.label -side left
entry .phone.entry
pack .phone.entry -fill x
```

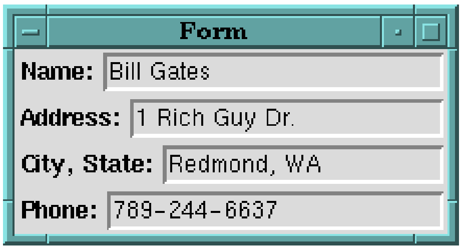


Figure 6-1. Tcl/Tk data entry interface

The coupling of label and entry widgets to provide a entry field is probably the most redundant pattern seen in your typical Tcl/Tk program. [incr Widgets] provides an entryfield mega-widget that is much simpler and cleaner. The same entry form can be coded using the [incr Widgets] entryfield in less than half the amount of code without all the extra frames.

```
entryfield .name -labeltext Name:
entryfield .address -labeltext Address:
entryfield .citystate -labeltext "City, State:"
entryfield .phone -labeltext Phone:

pack .name -fill x
pack .address -fill x
pack .citystate -fill x
pack .phone -fill x
```

Now we'll turn up the heat a little and add on some more requirements. Let's say we want to have the phone number field do some input verification, accepting only numbers and the "-" character. Also, we want the labels aligned and the background of the entry widgets to be different from the rest of the form. The [incr Widgets] entryfield widget provides for all of these needs with a couple of additional options. Thus, with the addition of a few more lines of code we can make the example enforce the requirements and improve its appearance.

```
option add *textBackground white

entryfield .name -labeltext Name:
entryfield .address -labeltext Address:
```

```
entryfield .citystate -labeltext "City, State:"
entryfield .phone -labeltext Phone: \
   -width 12 -fixed 12 \
   -validate {validate_number %P}

Labeledwidget::alignlabels \
   .name .address .citystate .phone

pack .name -fill x
pack .address -fill x
pack .citystate -fill x
pack .phone -fill x

proc validate_number {number} {
   return [regexp {^[-0-9]*$} $number]
}
```

When we created the .phone widget, we used the *-validate* option to indicate that the input would be checked by the validate_number procedure as the user entered it. Further down, we define validate_number, which returns either 1 (true) or 0 (false). What it actually returns is the value returned by a *regexp* call, which just determines whether the input consists of digits and hyphens using a regular expression.

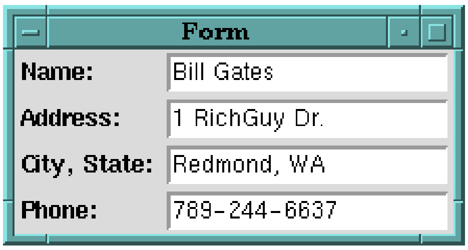


Figure 6-2. [incr Widgets] data entry interface

[incr Widgets] Basics

Construction, Destruction, and Commands

Creation of an [incr Widgets] mega-widget is exactly like that of a standard Tk widget: you invoke the class name as a command. The first argument is the path name of the new mega-widget, and succeeding arguments are option-value pairs. The following command creates an instance of a scrolledlistbox with a constant vertical scroll bar and a horizontal one that appears on demand.

```
ScrolledListBox .slb -vscrollmode static \
   -hscrollmode dynamic
pack .slb
```

Once the mega-widget has been created, a new object exists whose name is the mega-widget path name. Now a new set of commands that are specific to the object's class can be invoked. For example, the scrolledlistbox provides commands that support insertion and deletion of entries in the list.

```
.slb insert 0 Linux SunOS Solaris HP/UX Windows
.slb delete Windows
```

All the standard Tk commands that take a widget path name as an argument work as expected. For example, if you want to know the class an [incr Widgets] mega-widget, use the standard *winfo class* command.

```
% winfo class .slb
Scrolledlistbox
```

Finally, to destroy an [incr Widgets] mega-widget, use the same Tk *destroy* command you normally would.

```
destroy .slb
```

Configuration

Just like Tk widgets, [incr Widgets] mega-widgets keep a set of configuration options that mirror their current state. They may be used when creating a mega-widget and modified subsequently using the *configure* command. In addition, many of these options affect the physical layout of the mega-widget, allowing the visual appearance to be dynamically tweaked. [incr Widgets] provides this capability under the principle that flexibility yields reuse. If you'd rather have the label for the entryfield on top rather than the side, no problem, the option has been included. If you want scrollbars on demand for your scrolledlistbox, you got it. Each mega-widget has been designed to allow exactly this sort of modification of the visual aspects of the components through a rich option suite. In addition, default option values may be specified in the option database.

This type of dynamic configuration is a very useful quality. The fact that Tcl/Tk is interpretive, accentuates this virtue, allowing interactive interface design. Programs can be built that change appearance on the fly based on user input or program needs. An application that demands multiple flavors of a message dialog such as confirmation and warning dialogs, with different messages, can create one instance and change the options between uses to create different appearances. This can be much more efficient than creating separate dialogs, since construction time is much more costly than the time required to modify and map the widget.

Consider an application that must confirm user exit requests prior to leaving an application. Should the user respond "Yes", then we'll need to ask them if they'd like to save any changes. To illustrate, the following code segment creates the initial message dialog and configures the message to ask "Are you sure ?". The dialog is then activated, which pops it on the display. Should the user respond positively, then the message is changed to "Would you like to save changes ?" and displayed a second time. This eliminates the need for two message dialogs with different messages when one would be more efficient.

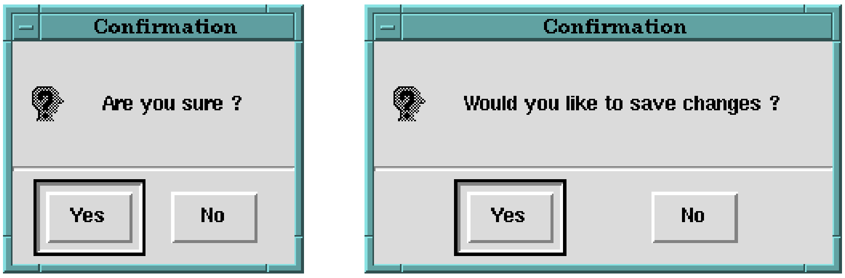


Figure 6-3. Confirmation dialog configuration

```
messagedialog .md -modality application \
   -title Confirmation -bitmap questhead \
   -text "Are you sure ?"
.md buttonconfigure OK -text "Yes"
.md buttonconfigure Cancel -text "No"
.md hide Help

if {[.md activate]} {
   .md configure \
         -text "Would you like to save changes ?"

   if {[.md activate]} {
         #
         # Save changes
         #
   }

   exit
}
```

We can change much more than just the message text. This same message dialog can be configured on-the-fly into an error dialog. Since all the options can be dynamically changed, it is possible to change not only the bitmap but its location as well. Furthermore, we can modify the text of the buttons and make the dialog modeless.

Now let’s make a change after the dialog has been activated. The *activate* command shows the dialog to the user, mapping it to the display. With the dialog visible, we’ll change the orientation and position of the buttons to be vertical along the right hand side. It is important to realize that no new message dialog has been created. Instead, one has been configured differently a total of four times. Also note that configuration options may be modified at any time, regardless of the current mapped status of the mega-widget.

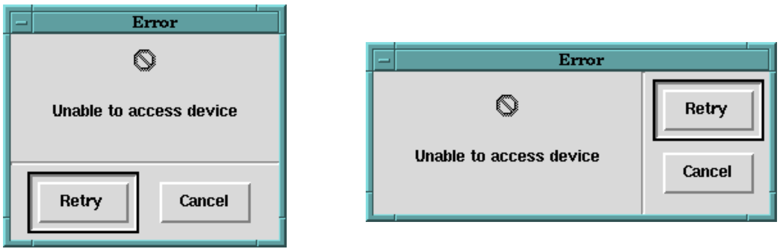


Figure 6-4. Error dialog configuration

```
.md configure -bitmap error -imagepos n \
   -text "Unable to access device" \
   -modality none -title Error
.md buttonconfigure OK -text Retry
.md buttonconfigure Cancel -text Cancel

.md activate

.md configure -buttonboxpos e
```

Childsites

Childsites are an integral and innovative concept in the [incr Widgets] mega-widget set. Basically, a childsite is a place holder. Let me explain with a concrete example.

The dialog mega-widget may provide, say, some buttons like “OK” and “Cancel” for operations you embed. What you provide in the body of the dialog may vary widely from application to application. It may be a scrolledtext widget in one case, a set of entryfields in another, and so on. So a lot of the [incr Widgets] mega-widgets offer a general place, a childsite, to insert whatever you want. The contents can be any widgets or mega-widgets you choose.

In terms of implementation, a childsite is actually a standard Tk frame widget. You acquire the widget path of this frame via the *childsite* command. With the path in hand, you can start packing in other widgets with the childsite as the parent, filling your application's needs.

Now let's illustrate the use of childsites. We'll make a dialog that allows user specification of a RGB color values. Once constructed, we'll grab the childsite path and stuff it into a variable. Next, we'll create three scales, one for red, green, and blue within the dialog's childsite using this variable. We'll use the scale's *-to* option to limit the range to 255. Finally, the dialog is made active.

```
dialog .rgb -title Colors
set cs [.rgb childsite]

scale $cs.red -orient horizontal \
   -label Red -to 255 -troughcolor Red
pack $cs.red -fill x

scale $cs.green -orient horizontal \
   -label Green -to 255 -troughcolor Green
pack $cs.green -fill x

scale $cs.blue -orient horizontal \
   -label Blue -to 255 -troughcolor Blue
pack $cs.blue -fill x
.rgb activate
```

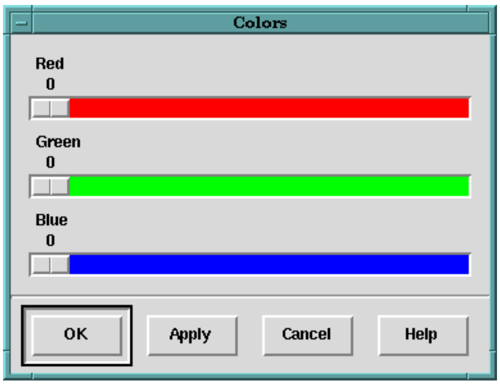


Figure 6-5. Colors dialog using a dialog childsite

One more quick example. The childsite of the entryfield mega-widget can be useful when you need a units label or a button that invokes a procedure. For example, lets create an entryfield for file name entry and alternatively provide a button that can be used to select a file using a file selection dialog. We'll create an entryfield and fill its childsite with a button labeled "...". We'll also create a fileselectiondialog and use it in a procedure tied to the button. Upon selection of the button, the procedure activates the dialog and waits for a response. If the user selects the "OK" button, the filename in the entryfield is replaced with the one from the fileselectiondialog.

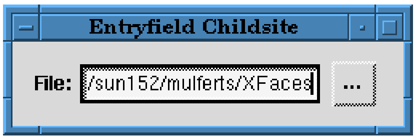


Figure 6-6. File entry field using an entryfield childsite

