The Fast Light Toolkit Extension for Tcl/Tk

Version 0.4

Iain B. Findleton

This document describes the binding between the Fast Light Tool Kit (FLTK) and the Tcl/Tk programming language. The language binding enables the creation of graphical user interface based applications that are built using the widget set provided by the FLTK library. Both the FLTK library and the Tcl/Tk application development language are distributed under various flavours of the GNU Public Licence.

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

1 Introduction

The Fltk extension is a dynamically loaded extension to the Tcl application environment that provides an interface between Tcl and the Fast Light Toolkit (FLTK) GUI toolkit. The Fltk extension implements the collection of widgets available from the FLTK toolkit, and a number of supporting commands and features that provide an application development support environment that resembles the Tk application development environment.

The set of Tcl commands that are implemented by the Fltk extension have names that are similar to the standard set of Tk commands. Many commands support options that are identical in form and content to those that are implemented by the Tk application environment. The default usage of the Fltk extension distinguishes between Tk commands and Fltk commands by the use of capitalization of the first letter of the extension commands. This allows the Fltk extension to co−exist with the Tk command set so that applications can make use of both GUI development environments at the same time.

The Fltk extension provides a set of widgets that, while they may offer similar functionality to the corresponding Tk widgets, will not provide identical functionality, and hence, will usually require configuration options that are not the same as those of the Tk widget. For example, the implementation based on the Fltk toolkit defines more than 6 types of button widget, while the Tk toolkit defines only a single, configurable button widget. Many of the options of the Fltk extension's *Button* widget are the same as those of the Tk *button* widget, but some additional features of buttons provided by the toolkit are also implemented.

1.1 Features of the FLTK Tool Kit

The Fast Light Tool Kit (FLTK) is a platform independent GUI development library that delivers a generic interface to a minimalist windowing system. FLTK implementations are available for a wide variety of computing platforms, including the popular Microsoft Windows and UNIX environments. The strategy of the FLTK approach is to interact with the native window manager at the level of a generic, largely undecorated window, and to provide through the use of a limited set of drawing primitives all of the widgets that the tool kit supports. This approach contrasts with, for example, that of the WIN32 API which provides an interface between a large variety of windows and pre−defined widgets.

The principle advantages of the FLTK approach are that the interface to the window manager on any given platform is always native, so the speed of the widget drawing code is always as fast as the platform will support, and that the application development API is always the same, regardless of the platform in use. This latter feature makes cross platform development much simpler.

A third feature of the design of FLTK is the use of generic geometric and text generation functions to build up the appearance of widgets. FLTK uses boxes and labels for most widget rendering operations. The underlying functions that implement these generic operations are accessed through a lookup table based on a type specification. Because of this design, applications can replace the standard box and label functions with their own versions, allowing the rendering of widgets using alternative GUI tool kits, such as the OpenGL tool kit, without the need to change any of the code used to draw the widgets themselves. This powerful design feature is used in the Tcl extension to implement all widgets as either OpenGL based or FLTK based graphics.

The design and implementation decisions characteristic of the FLTK tool kit have the happy result that FLTK based widgets have the same appearance regardless of the implementation target, and, applications have complete control over widget appearance.

1.2 Limitations of the FLTK Tool Kit

FLTK is fast, but it is also light. Conspicuously absent is a device abstraction, so printing is a labour. The number of colors available is limited to a 256 color palette, and there are limits to just about everything, including fonts, boxes and labels. The tool kit is decidedly less feature rich than either the X tool kit or the Windows API. The dearth of features definitely limits the scope available for the creation of truly exotic appearances.

FLTK is a C++ based API, and is therefore limited to platforms that have available a good C++ compiler.

Because FLTK draws all its own widgets, it does not benefit from the investment made by purveyors of some operating systems in advanced widget behaviours. For example, FLTK widgets may appear rather plain compared to those of the Windows XP operating system, or the latest GNOME widget set. For those who are sensitive to the appeal of highly polished widget sets FLTK is probably not attractive, unless there is the will to write appropriate drawing routines to get the needed effects.

1.3 FLTK and TCL/TK

The extension package that implements the FLTK bindings is a TCL extension package that can co−exist with the TK extension package. Both types of widgets, FLTK and TK, can appear on the screen as part of the same application, however, the management of windows for each package must be effected using their respective package specific commands. Windows can not be intermixed, although, it is possible to wrap FLTK windows in a TK window, and vice versa.

In general, TK widgets are more primitive, and hence much more configurable than are the FLTK widgets. Most commonly used FLTK widgets are actually mega−widgets, combining the functionality of more than one basic widget to produce something that is ready to use out of the box. Because the widgets are typically compound objects, they are necessarily less configurable than are TK widgets. On the other hand, the amount of code needed to produce an FLTK application will typically be considerably less than that found in a TK application.

In developing the FLTK extension, emphasis has been on providing basic functionality in as highly automated fashion as possible. This has left the FLTK extension package highly functional while missing many of the advanced features of the TK package. There is nothing in the FLTK package that compares favourably to the TK text widget, for example, although there are features in the FLTK package, such as state variable bindings, that make application development a lot less time consuming than would be the case with TK.

A final note relates to geometry management. The TK package has extensive and elaborate geometry management features implemented in several different fashions. The FLTK package has relatively limited geometry management. This can be a significant constraint on the development of certain styles of advanced GUI applications, but it also greatly reduces the time spent in configuring various geometry managers for various platforms. Because the FLTK geometry management approach is pixel based, widgets will always appear the same on all platforms and on all displays.

1.4 FLTK and Other Extensions

The FLTK extension is driven by the TCL event loop. To the extent that another extension may interfere with the TCL event mechanism, the FLTK extension should co−exist happily with any other extension to either TCL or TK. The only other issue is the use of the TCL name space. Like all TCL applications, the FLTK extension uses up the TCL name space for its command set. If required, the extension can be set to use its own name space so that name conflicts can ultimately be resolved using the TCL namespace mechanism.

2 Acquiring and Installing the FLTK Extension for TCL

In order to use the FLTK extension for TCL you must have installed on your platform a version of the Tcl distribution that is at release 8.1 or later. The extension makes use of the Tcl stubs mechanism to integrate itself with the Tcl development environment. Earlier releases of Tcl did not support stubs, so those wishing to make use of this extension with earlier Tcl releases will have to modify the source code and compile a private version of the extension.

If you already have Tcl or Tcl/Tk installed on your computer, then the most direct method of installation is to acquire one of the binary distributions of the FLTK extension package and install that on your machine. Binary distributions are available for Microsoft Windows and the Red Hat Linux operating systems. Other operating systems may require that you build the extension from the source distribution.

The binary distributions of the FLTK extension package are built with the static versions of the FLTK library, so, if you do not need to build from source, you do not need to have FLTK installed on your machine. If you wish to modify the extension, or if you need to build from the source distribution, you will need to get the source distribution of the FLTK package and install that on your machine.

2.1 TCL/TK Distributions

As of December, 2001, the current preferred source for Tcl/Tk distributions is www.activestate.com. Active State is distributing a number of commercial and public domain versions of various scripting languages and development tools. The Tcl/Tk distribution is available for free download from their site. Distributions are available in both source and binary formats for Windows and for Linux. Tcl/Tk has fairly wide penetration in the UNIX world, and if you are running on a UNIX/Linux based machine it is probable that Tcl/Tk is already installed.

Tcl/Tk has a large and active user community that can provide help with installation and programming issues should you need it. There are also several books available on Tcl/Tk and the internet has a large amount of online documentation, example applications, tutorials and other resources that make Tcl/Tk a very good target language for both small and enterprise scale application development projects. If you need to access these Tcl/Tk resources just post your questions to the comp.lang.tcl usenet group.

2.2 FLTK Distributions

As of December 2001 the preferred source for the Fast Light Tool Kit distribution is www.fltk.org. FLTK has an extensive user base on a wide variety of platforms. Distributions are available in both source and binary format for Windows and Linux machines. FLTK is distributed freely on the internet.

The FLTK web site provides various user resources to aid in the implementation of the tool kit and in the development of FLTK based applications. There are many examples of FLTK applications included with the distributions, extensive documentation, and a point and click style application generator that can be used for rapid widget development. The FLTK community has an active chat group and a mailing list which can serve as a good access point to the available FLTK knowledge base.

2.3 Distributions of the FLTK Extension for TCL/TK

The current source of extension distributions is the Custom Clients web site at http://pages.infinit.net/cclients/software.htm. This site contains a number of distributions for Tcl/Tk extensions, including both the source and binary distributions available for the FLTK extension. Instructions for downloading and installing the various distributions are maintained on the web site.

The FLTK extension for Tcl/Tk is distributed freely in source and binary formats. Binary distributions are available for both the Windows and the Red Hat Linux operating systems. Currently, support for the package is limited to e−mail based queries to the author.

3 Introduction to Tcl Programming

The following chapter contains a very brief overview of the Tcl language and its use as an application development environment. Tcl is a widely used scripting language that has enjoyed many years of development. Tcl distributions come with extensive documentation and there is a large amount of information on the use of Tcl available on the internet. While this overview will get one started with the language, it is not a complete reference to Tcl and its facilities. Readers are encouraged to consult one of the many excellent books on Tcl that are available. To be able to effectively program in Tcl, readers will need to become familiar with the contents of the on line documentation that is included in standard Tcl distributions.

Another useful resource is the Tcl'ers Wiki. This searchable database contains a large number of hints, tips, explanations and code examples that are useful to all levels of Tcl developers. It is available on the internet at http://wiki.tcl.tk. The Wiki has extensive links to other Tcl resources on the internet and is the place to look for answers to Tcl questions. Additionally, there is an active usenet group, news://comp.lang.tcl, where technical issues about Tcl are discussed.

3.1 Writing Tcl Programs

The Tcl language is a fully functional application development language. Tcl applications are text files that contain a sequence of statements that form a Tcl script. Tcl scripts are executed by an application that interprets the Tcl statements in the script. The interpreter reads the script files, parses the statements in the script and executes the Tcl commands that are found in the statements. The Tcl language specification provides a set of commands that can be used for creating and initializing variables, evaluating expressions, creating code blocks which can be executed as subroutines with variable parameters, controlling the flow of program execution, and for performing input/output operations to various types of channels.

There are several different Tcl interpreters commonly available. The standard Tcl shell, *tclsh*, is an interactive program that runs on most computer systems which will accept, in addition to the set of commands characteristic of the local platform, any Tcl statement. Similarly, the *wish* shell is a version of the Tcl interpreter that can be used to develop GUI applications using Tcl scripts based on the standard X Windows API. The *fltkwish* interpreter that is described in this document is a version of the Tcl interpreter that, like *wish*, is used to develop GUI applications based on the Fast Light Tool Kit API. The *expect* interpreter is a version of the Tcl interpreter that is adapted for use in the automation of applications that need operator control. All of these interpreters, and many others, implement the Tcl language as the basis of script development, and therefore have a common language syntax and basic command sets.

Tcl provides a rich set of commands that take a fairly large number of switch options and parameters. Tcl interpreter installations typically include extensive on line documentation that describes the details of the available commands and their options and parameter meanings. A complete discussion of these parameters and options is beyond the scope of this text, however, the reader can readily access the on line Tcl documentation for the relevant Tcl interpreter to discover the details of available commands. Under UNIX systems, the documentation is available using the *man* command, while under Windows environments, a *WinHelp* database is available.

Developing a Tcl application can be accomplished either by using a text editor to create a Tcl script which has the list of Tcl commands to be executed and then passing the script to the standard input stream of an interpreter, loading the text file into the interpreter using the *source* command, or by starting up an interpreter and typing the commands directly into the command prompt. Here is the standard test program for computer languages as used in Tcl.

```
puts "Hello, world!"
```

This program will write the text string "Hello, world!" to the standard output of an interpreter. If this statement was in a text file named "myfile.tcl", then at the interpreter shell prompt one could load the program using the command:

```
source myfile.tcl
```

Most interpreters will also accept the name of the source file as a parameter on the command line that invokes the interpreter. Entering a command like:

```
tclsh myfile.tcl
```

at a Unix or Windows command prompt will start the interpreter and automatically load the file.