```
fileselectiondialog .fsd -modality application
proc SelectFileProc {} {
   if {[.fsd activate]} {
         .filename delete 0 end
         .filename insert end [.fsd get]
   }
}

entryfield .filename -labeltext File:

set childsite [.filename childsite]

button $childsite.button -text "..." \
   -padx 5 -command SelectFileProc
pack $childsite.button -padx 5

pack .filename -padx 10 -pady 10 \
   -fill x -expand yes
```

[incr Widgets] Hierarchy

The [incr Widgets] class hierarchy is derived from several base classes provided by [incr Tk]. Figure 6-7 depicts the [incr Widgets] inheritance hierarchy. They provide option management, standard commands, and a parent for components called the hull widget. The hull is just like a childsite. It is a placeholder for widgets, which [incr Tk] refers to as components. In addition, the hull's path is stored in a protected class variable named itk_interior.

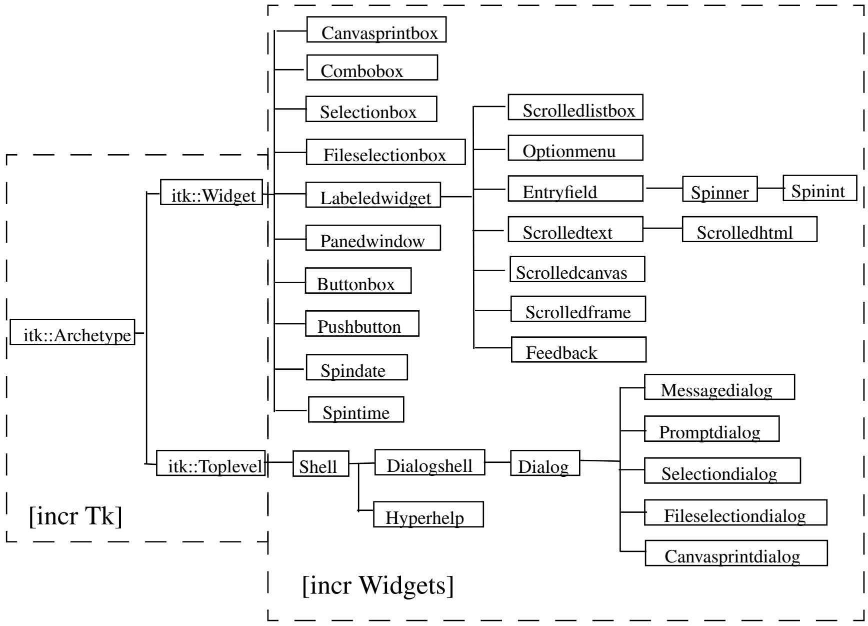


Figure 6-7. [incr Widgets] inheritance hierarchy

Many mega-widgets within [incr Widgets] successively maintain the hull variable in the hierarchy, passing it along from parent to child. When a mega-widget is constructed, new components are built off the hull's path. As mega-widgets are derived one from another, they may wish to redefine the hull. To do this, the mega-widget constructs a frame which it would now like to call the hull, setting the itk_interior variable to this new hull's path. This allows for the hull to change from parent to child as each mega-widget sees fit.

For example, the [incr Tk]'s Toplevel class defines a hull that is filled by the [incr Widgets]'s Dialogshell class with a buttonbox for push buttons, a separator, and a frame on top. The Dialogshell redefines the hull to be this frame and adds a *childsite* command to provide access for composition purposes. The Dialog class is derived from Dialogshell, adding specific buttons and leaving the hull just as it was defined by the Dialogshell class to be the area above the separator. The Selectiondialog, next in the hierarchy, is derived from the Dialog class, filling the hull with a scrolledlistbox, an entryfield, and a frame. It changes the hull to now be this frame and provides a new *childsite* access command as well. Thus, you can see that as the definition of the hull may change throughout the hierarchy. Each mega-widget passes it along, defining it as needed for the next class and narrowing its focus.

Creating new mega-widgets

Thus far, the childsite examples presented have extended an [incr Widgets] mega-widget by just sticking widgets or even other mega-widgets inside one other using the *childsite* command--a technique known as composition. Mega-widgets may also be extended using inheritance which creates a parent-child relationship between mega-widgets. The new child mega-widget builds on the commands and options already defined in our parent class. [incr Tk] provides the mechanism and [incr Widgets] furnishes the hooks.

To illustrate, let's make a colordialog mega-widget using inheritance that will contrast with the composition example previously presented. As a new mega-widget we'll call the class Colordialog. We'll also take the opportunity to make the colordialog less hard-wired to RGB values. Since we have more control over options at this level, we'll keep them more generic. Instead of naming the scales red, green, and blue, we'll use value1, value2, and value3. We'll use this convention in naming the label and to options as well. The benefits to this less specific approach will become apparent later.

Let's look at the [incr Tk] code required to create a more generic colordialog mega-widget. First, let's define the class. It will be derived from the [incr Widgets] Dialog class. We'll also need a constructor and a *get* method for retrieving the values from our scales. This method should be able to query the value for a specific scale or default to returning all three values in a list.

```
class Colordialog {
   inherit iwidgets::Dialog

   constructor {args} {}

   public method get {{which all}}
}
```

Now we'll need to specify the body for our constructor. It will need to create three scales as components using `itk_component add`. Each will have the hull as its parent, which is stored in the *itk_interior* variable. The result is that the scales are created in the dialog's childsite. We'll keep specific options offered by the scale widget, renaming those which might otherwise conflict by prefixing the scale's name to the *-to*, *-label*, and *-troughcolor* options. This will make them unique and allow each to be independently modified. The last thing the constructor will need to do is evaluate any option arguments.

```
body Colordialog::constructor {args} {
   itk_component add value1 {
```

```
            scale $itk_interior.value1 \
                  -orient horizontal
      } {
            keep -activebackground -background \
             -cursor -font -foreground \
             -highlightbackground -highlightcolor \
             -highlightthickness -repeatdelay \
             -repeatinterval -showvalue

            rename -to -value1to value1To To
            rename -label -value1label \
                  value1Label Label
            rename -troughcolor -value1troughcolor \
                  value1TroughColor TroughColor
      }
      pack $itk_component(value1) -fill x
```

Similarly for value2 and value3 scales ...

```
      eval itk_initialize $args

      hide Help
   }
```

Next, we'll define the body of the *get* method. It will take a default argument of "all" and allow individual scale values to be retrieved. We'll also install an error check for invalid arguments.

```
   body Colordialog::get {{which all}} {
      switch $which {
            value1 {
                   return [$itk_component(value1) get]
            }
            value2 {
                   return [$itk_component(value2) get]
            }
            value3 {
                   return [$itk_component(value3) get]
            }
            all {
                  return [list \
                   [$itk_component(value1) get] \
                   [$itk_component(value2) get] \
                   [$itk_component(value3) get]]
            }
```

```
            default {
                  error "bad get argument \
                   \"$which\",should be:\
                   value1, value2, value3, or all"
            }
      }
}
```

Currently, construction of a new colordialog would require us to use the uppercase class name. This doesn't really follow the Tk model of lowercase commands. So, we'll provide a bit of syntactic sugar to remedy this.

```
proc colordialog {args} {
   uplevel Colordialog $args
}
```

This completes the code required for our colordialog class implementation. Next, we'll make use of the option database to define defaults that will map to our previous example. Mega-widgets, when created, consults the option database for option values not defined at construction. We'll define default values for the scale labels setting them to "Red", "Green", and "Blue". We'll also default the scale's *-to* options be 255.

```
option add *Colordialog.title \
         "Color Dialog" widgetDefault
option add *Colordialog.value1Label \
         Red widgetDefault
option add *Colordialog.value2Label \
         Green widgetDefault
option add *Colordialog.value3Label \
         Blue widgetDefault

option add *Colordialog.value1To \
         255 widgetDefault
option add *Colordialog.value2To \
         255 widgetDefault
option add *Colordialog.value3To \
         255 widgetDefault
```

The example presented earlier can now be created much more directly, using a more simpler syntax.

```
colordialog .cd
.cd activate
```

The [incr Tk] base classes enable the colordialog to act exactly like a standard Tk widget. Commands like *configure* and *cget* work as expected. Other typical Tk commands work as well.

```
% .cd configure -value1troughcolor Red \
        -value2troughcolor Green \
        -value3troughcolor Blue
% winfo class .cd
Colordialog
```

This version of the colordialog is much more versatile than before. We can easily create a colordialog for entering HSV (Hue Saturation Value) values by changing the scale label and to option values.

```
colordialog .cd -value1label Hue -value1to 360 \
   -value2label Saturation -value2to 100 \
   -value3label Value -value3to 100
.cd activate
```

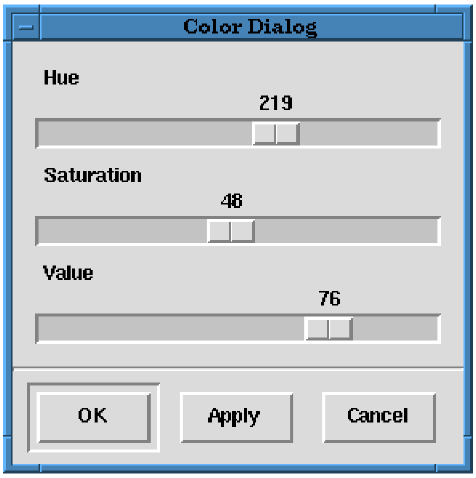


Figure 6-8. Color dialog mega-widget

[incr Widgets] Tour

The [incr Widgets] mega-widget classes can be divided into widgets, scrolled widgets, managers, and dialogs. Each of these will be detailed in separate sections.

General Characteristics

The widgets that allow textual input or selection maintain an additional option called textbackground. This is independent of the background option and enables the user to specify a separate color for the text area as opposed to the complete background. The entryfield, scrolledlistbox, and scrolledtext widgets all provide the textbackground option. For example, Figure 6-9 depicts two scrolledlistboxes, the first one with a standard background of grey and the second with a textbackground of GhostWhite.

```
scrolledlistbox .grey
scrolledlistbox .gwhite -textbackground GhostWhite

pack .grey -side left -padx 10 -pady 10
pack .gwhite -side left -padx 10 -pady 10

set OSs [list Linux SunOS Solaris HP/UX AIX IRIX]

foreach item $OSs{
   .grey insert end $item
   .gwhite insert end $item
}.
```

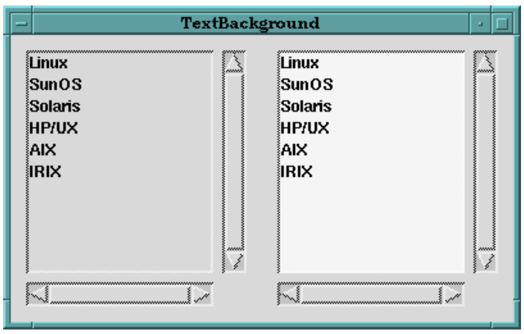


Figure 6-9. Textbackground option

The use of *-textbackground* produces a much more aesthetically pleasing interface. So rather than duplicate it for every example, we'll just put it into our .Xdefaults file and use it implicitly throughout the rest of this chapter, taking advantage of its visual affect.

```
*textBackground: GhostWhite
```

Widgets

[incr Widgets] widgets represent the most basic of mega-widgets. Tk widgets are combined and enhanced, expanding the command and option suites, and providing a simpler means of creating typical Tk widget patterns.

Labeledwidget

The labeledwidget is the most primitive widget in the [incr Widgets] set, providing label support for many of the other classes. The class contains a label, a margin, and a childsite that can be filled with other widgets of your choice. The options provide the ability to position the label around the childsite, modify the font, and adjust the margin distance. Valid locations for label position follow the notation established by Tk: n, ne, e, se, s, sw, w, and nw. It should also be noted that labels for labeledwidgets are not just limited to text strings, both bitmaps and images are supported as well.

The following example creates a labeledwidget with a canvas widget in the childsite. The label is set to “Canvas” and initially located south of the childsite. Next, the label is moved around the childsite while the margin is set to various distances.

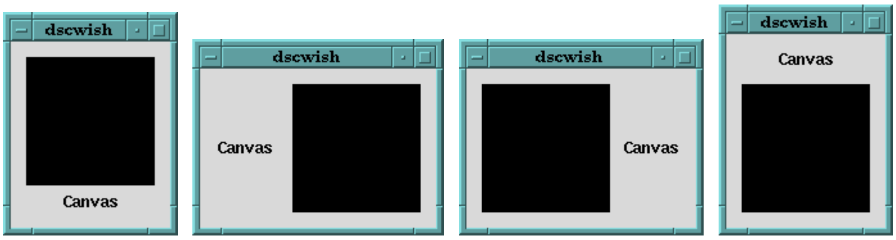


Figure 6-10. Labeledwidget label positions

```
labeledwidget .lw -labeltext "Canvas" -labelpos s
set childsite [.lw childsite]
canvas $childsite.c -relief raised -width 100 \
   -height 100 -background black

pack $childsite.c
pack .lw -fill both -expand yes -padx 10 -pady 10

update; after 1000
.lw configure -labelpos w -labelmargin 10

update ;after 1000
.lw configure -labelpos e -labelmargin 5
```

```
update; after 1000
.lw configure -labelpos n -labelmargin 7
```

A typical problem with a set of labeled widgets is alignment. You'll feel the need for this whenever you pack together a group of widgets, such as entry-fields, with labels of different lengths. The labeledwidget class solves this problem by providing a command, *alignlabels*, that performs alignment by adjusting the margins so the fields all start at the same position. This command is actually a class command, global to all instances of labeledwidget and any other classes based on labeledwidget. As a class command, it has an unusual syntax in that the class name precedes the *alignlabels* command. The command takes as parameters the widgets to be aligned, which must be all derived from Labeledwidget. The following example creates a second canvas with a longer label text then the last example and aligns the labels. One trick to remember is to always pack the widgets to be aligned using the *-fill* option.

Example 6-2.

```
.lw configure -labelpos w -labelmargin 1

labeledwidget .lw2 -labeltext "Another Canvas" \
    -labelpos w
set childsite [.lw2 childsite]
canvas $childsite.c2 -relief raised -width 100 \
    -height 100 -background white

pack $childsite.c2
pack .lw2 -fill both -expand yes -padx 10 -pady 10

Labeledwidget::alignlabels .lw .lw2
```

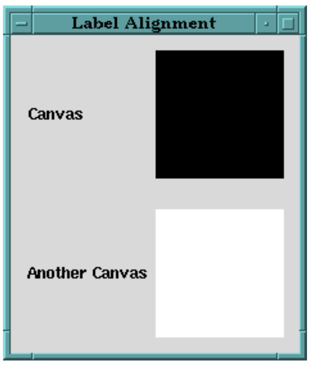


Figure 6-11. Labeledwidget label alignment

Entryfield

The entryfield class associates a label with an entry widget for text entry. The combination has been further enhanced by providing length control and validation capabilities. Since the class is based on the labeledwidget class, all the options and commands for labeledwidgets are supported in entryfields. Also, the commands for the standard Tk entry widget have been included such as *insert*, *delete*, *get*, and *scan* to name a few.

Anyone who has worked significantly with the standard Tk entry widget has become aware of its limitations. The entry widget lacks any restrictions on length and provides no simple means for input validation. [incr Widgets] provides a remedy. The entryfield widget furnishes a *-fixed* option, which limits the number of characters allowed to be entered in the widget. Once this character limit has been reached, input is discarded and the user is alerted via the bell. The entryfield widget lets you restrict input to particular types of data through the *-validate* option. Character input can be limited to alphabetic, numeric, alphanumeric, integer, hexadecimal, and real.

The default action that occurs following the reception of invalid input is to ignore the character and ring the bell. Should more drastic action be warranted, include the *-invalid* option and the name of a procedure that is to be called whenever user input fails to meet your requirements. To illustrate, the following example creates an entryfield for hexadecimal entry. Also, should the user enter invalid characters, an error message is written out to standard error.

Figure 6-12. Hexadecimal entryfield validation

```
entryfield .hex -labeltext Hex: \
   -validate hexadecimal \
   -invalid {puts stderr "Invalid Hex Character"}
pack .hex -padx 10 -pady 10
```

Input can also be restricted to a finer level of granularity. Besides the type keywords previously given, the *-validate* option accepts a command script to executed upon detection of character input. The script must return a boolean value. A false value triggers the execution of the *-invalid* option. If the validation command contains any % characters, the script will not be executed directly. Instead, a new script is generated by replacing each % and the character following it with information from the entryfield. The replacement depends on the character following the %. These substitution characters allow scripts to selectively process the current input character, the contents of the entryfield following the input, or the previous contents. For example, let's suppose we want an entryfield that accepts only uppercase alphabetic characters and the '_' character. We can use the -*validate* option and %c to specify a command procedure to take the current input character as an argument. The procedure then uses a regular expression to validate the argument, returning a boolean result.

```
proc RegExpValidate {inputChar} {
   return [regexp {[A-Z_]} $inputChar]
}

entryfield .e -labeltext "Restricted Input:" \
   -validate {RegExpValidate %c}
pack .e -padx 10 -pady 10
```

Pushbutton

The pushbutton class augments the Tk button widget, adding default button display using a recessed ring. The primary use for the pushbutton is as components of the buttonbox class. In addition to furnishing the commands and options that the pushbutton gets from the Tk button widget, the pushbutton provides options for enabling or disabling the display of the default ring. The amount of padding within the default ring is adjustable using the *-defaultringpad* option. The following code segment creates a pushbutton with the default ring shown.

```
pushbutton .pb -text Pushbutton \
   -defaultring yes -defaultringpad 4
pack .pb -padx 12 -pady 12
```

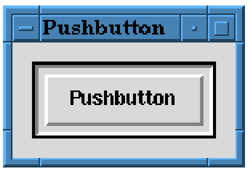


Figure 6-13. Pushbutton

Optionmenu

The Optionmenu class allows users to select one item from a set. Only the selected item is displayed, until the user selects the option menu button and a popup menu appears with all the choices available. Once a new item is chosen, the currently selected item is replaced and the popup is removed from the display. The [incr Widgets] optionmenu class is based on the labeledwidget class and therefore supports label specification, position, and alignment.

Figure 6-14. Optionmenu

The best way to understand the capabilities of the optionmenu widget is by example. The following creates a labeled optionmenu of various flavors of operating systems. It also associates a command procedure with the selection that will print out the selection. The procedure is called only upon change in current selection.

```
proc SelectOSProc {} {
   puts "The best OS is : [.om get]"
}

optionmenu .om -command SelectOSProc \
   -labeltext "Operating Systems:" \
   -items {SunOS HP/UX AIX OS/2 Windows DOS}
pack .om -padx 10 -pady 10
```

Widget commands for manipulating the menu list contents have also been included such as *insert*, *delete*, *select*, *disable*, *enable*, and *sort*.

```
.om insert end Linux VMS
.om disable DOS
.om delete 1 2
.om sort ascending
.om select Linux
.om configure -cyclicon true
```

Cyclic item selection is offered as an extension to the basic operations of standard option menus. This allows the right mouse button to cycle through the menu without displaying the popup. The right mouse with the shift key depressed cycles though the items in the reverse order.

Combobox

The [incr Widgets] combobox provides an enhanced entryfield widget with an optional label and a scrolledlistbox. When an item is selected in the list area of a combobox its value is then displayed in the entry field text area. Functionally similar to an optionmenu, the combobox adds optional list scrolling and the ability to edit and insert items.

There are two basic styles of comboboxes: dropdown and simple. The dropdown style adds an arrow button to the right of the entry field, which when selected pops up the scrolledlistbox beneath the entryfield widget. The simple, non-dropdown, combobox permanently displays the listbox beneath the entry field and has no arrow button. Both styles provide an optional label. Figure 6-15 illustrates the various flavors of comboboxes.

Example 6-3Comboboxes.

```
#
# Non-editable dropdown combobox
#
set monthList \
    {Jan Feb Mar Apr May June \
     Jul Aug Sept Oct Nov Dec}

combobox .noneditcb -labeltext Month: \
    -editable false -items $monthList
pack .noneditcb -padx 10 -pady 10 -fill x

#
# Editable dropdown
#
combobox .editablecb -fliparrow true \
    -popupcursor hand1 -listheight 100 \
    -items {Linux HP-UX SunOS Solaris Irix} \
    -labeltext "Operating Systems:"
pack .editablecb -padx 10 -pady 10 -fill x

#
# Empty editable dropdown
#
combobox .numericcb -unique true -labelpos nw \
    -validate numeric -labeltext "Numeric Combo:"
pack .numericcb -padx 10 -pady 10 -fill x

#
# Simple combobox
#
combobox .simplecb -dropdown false -textfont 9x15 \
    -labelpos nw -labeltext "Font:" \
    -items [exec xlsfonts] -listheight 220
pack .simplecb -padx 10 -fill both -expand yes
```

Feedback

The feedback mega-widget is a simple progress indicator. It has commands for stepping along and resetting the indicator. Options enable the width, height, and color of the status bar to be configured. Its application can be seen in the hyper-

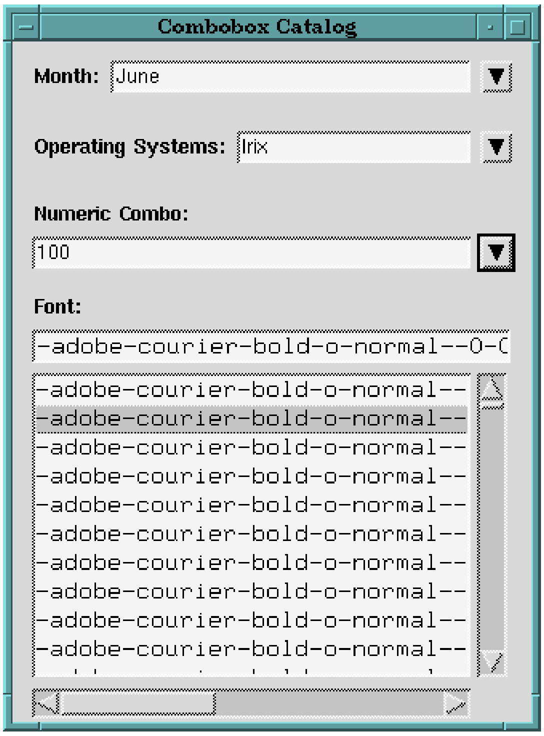


Figure 6-15. Comboboxes

help mega-widget, which displays a feedback widget during html rendering. For a more simple example, we'll cycle through a loop and update our progress once a second.

```
feedback .gage -steps 10 -labeltext Progress
pack .gage

for {set i 0} {$i < 10} {incr i} {
   after 1000
   .gage step
}
```

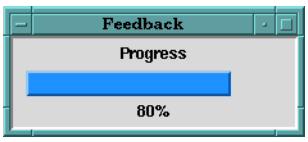


Figure 6-16. Feedback

Spinners

Spinners constitute a set of widgets that provide entryfield functionality combined with increment and decrement arrow buttons. A value may be entered into the entry area explicitly or the buttons may be pressed, which cycle up and down through the choices. This latter behavior is called spinning. The actions associated with the arrow buttons vary from class to class, offering integer, time, and date spinning.

Spinner. The spinner class is the basis for all the spinner widgets including the more specialized spinint, spintime, and spindate. Since it is based on the entryfield class, all the options and commands associated with entryfield widgets are provided as well as additional options that enable a command to be specified for increment and decrement actions. The other spinner classes make use of this capability and have predefined actions for the command that fill their functionality.

The spinner class is simple and generic. [incr Widgets] builds a number of fancy classes on it, and you can develop a custom spinner as well. For example, let's use the spinner class to make a month spinner. The months will be stored in a list, which a spinMonth procedure cycles through. The Spinner *-increment* and *-decrement* options invoke this procedure with a direction argument of either 1 or -1.

```
set months {January February March April \
         May June July August September \
         October November December}
proc spinMonth {direction} {
   global months

   set index [expr [lsearch $months [.sm get]] \
         + $direction]

   if {$index < 0} {set index 11}
   if {$index > 11} {set index 0}

   .sm delete 0 end
   .sm insert 0 [lindex $months $index]
}

spinner .sm -labeltext "Month : " \
   -fixed 10 -validate {return 0} \
   -decrement {spinMonth -1} \
   -increment {spinMonth 1}
```

```
.sm insert 0 January
pack .sm -padx 10 -pady 10
```

Figure 6-17. Spinner

The default orientation of the spinner arrows is vertical, but this can be modified by the *-orient* option that allows for a side by side, horizontal presentation as well. The previously created month spinner can be configured to illustrate this option.

```
.sm configure -arroworient horizontal
```

Figure 6-18. Spinner arrow orientation

Spinint. Integers are the most common data type needing a spinning behavior. The spinint class offers this capability. Additional options are provided for step and range values that vary and limit the cycling. You can include the boolean *-wrap* option to let the user go straight from the maximum value to the minimum, or vice versa.

The following code segment creates a water temperature integer spinner widget labeled appropriately. The widget limits the range of values to be between freezing and boiling, and specifies a step value of two.