3.2 Tcl Language Syntax

The Tcl language syntax is based on the use of tokens delimited by white space. Tokens are strings of characters in the printable ASCII subset. A token may contain white space, such as blanks, tabs and newline characters, if it is protected by quotation marks or brackets. In the above example, the string "Hello, world!" is a token because the quotation marks protect the string between them. For this reason, the presence of a blank in the string does not result in the creation of 2 tokens. This example could also have been written as follows:

```
puts { Hello, world! }
```

where in this case the curly brackets have the effect of protecting the string and producing a single token. Tcl also uses square brackets in its syntax to effect protection of the contents between them, however, square brackets have the additional effect of causing immediate evaluation of the string between them as a Tcl statement. For example, the Tcl statement:

```
puts [expr 5 + 3]
```

will print the number 8 on the standard output stream. Here, the token created by the square brackets is "expr 5 + 3" which is a Tcl statement that says "compute the result of the expression 5 + 3".

There are 4 characters of special significance in Tcl. A *newline* character will signal the end of a statement, unless it occurs within a set of brackets or quotation marks that protect it as part of a token. The *backslash* character will allow for the continuation of a statement beyond a *newline*, provided that it is the last character of the statement before the *newline*. A *semi−colon* character will signal the end of a statement, so by using a *semi−colon* many statements can be placed on a single line. The # character, when it is the first character of a statement, indicates a comment. All of the text on a comment line, up until the end of the statement signaled by a semi−colon or a newline character, is ignored by the interpreter.

Here is a block of code that demonstrates the use of these special characters:

```
# A comment line

puts "Hello, world" ;# 2 statements on a single line. The second statement is a comment!

# A backslash is used to extend a statement to more than 1 line

puts "now is the time \
     for all good men to come to the aid of the party!"
```

The backslash character has an additional use as an escape that specifies special character patterns. The common use is to insert tabs, newlines or other non printable characters into text strings. For example, a text string might have embedded tab characters like the following:

```
puts "\tThis\tIs\ta\ttabbed\tstring\n"
```

The list of valid escapes is described in the Tcl documentation and is similar to that used by the C programming language's *printf* function.

A final special case is the $ character. This character is used as a dereferencing operator when it is the first character in a token. The effect of this character is to return the current value of the variable identified by the remainder of the string. For example, the statement:

```
puts $Data
```

returns the current value of a Tcl variable named *Data.*

3.3 Variables

Tcl has only 1 type of variable, the string. The strings used in Tcl may contain any type of data, including binary data, and the internal representation of the data within the Tcl interpreter will, in general, be related to the machine architecture of the computer platform that is being used to run the interpreter. The external representation, however, is the only representation that the language exposes to the user, and this external representation is always a string representation of the variable.

Variables are created using the *set* command and deleted using the *unset* command. For example, the command:

```
set Number 10.352
```

creates a variable named *Number* which would, presumably, be used to perform some calculation. One could get rid of this variable by using the following command:

```
unset Number
```

Typically, the *unset* command is not used because Tcl variables are automatically deleted when they go out of scope. A variable gets its scope depending on where it is defined in a script, and it goes out of scope according to how the program flow within the script occurs. Tcl scope defines 3 categories of variables, *local*, *global* and *namespace*. *Local* variables go out of scope when the program flow leaves the code block within which they are created. *Global* and *namespace* variables go out of scope when the application terminates.

Variable names can be formed from any string of characters that does not contain white space. The only qualification to this statement is that of *namespace* variable. The *namespace* mechanism is used to partition the variable names space for ease of manageability. *Namespace* variable names employ a special syntax to qualify the variable name. Here is an example of the use of a *namespace* variable:

```
set Data::Number 10.352
```

Note the use of the double colon to signify the *namespace* qualifier. *Namespace* is convenient because Tcl *global* variables must be unique across an application. The *namespace* mechanism makes it possible to define *global* variables that have the same name, but exist in different namespaces.

There is one more syntax that is used in Tcl to implement arrays of variables. The syntax uses regular brackets to specify array elements. The following statement:

```
set Data(Number) 10.352
```

defines an element of an array named *Data* whose element is *Number* with a value of 10.352. As with variable names, there are no particular restrictions on the formation of the array index names, other than the use of white space. For instance, simulation of a 2 dimensional array element might be done with a statement like:

```
set Data(Number,1) 10.352
```

The value of a variable can be accessed in one of 2 ways. The *set* command can be used with only 1 parameter:

```
set Number
```

This statement will return the current value of the variable *Number*. The other syntax is the use of the dereferenceing operator $. The statement:

```
puts $Data(Number,1)
```

will print the current value of the array element.

3.4 Tcl Lists

A string of tokens separated by white space and grouped using brackets or quotation marks is a Tcl list. Lists are used extensively in Tcl programs for manipulating data, and the language provides a number of commands specifically for the purpose of manipulating lists. There is no formal distinction between a text string used in a statement such as:

```
set text "This is a text string"
```

and in a statement such as:

```
set text {This is a text string}
```

In both cases, the data stored in *text* will appear to be identical, however, the latter syntax creates a Tcl list, while the former does not. Regardless of how the variable *text* is initialized, Tcl list processing commands will produce the same results. This is because the interpreter will attempt to treat any string as a list if a list command is applied to it.

Lists are useful because the list processing commands available in Tcl allow easy access to and manipulation of list elements. Using brackets, a list of lists can be easily created as follows:

```
set fruit { { apple red } { orange orange } { grape green } }
```

This creates a list of 3 elements, each of which is itself a list of 2 elements. One possible view of the Tcl language is that everything is a list, and that all of the Tcl commands are operators on lists. For this reason, it is worth the effort to fully understand the available documentation for the Tcl list functions.

3.5 Command Evaluation

Tcl is a language that evaluates its statements through a process of string substitution. The result of every Tcl statement is a string. When a Tcl statement is passed to the interpreter for evaluation, the statement is parsed into tokens, and the tokens are assembled into commands and parameters for those commands. A token itself can be a Tcl statement, so the evaluation process can be recursive.

The effect of passing a Tcl statement to the interpreter for evaluation is the removal of one level of brackets from the string that represents the command. All statements have an implied set of brackets around the statement itself. The interpreter will act to remove the inner most set of brackets first, replacing that token with the result of its evaluation, then proceed to process the next innermost set of brackets, continuing until the evaluation process is complete. For example, the statement:

```
puts [expr 5 + 3]
```

is evaluated first to the statement:

```
puts 8
```

and then evaluated to execute the command *puts* which writes the parameter 8 to the output stream.

In contrast, the statement:

```
puts { expr 5 + 3 }
```

will be first evaluated to the statement:

```
puts "expr 5 + 3"
```

which will write "expr 5 + 3" on the standard output stream of the interpreter. In both cases, the interpreter removed one level of brackets. In the former case, the type of brackets indicated that the resulting token should be evaluated as a command, in the later case, no evaluation occurs.

By default, Tcl interpreters always evaluate input statements. The language provides a special command, *eval*, to initiate this process within a script sequence itself. The *eval* command concatenates all of the tokens following it in the statement into a string and then carries out a bracket reduction operation. For example, the command:

```
eval  exec start http://pages.infinit.net/cclients
```

could be used on a Windows computer to start Internet Explorer and load the Custom Clients home page. In this particular case, there are no brackets to remove, so the operation is equivalent to the command:

```
exec start http://pages.infinit.net/cclients
```

There are sometimes occasions when it is desirable to carry out the bracket reduction operation but to not evaluate the resulting string as a command. Tcl provides the *subst* command for this purpose. The commands:

```
set a http://pages.infinit.net/cclients
set cmd [subst "exec start $a"]
```

will put the string "exec start http://pages.infinit.net/cclients" into the variable cmd. The *eval* command can then be used to invoke the command as follows:

```
eval $cmd
```

One of the useful attributes of the *eval* command is that it can be used to apply the same command to all of the values in a list. Suppose that a list of items is created in a Tcl variable as follows:

```
set list { a b c d e f g }
```

and suppose that these items are to be added to a list box for display to the user. The list box is created using the TclFltk command *Listbox*. This widget supports the *add* function command. The following set of commands create the list box and add all of the elements of the variable *list* to the list box:

```
Listbox t.l; eval { t.l add } $list
```

In this example, the effect of the *eval* command is to expand the token "t.l add" with each element of the *list* and then evaluate the resulting Tcl statement. This is equivalent to a series of commands that would look like the following:

```
t.l add a b c d e f g
```

Since this is a valid command for the *Listbox* command, it is an easy method of adding lists to the widget.

3.6 Expressions

Expressions are used for performing computations, such as the addition of numbers. Tcl provides the *expr* command that takes its parameters and evaluates the expression that is defined by the parameters. The parameters are operators and operands ordered in the usual manner of algebraic expressions. The *expr* command supports the usual arithmetic operators, the use of parentheses for specifying precedence, and can access a library of built in functions typical of most programming languages.

For well formed expressions, the result of the *expr* command is the mathematical result of the expression. Tcl performs all of the required conversions to reduce the expression to a homogeneous set of operands, and then applies the operators to the operands. The result is converted back to a string and returned to the script. For example, the command:

```
expr sin(2 * 3.14159 * 25.0 / 100.0)
```

will compute the trigonometric sine of the value inside the brackets. The documentation for the *expr* command contains the details of the available built in functions, the available operators and the precedence of the operators.

3.7 Procedures

A Tcl procedure is a block of Tcl statements that is identified by the *proc* keyword. The form of a procedure is:

```
proc name { parameter list } { body }
```

where *name* is the unique name of the procedure, *parameter list* is a possibly empty list of tokens that represent parameter values, and *body* is a set of Tcl statements. The name can be any set of characters that does not contain white space, the dereferencing operator, or brackets that are meaningful to Tcl. The name is only unique in the sense that the Tcl namespace can have only one active procedure with a given name. Should the procedure associated with a name be redeclared, the effect is that the new procedure body replaces the original procedure body. The original procedure body is made inaccessible by such an operation.

Here is a simple procedure that will print a string:

```
proc Print { { what Nothing } } {
    puts "$what"
    }
```

This procedure might be invoked using the command:

```
Print "Hello, world!"
```

which would result in the string "Hello, world!" being written to the standard output stream of the interpreter. The syntax shown means that the procedure Print takes one parameter which has a default value "Nothing". Should the command:

```
Print
```

be encountered, the procedure will write the text "Nothing" to the standard output stream of the interpreter. Another way to write the Print procedure would be the following:

```
proc Print { what } {
    puts "$what"
    }
```

in which case an incidence of the *Print* command without any parameters would result in an error because the value for the parameter is missing. A third syntax for procedures is the following:

```
proc Print { args } {
    puts  $args
    }
```

Here the special keyword *args* indicates that a variable number of parameters may be present. In this case, each argument is printed on the standard output of the interpreter.

Procedures return when the statements in the body are exhausted, or when a *return* statement is encountered. When no *return* statement is present, the value returned by a procedure is the value returned by the last statement executed in the procedure *body*. To return something specific, the procedure can be written as:

```
proc Print { args } {
    puts  $args
    return 1
```

```
        }
```

Here, the string 1 is returned, regardless of the contents of the arguments or the results of the *puts* commands.

Typically, a Tcl application will consist of a number of procedures that are called to implement the functions of the application. Because of the design of the Tcl interpreter, there is no loss in performance associated with breaking a large script down into a number of procedures. This is because the Tcl interpreter parses a procedure only once, then saves the parsed procedure as a byte code that is executed each time a procedure is invoked.

The default scope for a procedure name is *global*, so the name is known everywhere within the interpreter in which it is defined. Local scope can be obtained by defining a procedure within another procedure. The *namespace* mechanism can also be used to qualify procedure names for the purpose of organizing the Tcl name space. For example, an instance of the *Print* procedure defined using the statements:

```
namespace eval Printer { proc Print { args } { puts  $args } }
```

will result in a globally available procedure name Printer::Print that will execute the *Print* procedure. Here the qualifier *Printer* is the *namespace* name. Inside of this *namespace*, the function is known simply as *Print*. For example:

```
namespace eval Printer { Print "Hello world!" }
```

will invoke the previously defined procedure.

3.8 Control of Statement Execution

Control of statement execution in Tcl is accomplished using the implied function call method provided by the square bracket syntax, through the use of the *if* function, through the use of the *switch* construct, and through the use of the *break*, *continue* and *return* statements. These constructs are very similar to those found in other programming languages such as the C programming language.

The *if* command has the format:

```
if { condition1 } { body1 } elseif { condition2 }  { body2 } elseif { condition 3 } { body3 }... else { bodyn }
```

where *condition* is an expression that evaluates to zero or not zero. The *body* items consist of one or more Tcl statements. The last clause is identified by the *else* keyword. Execution of a body depends on the result of the evaluation of the expression. Typically, expressions are logical operations such as comparison, or arithmetic operations that result in some value that will suffice to determine the test result.

The switch construct has the format:

```
      switch { key } {
item1           { body1 }
item2           { body2 }
        ...
 default        { body n }
                }
```

where the *key* is compared to the *items*. A match to an *item* will result in the execution of the associated *body*, which is a series of Tcl statements. If no match occurs and there is a default item, its body is evaluated. The switch construct can take some options that allow searching to proceed according to several methods, such as *exact*, *glob* and *regexp*. These alternative specifications can be useful when looking for matches against sub strings or classes of string representations. Read the documentation on this command to gain a full appreciation of the power of some of these options. The default search scheme is *exact* matching.

Looping over a set of Tcl statements can be accomplished using constructs such as *for*, *foreach*, and *while*. The *for* command has the following format:

```
for { init } { condition } { increment } { body }
```

where *init* is an initialization statement, *condition* is the limiting condition for the loop, and *increment* is the method of changing the loop variable. Each of these tokens can be complex Tcl scripts that have as their results values appropriate to their function. Here is a simple example of a *for* command:

```
for { set i 0 } { $i < 100 } { incr i } { puts "Hello, world for the ${i}th time!" }
```

This statement will produce 100 lines of output on the standard output stream of the interpreter. In a similar manner, the while loop has an implementation like the following:

```
set i 0; while { $i < 100 } { puts "Hello, world for the [incr i]th time!" }
```

This series of commands will also print out 100 lines of output. Finally, if a list of items is available in a variable named *list,* then a construct like:

```
foreach item $list { puts "This is item $item" }
```

would iterate over the elements of *list* and print them out.

Where a loop construct contains an *if* construct, the iteration process can be controlled based on some test condition. Consider the following code fragment:

```
foreach item $list {

    if { $item == c } { break }
    elseif { $item != d } { continue }
    else { puts $item }

    }
```

Here the iteration is interrupted when the value of temporary loop variable i*tem* is c. Similarly, the *continue* keyword could be used to implement some type of flow control based on the results of some test condition. While *break* terminates the loop iteration, the *continue* keyword will continue the loop iteration with the next value of the loop variable.

The final and most common method of execution flow control is the procedure call. Where a procedure has been defined, then it can be invoked using its name. Here is a construct that uses the Print procedure:

```
if { 1 != 0 } { Print "1 != 0" } else { Print "1 == 0" }
```

3.9 Error Handling

The evaluation of a Tcl statement can result in an error condition. Error conditions occur because the statement is malformed, or because there are references to undefined procedures or variables, or because the operation requested of a command can not be successfully completed. When an error condition occurs, the Tcl interpreter will, baring other instructions, terminate execution of the current script and return an error message somewhat descriptive of the error that occurred.

Tcl provides a mechanism for handling error conditions that occur in applications. The *catch* command can be used to wrap any block of Tcl code and trap any error conditions that are encountered. The format of the *catch* command is:

```
catch { script } result
```

where *script* is the body of Tcl statements to monitor for execution errors and *result* is the name of the Tcl variable that is to receive the result of the *script,* or the error message that is indicative of the error encountered. For example, the construct:

```
if [catch { Print "Hello, world!" } result] { puts "An error happened because : $result" }
```

will print out the reason for an error, or if no error occurs, the string "Hello, world!". The result of the *catch* command is either 0 for the case where no error is detected, or 1 for the case where an error occurs. Clearly, by using catch within an *if* command construct, elaborate error handling can be implemented for Tcl applications.

3.10 Input and Output

Tcl implements the concept of channels for the purpose of input/output operations. A channel can be any type of input/output device for which the idea of sending and receiving character data is meaningful. By default, all Tcl interpreters create 3 channels, the standard input channel, the standard output channel and the standard error channel. These 3 channels are character stream channels that are typically connected to the equivalent channels of the platform console. The channel descriptors for the standard channels are the keywords *stdin, stdout*, and *stderr*.

Other types of channels can be used by Tcl applications. Tcl has a very easy to use channel implementation for TCP/IP based socket communications, and there are language extensions that implement channels for various types of computer hardware, such as game ports, serial communications devices and digital input/output interfaces.

Input and output operations are carried out using the *gets*, *read* and *puts* commands. Connection support is provided using the *open, close* and *seek* commands. Both byte stream and block operations are supported. Tcl also provides an event driven interface for use with input/output channels that makes it easy to monitor channels for activity without the need to use a polling construct. Extensive configuration options are available for channels to support various types of buffering and character translations. Usually, the default configurations for standard channels will meet most application needs.

Here is a typical construct that will read data from a text file:

```
set fd [open datafile.txt r]
 if { $fd == "" } { puts stderr "Failed to open file datafile.txt! File not found or permissions not valid" ; exit }
while { [gets $fd line] != −1 } { puts "$line" }
close $fd
```

Here the *open* command gets a channel descriptor for the file *datafile.txt.* If the *open* operation were to fail, then the descriptor will be an empty string. The *while* loop reads a single line from the channel and prints it out until it comes to the end of the file. The file is then closed, freeing up the channel and invalidating the channel descriptor.

Note that the general form of the *puts* command is:

```
puts options stream text
```

where *options* are command options for controlling the output to the channel, *stream* is the channel descriptor, and *text* is the character stream to write. In the previous examples, no *options* are specified, so the command writes the text followed by a newline character. Since no *stream* is specified, the command assumes the standard output channel.

3.11 Events

Some Tcl applications, and all GUI based Tcl applications use an event loop to manage the interaction between external events and the application. External events are such things as mouse clicks, keyboard activity, data available on channels and timer interrupts. The basic *tclsh* interpreter does not automatically enter an event loop, so if an application is designed to use the event constructs, the script must enter an event loop by specifically calling a Tcl command such as *vwait* or *after* to initiate event pooling.

Interpreters such as *wish* and *fltkwish* always enter an event loop when started. Using commands such as *fileevent* and *Bind* execution of scripts can be structured to respond to external events, greatly simplifying application development. Here is an example of using events to monitor traffic on a TCP/IP socket:

```
# Establish a server socket listening for connections on port 3079

set s [socket −server ConnectProc 3079]
```

```
# Handle a connection from a client

proc ConnectProc { client port address } {

    puts "Connection on port $port form $address"

     fconfigure $client −buffering line
     filevent readable $client "HandleData $client"
    }

# Get a line of data from the client socket

proc HandleData { client } {

    if { [gets $client line] != −1 } {

       puts "$client : $line"
       }
```

The structure of the above code fragment is entirely event driven. When a remote client connects to the socket on TCP port 3079, the *ConnectProc* procedure is executed. This procedure configures the socket to buffer full lines of input before signaling that data is available. The *fileevent* statement will cause the *HandleData* procedure to be executed when a line of data is available. Because the socket identified by the client parameter is a Tcl channel, the general stucture of I/O event handling is the same as with a data file that was being read for, for example, user input, or as part of a pipe.

When using the TclFltk extension, an event loop must be entered in order for the display to be updated. if you use the *tclsh* shell to run Fltk scripts, then the last line of your script should use the Tcl *vwait* command to start the event loop. If you use a version of the Tcl interpreter that already initiates the event loop, such as *fltkwish*, then you need do nothing to start the event loop.

Here is an Fltk example of the use of events to control program execution:

```
# An example of the use of Events in Fltk

package require Fltk 0.4

Image t.i −file $Fltk(Library)/images/ashley.gif

Bind t.i <motion> { puts { %x %y } }

Show t

Wm title t "Event example"
```

This script will create a window with an *Image* widget inside of it. When the mouse is moved over the *Image* widget, the window relative coordinates of the mouse will be written to the standard output stream of the Tcl interpreter. The *Bind* command causes the script fragment that prints the mouse location to be executed whenever the mouse moves over the *Image* widget. Since there are event names defined for all of the principle user interaction events, it is possible to create an application that will respond to mouse clicks, keyboard activity, communications line activity, socket input and a few other things. Event driven programs have wide application in GUI environments, amongst other places.

3.12 Library Code and Extensions

Commonly used Tcl scripts can be collected into script libraries and bundled into packages. Extensions, which add additional commands the the Tcl command set, can be written using compiled languages such as C or C++ and bundled into packages as well. Tcl scripts can find library procedures and extensions using the *package* mechanism.

A Tcl *package* is known by its name and its version number. It is located by a package index file. A package index file is a Tcl script that conditionally loads other Tcl scripts or compiled extensions into an interpreter. Typically, extensions and library code are stored in a location that the Tcl interpreter will know about, such as its library path. When the interpreter is started it will search its library path for package index files. These files will specify the conditions under which a named package should be loaded, and will have instructions for loading the package.

Within an application, extensions and library packages are invoked using the *package* statement in the following format:

```
package require options name version
```

where *options* are optional flags controlling the identification of the package, *name* is the name of the package, and *version* is a version number string. By default, the package loader will load the most recent package version available. If a *version* string is specified, the package loader will load the specified version or a later version, but will fail if the requested version is more recent than the latest available version.

Here is an example of the method of loading the TclFltk package:

```
package require −exact Fltk 0.4
```

This statement will result in an error for all versions of the package other than the 0.4 version.

3.13 Introspection

Introspection is the ability to interrogate the application environment about itself. Tcl is a language that implements many mechanisms that allow applications to interrogate aspects of the application environment and even the application itself while it is running. A basic tool for introspection is the *info* command. Tcl's *info* command can be used to get information about the existence of variables, the source of procedures, the arguments used by procedures, the list of commands currently available, and many other potentially useful aspects of the running application and its environment.

One very common application of introspection is to determine whether a variable is currently accessible to a procedure or script. A variable is accessible if it exists within the currently accessible scope of variable name spaces. Applications might wish to perform such a test, for instance, when one part of an application initializes the variable for use by another part of the application. To prevent script errors from aborting the current script, the *info* command can be used to test if the variable was, in fact, created and initialized. For example, the command:

```
info exists MyVariable
```

will return the value 1 if there is currently accessible a Tcl variable named *MyVariable*, or the value 0 if no such variable currently exists.

The command:

```
set p [info body MyProc]
```

will return the current source for the body of the procedure named *MyProc*. Applications can then inspect and possibly modify the body of this procedure. Some applications, for instance, implement schemes for generating procedures based on a template using this type of technique.

Most Tcl commands that are used by GUI environments, such as those that implement widget constructors, provide a mechanism for interrogating the current configuration parameters of the widget. All Fltk widgets, and all Tk widgets, implement the *cget* sub−function which can be used to retrieve the current value of any of the widget configurable parameters. For example, the command:

```
$w cget −width
```

will return the current width in pixels of the widget whose command token is contained in the variable *w*. Typically, the use of the *cget* sub−function without any parameters will return the list of all of the option names that can be queried for the widget.