```
spinint .temp -labeltext "Water Temperature:" \
   -fixed 5 -range {32 212} -step 2
pack .temp -padx 10 -pady 10
```

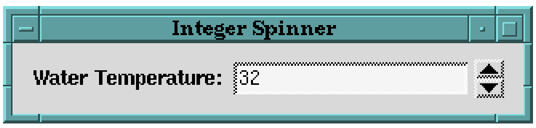


Figure 6-19. Spinint

Spintime. The spintime class provides a set of spinners for entering various units of time. The set includes an hour, minute, and second spinner widget.

Options allow each spinner to be hidden or shown. Also, the format of the time is configurable between military and normal display.

```
spintime .st
pack .st -padx 10 -pady 10
```

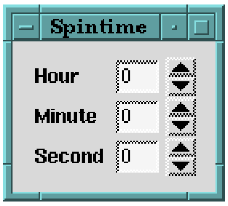


Figure 6-20. Spintime

Spindate. The spindate class supports date value entry. Just as with the spintime class, the display of each spinner can be separately controlled. The format of the month display may also be varied between text and integer display.

```
spindate .sd -monthformat string
pack .sd -padx 10 -pady 10
```

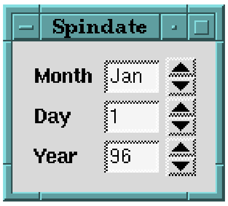


Figure 6-21. Spindate

Scrolled Widgets

Another common pattern in typical Tk scripts is the attachment of scroll bars to those Tk widgets that support scrolling. Sometimes you want just a vertical scrollbar, other times just a horizontal one, and sometimes both. These combinations were a prime candidate for replacement by [incr Widgets], which provides a scrolled listbox, text, canvas, and frame.

Each of the [incr Widgets] widgets that support scrolling offer wide-ranging control of scrollbar display and appearance. Figure 6-22 presents the four basic visual formats of scrolled widgets. Both the vertical and horizontal scrollbars can be separately configured for three different modes of display: static, dynamic, and none. In static mode, the scrollbar is displayed at all times. Dynamic mode displays the scrollbar as required and none disables scrollbar display. The default is static. The width of the scrollbars and the margins between the bars and the widget being scrolled can also be adjusted via options.

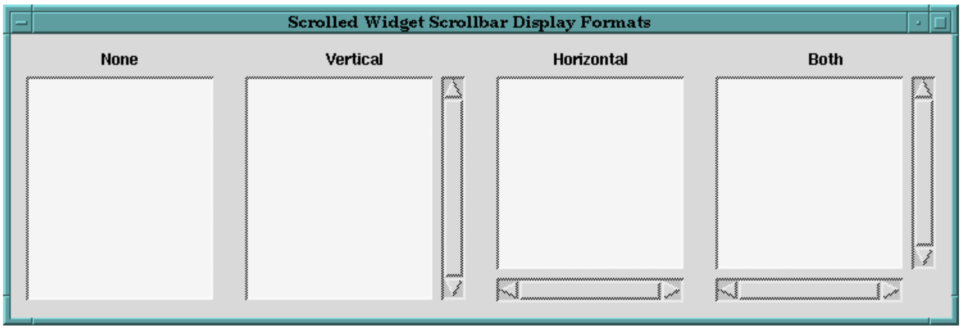


Figure 6-22. Scrollbar positioning

Each of the [incr Widgets] scrolled widgets provides -*width* and -*height* options for geometry specification. These dimensions apply to the widget as a whole, inclusive of the other components: label, margins, and scrollbars. Any additional space required to display these components comes for the space allocated to the primary widget to be scrolled. The dimensions are fixed in that the values given apply to the overall frame.

The scrolledlistbox and scrolledtext widgets offer an alternative method of geometry specification. Both of these widgets provide a -*visibleitems* option that specifies the width in characters and height in lines for the listbox or text widget. The dimensions of the listbox or text widget become fixed. The outer frame expands as needed to display the label, margins, and scrollbars. This option is effective only if the -*width* and -*height* options are both set to zero. Otherwise, the -*visibleitems* option applies by default.

Scrolledlistbox

The scrolledlistbox extends the standard Tk listbox widget with vertical and horizontal scrollbars and a label. The set of options has been expanded to let you specify list items and bind single and double click selections to procedures.

The following example shows a multiple selection scrolledlistbox of common weeds, which exist in my yard. The widget has a statically displayed vertical scrollbar and a horizontal one that appears on demand. A command procedure has been associated with the single select action to print the item.

```
proc SelectProc {} {
   puts "Oh no ! Not [.slb getcurselection]"
}
```

```
scrolledlistbox .slb -vscrollmode static \
   -hscrollmode dynamic -selectmode multiple
   -items {Crabgrass Dallisgrass Nutsedge} \
   -scrollmargin 5 -labelpos n -labeltext weeds \
   -selectioncommand SelectProc
pack .slb -padx 10 -pady 10 -fill both -expand yes
```

The standard listbox commands have been amended with ones for sorting the list contents and a short cut for getting the current selection. The *getcurselection* command combines the actions of the *curselection* and *get* commands so that the text of a selected item may be retrieved in a single action.

```
.slb insert 2 Sandbur Goosegrass Barnyardgrass
.slb insert end Chickweed Johnsongrass Puncturevine
.slb sort ascending
```

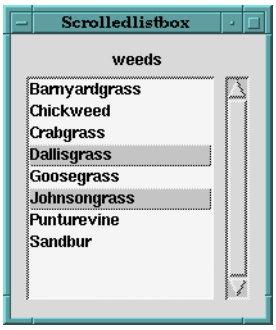


Figure 6-23. Scrolledlistbox

Scrolledtext

The scrolledtext widget provides all the functionality of the standard Tk text widget along with scrollbar and label control. The commands have been extended to include file import and export. To illustrate, we'll create a scrolled-text widget that imports the password file from my Linux machine.

```
scrolledtext .st -labeltext "passwd" -wrap none \
   -visibleitems 80x10

pack .st -padx 10 -pady 10 -fill both -expand yes

.st import /etc/passwd
.st yview end
```

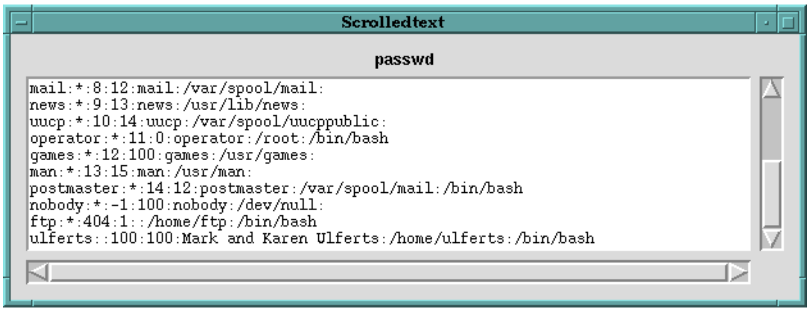


Figure 6-24. Scrolledtext

Scrolledframe

The scrolledframe combines the functionality of scrolling with that of a standard Tk frame widget, yielding a clipable viewing area whose visible region may be modified with the scroll bars. This allows the construction of visually larger areas than could normally be displayed, containing a heterogenous mix of widgets.

Once a scrolledframe has been created, the childsite can be accessed and filled with widgets. One ideal usage is for tables. For example, using the childsite as a parent for the combination of labels and entry widgets, we can create a checkbook.

```
scrolledframe .sf \
   -vscrollmode static -hscrollmode static \
   -labeltext CheckBook -width 7i -height 2i

set childsite [.sf childsite]

frame $childsite.f
pack $childsite.f
label $childsite.f.date -text Date -width 8 -bd 4
pack $childsite.f.date -side left

...

for {set i 0} {$i < 30} {incr i} {
   frame $childsite.f$i
   pack $childsite.f$i

   entry $childsite.f$i.date$i -width 8
   pack $childsite.f$i.date$i -side left
```

```
    ...
}
pack .sf -expand yes -fill both
```

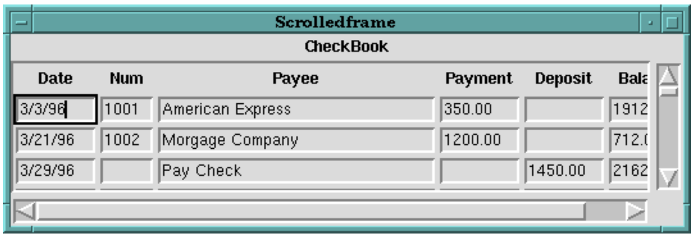


Figure 6-25. Scrolledframe

Scrolledcanvas

The scrolledcanvas applies scrollbars and display options to a Tk canvas widget. All the usual canvas commands and options have been kept. A new option, -*autoresize*, has been added, allowing automatic resizing of the scrolled region to be the bounding box covering all the items. The region is adjusted continuously as items are created and destroyed via the canvas commands, affecting the display of the scrollbars.

```
scrolledcanvas .sc -labeltext Floorplan \
   -width 300 -height 300
.sc create line 386 129 398 129 -fill black
.sc create line 258 355 258 387 -fill black
...
```

Managers

[incr Widgets] manager widgets include buttonbox, radiobox, toolbar, panedwindow, and tabbednotebook classes. Each acts as a container, providing geometry management for its children. This entails arranging the relative positions of the widgets placed inside, and adjusting the size of the container that holds them.

Buttonbox

The buttonbox performs geometry management for pushbuttons. Commands allow the user to add new pushbuttons, define the default button, control their display, and manage pushbutton configuration. The primary tasks of a buttonbox is to equally distribute the visible pushbuttons and maintain their

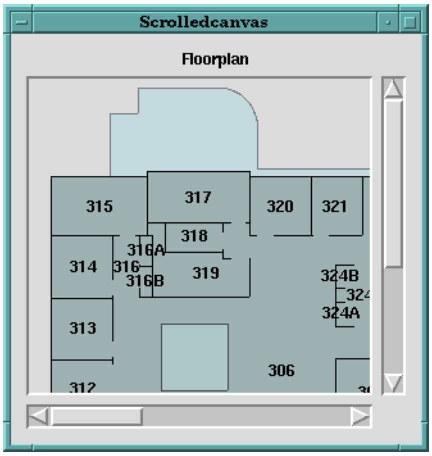


Figure 6-26. Scrolledcanvas

widths consistently. When another pushbutton is added that has a much longer text than those which are currently displayed, the widths of all the buttons are adjusted, making them of equal size. This class is used to manage the buttons for all the dialogs in the [incr Widgets] widget set.

When you create a buttonbox it initially contains no buttons. Add them through the *add* command, which accepts a tag name and a series of options to be applied to the pushbutton. The tag can later be used in other commands that employ an index as a reference to the button. Indexes may be specified numerically, using keywords, or a pattern that is matched against the tags. For example, the following code segment creates a buttonbox, adds buttons, and specifies the default using a numerical index. For simplicity, we'll initially keep the text the same as the tags.

```
buttonbox .bb -padx 10 -pady 10

.bb add OK -text OK -command {puts OK}
.bb add Cancel -text Cancel -command {puts Cancel}
.bb add Help -text Help -command {puts Help}
.bb default 0

pack .bb -expand yes -fill both
```

The *add* command appends pushbuttons to the end of the buttonbox. Another command, *insert*, allows pushbuttons to be placed before another pushbutton. The command takes as arguments the index of an existing pushbutton, followed by a tag, and finally the options to be applied to the new pushbutton. For

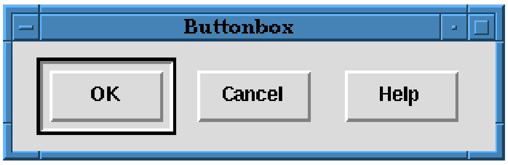


Figure 6-27. Buttonbox add command

example, we can place an Apply pushbutton between the OK and Cancel pushbuttons using the tag of the Cancel button as the index.

```
.bb insert Cancel Apply -text Apply
```

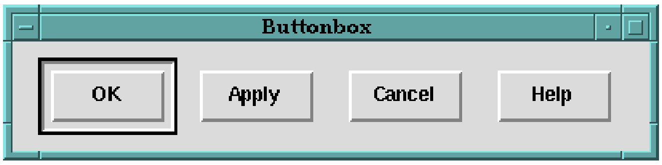


Figure 6-28. Buttonbox insert command

Commands have also been included to *hide*, *show*, or *delete* a pushbutton. In addition, the options for a tagged pushbutton can be configured using the *buttonconfigure* command. Now, we'll take our buttonbox and hide the Help button, and change the text of the button tagged OK to be Commit.