GUI implementations, such as Fltk and Tk, also implement commands for the interrogation of various aspects of the widget tree and the associated geometry manager and window manager. An example of such a command is the *Winfo*, which will return details of the geometry of currently displayed widgets.

3.14 Summary

To develop a Tcl application, use a text editor to write the series of Tcl statements that implement the desired application functionality. Start by loading the required library packages and extensions, compose the necessary procedures, initialize the required variables, create the desired user interface, then either enter the event loop or call the main entry point.

Here is a simple Tcl application that uses the TclFltk extension to implement a command line calculator. This script is designed to run as a command under a Unix operating system. It will start the *wish* shell, which is a form of the Tcl intepreter, and then load the Fltk extension, and produce a widget window that has an input area for use by the application. The same script can be used directly under Microsoft Windows operating systems by starting the *wish* shell and reading in the script file using Tcl's source command. Note that the *wish* shell will automatically start the event loop, so the GUI elements will be displayed correctly.

```
#!/bin/sh
# \
exec wish "$0" ${1+"$@"}

# A simple calculator application in Fltk

package require Fltk 0.4

set Data ""

Input t.c −command { catch { eval expr $Data } Data } −variable Data −w 200

Show t

Wm title t "Calculator"
```

You can enter arithmetic expressions into the *Input* widget and when you press the enter key you will see the calculated result, or an error message, appear in the widget window. This simple application can be terminated using the standard system menu items on the application window.

In the above script, the first 2 lines are, to a Tcl interpreter, comments. Under a UNIX operating system, the first line will cause the default command shell to start in batch mode and begin executing commands at the *exec* statement. This statement tells the operating system to start the *wish* shell, pass any command line parameters to the shell, and send the rest of the input file to the shell as its standard input. It does not matter if you just use the source command to read this file as is into a Tcl interpreter, because the effect of the second line is to make the *exec* command a comment. Under the Microsoft Windows operating system, the first three lines are always treated as comments because the only way to execute a script under Microsoft Windows is to pass it to an interpreter through its standard input.

4 How to Write Applications Using the Fltk Extension

The Fltk extension adds a new set of commands to a Tcl interpreter that can be used to construct GUI elements called widgets. Widgets are useful components of a Graphical User Interface (GUI) that provide an interface between the user of an application and the application itself. Typically, the user will interact with a widget through mouse actions or keyboard actions which the widget then translates into some desired functionality.

In an application built with the Fltk extension, the functions of the application will typically be implemented as Tcl procedures. User actions that apply to a widget invoke the appropriate procedures. These procedures may or may not change the appearance of the widget, affect the appearance of the user interface, or invoke other applications that themselves may be built using the Fltk extension.

The basic steps in building an application with the Fltk extension are:

- Design the target user interface using Fltk widgets
- Write the Tcl procedures that implement the required functionality
- Use the Fltk widget construction commands the build the user interface
- Bind the procedures to the widgets
- Activate the application by causing the widgets to be displayed

4.1 Designing User Interfaces

The design of a user interface is a subject that has received a lot of attention over the history of computing. The commonly seen varieties on modern day computers are those that implement a window paradigm, such as that of the Microsoft Windows family of operating systems, or the X Windows based user interfaces used by the UNIX operating systems. The idea is that an application presents itself as a frame window that contains a number of specialized sub−windows, each of which implements some function of the application. Over the years all of these user interface efforts have drifted to standard types of layouts which have some or all of the following elements:

- A frame window having a title bar and some icons that implement system functions, such as maximizing the application window or terminating the application
- A menu bar with various types of drop down menu selection features
- One or more button bars that implement through single button presses elements of the application menus
- An application area which displays various aspects of the application functionality
- A status bar that display status information and provides flyover help information as the mouse moves over an application menu or widget

In the Fltk context, all of the elements of this type of GUI are widgets, and the entire GUI is assembled by constructing the widgets and placing them inside a frame window, either by specifically specifying their location and size, or by using special container widgets that arrange their child widgets according to preset rules.

It is always possible to specify the layout of a GUI using the *standard widget options* that fix the top left hand corner and the horizontal and vertical dimensions of the widget. This method is, however, somewhat tedious, particularly when there are more than a few widgets involved and the GUI is changing for some reason or other during the running of the application.

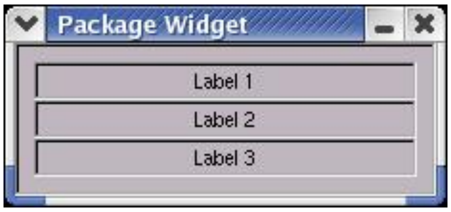
The Fltk extension provides support for the layout of GUIs by implementing the idea of container widgets that themselves provide geometry management functionality that operates on child widgets that are constructed inside of the containers. There are 7 basic types of containers, the *Toplevel* widget, the *Group* widget, the *Package* widget, the *Scroll* widget, the *Tabs* widget, the *Wizard* widget and the *Tile* widget.

The *Toplevel* widget creates application frame windows. A *Toplevel* widget creates the root widget in which a collection of child widgets can be constructed. When a *Toplevel*widget is minimized, all of the children of the *Toplevel* are minimized. All widgets that are not either *Toplevel* widgets or popup menus must be children of a *Toplevel* widget. Aside from system level geometry management, the *Toplevel* widget does not explicitly manage the internal layout of its children, except in the case where it is constructed implicitly. Implicit construction of a *Toplevel* widget will cause it to resize itself such that it wraps all of its children inside a 2 pixel border. Implicit *Toplevel* construction is a convenience useful for short GUI applications.

The *Group* widget is a simple container that can be used to group a collection of child widgets. The *Group* widget provides no special geometry management functionality, other than allowing the displacement of its children while preserving their relative positions.

The *Package* widget operates on its children by resizing them all to the same dimension along one of its axes, and packing the widgets together along the other of its axes. By constructing a hierarchy of *Package* widgets, it is possible to layout widgets in any desired manner. Once constructed inside of a *Package*, child widgets take their resize behaviour from the *Package*, not their internal geometry specification. Here is an example of a simple *Package* that will align some *Label* widgets vertically:

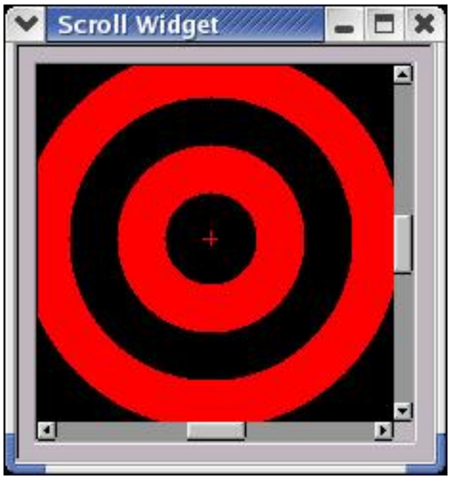
```
# Construct an empty Package

Package t.p −width 200 −orientation vertical

# Add some child widgets

Label t.p.l1 −text "Label 1"
Label t.p.l2 −text "Label 2"
Label t.p.l3 −text "Label 3"

# Display the GUI. Note the implied creation of the Toplevel widget t

Show t
```



This script shows the use of the widget path name convention used by the Fltk extension. A widget path name is a string that has elements separated by a period. The first element of the path name is the root widget name, and the parents of any particular widget are evident from the list of elements. Root names can not begin with a period. The script also shows an example of implicit construction of the root widget. No *Toplevel* construction command is present, so an application frame window is automatically constructed that will nicely wrap the *Label* widgets.

The *Scroll* widget is a container that allows the construction of child widgets whose client areas are larger than that of the *Scroll*. The*Scroll* will automatically manage scroll bars to provide visibility over all children of the scroll and their client areas. This is a very convenient container as evidenced by the following

```
# Create a Scroll

Scroll t.s −w 200 −h 200

# Put a drawing in it that is large

Drawing t.s.d −w 1000 −h 1000 −variable d

set d "cs fl 1 bg black cr 400 bg red cr 300 bg black cr 200 bg red cr 100 bg black cr 75 bg red cr 50 bg black cr 25"

# Show it

Show t
```



Here the *Drawing* is large (1000 x 1000) compared to the client area of the *Scroll*(200 x 200). This will result in the appearance of scroll bars that will allow the user to scroll the *Drawing* so that all parts of it are visible.

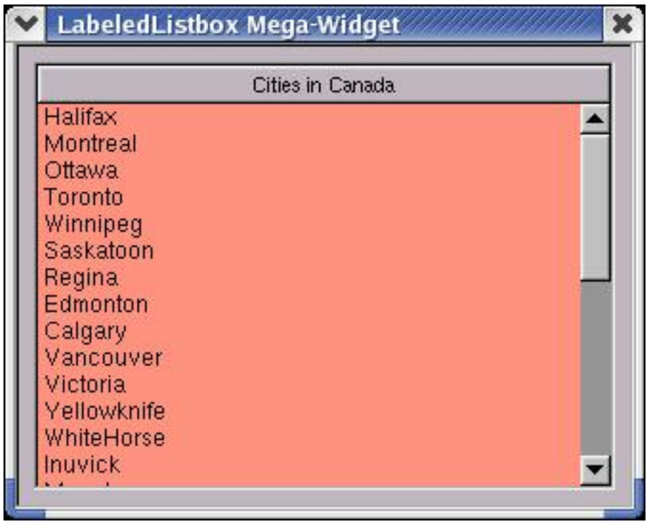
The *Tile* widget is a container into which widgets can be packed using their own geometry specifications, such as the location of the top left hand corner and their width and height. Once in a *Tile*, the internal borders that separate the widgets can be dragged with the mouse to resize the child widgets. This type of feature is used for things like paned windows where the panes can be resized.

The *Tabs* widget is a fifth type of container. It presents a series of tabs using a file folder paradigm the can be selected using the mouse. Each tab is a container which can have child widgets that implement different aspects of application functionality. The *Tabs* widget does not, in itself, provide for any geometry management of its child widgets, although a *Tabs* container can have as children any of the other container widgets.

The *Wizard* widget is a container that is used to build an interface that can carry the user through a structured set of steps. The *Wizard* widget is similar to the *Tabs* widget, except that the contained widgets are exposed under control of the script rather than through direct user interaction. Any of the other container widgets can be one of the children of a *Wizard* widget.

4.2 Creating Custom Mega−Widgets

The container widgets provide the foundation for the construction of custom mega−widgets using the standard set of widgets provided by the Fltk extension. A mega−widget is a widget that is built using a collection of basic widgets to provide enhanced functionality. An example of a mega−widget is a the labeled listbox widget. This widget is built up using the *Label* widget and the *Listbox* widget to form a mega−widget that provides a label at the top of the *Listbox*.



Here is the code needed to implement the *LabeledListbox* widget:

```
    proc LabeledListbox { w  args } {

            global Data

            set f [Package $w -orientation vertical -relief sunkenframe]

            eval { Label $f.label -text $w -relief raised -qn true } $args

            eval { Listbox $f.list -relief flat -bg tan } $args

            return $f.list

            }
```

The *LabeledListbox* procedure will construct a mega−widget with a path name set to the contents of the *w* parameter and will apply the configuration options supplied via the *args* parameter. The maga−widget is constructed using a *Package* widget set to pack its child widgets vertically. The *Package* will auomatically resize the children in the horizontal dimension so that they will all have the same width. The chosen width is the width of the widest child.

There are 2 child widgets, the *Label* and the *Listbox*. Typically the only options that are of interest for the *Label* part of the mega−widget are the displayed text and its color rendition. The vertical size is left to the widget default, and the width is determined automatically based on the width of the *Listbox* widget. The qn option supplied for the *Label* widget will cause the *Label* not to respond to any options in the args parameter that are not specifically qualified to refer to the *Label*.

The *Listbox* child widget will accept both qualified and unqualified option names. The widget constructor for the *LabeledListbox* could then look something like the following:

LabeledListbox t.list −w 300 −h 200 −label.text "Labeled Listbox Widget" −variable Data(Selection) −command "Select %W.list"

This constructor will create a mega−widget with a *Listbox* child that has the dimensions 300 x 200 pixels, bound to the Tcl variable *Data(Selection)* and with a command *Select* that is to be executed when the user makes a selection. The title displayed in the label component will be *Labeled Listbox Widget*. Note that all of the unqualified option names, such as *w*, *h*, *variable* and *command*, are ignored by the *Label* child. The qualified option name *label.text* is the only option that the *Label* will process, because the qualifier *label* is a component of its path name.

The *LabeledListbox* procedure returns the path name of the *Listbox* as its result. This makes it convenient to make use of*Listbox* commands, such as the add command, the load the *Listbox* with items to be selected. A complete code fragment might look like the following:

set list [LabeledListbox $f.list −w 300 −h 200 −label.text "City Names" −variable Data(City)]

eval { $list add } $CityList

Here, the variable *CityList* is presumed to hold a list of cities for the user to choose from.

4.3 Binding Tcl Procedures to Widgets

The Fltk extension provides the programmer with a lot of help when it comes to binding user actions to Tcl procedures. Every widget implements options to specify a widget command that is executed according to the occurrence of user actions. The specific actions vary according to the widget. All widgets also provide for the automatic binding of the widget to a Tcl variable that will be maintained synchronous as to contents between the Tcl variable and the widget. All widgets also support a binding to a second Tcl variable that controls the state of the widget, and controls the invocation of a command that occurs whenever the state of the widget changes. A final mechanism is the use of the *Bind* command which allows the binding of Tcl procedure to both system defined and user defined events that occur while a particular widget has input focus.

The simple example of a binding between a widget and a Tcl procedure is the use of the *Button* widget to activate some function:

```
# This is the procedure to be invoked

proc ButtonProc { w } {

    global ButtonState

    puts "Hello, world from button $w"

    incr ButtonState −1
    }

# Here is the button constructor

Button t.b −text "Press Me!" −command { ButtonProc %W } −statevariable ButtonState

# Here is the button state variable

set ButtonState 10

Show t
```

When this script is executed, the *Button* will initially be enabled, because the value of its state variable is not zero. Each time the *Button* is pressed, it will print a message on the standard output stream of the Tcl interpreter, and decrement the value in *ButtonState*. When this value reaches zero, the *Button* will become inactive, and stop responding to button press actions.

The Fltk extension extends the ideas of variable bindings, state bindings and command invocation for both state and variable changes to all widgets. The type of widget determines the meaning and utility of these bindings. For instance, a *Counter* will typically be bound to a variable that is being controlled by the widget, while a *Drawing* might be bound to a variable that contains the current script needed to create the image in the *Drawing*. Using these mechanisms greatly simplifies the construction of an application using Fltk as compared to other available Tcl GUI bindings.

4.4 Using Options and Application Data

The Fltk extension includes a facility for setting the values of widget options in a data base that can be used to configure the widgets of an application. This facility is useful when a number of widgets in a GUI need to have common behaviour, such as having the same background color, dimensions, or have their state controlled by a single state variable. The *Option* command is used to manipulate the contents of the option database.

Widgets constructed using the Fltk extension have a class property and a name property. The name property has a value that is the path name of the widget itself. The class property has a value that is, as a minimum, the class name of the widget. Class names are typically the same as the name of the widget construction command, so, for example, the class name of a *Button* widget is *Button*, and the class name of a *Label* widget is *Label* . Class names are a bit more flexible than path names in that a widget may have any number of class names. Applications can set additional class names for a widget using the class option of the widget.

Several widgets will be automatically given more than 1 class name. All widgets that behave as buttons have are members of the *Button* class. For example, the *ImageButton* widget is a member of the *ImageButton* class and the *Button* class. Additional class memberships can be established through the use of the class option of the widget constructor command.

An application can specify the value of a widget configurable option by adding an entry in the option data base that specifies it value. This is done with a command of the form:

```
Option add name.option value
```

where name is either a widget path name or a widget class name, option is the name of the option, and value is the value to be used. In the following example,

```
Option add Button.foreground red
```

an entry is added to the option data base that applies to all widgets which have membership in the *Button* class. This entry will cause the *foreground* property of these widgets to be set to the color red. Using the option database is a very powerful way to configure the look and feel of an application, and provides a convenient method of providing parameters for some types of widget layout schemes.

Application data is user data that can be manipulated using the *Application* command. This data is typically used to implement version control for applications, and to pass parameters to applications that is used to configure a generic application script to some specific purpose. The *Application* command provides an interface that manages a few generic application parameters, such as the application name, the default language for messages, the application version, and some general purpose data. Using the application data is one way of preparing an application for the use of international languages.

4.5 The Fltk Global Array

The global array named Fltk has a number of elements that are initialized by the extension package to provide information to applications about the version of the Fast Light Tool Kit used to generate the extension, the location of the package library, and the version of the extension itself. The array has the following elements that applications may query:

ToolkitName	For this extension it is "Fast Light Tool Kit"
ToolkitVersion	The release version numbers for the tool kit used to compile the extension
Version	The version of the extension
PatchLevel	The patch level of the extension
Copyright	Copyright notice for the extension
Library	Location of the extension library

The Tcl set command will return the values of these variable. For example, the command:

```
set Fltk(Library)
```

will return the path to the extension library directory. Note that while it is possible to modify the vaues of the variables within a script, once modified the information they contain is no longer reliable.

4.6 Running the Application using the fltkwish Interpreter

The *fltkwish* interpreter is a version of the Tcl interpreter that automatically loads the Fltk extension. Tcl scripts that make use of Fltk extension commands can be run by starting the *fltkwish* interpreter and passing the scripts to its standard input stream. This can be done by specifying the script file name on the *fltkwish* command line, or by issuing the Tcl source command at the interpreter command prompt.

The *fltkwish* command line has the following general form:

```
fltkwish options file
```

where the *options* are command line switches that control the behaviour of the interpreter and *file* is the name of the file to interpret. If no *file* is specified, the interpreter will start up as an interactive console application and present the user with the usual Tcl command prompt. If a *file* is specified, the interpreter will not present the interactive command prompt, but will interpret the commands in the *file*.

Under UNIX operating systems, the options can be any of the standard X toolkit options supported by the platform. Under the Windows operating systems, the options are limited to a small set of keywords that directly relate to the behaviour of the FLTK toolkit and the Fltk extension.

Here is the list of Fltk related options:

- −fg Set the default foreground color
- −bg Set the default background color
- −bg2 Set the default alternate background color
- −namespace Set the name of the Tcl name space to use

The *−fg*, *−bg* and *−bg2* colors have defaults that are typically established by the window manager in use. The *−namespace* option can be used to tell the interpreter to create its command set in a specific Tcl name space, a facility that is useful where there exists the possibility of name conflicts in the Tcl global name space.

Here is the typical method of running an Fltk application:

```
fltkwish myapp.tcl
```

This command will start the *fltkwish* interpreter and begin interpretation of the script file *myapp.tcl*.

5 Commands

The following is the list of commands added to a Tcl interpreter by the Fltk based toolkit extension. Note that the capitalization is important in the use of the command names.

- Alert Display an alert message
- Ask Ask a question to the user
- Adjuster Adjust values
- Application Set or get application variables
- Bind Associate a script with an event and a widget
- BindTags Specify event processing order
- Button Create a button widget
- Canvas Create a canvas widget
- Chart Create a chart widget
- CheckEvents Process pending events
- CheckButton Create a check button widget
- Choose Ask the user to choose an option
- ChooseColor Display a color selection dialog
- ColorName Find the name of a color description
- Combobox Create a combo box widget
- Counter Create a counter widget
- Destroy Destroy one or more widgets
- Dial Create a dial widget
- DiamondButton Create a diamond button
- Drawing Create a turtle graphics drawing widget
- Dummy A command that does nothing
- Exit Terminate the FLTK application
- Frame Create a frame widget
- Focus Set or query the input focus
- GetInput Get some input from the user
- GetFileName Get a file name from the user
- GetPassword Get a password from the user
- Group Create a group container widget
- Help Display help information
- HelpDialog A dialog box with help information
- Hide Hide windows
- HtmlViewer Display HTML format text
- HtmlWidget An HTML viewer with navigation controls
- Image Create an image widget
- ImageButton Create an image button widget
- Input Create an input widget
- Iterator Construct a list iterator button
- Knob Create a knob widget
- Label Create a label widget
- LabeledInput Create a labeled input widget
- LabeledText Create a text box with a configurable label
- LightButton Create an illuminated button
- Listbox Create a list box widget
- Menu Create a menu widget
- Message Display a message box
- Option Get or set option database values
- Output Create an output widget
- Package Arrange widgets in a frame
- ProgressBar Create a progress bar widget
- RadialPlot Create a radial plot widget
- Region Create an event region widget
- RepeatButton Create a repeating button
- ReturnButton Create a button that handles the enter key

- Roller Create a roller widget
- RollerInput Create a roller with an input widget
- RoundButton Create a button that has a round shape
- Scheme Specify a widget rendering scheme
- Scroll Create a scrolling container widget
- ScrollBar Create a scroll bar widget
- Signal Generate an event
- Slider Create a slider widget
- Show Show windows
- Table Create a table widget
- Tabs Create a set of notebook tabs
- TestWidget Create a test widget
- Tile Create a tile container widget
- Text Create a text widget
- Toplevel Create a top level widget
- Update Redraw specified widgets
- UserButton Creates a custom button
- ValueSlider Create a slider with a value display
- Vu Create a digital volume control
- Windows Interrogate the widget list
- Winfo Interrogate widget characteristics
- Wm Interact with the window manager
- XYPlot A plotting widget for displaying data

6 Widgets − Standard configurable widget options

All widgets supported by the Fltk extension accept a common list of configurable options as well as a set of widget specific options. Widget commands support 2 functions, the *configure* function and the *cget* function. The *configure* function is used to set the values of the configurable options for a widget, while the *cget* function is used to interrogate the current value of the configurable options of a widget.

The list of standard configurable options is:

- alignment Alignment of the widget label
- anchor How to anchor text
- background Color of the widget background
- borderwidth Width of the widget border
- class Class name of the widget
- command Widget command script
- cursor Cursor to use in a window
- data User data for the widget
- eventdefault If default event handling is used
- font Label font
- fontsize Size of font characters
- fontstyle Style of the label
- foreground Text foreground color
- highlightbackground Background color when highlighted
- highlightcolor Color to use when highlighted
- highlightthickness Border thickness when highlighted
- height Height of the widget
- invertstate Invert the state of the widget state variable
- label Label string for the widget
- limits Range of values for window size
- nocomplain If errors in options are ignored
- padx Internal horizontal padding
- pady Internal vertical padding
- qualifiednames If option names must be qualified
- relief Widget relief
- resizeable If a window can be resized
- state Set the state of the widget
- statevariable The variable to monitor for the state
- statevariablecommand Command to execute on a state change
- tooltip The tooltip text for a widget
- underline State of the underline text
- variable Name of the associated text variable
- variablecommand Command to execute when a variable changes
- wraplength If text should be wrapped
- wallpaper Name of the wallpaper image
- width Width of the widget
- x Horizontal location of the widget
- y Vertical location of the widget

While the widget command for all widgets will process all of the standard options, not all options are meaningful to all widgets. Specific widgets may also support additional configurable options.

The Fltk extension uses a system of keyword aliases that provides for the use of alternate names, abbreviations and translations into multiple languages of the option names listed above. Depending on the option, there may be one or more names that will be recognized for an option. Common examples are the use of *w* for *width*, *h* for *height*, and *justify* for *alignment*. The option names listed above are those that are guaranteed to be valid when the English language message table is used.

6.1 Getting and Setting Widget Option Values

Widget option values can be set when the widget is constructed, or by using the widget command. The widget command is the command whose name is returned when a widget is constructed and will typically be the path name of the widget. *Menu* items can also return a widget command which will be the path name of the *Menu* qualified with the index number of the item in the widget.