```
.bb hide Help
.bb buttonconfigure OK -text Commit
```

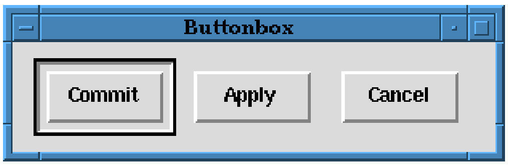


Figure 6-29. Buttonbox buttonconfigure command

The orientation of the buttonbox, and the distance between the buttons and the perimeter may also be modified through options. For example, the buttonbox can be made horizontal with additional padding using the -*orient*, -*padx* and -*pady* options.

```
.bb configure -orient vertical -padx 20 -pady 20
```

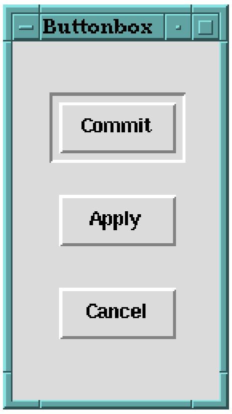


Figure 6-30. Buttonbox orient and pad options

Radiobox

The [incr Widgets] radiobox manages radiobuttons, encapsulating the global variable needed for Tk radiobutton usage. Once the radiobox has been created, radiobuttons are added with a tag using the *add* command. The tag can be used in other commands such as *insert*, *delete, select, deselect,* and *invoke*. The current setting can be obtained using the *get* command. A *buttonconfigure* command allows the options associated with a tagged radiobutton to be modified and queried. The radiobox is based on the labeledwidget class, so all the options for label management are available. Let's create a font radiobox, demonstrating command and option usage.

```
radiobox .rb -labeltext Fonts -relief raised

.rb add times -text Times
.rb add helvetica -text Helvetica

.rb insert times courier -text Courier
.rb insert 2 symbol -text Symbol

pack .rb -padx 20 -pady 20 -fill both -expand yes

.rb buttonconfigure times -command {puts Times}
.rb select times
.rb get
```

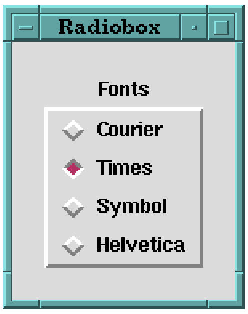


Figure 6-31. Radiobox

Toolbar

A toolbar makes for a very nice addition to any application's main window. It displays a collection of widgets such as buttons, radiobuttons, or optionmenus, offering quick access to a set of commonly needed functions for user convenience. The toolbar can also be used as a mouse bar. Placed horizontally beneath a menubar, you get a toolbar, oriented vertically and placed on the side, you have a mouse bar. As an extra feature, the toolbar class offers string and balloon help for each widget that it manages. This comes in quite handy when filling a toolbar with buttons that are labeled with bitmaps. It allows the user to see a simple help string associated with the item the mouse happens to be positioned over prior to selecting it.

The following example illustrates these features of the toolbar, creating an instance containing buttons with various bitmaps. The toolbar is created with a help variable, which is a global variable. As each button is added, a help string and balloon help value are associated with the component. The buttons use the standard "@" syntax in the *-bitmap* option to locate the bitmap file. Finally, a Tk entry widget is added for displaying help strings using its *-textvariable* option. As the mouse passes over each toolbar component, the help string is written to the entry widget. Should the mouse pause over a button, the balloon help pops up.

```
toolbar .tb -helpvariable statusVar

.tb add button line -helpstr "Draw a line" \
   -bitmap @line.xbm -balloonstr "Line" \
   -command {puts LINE}
.tb add button box -helpstr "Draw a box" \
   -bitmap @box.xbm -balloonstr "Box" \
   -command {puts BOX}
.tb add button oval -helpstr "Draw an oval" \
```

```
    -bitmap @oval.xbm -balloonstr "Oval" \
    -command {puts OVAL}
.tb add button points -helpstr "Draw poly points" \
    -bitmap @points.xbm -balloonstr "Points" \
    -command {puts POINTS}
.tb add frame filler -borderwidth 1 \
    -width 10 -height 10
.tb add button text -bitmap @text.xbm \
    -helpstr "Enter text" -balloonstr "Text" \
    -command {puts TEXT}

pack .tb -side top -anchor nw -fill x

entry .e -textvariable statusVar
pack .e -side top -anchor nw -fill x
```

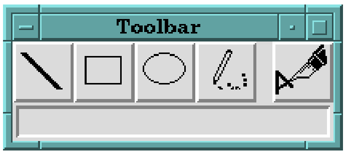


Figure 6-32. Toolbar

Panedwindow

Most of the [incr Widget] classes contain at most one childsite. The panedwindow is unique in that it manages multiple childsites. The mega-widget is composed of panes, separators, and sashes for adjustment of the separators. The sash is the little button used to grab the separator running beneath it and change the size of the pane. Each pane is a childsite whose path may be obtained using the *childsite* command. Once the paths have been acquired, you may fill them with widget combinations.

Once a panedwindow has been created, panes can be added and inserted using the *add* and *insert* commands. Each pane is assigned a unique tag identifier. Other commands such as *hide*, *show*, and *delete* take the tag as an argument. The viewable percentage of each pane can be modified using the *fraction* command. To illustrate, the following example creates a panedwindow with three panes, assigning a viewable percentage, and fills the childsites with scrolledlistboxes.

```
panedwindow .pw -width 400 -height 500
.pw add top
.pw add bottom -margin 10
.pw insert bottom middle -minimum 70
```

```
.pw fraction 50 30 20

pack .pw -fill both -expand yes

foreach pane [.pw childsites] {
   ScrolledListBox $pane.slb \
         -vscrollmode static -hscrollmode static
   pack $pane.slb -fill both -expand yes
}
```

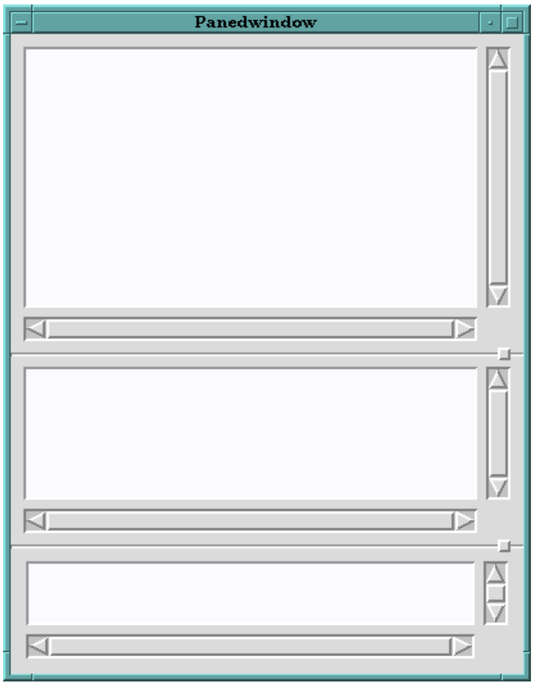


Figure 6-33. Panedwindow

As new panes are added, options may be specified that apply to the individual pane. These options include -*margin* and -*minimum*. The margin is the distance between the pane and the actual childsite. The -*minimum* option sets the minimum size in pixels that a pane's contents may reach. The *paneconfigure* command allows modification of the option values previously specified in an *add* or *insert* command.

Once visible, the separators on the panedwindow can be adjusted by selecting the square sash button and with the mouse button held down, moving the separator to the desired location. Upon mouse button release, the size of panes will be updated. One feature of the panewindow that is evident with three or more panes is that separator movement is allowed to bump and move other separators. Any minimum value previously set is maintained at all times.

The panedwindow offers significant control over its presentation through a rich set of options. The option set allows specification of the orientation, separator thickness, dimensions, sash position, and sash cursor. The *sashindent* option specifies the placement of the sash button as an offset. Positive values offset the sash relative to the near (left/top) side, whereas negative values denote a offset from the far (right/bottom) side.

Next, let's take this same panedwindow previously created and highlight some more of the commands and options we just talked about. We'll hide the center paned which was tagged "middle" and configure the minimum size of the top pane to be 100 pixels. Also, we'll change the orientation to be vertical using the -*orient* option. The -*sashindent*, -*sashheight*, -*sashwidth*, -*sashcursor* and -*thickness* options will be used to alter the appearance of the sash and separator.

```
.pw hide middle
.pw paneconfigure top -minimum 100

.pw configure -orient vertical \
   -sashindent 20 -sashheight 15 \
   -sashwidth 15 -thickness 5 -sashcursor gumby
```

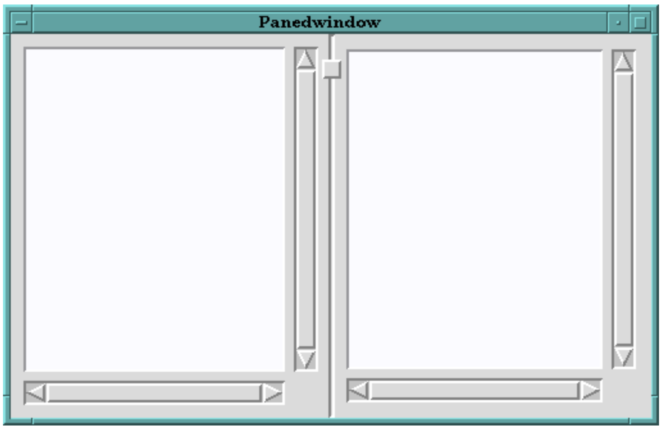


Figure 6-34. Panedwindow configuration

Tabnotebook

The tabnotebook mega-widget is one of the stars in the [incr Widgets] parade. Its unique presentation allows users to organize their interface as a set of tabbed pages, displaying one page at a time. When created, tabnotebooks have no pages. The *add* command allows pages to be affixed to the notebook. Additional widget commands support page deletion and configuration. Each tabnotebook page contains a single childsite that serves as a container for user defined widgets. Tabs are shown on the page borders serving as page selectors and identifying each page with a label. As a tab is selected, the contents of the associated page are displayed. Visually, the selected tab maintains a three-dimensional effect, making it appear to float above the other tabs.

Tabs themselves are quite configurable. The richness of the option suite permits a wide variety of styles and appearances, including Microsoft, Borland property, or Borland Delphi styles. Positionally, tabs may be located along the north, south, east, or west sides of the notebook. North and south tabs may appear angled, square, or bevelled. West and east tabs may be square or bevelled. Additional tab options allow specification of their gap, margin, internal padding, font, and label. To illustrate, we'll take a page from Quicken, no pun intended, and create a tabnotebook for bank, credit, investment, and other types of accounts. Each page will contain a set of specific accounts.