Here is an example of a widget construction:

```
set w [Label t.l −label "This is a label" −foreground red −relief raised]
```

This command will construct a *Label* widget whose widget command is *t.l,* which, in this case, is stored in the Tcl variable *w*. The form of a widget command may be either:

```
path config ?−opt? ?value? ...
```

or

```
path cget ?−opt?
```

Where *opt* is the name of one of the options and *value* is the value to set for the option. Both the *configure* and the *cget* functions will report available options if the widget command line includes none. In general, each type of widget will report a list that contains the standard options indicated here and the widget specific options documented for the widget.

6.2 Qualified Option Names

Option names can be either *qualified* or *not qualified*. A *qualified* option name has the form:

```
qualifier1,...qualifiern.name
```

where the *qualifiers* can be either widget *path names* or widget *class names*. If no *qualifier* is specified, then the option is applied to the widget. If the widget *path name* or one of the widget *class names* matches one of the qualifiers, the option is applied to the widget, otherwise, the option is ignored.

Here is an example of a widget command that uses *qualifiers* for the options:

```
$w configure −width 100 t.v.r,t.v.l.height 200 Package,Label.relief flat
```

In this example, the *width* option will be applied to any widget, the *height* option will be applied only to widgets whose path name is *t.v.r* or *t.v.l*, and the *relief* option will be applied to widgets that have the *Package* or *Label* class name.

Qualified option names are convenient when a list of options is to be applied to several widgets. This situation occurs when compound widgets are built up in scripts. By using *qualifiers* with the widget option names, there is no need to filter the option list. The *qualifiers* will act to filter the relevant options to the widgets that make up the compound widget.

Here is an example of a compound widget that makes use of qualified option names for its initialization. The compound widget consists of a *Label* widget and a *Counter* widget. The following procedure will construct the new widget and process any arguments that are supplied:

```
proc LabeledCounter { w args } {

    set f [Package $w −orientation horizontal]

    Label $f.label −relief flat −width 70 −align left,inside −qualifiednames true
    Counter $f.counter

    eval { $f.label set } $args
    eval { $f.counter set } $args
```

```
            return $f
            }
```

Here is the constructor for the *LabeledCounter* compound widget. The component widget class names are used to initialize options relevant to the components of the compound widget.

```
        LabeledCounter $f1.a −Label.label "Value of A" −variable a −label.anchor w
```

In this case, the *label* option will be applied to *Label* component, while the *variable* option will be applied to the *Counter* component The *anchor* option applies to the *Label* component, because its *qualifier* matches part of the path name of the *Label* at its lowest level in the widget tree.

When constructing compound widgets from the standard widget set, it is sometimes convenient to limit access to the widget options of the component widgets to only qualified option names. In this manner, the set of standard widget options can be selectively applied to component widget, while access to the options of specific widgets can still be garnered using qualified option names. The *qualifiednames* option can be used to restrict application of the options to only qualified option names. In the above example, the *width* option will apply only to the *Counter* widget, because it is not a qualified option name and the *Label* widget has its *qualifiednames* option set to *true*.

Default Widget Behaviour

All of the standard widget options are automatically initialized to a set of default values when a widget is constructed. The default values of the options define, amongst other things, the default behaviour of a widget when it receives event notifications, cause by, for instance, mouse or keyboard actions. Options such as the *foreground*, *background*, *highlightforeground* and *highlightbackground* ones define the appearance of the widget when it is active and has input focus. Options such as *width*, *height* and *x* and *y* define the geometry of the widget.

Standard Widget Option Reference

6.2.1 alignment

All widgets have the *label* property that is a text string that can be set to a string that can be displayed in a number of locations relative to the widget. The position of the *label* text is determined by the *alignment* property. The *alignment* property is composed of a number of specifiers that combine to define the position of the label text. The specifiers are:

centered	The text should be centered
top	The text should be at the top of the widget
bottom	The text should be at the bottom of the widget
left	The text should be at the left of the widget
right	The text should be at the right of the widget
inside	The text should be inside the widget

By default, the *alignment* value is *centered,* and the label text is drawn centered with respect to the widget rectangle. If the *inside* specifier is not present, then the label text is drawn outside the widget rectangle. Here is an example of a specification that would draw the label text vertically centered and right justified inside of a widget:

−alignment right,inside

Note that there are possible constructs that do not make any sense, such as:

−alignment top,bottom

These constructs will result in unpredictable label text positioning.

6.2.2 anchor

The *anchor* property is identical to the alignment property. It exists as an alias only.

6.2.3 background

The *background* property is used to specify the color of the background for a widget. The default *background* color is determined by the particular *Scheme* being used to draw the widgets, and possibly by any widget toolkit options specified on the command line used to start the interpreter being used to process the application script. For a standard invocation of *fltkwish* with no toolkit options and the default scheme, the *background* color will be *clear*, a specification that results in a background color determined by the current GUI desktop color scheme.

The background color for a widget can be set using the command:

```
$w configure −background color
```

where *$w* is the token that represents the widget command, and *color* is either the name of a color or a color specification. The Fltk extension has a database of color names that includes the usual set of primary colors (i.e. red, green, blue, orange), an extensive list of color names commonly found as part of X Windows color databases, and the names of color specifications used by desktop color schemes (i.e. app_workspace, color_buttontext).. Colors can also be specified using comma separated red, green and blue triplets as follows:

```
$w configure −background 193,24,86
```

There is also provision for the specification of gray scale colors using the form:

```
$w configure −background gray80
```

which will set the color to 80 percent gray.

The Fltk extension uses an internal color cube representation that limits the actual number of colors that can be displayed to 256 values. Actual color specifications are mapped to the closest color cube value for most widget purposes.

6.2.4 borderwidth

The *borderwidth* property can be used to specify the width of the internal border for container widgets. The Fltk tool kit specifies the window border width for the default *scheme*, so the *borderwidth* property has no effect on window borders. For the OpenGL *scheme*, the *borderwidth* is used to set the width of the OpenGL rendered borders.

Container widgets are those that have children, such as *Frames*, *Packages*, *Groups* and *Tiles*. The *borderwidth* property is used by these widgets to specify a border between the contained widgets and the window border.

6.2.5 class

The *class* of a widget is a list of comma separated strings that is useful for the specification of option values in the option database and for the specification of widget bindings for event handlers. All widgets are given a *class* specification when they are created that consists of at least the name of the widget command that created the widget. For example, all widgets created by the *Button* command will have the class *Button. LightButtons* will also have the class *LightButton.*

Typically, applications do not set the class string unless there is a particular need to do so. There is not much use in changing the class of widgets after they have been created as option database values are scanned during the widget creation operation, but a command of the form:

```
LightButton t.b1 −class Button,LightButton,MyLightButton ...
```

might have some use. Here, the default set of class specifiers is extended to add the specifier *MyLightButton*, which could then be used to bind events to all widgets with this class specifier. Using class specifiers helps when the option database is being used to provide configuration for a group os widgets that have characteristics that differ from the configured characteristics of a standard class. For example, suppose the option database is initialized using the following command:

```
Option add Button.foreground blue
```

which will cause all widgets that are members of the *Button* class to have their text colored blue. Subsequently, the following script fragment is executed:

```
Option add MyButton.foreground green

Button $f.mybutton −class MyButton ...
```

The result is that the widget *$f.mybutton* will collect options for the *MyButton* class instead of the options for the *Button* class. Using this technique, the contents of the option database can be set up to selectively configure the same type of widget in different places of an application script.

6.2.6 command

The *command* property of a widget can be used to specify a script that is executed when some event occurs that changes the state of a widget. The most common use of the *command* property is with *Buttons* and *Menus*. When a *Button* is pressed, the command script is executed to implement the action of the *Button* or *Menu* item. By default, widgets have no *command* script associated with them, and state changes have no effect on the application. Here is an example of a *Button* with a command script:

```
Button t.b1 −text Dismiss −command Exit
```

In this case, pressing the button will terminate the application. Command scripts are expanded to substitute any embedded keywords before execution. For example, the command:

```
Button t.b1 −text Dismiss −command { puts "Button %W was pressed!" }
```

will generate a message on the standard output stream of "Button t.b1 was pressed!" each time the button is pressed.

6.2.7 cursor

The *cursor* property is used to specify the name of the cursor that is to be used when the mouse pointer is over a widget. If no *cursor* is specified, the default cursor is the usual arrow cursor.

6.2.8 data

The *data* property is used to store widget specific data with the widget. By default, the *data* property contains an empty string. Applications can put whatever data that is desired into the widget data area using a command of the form:

```
$w configure −data { ...anything...}
```

and may later recover the data with a command of the form:

```
$w cget −data
```

where *$w* is the token that represents the widget command.

6.2.9 eventdefault

The *eventdefault* property determines whether a widget will benefit from default event bindings for the mouse and focus events. By default, for most widgets, the value of the *eventdefault* option is *false*, and the widget does not benefit from default behaviour. Some widgets, such as the *Button* class of widgets, have the eventdefault option value set to true. When the mouse enters the widget, the *highlightforeground* and *highlightbackground* colors are used to enhance the appearance of the widget. The effect is further enhanced when the widget has input focus.

When the *eventdefault* property has the value *true*, the entry of the mouse cursor into the area of the widget causes the text to take the color of the value of the *highlightforeground* options, and the widget background color to take the value of the *highlightbackground* option. If the widget has input focus, the text color is a lighter value of the *highlightforeground*, and the widget background is a lighter value of the *highlightbackground* color. When the mouse cursor leaves the widget, the text color reverts to the default *foreground* and *background* colors, unless the widget retains input focus.

6.2.10 font

The *font* property is used to specify the font that is used to draw the widget label text. The default *font* is normal weight helvetica 12 point text. Other fonts can be specified using a command separated list of font name and attributes. Here is an example:

```
$w configure −font times,bold,italic
```

where *$w* is the token that represents the widget command.

6.2.11 fontsize

The *fontsize* property is a numeric value that specifies the relative size of the font. The default value is 12 and the range of useful values is from 8 to 60.

6.2.12 fontstyle

The *fontstyle* property specifies the style of the current font. Font styles are specified as a command separated list of style names from the list *normal*, *shadow, engraved, none, symbol, bitmap, pixmap, image, multi, freeform,* and *embossed*. By default, the font style is *normal,* which represents a font style without any special effects. The *fontstyle* of *none* results in the relevant text not being displayed. This is sometimes useful when constructing widgets for which the *label* property is not useful.

6.2.13 foreground

The *foreground* property is used to specify the color used to draw label text for a widget. By default, the value of *foreground* is *black.* Here is an example of how to change the text in a *Button* to the color orange:

```
Button t.b1 −foreground orange
```

All of the color specification features described for the *background* option apply to the *foreground* color specification.

6.2.14 highlightbackground

The *highlightbackground* property specifies the color to be used to draw the background of the widget when it is highlighted. The default value is *clear,* a color that is defined by the current desktop scheme and tool kit options. The default behaviour of a widget is to set its background to the color specified for *highlightbackground* whenever the mouse moves into the area of the widget. If the widget has input focus, the color used will be a lighter version of the *highlightbackground* color. The *Bind* command can be used to alter the default behaviour of widgets.

6.2.15 highlightforeground

The *highlightforeground* property is used to specify the color used to draw the label text when the widget is highlighted. The default value is *red*. The default behaviour of a widget is to set the text color to the color specified by *highlightforeground* when the widget is notified that the mouse enters the area of the widget. When a widget has input focus, the color of the label text is set to a lighter version of this color. The *Bind* command can be used to alter the default behaviour of widgets.

6.2.16 highlightthickness

The *highlightthickness* property is used to specify the border width of the widget when it is highlighted. Because of the characteristics of the Fltk tool kit, this property has no effect on widgets. It exists for Tk compatibility.

6.2.17 height,width,x,y

The *height*, *width*, x and y properties are used to specify the geometry in pixels of a widget. All widgets have default values for these properties that are reasonable for the type of widget. Here is an example of a widget construction command that specifies the geometry:

```
Package t.p1 −x 100 −y 50 −o horizontal −height 20
```

In this case, a *Package* is being created that will limit the vertical dimension of the widgets it contains to 20 pixels. This type of command might be used for packing *Button* widgets into a horizontal button bar.

At any time the geometry of a widget can be changed by using the widget command to adjust any of these properties. The values supplied can use the *relative syntax* to adjust values relative to their current values. For example, a widget might be moved down the screen with the following command:

```
$w configure −y +20
```

This command will move the widget represented by the token *$w* down the screen by 20 pixels.

The geometry options can take special keywords instead of numerical values which may be useful in alignment of child widgets inside containers. For child widgets, the *width* and *height* options can optionally be specified as *width* and *height*. Using these keywords will cause the width and height of the child to be set to the current client area width and height of the containing parent widget.

The *x* option can be specified using the keywords *left*, *right*, or *centered*. Using one of these keywords causes the child to be positioned within its parent container accordingly. Similarly, the *y* option can be specified using the keywords *top*, *bottom* or *centered*. For example, the following command would construct a child of the container specified by the path name in *$w*, and position the child centered, along the top edge of the client area of the parent:

```
Button $w.button −x centered −y top
```

while a command of the form:

```
Image $w.image −w width −h height
```

might be used to create a child widget within a container that has the dimensions of the client area of the parent.

Finally, the *relative syntax* can be applied to the keywords to further adjust the computed locations and dimensions of the child widgets. For example, the following command would align the child along the right hand border of the parent client area with a gap between the child widget's border and that of the parent of 10 pixels:

```
Button $w.button −x right−10 −y centered
```

See the *wizard.tcl* script in the *scripts* directory of the distribution for an example of how keywords are used to aid in the layout of child widgets.

6.2.18 invertstate

The *invertstate* property determines how the value in the widget *statevariable*, if any, is to be treated. By default, *invertstate* is *false*, and the interpretation of the widget state variable is as described below. If the value of the *invertstate* property is *true*, then the interpretation of the value of the state variable is inverted.

For example, if the *statevariable* for a *Button* widget is a Tcl variable whose value is 0 and if i*nvertstate* is *true*, then the state of the widget will be *normal*. If, however, *invertstate* is false, then the state of the widget will be *disabled*.

Where a widget has no *statevariable*, the effect of this option is nothing. If a *statevariablecommand* has been specified for the widget, the *invertstate* option has no effect. The *statevariablecommand* script must set the widget state.

6.2.19 label

The *label* property is used to specify the text of the widget label. Widget labels are used for various purposes by the Fltk tool kit, and many of the Fltk extension widgets use the *label* text to display values that are either labels in the usual sense of the word, or the variable contents of the widgets themselves.

Here is an example of a widget command that sets the widget *label* text:

```
$w configure −label "This is label text"
```

Here *$w* is a token that represents the widget command. If the widget is a *Button* , the label text would become the text displayed in the button. If the widget is a *Label*, then the text would be the contents of the label. If the widget is an *Input*, then the label text would be displayed beside the widget as a label.

6.2.20 limits

The *limits* property is used to specify the range of values that the widget geometry can take. By default, widgets can be resized to any dimensions. Specifying limits will limit resize behaviour to the ranges given.

6.2.21 nocomplain

The *nocomplain* option is used to specify whether or not errors in configuration option names result in a command failure. By default, the value of *nocomplain* is *false*, and the presence of an invalid option name on a widget command will result in an error message. If the value of *nocomplain* is *true*, then invalid options are simply ignored. This option is useful when constructing compound widgets. One set of configuration options can be passed to all of the widgets in the compound widget. Invalid options for specific widgets will be ignored.

6.2.22 padx,pady

The *padx* and *pady* properties are used to specify internal padding values for widgets. By default these values are both 0. Widgets such as the *Package* and *Image* widgets use these value to position their child widgets.

6.2.23 qualifiednames

The *qualifiednames* option is used to force the widget configuration and query functions to respond only when passed a properly qualified option name. By default, the value of the *qualifiednames* option is *false* and the widget will accept either qualified option names or unqualified option names. By setting the value of this option to *true*, the widget will process only qualified option names.

6.2.24 relief

The *relief* property is used to specify the relief that is used to draw the widget. The Fltk tool kit defines two classes of *relief*, frame relief and filled relief. Frame relief draw only the borders of the widget in the specified relief style, while filled relief will draw the internal background of the widget as well. Usualy, a frame style is used when the entire inside contents of a widget are filled with other widgets.

The Fltk extension provides a large number of relief types. The *Help* command can be used to list all of the values. Commonly used values are:

● none	No relief
● raised	Raised
● sunken	Sunken
● flat	No relief
● ridge	A Ridged relief
● groove	A grooved relief

Note that the value *none* is not the same as the value *flat*. Where *none* is specified, nothing is drawn, and unless the widget contains some other drawings, you see the desktop background through a transparent area that represents the widget.

6.2.25 resizeable

The *resizeable* property is currently a boolean value that determines whether a widget may be resized. By default this value is *true*.

6.2.26 state

The *state* of a widget can be either *normal* or *disabled*. By default all widgets are created in the *normal* state. When *disabled* a widget will not process any input events when it has focus.

6.2.27 statevariable

The *statevariable* property is used to set up a relationship between the state of a widget and a Tcl variable. The Tcl variable should have a binary behaviour that can be used to deduce the state of the widget. This means that the state of the widget will be *disabled* when the Tcl variable is either an empty string or has the value of 0. When the variable is non zero, or contains a non empty string, the state of the widget will be *normal*.

Using *statevariables* is a convenient method of setting the state of widgets in an application based on something that may be going on in the application. For example, in a client and server application, the *statevariable* might be a socket connection. When a connection occurs, all of the widgets monitoring the connection handle will change state.

Here is an example of a *Button* that is tied to a Tcl variable for its state:

```
set Data(ButtonState) 0

Button t.b1 −statevariable Data(ButtonState) −label Disconnect −command Disconnect
```

In this example, the initial state of the *Button* will be *disabled* because its state variable is 0. When the state variable changes to a nonuser value, the state of the *Button* will also change to *normal*. Here, when active, the *Button* will, presumably, disconnect the connection and restore the state variable to 0 again, thereby disabling the *Button* .

The Tcl variable used for *statevariable* bindings must be a Tcl global variable. They can be simple variable or members of Tcl arrays, as in the above example.

6.2.28 statevariablecommand

The *statevariablecommand* property is used to specify the script that is to be executed when the *statevariable* of a widget changes. If a widget has a *statevariable* and it changes value because of some application related action, the script specified for the *statevariablecommand* is executed. By default, widgets have no *statevariable* associated with them, and the value of the *statevariablecommand* has no effect.

The scripts specified for the *statevariablecommand* property are first expanded in the manner of *command* scripts to substitute and embedded keywords, then are executed. If a *statevariablecommand* script is specified, then the command must set the state of the widget. If no *statevariablecommand* script is specified, then the combination of the contents of the *statevariable* and the *invertstate* options will determine the state of the widget. For example, the following command:

```
$w set −statevariable MyVar −statevariablecommand "if { $MyVar == Off } { %W set −state disabled } else { %W set −state normal }"
```

would cause the interpreter to check the value of the Tcl variable *MyVar* whenever it changes, then set the state of the widget whose path name is contained in the variable *$w* according to the contents of MyVar.

6.2.29 tooltip

The *tooltip* property can be used to specify a tool tip style help text string for the widget. Tool tips are short captions that appear when the mouse pointer lingers over a widget. By default, widgets are created without any *tooltip* text. If *tooltip* text is specified then the tool tip feature is automatically activated for the widget.

The text specified for a *tooltip* can contain the following embedded keywords:

%w	The path name of the widget
%l	The label text of the widget
%d	The current widget data
%v	The current widget variable
%s	The current widget state variable

Before a tool tip is displayed, the keywords are replaced by their relevant values. Here is an example of a *Button* with a *tooltip* text:

```
Button t.b −tooltip "Help for the %l button" −label Dismiss
```

This command will produce the tool tip "Help for the Dismiss button" when it appears.

6.2.30 underline

The *underline* option controls the state of underlined text in the label. By default, the value of this option is *true.*

6.2.31 variable

The *variable* property is used to associate the *value* of a widget with a Tcl variable. This option only applies to widgets that have the *value* property, such as *Buttons*, *Scrollbars*, *Input* and *Output* widgets and a few others. By default, the value of the *variable* property is an empty string and no Tcl variable is bound to the *value* of the widget.

If the name of a Tcl variable is specified for the *variable* property, then whenever the *value* of the widget changes, the new *value* will be stored in the Tcl variable. Similarly, should the value of the Tcl variable change, the *value* of the widget is automatically updated to reflect the change.

The Tcl variable can be a simple variable or an array element, but must be a global variable. If the variable does not exist when the *variable* property is set, a suitably named global variable will be created and initialized from the *value* of the widget. If, conversely, a Tcl variable with the specified name does exist, the *value* of the widget will be initialized from the Tcl variable.

Here is an example of a *Label* widget whose text is bound to the contents of a Tcl variable:

```
set Data(LabelText) "This is some text"

Label t.l −variable Data(LabelText) −width 200 −alignment left,inside
```

The result will be a *Label* widget which contains the text in the bound variable aligned in a left justified fashion.

6.2.32 variablecommand

The *variablecommand* property is used to specify a script to be executed whenever the *value* property of a widget changes. The script can contain keywords that are replaced with appropriate values before the script is evaluated. See the discussion on script expansion for the details on the available keywords.

By default, widgets have no *variablecommand* script. If a widget is not bound to a Tcl variable, the value of the *variablecommand* property has no effect. If a script is specified, then the script should set the appropriate *value* option of the widget when it is invoked. If no script is specified, then the *value* option of the widget is set by the automatically. For example, the following command:

```
$w set −variable MyVar −variablecommand "%W set −value $MyVar"
```

would be equivalent to the command:

```
$w set −variable MyVar
```

In either case, the value of the *value* option of the widget whose path name is in the variable *$w* would be set the the contents of the Tcl variable *MyVar*.

6.2.33 wraplength

The *wraplength* property is used to specify the character position in a text string to use for wrapping the text. In Fltk the determination of this value is automatic, so this option is ignored.

6.2.34 wallpaper

The *wallpaper* property is used to specify the name of an image file that contains an image that is to be used as the background image for the widget. not all widgets support this option. Typically, *wallpaper* images are used with *Toplevel* widgets to implement themes. By default, widgets are created with no wallpaper image.

The image files should be in one of the image formats that is supported by the *Image* widget. Widgets such as the *Toplevel* widget support features such as centering and tile placement of the background image.

6.3 Configurable Options and the Option Database

All configurable options can be initialized using values in the option database. The *Option* command is used to specify the contents of the option database. The contents of the database consist of string keys and priority based values that are retrieved when widgets are constructed or configured.

Widget options that are integer values, such as *x, y, width* and *height* can be configured using a special syntax that relates the configured value to the current contents of the value applied to the widget in the option database. The syntax is recognized by pre pending to the specified configuration value a non digit operator symbol that specifies how to apply the value following the operator to the value in the option database to produce a value with which to configure the widget.

The list of operators is:

+	Add the value to the database value
−	Subtract the value from the database value
*	Multiply the value by the database value
/	Divide the database value by the value
%	Take the modulus of the database value and the value
&	Take the logical AND of the value and the database value
|	Take the logical OR of the value and the database value

If there is no option value set in the option database that applies to the widget being configured, the database value applied is zero. For example, the widget command:

```
$w config −x +10 −y +10
```

would place the widget identified by $w at the location computed using the current option database values for x and y and adding the value 10 to each.