```
tabnotebook .tn -tabpos n -angle 15\
   -width 400 -height 200

.tn add -label "Bank"
.tn add -label "Credit"
.tn add -label "Other"
.tn add -label "Invest"

scrolledlistbox [.tn childsite "Bank"].bank \
   -hscrollmode none \
   -items {"Joint Checking" "Savings"}
pack [.tn childsite "Bank"].bank \
   -fill both -expand yes -padx 10 -pady 10

scrolledlistbox [.tn childsite "Credit"].credit \
   -hscrollmode none \
   -items {"Visa" "Mastercard" "American Express"}
pack [.tn childsite "Credit"].credit \
   -fill both -expand yes -padx 10 -pady 10

scrolledlistbox [.tn childsite "Other"].other \
   -hscrollmode none
pack [.tn childsite "Other"].other \
   -fill both -expand yes -padx 10 -pady 10
```

```
scrolledlistbox [.tn childsite "Invest"].invest \
   -hscrollmode none -items {"Stocks" "Bonds"}

pack [.tn childsite "Invest"].invest \
   -fill both -expand yes -padx 10 -pady 10

pack .tn -fill both -expand yes -padx 10 -pady 10

.tn select 0
```

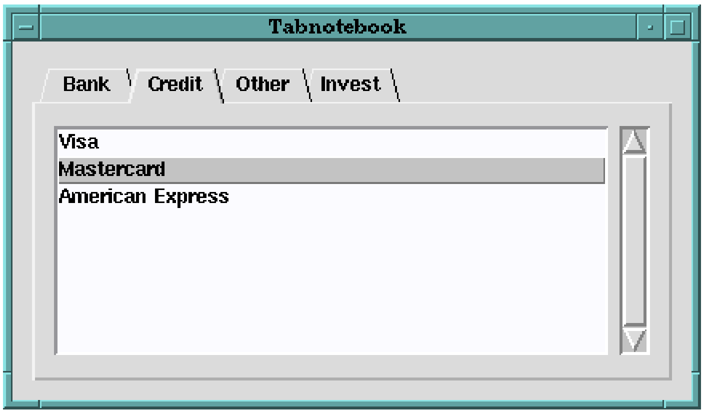


Figure 6-35. Tabnotebook

Dialogs

[incr Widgets] provides multiple flavors of toplevel dialog mega-widgets. They range from the bare-bones dialogshell up to various specialized dialogs. The suite includes the Motif favorites: promptdialog, messagedialog, selectiondialog, and fileselectiondialog. The [incr Widgets] designers have pulled out the basic dialog functional requirements, such as modal operation and button management, and spread them over the shell, dialogshell, and dialog classes. The shell provides mapping capabilities. The dialogshell adds button management and the dialog class completes the feature set by presenting a standard set of buttons.

Once a dialog has been created, it must be mapped using the *activate* command when you want to display it. This is analogous to packing a widget. Following activation, sooner or later the dialog will need to be removed from the display. This is facilitated by the *deactivate* command, which is similar to pack forget.

Modal dialog levels include global, application, and none. The difference being the degree of blocking. Global modal dialogs block all applications, whereas application modal dialogs block the current application. This allows processing of the dialog contents following user response and dialog termination. Modeless dialogs are of the modal type "`none`". They are non-blocking, enabling the application to continue. In this case, the actions attached to the buttons should perform all processing of the dialog contents including deactivation of the dialog itself. The default modal type is none.

The desired modal level is specified via the *-modality* option upon creation of a dialog, which takes effect upon activation. For application and global modal dialogs, the *activate* command does not immediately return. Instead, it waits until the dialog is deactivated. The more specific dialogs such as messagedialog, promptdialog, selectiondialog, and fileselectiondialog, are deactivated automatically upon selection of the OK or Cancel buttons. In this case, the OK button returns the value of 1 and Cancel of 0 to the waiting *activate* command. This feature comes in quite handy when you wish to wait for the user response to a dialog and process it only if they selected OK. For example, we can create an application modal promptdialog and wait for a value of 1 to be returned, signifying the user selected OK. Next, we can process the prompted value retrieved via the *get* command.

```
promptdialog .pd -modality application

if {[.pd activate]} {
   set response [.pd get]
   #
   # Process response
   #
}
```

Figure 6-36. Dialog activation

Sometimes we may want to catch the selection of OK or Cancel and not just let the dialog deactivate automatically. Consider the promptdialog again. Let's say we want to only *deactivate* the promptdialog if the user has non-blank input in the entryfield. In this case, we'll need to modify the behavior of the OK button to invoke a procedure that can check for this. The *buttonconfigure* command will allow us to modify the *-command* option associated with the OK button using the button's tag. By default, the tag for the OK button is OK. So now let's take our promptdialog example and add a checkInput procedure that uses promptdialog's *get* command combined with a regular expression test to make sure the input isn't just blank. In this case, we'll deactivate the dialog and process the input.

```
promptdialog .pd -modality application
.pd buttonconfigure OK -command checkInput

proc checkInput {} {
   if {[regexp {[^ \t]} [.pd get]]} {
         .pd deactivate
         #
         # Process response
         #
   }
}

.pd activate
```

The *buttonconfigure* command is capable of configuring all the options associated with the buttons, including the labels. The *deactivate* command accepts an optional argument that will be returned as a result of the activate command for global and application modal dialogs. This allows control over status notification. By default OK returns 1 and Cancel returns 0 to the *activate* command, but this can easily be modified.

Let's consider a messagedialog that notifies the user of impending doom and requests a response of either ignore, retry, or abort. To implement this we'll need to create a messagedialog, change the labels of the buttons, adjust the default button, and deactivate explicitly, passing the button label to the deactivate command. The default button is displayed with a sunken ring. Pressing the return key within the dialog invokes the default button. We'll choose to make the abort button be the default. Once activated, the messagedialog waits until deactivated, reacting to the argument passed to the deactivate command.

```
messagedialog .doom -bitmap error \
   -text "Severe Application Error" \
   -modality global

.doom show Apply
.doom hide Help
.doom default Cancel
```

```
.doom buttonconfigure OK -text Ignore \
   -command {.doom deactivate ignore}
.doom buttonconfigure Apply -text Retry \
   -command {.doom deactivate retry}
.doom buttonconfigure Cancel -text Abort \
   -command {.doom deactivate abort}

switch [.doom activate] {
   ignore {
         # Process ignore ...
   }
   retry {
         # Process retry ...
   }
   abort {
         # Process abort ...
   }
}
```

Figure 6-37. Dialog deactivation

Dialogshell

The dialogshell is the most primitive of the [incr Widgets] dialogs, composed of a buttonbox and childsite. It provides modal operation, button management support, and a childsite that lies above the separator. Button management includes the ability to add, insert, delete, hide, show, and configure buttons based on a tag. Since no buttons exist by default, deactivation must be explicitly handled in any buttons added to the dialogshell.

Even given only the basics, many dialogs can be constructed using the dialogshell class. For example, let’s create a database login dialog. We need to add buttons for opening a new or existing database, and exiting the dialog. The command associated with each button will deactivate the dialog and pass its label back to the activate command. The childsite will be filled with entryfield prompts for name, password, and database. The code would look like this.

```
dialogshell .dblogin \
   -title "Database Login" -modality global

.dblogin add New -text New \
   -command {.dblogin deactivate new}
.dblogin add Open -text Open \
   -command {.dblogin deactivate open}
.dblogin add Exit -text Exit \
   -command {.dblogin deactivate exit}

.dblogin default New

set childsite [.dblogin childsite]

entryfield $childsite.name \
   -labeltext Name:
pack $childsite.name -fill x
entryfield $childsite.password \
   -labeltext Password: -show \267
pack $childsite.password -fill x
entryfield $childsite.database \
   -labeltext Database:
pack $childsite.database -fill x

Labeledwidget::alignlabels $childsite.name \
   $childsite.password $childsite.database

if {[set type [.dblogin activate]] != "exit"} {
   switch $type {
         new {
             # Open new database
         }
         open {
             # Open existing database
         }
   }
}
```

Dialog

The [incr Widgets] dialog class is basically a dialogshell with several predefined buttons. They include the standard OK, Apply, Cancel, and Help. The command options for both the OK and Cancel buttons have also been preset. The OK command is *deactivate 1*, and the Cancel is *deactivate 0*.

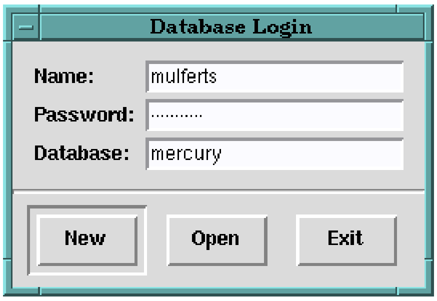


Figure 6-38. Dialogshell

Revisiting the database login example, we see that much of the code was spent creating, and configuring the buttons. Since the dialog class already provides the basic buttons, we can create the same login dialog without the need to add buttons. Instead, we'll just configure the labels and commands of the existing buttons using the *buttonconfigure* command and the button tags.

```
dialog .dblogin \
   -title "Database Login" -modality application

.dblogin hide Help

.dblogin buttonconfigure OK -text New \
   -command {.dblogin deactivate new}
.dblogin buttonconfigure Apply -text Open \
   -command {.dblogin deactivate open}
.dblogin buttonconfigure Cancel -text Exit \
   -command {.dblogin deactivate exit}

.dblogin default OK

...
```

Messagedialog

The [incr Widgets] messagedialog combines the display of an image and text together with dialog behavior. The option set allows the image to be located at various locations relative to the text: n, s, e, and w. Padding options have been included to provide extra space surrounding the text. Since the messagedialog is a dialog, all the button management commands from the buttonbox class are also available.

Many examples of typical messagedialog usage have been previously illustrated. A more unusual example is one of a copyright. The following code segment creates a copyright dialog and activates it. The dialog is non-modal and unnecessary buttons have been hidden.

```
messagedialog .cr -title “Copyright” \
   -bitmap @ora.xbm -imagepos n \
   -text “Copyright 1996\nAll rights reserved”

.cr hide Apply
.cr hide Cancel
.cr hide Help

.cr activate
```

Figure 6-39. Messagedialog

By default, selection of the OK button will deactivate the copyright dialog. Yet, if this were the first dialog for our application, forcing the user to explicitly remove it repeatedly would be quite annoying. As a matter of fact, it would be user hostile. A better approach would be to display the copyright for a finite period of time and then deactivate it ourselves. The Tk *after* command could be used to create this effect. This would provide two means by which the copyright notice may be removed, user action or timer.

```
after 10000 {.cr deactivate}
```

Promptdialog

The [incr Widgets] promptdialog reflects the functionality of the Motif prompt dialog. It is a dialog whose childsite contains an entryfield widget. All the entry-field options are available along with the buttonbox button management commands. For example, a password prompt can quickly be created that inhibits the display of the input text using the -*show* option to display “*”s in place of each character entered in the entryfield.

```
promptdialog .pd -modality global -show * \
   -title Promptdialog -labeltext Password: \
.pd hide Apply
.pd hide Help

if {[.pd activate]} {
   puts "Password entered: [.pd get]"
} else {
   puts "Password prompt cancelled"
}
```

Figure 6-40. Promptdialog

Selectiondialog

The selectiondialog class combines a scrolledlistbox of items, an editable entry-field for the selected item, and a pair of labels in a dialog, allowing the user to select or enter one item from a list of alternatives. It is a dialog based extension of the [incr Widgets] selectionbox class. The selectiondialog also provides a childsite and an option to control its position. This enables the widget to be extended by placing other widgets in the childsite, specializing the functionality.

The selectiondialog includes a wide assortment of options. Since both the scrolledlistbox and entryfield component widgets have labels, their position control options have been kept. Options have also been included to hide each of the two major components.

Let’s consider an application that requires an icon selection dialog, visually displaying the icon as the textual name is selected from the list. We’ll use the childsite in the selectiondialog, filling it with a canvas for icon display. We’ll also modify the default behavior of the selection command for the items list to place the icon bitmap in the canvas. Finally, we’ll force the user to select icons from the list rather than be able to type them by turning off the display of the selection entryfield.

```
selectiondialog .icons -title "Icon Selector" \
   -itemslabel Icons -itemscommand SelectProc \
   -selectionon no -items [exec ls]
.icons hide Help
.icons hide Apply

set cs [.icons childsite]
pack $cs -fill x

canvas $cs.canvas -height 70 \
   -relief raised -borderwidth 2
pack $cs.canvas -fill x -expand yes

proc SelectProc {} {
   global cs

   .icons selectitem

   $cs.canvas delete all
   $cs.canvas create bitmap \
         [expr [winfo width $cs.canvas] / 2] \
         [expr [winfo height $cs.canvas] / 2] \
         -bitmap @[.icons get]
}

.icons activate
```

Fileselectiondialog

The [incr Widgets] fileselectiondialog combines the fileselectionbox and dialog classes to achieve a Motif style file selection dialog. It consists of a file and directory list as well as a filter and selection entry widget. The display of each of these components can be controlled through options. A childsite has been provided along with a position option. The site can be located at the n, s, e, w, and center positions. The option set is quite extensive, providing specification of the initial directory, search commands, filter mask, no-match string, and margins to name but a few.

To illustrate, the following example creates a fileselectiondialog for selection of c language source and header files within directories using the *-mask* option. It also turns off the display of the selection entry field, forcing users to select only

Figure 6-41. Selectiondialog

existing files, rather than allowing them to type in the name of a non-existent one.

```
fileselectiondialog .fsd -selectionon no \
   -mask "*.\[ch\]"

.fsd activate
```

Canvasprintdialog

[incr Widgets] provides a handy dialog for pre-formatting the postscript output of a Tk canvas widget called the canvasprintdialog. This dialog displays a shrunken copy of a canvas in a stamp area. It supports configuration of the page size, orientation, and print command. The output may be posterized or in other words spread out over MxN pages, and even stretched to fit on your selected page size. In addition, the widget provides a command for easy computation of typical paper sizes. This can be quite useful when constructing your input canvas size.

The basic functionality of the canvasprintdialog is contained in a separate mega-widget called canvasprintbox. Structurally, the canvasprintdialog is a dialog mega-widget containing a canvasprintbox. Since the canvasprintdialog is dialog

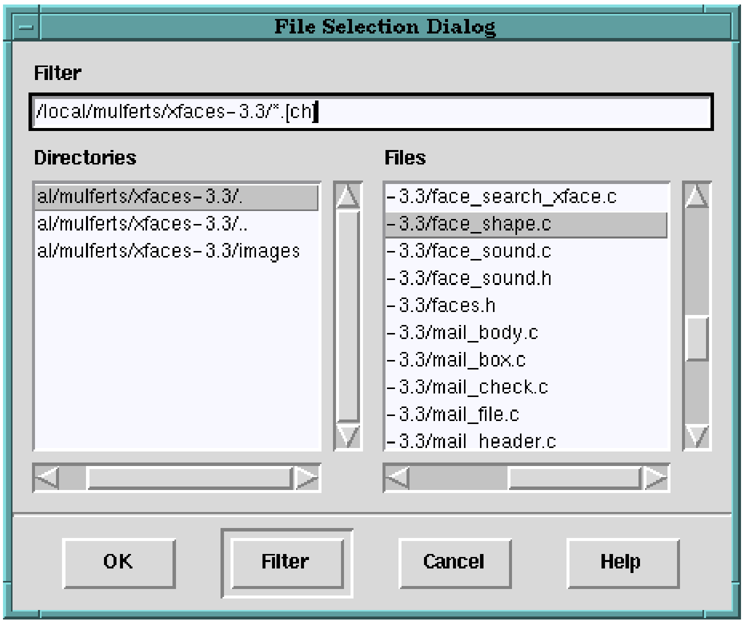


Figure 6-42. Fileselectiondialog

based, all the dialog commands for controlling button display and their callback commands are available.

The canvasprintdialog layout contains option menus for output, paper size, and orientation. Selection of the output option modifies the display accordingly. Specifically, file output selection demands file name input whereas printer output requires a printer command. The stamp appears centered in the dialog above the checkbuttons for posterization and stretching. The following example depicts the canvasprintdialog displaying an source canvas filled with various geometric shapes.

```
canvas .c
pack .c -expand 1 -fill both

# Fill canvas with shapes.

Canvasprintdialog .pcd -modality application \
   -pagesize A4

.pcd setcanvas .c

if {[.pcd activate]} {
```

```
    .pcd print
}
```

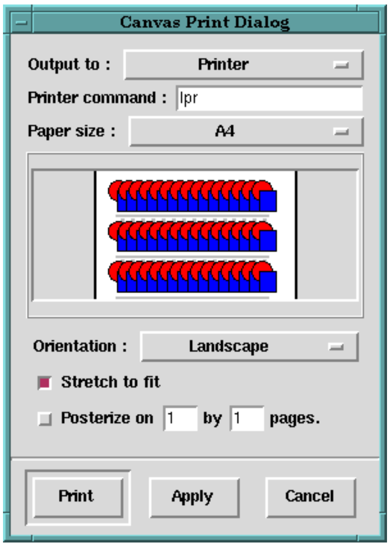


Figure 6-43. Canvasprintdialog

Hyperhelp

The [incr Widgets] hyperhelp widget is a HTML hypertext help viewer. HTML is the language of not only the web, it has become one of the more common formats for help files. At the time of this writing, the hyperhelp widget is able to process HTML 1.0 codes. Support for the more advanced codes is forthcoming.

The hyperhelp interface is simple. Help files are displayed in a scrolled region below a menubar containing topic and navigate pulldowns. The topic menu lists the configured topics. As they are selected, the associated html help file is rendered in the scrolled area. The navigate menu supports moving through the help file history.

From a programming perspective, the hyperhelp mega-widget is similar to other dialogs. The widget supports modal blocking, activation, and deactivation. In addition, a *showtopic* command is provided to initialize or change the current topic. Topics are directly mapped to html help files. The option set enables the help file source directory and topic list to be configured.

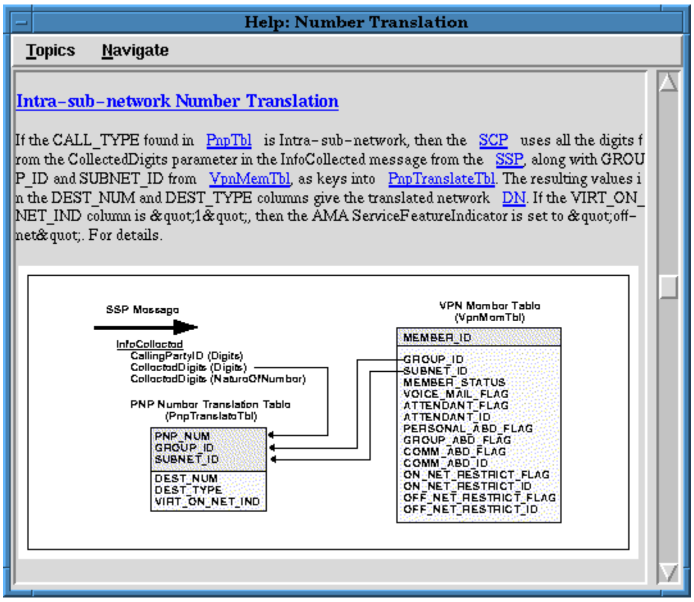


Figure 6-44. Hyperhelp

Example Application

Thus far, each [incr Widgets] mega-widget has been presented separately. In this section we'll put it all together and illustrate the use of [incr Widgets] in the construction of a complete application. We'll build a multi-file text editor, capable of splitting the screen either vertically or horizontally. All in all, we'll do the whole thing in less than 170 lines of Tcl code, illustrating the use of dialogs, panedwindows, and scrolledtext [incr Widgets] mega-widgets.

The main window for the editor will contain a menubar and panedwindow. We'll create scrolledtext widgets on demand as children of the panes. The menubar will house the actions available. The following code segment creates our main window and sets the dimensions using the option database.

```
option add *edit.width 6i startupFile
option add *edit.height 6i startupFile

frame .mbar -borderwidth 2 -relief raised
pack .mbar -side top -fill x

panedwindow .edit
pack .edit -expand yes -fill both
```

The menubar needs a file menu for loading and saving files. The actions will activate dialogs for file selection and possible error notification. We’ll also need a means for quitting the application. The following code adds the file menu to the menubar and binds the accelerators to the window. The procedures for loading, saving, and clearing the text areas will be defined later.

```
menubutton .mbar.file -text “File” \
   -underline 0 -menu .mbar.file.menu
pack .mbar.file -side left -padx 4

menu .mbar.file.menu
.mbar.file.menu add command -label “Load...” \
   -accelerator “  ^L” -underline 0 \
   -command file_load
bind . <Control-KeyPress-l> \
   { .mbar.file.menu invoke “Load...” }

.mbar.file.menu add command -label “Save As...” \
   -accelerator “  ^S” -underline 0 \
   -command file_save_as
bind . <Control-KeyPress-s> \
   { .mbar.file.menu invoke “Save As...” }

.mbar.file.menu add separator

.mbar.file.menu add command -label “Quit” \
   -accelerator “  ^Q” -underline 0 \
   -command {clear_text; exit}
bind . <Control-KeyPress-q> \
   { .mbar.file.menu invoke Quit }
```

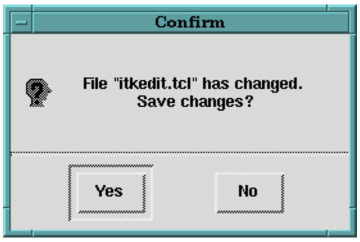
Next, we’ll add menubar commands for splitting the screen and changing the orientation under a view menu. The changing of orientation requires us to configure the -*orient* option of the panedwindow appropriately. The procedure for splitting the window will come later.

```
menubutton .mbar.view -text “View” \
   -underline 0 -menu .mbar.view.menu
pack .mbar.view -side left -padx 4

menu .mbar.view.menu
.mbar.view.menu add command -label “Split” \
   -underline 0 -command split_view
.mbar.view.menu add separator
```

```
.mbar.view.menu add command -label "Horizontal" \
   -command {.edit configure -orient horizontal} \
   -underline 0
.mbar.view.menu add command -label "Vertical" \
   -command {.edit configure -orient vertical} \
   -underline 0
```

We'll be needing a couple of messagedialogs for error reporting and exit confirmation. The type of errors we need to report include not being able to write out the file to disk. Exit confirmations request the user to save changes prior to exiting. Since both these dialogs will need to vary their text message to include the file name, we'll create the dialogs now and activate them later whenever we need, configuring the text as appropriately.

```
messagedialog .notice -title "Notice" \
   -bitmap info -modality application
.notice hide OK
.notice hide Help
.notice buttonconfigure Cancel -text "Dismiss"

messagedialog .confirm -title "Confirm" \
   -bitmap questhead -modality application
.confirm hide Help
.confirm buttonconfigure OK -text "Yes"
.confirm buttonconfigure Cancel -text "No"
```



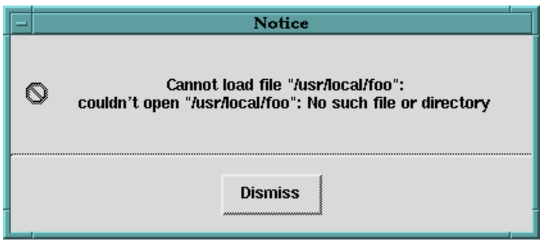


Figure 6-45. Multi-file text editor message dialogs

We'll need a means of selecting files for loading or saving. The straight out of the box [incr Widgets] fileselectiondialog almost fits the bill, but not quite. When users select a file to be loaded, they also need to select a pane to place it in. What we need is a fileselectiondialog with an embedded optionmenu for pane selection. Fortunately, the fileselectiondialog can easily be extended using the childsite provided for just this sort of situation. We'll just stick an optionmenu in its childsite and fill the menu with the list of available panes, adjusting the contents with each split of the editor. Since we'll have to access the optionmenu later in various procedures, we'll save its widget path name in a variable.

```
fileselectiondialog .files -title "Files" \
   -childsitepos s -modality application
.files hide Help

set PaneMenu "[.files childsite].panes"
optionmenu $PaneMenu -labeltext "Edit Window:"
pack $PaneMenu -pady 6
```

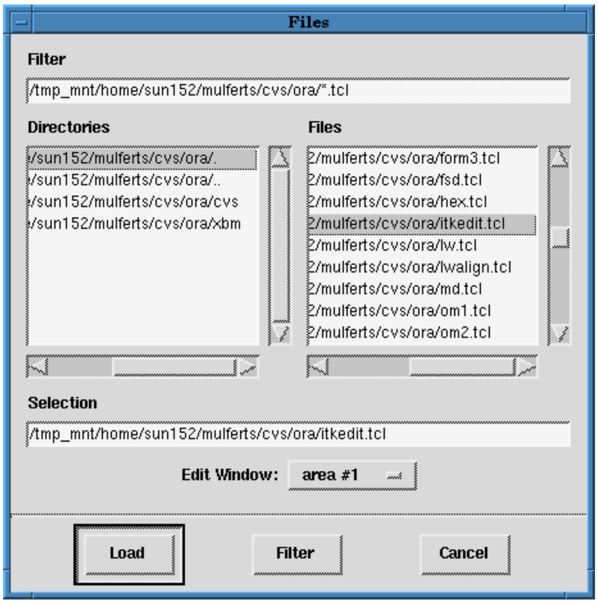


Figure 6-46. Multi-file editor file selection dialog

All that is left is to write a few procedures. First we need the procedure to load a file into a selected pane. It will be tied to the load menu option and need to activate the fileselectiondialog, waiting for a selection. Once a file has been chosen, we'll retrieve it from the dialog and get the selected pane from the option menu. Next, we'll clear the desired pane and import the file. Should an error occur during loading, we'll use the notification messagedialog to alert the user of the problem. Finally, we'll set the title for the scrolledtext widget to be the filename.

```
proc file_load {} {
   global FileName PaneMenu

   .files buttonconfigure OK -text "Load"

   if {[.files activate]} {
         set fname [.files get]
         set pane [$PaneMenu get]
         set win [.edit childsite $pane]
         set cmd {$win.text import $fname}

         clear_text $win

         if {[catch $cmd err] != 0} {
               .notice configure -bitmap error \
                     -text "Cannot load file\
                           \"$fname\":\n$err"
               .notice activate
               return
         }

         $win.text configure \
               -labeltext "file: $fname"
         set FileName($win) $fname
   }
}
```

Similarly, we need procedures for saving files. We'll use the same fileselection-dialog that was used before, configuring the label of the buttons appropriately. Upon selection of the save button, we'll determine the pane to save along with the file name and attempt to perform the operation. Again, should problems occur, the notification dialog will be used. We'll separate this process into two different procedures. The file_save_as procedure prompts for the pane and output file name and the file_save procedure does the actual exporting of the file. By dividing the work, we can reuse the file_save procedure later when we need to save changes as we clear the text. This way we won't need to repeatedly prompt when saving multiple files.

These two procedures also introduce a couple of global variables. The first variable, FileName, is an array that associates panewindow childsites with the filename occupying the space. Since the "file save as" action allows the user to change file names, we'll need this array to hold all the names associated with different panes. In order to remember where the user has made changes since the last save, we'll associate the array of panes with an array of dirty flags (boolean values that are cleared when the file is saved and set when a keystroke is made).

```
proc file_save_as {} {
   global FileName PaneMenu

   .files buttonconfigure OK -text "Save"

   if {[.files activate]} {
         set pane [$PaneMenu get]
         set win [.edit childsite $pane]
         set FileName($win) [.files get]

         file_save $win
   }
}

proc file_save {win} {
   global FileName FileChanged

   set cmd {$win.text export $Filename($win)}

   if {[catch $cmd err] != 0} {
         .notice configure -bitmap error \
               -text "Cannot save file\
                     \"$FileName($win)\":\n$err"
         .notice activate
         return
   }

   set FileChanged($win) 0
   $win.text configure \
         -labeltext "file: $FileName($win)"
}
```

Another procedure we'll need is one to clear a pane and save any pending changes. We'll also add in some versatility. The areas parameter will take the childsite of the pane to be cleared as a argument. Should one not be provided, then we'll set it to all the panes using the *childsite* command. This allows us to kill two birds. We can use this procedure to clear a single pane as we load in new files and also to clear out all the panes during cleanup prior to exiting.

```
proc clear_text {{areas ""}} {
   global FileName FileChanged FileWindows

   if {$areas == ""} {
         set areas [.edit childsite]
   }

   foreach win $areas {
         if {$FileChanged($win)} {
               set fname \
                     [file tail $FileName($win)]
               .confirm configure \
                     -text "File \"$fname\"\
                           has changed.\
                         \nSave changes?"
               if {[.confirm activate]} {
                     file_save $win
               }
         }

         $win.text clear
         set FileChanged($win) 0
   }
}
```

Last, we'll need a procedure that splits the screen, adding a new pane on the bottom and filling its childsite with a scrolledtext widget. We'll also initialize our global variables used for tracking changes and associating file names with panes. In addition, we'll update the contents of our optionmenu in the fileselectiondialog to include the new pane and do the dirty flag binding.

```
proc split_view {} {
   global FileName FileChanged FileWindows PaneMenu

   set pane "area #[incr FileWindows]"
   .edit add $pane -minimum 100
   $PaneMenu insert end $pane

   set win [.edit childsite $pane]
   set FileName($win) untitled.txt
   set FileChanged($win) 0

   scrolledtext $win.text -wrap none \
         -labeltext "file: $FileName($win)" \
         -hscrollmode none -vscrollmode dynamic
```

```
        pack $win.text -expand yes -fill both

        bind [$win.text component text] <KeyPress> \
              "set FileChanged($win) 1"
    }
```

With our procedures defined, we need one last bit of initialization to kick start our editor. We'll need to set our global variable that tracks the number of panes to 0 and then split the screen the first time. This adds our first pane and scrolled text area, producing the following screen layout.

```
    set FileWindows 0
    split_view
```

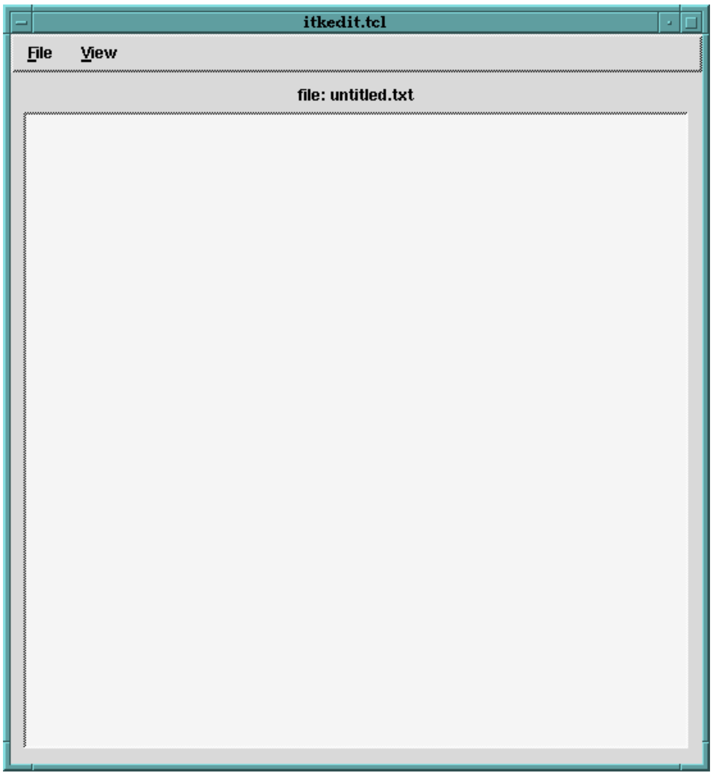


Figure 6-47. Multi-file text editor initial layout

Now as we use it, splitting the screen and loading new files it would look like this. Not a bad little split screen editor for 170 lines of code.

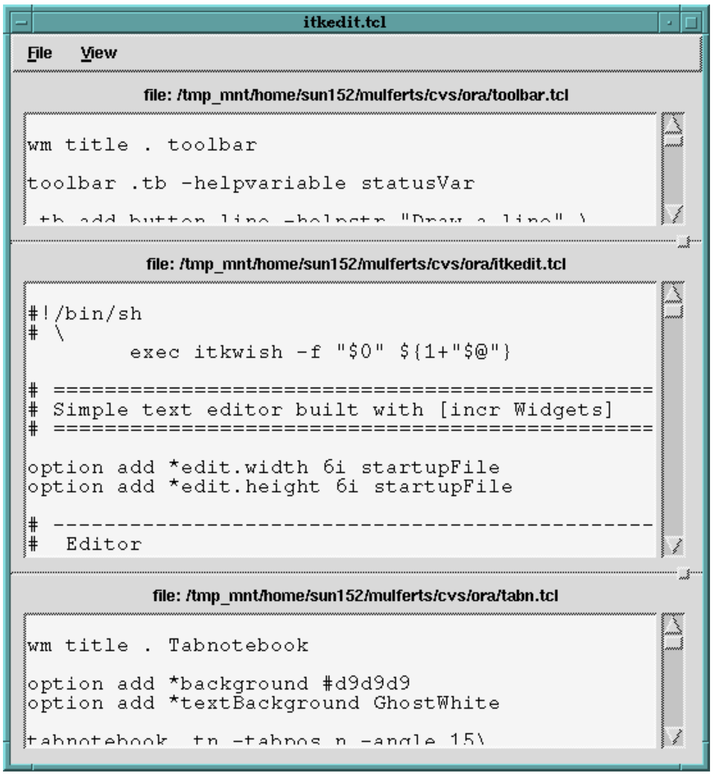


Figure 6-48. Multi-file text editor with several files opened

Contributing to [incr Widgets]

[incr Widgets] is a continuing effort. We have many new mega-widgets currently under development and contributions are greatly appreciated. [incr Widgets] is a good start in the direction of establishing a strong set of object-oriented mega-widgets, but I swear that every time I go back and look at the source I see an even better way something could have been accomplished. Should anybody within the Tcl/Tk community come upon an good improvement, a great enhancement, or an awesome new mega-widget altogether, please don't hesitate to send it to the author listed in the header or myself, mulferts@spd.dsccc.com, as moderator. I or any of the development team members are always available via email for a technical interchange of ideas.

The [incr Widgets] distribution is moderated. Contributed mega-widgets must be of good quality and complete with documentation, tests, and demonstrations. Please follow the coding style found in the distribution source. This includes man page and test script formats as well. The languages and extensions on which [incr Widgets] is based are of high standards. [incr Widgets] strives to attain this same level.

The mega-widgets we are looking for include but are not limited to the following list. Should you be interested in signing up for any of those listed here, please send email to mulferts@spd.dsccc.com. Also, should you have any ideas on others not listed, please send email to the same.

Timer	Fontdialog	Biswitch	Colordialog
Ballonhelp	Instrumentation	Graphs	Table
Tree	Spreadsheet	Printdialog	Labeledframe

[incr Widgets] Distribution

The [incr Widgets] distribution is included with [incr Tcl] version 2.0 and greater. It is available via ftp at *ftp.aud.alcatel.com/tcl/extensions*. Consult the included release documentation for installation notes. For the latest in distribution information, consult our web site: *http://www.tcltk.com/iwidgets*.

The [incr Tcl] distribution will always include the most current release of [incr Widgets] possible at the time of its release. It is anticipated that [incr Widgets] will change more rapidly than [incr Tcl]. This being the case, in between [incr Tcl] releases, the latest version of [incr Widgets] is also separately available via ftp at the same location as well as the web site.

The version number of [incr Widgets] tracks the release of [incr Tcl], identifying its compatibility. The version numbering system for [incr Widgets] includes an extra number. For example version 2.0.3 of [incr Widgets] is compatible with [incr Tcl] 2.0. As the minor number of [incr Tcl] increases the second digit of the [incr Widgets] version varies. This makes for easy release compatibility identification.

Acknowledgments

Thanks to the original development team, comprised of Mark Ulferts, Bret Schuhmacher, Sue Yockey, Alfredo Jahn, John Sigler, and Bill Scott. Also thanks to Mark Harrison for his influence, confidence, and ideas. Much credit goes to Michael McLennan, creator of [incr Tcl] and [incr Tk], for providing beta copies, training, assistance, and his infectious enthusiasm. Thanks also to DSC Communications for picking up the copyright and supporting the public release of this software. Specifically, Allen Adams, Larry Sewell, Gil Stevens, Mahesh Shah, Pardeep Kohli, and Pete Kielius.

Additional thanks to Ken Copeland for enhancing the panedwindow, smoothing out the scrolling and making panes bumpable. Also, thanks to our many contributors. Most notably, Tako Schotanus for contributing the canvasprintdialog mega-widget. Also, Kris Raney who provided the Hyperhelp and Feedback mega-widgets based on Sam Shen's work with tkinspect.

Thanks to John P. Davis for creating the [incr Widgets] "Flaming Toaster" logo that can be seen at the [incr Widgets] home page *http://www.tcltk.com/iwidgets* and to WebNet Technologies, http://www.wn.com/, for their assistance is designing the [incr Widgets] web site, as well as hosting it.

Special thanks to my wife Karen for supporting this effort and to our two girls, Katelyn and Bailey, who occasionally shared the PC with me. Also thanks to my Discman and its relentless power supply as well as my CD collection. No tunes ... no mega-widgets.