6.4 Initialization of Widgets from the Option Database

When a widget is created its characteristics are initialized by first searching the option database for key patterns that could refer to the widget and applying any matches that are found to the initial values of the widget configurable options.

All applications define an application *name string* and a *separator string* that are used in the generation of keys for searching the option database. The database is searched for keys in the following order:

```
global
application name.global
widget class
widget class.widget name
application name.widget class
widget name
application name.widget name
application name.widget class.widget name
```

Here, the *separator* is a period, and the *application name* is the current application *name string*. These two parameters can be set using the *Application* command. The *widget name* is just the path name of the widget and the *widget class* is just the class name of the widget.

The keyword *global* means that an option should be applied to all widgets in all applications that make use of a specific option database. The option database itself could be saved in a file that is automatically read whenever the Fltk extension is loaded, in which case the global keyword could be used to set the default values for all widgets in any application that is run using the extension. For example, the default location for all widgets and the default background color for all widgets could be set using the following commands:

```
Option add global.x 50
Option add global.y 50
Option add global.background tan
```

The default application name is "Fltkwish" so the relief for all widgets of class *Button* could be preset using the following command:

```
Option add Fltkwish.Button.relief raised
```

which will cause the default relief for all buttons in that application to be raised.

Typically, a non permanent copy of the option database is created for each application run using the Fltk extension. Option values are set to configure the widgets used in the GUI so that all widgets, or all widgets in a specific class have common default characteristics. For example, the following command will configure the default *relief* of widgets in the *Button* class:

```
Option add Button.relief ridge
```

Since the usual default relief for *Button* widgets is *raised*, any widgets created by this instance of the Fltk extension would now have a raised *relief.*

6.5 Using Widget Commands

When a widget is constructed, the result of the widget creation command, if no error is detected, is a token that represents a widget command. The widget command supports the functionality of the widget. For example, the command:

```
set w [Frame t.f]
```

creates a *Frame* widget and returns the token that represents the widget command to the Tcl variable *w*. This token can then be used as a widget command to access the features of the *Frame* widget.

Here is an example of the use of a widget command:

```
$w config −x 20 −y 40 −label "My Widget"
```

This command will configure the widget identified as *$w* to be position at location (*x*,*y*) relative to its parent widget, and to have a *label* of "*My Widget*". The use of *label* strings varies with the type of widget. For a *Toplevel* widget, the specified *x* and *y* coordinates will be relative to the screen.

6.6 Widget Construction

Widgets are created using the appropriate Fltk extension command, such as *Toplevel, Frame* or *Button.* The general format of these commands is:

```
Widget path ?−opt? ?value? ...
```

Where *Widget* is the actual command that creates the widget and *path* is the path name of the widget. A valid path name consists of a set of strings separated with periods (.) that specify a route to a root node in a hierarchical tree. The first element of the path is the name of the top level widget that is the root parent of the widget being created, and the last element is the unique name of the widget itself. For example, the path name:

```
root.frame.child
```

says that *child* is a child of *frame* which is itself a child of *root*.

Generally speaking, the elements of a path name can be any sequence of characters that are not prohibited by Tcl. Tcl uses some characters to indicate special treatment during the evaluation of a command. Quotation marks ("), dollar signs ($), square ([]) and curly ({}) brackets must be used with circumspection as path name elements. The Fltk extension does not care what the string contents are, depending only on the period for its proper functioning in finding widget parents. Names such as:

```
$...[help]{}
```

will probably cause undesirable results when running your applications.

The special root widget name '.' is intrinsic to the Tk package. If the Fltk extension is running while Tk is active, then you may not name a root widget '.'. Doing so will cause Tk to assume the widget is one of its own set, and unless this is what you intend, you will have some problems. If Tk is not active, then you can use the name '.' for your root widget.

6.7 Widget Destruction

Widgets that belong to the Fltk extension can be destroyed by three basic mechanisms, using the *Destroy* command, closing the widget's parent container widget, or using the Tcl *rename* command.

The *Destroy* command is the direct method of destroying a widget. The named widgets in the *Destroy* command are closed and removed from the master list of widgets managed by the Fltk extension.

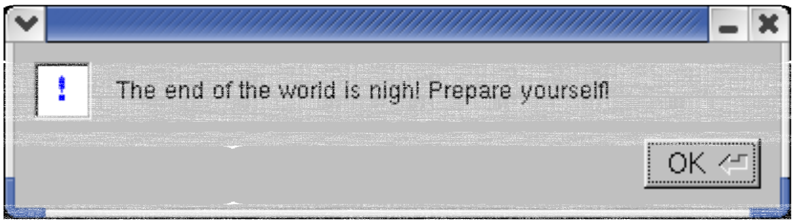
If the parent container of a widget is destroyed, then the widgets that are children of the container are also closed. A parent container is either a top level widget, which might have been closed using its system menu, or it might be a *Frame* widget that acts as a group container for a number of widgets that have their geometry managed by the *Frame*.

The Tcl *rename* command can be used to destroy a widget by renaming its widget command to an empty string. The Fltk package initialization procedure actually captures the Tcl *rename* command to check for the case of a rename with implied destruction, and calls the *Destroy* command when needed.

Of course, the Tcl *exit* command, or the Fltk *Exit* command, will also close all open widgets.

7 Alert − Display an alert message

The *Alert* command can be used to display a message box that contains a message. The user responds by pressing a button to indicate that the alert message is acknowledged.


The end of the world is nigh! Prepare yourself!

The format of the command is:

```
Alert message
```

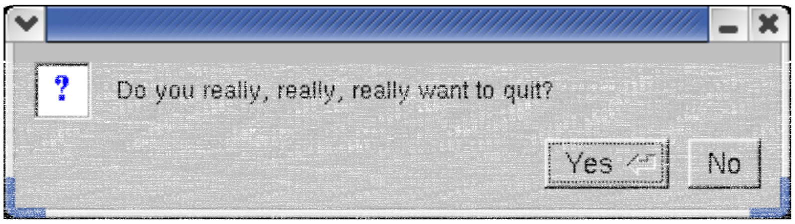
where *message* is the text of the alert message. If no message is supplied an error message is returned, otherwise, nothing is returned. For example:

```
Alert "The world is coming to an end! Prepare thyself!"
```

is a method of either alarming or amusing the user.

8 Ask − Ask a question

The *Ask* command will display a message box with two buttons that present the user with two possibilities for answering a question, either *yes* or *no*.


Do you really, really, really want to quit?

The format of the command is:

```
Ask message
```

where *message* is the question to be posed. If the user selects the *yes* response button, the command returns without raising an error, if the user selects the *no* response, the command raises an error.

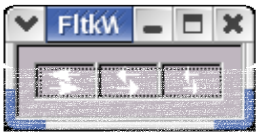
For example:

```
if { catch { Ask "Do you want to continue?" } result } { exit }
```

would terminate an application if the user selects *no*.

9 Adjuster − Create an adjuster widget

An *Adjuster* is a widget that changes its value property through user interaction with the mouse. The classic spinner is an example of an *Adjuster*.



The format of the command is:

```
Adjuster path ... options ...
```

Where *path* is a valid widget path and *options* are the list of option and value pairs used to configure the widget. In addition to the standard set of widget options, the *Adjuster* supports the following widget specific options:

● max	Set the maximum value for the widget value
● min	Set the minimum value for the widget value
● orientation	Set the orientation of the widget
● step	Set the step value for the widget value
● value	Specify the current value for the widget

By default, the values of *min* and *max* are 0 and 100 respectively, the value of *step* is 1, and the value of *value* is 0. *Orientation* can be used to set the layout of the *Adjuster* controls to either *horizontal* or *vertical*. The default orientation is *horizontal*.

For example, the following command will create a spinner that steps by 5 units from −50 to 50:

```
Adjuster t.a −min −50 −max 50 −step 5 −value 0 −orientation vertical −variable CurrentValue
```

This *Adjuster* updates the Tcl global variable *CurrentValue* each time the user changes its value by clicking on one of its controls.

10 Application − Specify application data

The *Application* command is used to query or define application data. The command supports the sub functions *configure* and *cget. Configure* is used to set the values of the application variables while *cget* is used to obtain the current values.

The list of application variables available is as follows:

● name	The name of the application
● version	The application version string
● copyright	The application copyright string
● comment	A comment string
● data	A string of user data
● separator	A single character that is used for option parsing
● compatibility	A boolean value controlling Tk compatibility features
● options	A string of values that may be used to pass options to an application.

All of the variables are strings. In a safe interpreter, the *configure* function is not available, so the initialized values set at the compile time of the Fltk extension in use are fixed.

The format of the command is:

```
Application cget ?−var? ...
```
or
```
Application configure ?−var? ?value? ...
```

where *var* is the name of the application variable and *value* is a string that specifies the new value for the variable. If no parameters are specified after the function name, then the list of options available will be returned.

For example, to set the application *name* and *version*, use the command:

```
Application configure −name "My Application" −version "1.01"
```

and to determine whether the Tk compatibility mode is active, use the command:

```
Application cget −compatibility
```

This latter command will return the string *true* or *false* according to the state of the compatibility mode. When compatibility mode is *true* commands that begin in lower case are treated as Tk commands, while commands that begin in upper case are treated as commands for this package. This is the default mode, and it allows Tk to co−exist with the toolkit extension. If compatibility mode is *false* Tk should not be loaded when using this extension.

The default values of the application data are set when the Fltk extension is compiled. They have the following values:

name	"fltkwish"
version	The current version number of the extension
copyright	"Copyright(C) I.B.Findleton, 2000,2001. All Rights Reserved"
separator	"."
compatibility	true
options	""

all other values are empty strings.

11 Bind − Manage event bindings for widgets

The *Bind* command is used to associate an event with a widget. An event can be either one of the pre−defined events that are associated with keyboard, mouse, or window manager actions, or it may be a user defined event that is activated by some mechanism such as the *Signal* command.

The format of the *Bind* command is:

```
Bind widget name script
```

where *widget* is either the *path name* of a widget or a widget *class name*, *name* is a string that names the event and *script* is a Tcl script that is to be executed when the widget receives the event.

The *widget* parameter can be the special keyword *all*. In this case, the binding is a global binding that will be applied to all of the widgets in the current widget list. If *widget* is not *all* and it is not the name of an existing widget, it is assumed to be a widget *class name*. The event will be automatically bound to all of the widgets of the specified class that exist or are subsequently created.

The following command would bind an event script to all of the widgets currently in the widget list:

```
Bind all <ButtonPress> { HandleClick %W %x %y }
```

while the following command:

```
Bind Button <ButtonPress> { HandleClick %W %x %y }
```

would bind the event script to all widgets of class *Button*. Given that there is a widget with the path name of *t.b* , then the following command would bind the event script to it:

```
Bind t.b <ButtonPress> { HandleClick %W %x %y }
```

Note that in the case of a specific widget, the widget must exist at the time the *Bind* command is executed if it is to be bound to the event handler.

11.1 Event Names

An event name can be any string of characters. The list of pre−defined events include the following:

• <Nothing>	No event
• <Resize>	A resize or reposition event
• <ButtonPress>	A mouse button is pressed
• <ButtonRelease>	A mouse button is released
• <KeyPress>	A keyboard key was pressed
• <KeyRelease>	A keyboard key is released
• <Enter>	The mouse entered a widget
• <Leave>	The mouse left a widget
• <Motion>	The mouse moved in a widget
• <FocusIn>	The widget received focus
• <FocusOut>	The widget lost focus
• <Activate>	The widget is activated
• <Deactivate>	The widget is deactivated
• <Destroy>	The widget is destroyed
• <Map>	The widget became visible
• <UnMap>	The widget became invisible
• <Paste>	The widget is receiving data
• <Selection>	The widget should have a selection

- The widget is entered for drag and drop
- <DNDRelease> Drag and drop released
- <MouseWheel> Mouse wheel motion in the widget

These built−in names for events are typical of many of the window managers that are used on GUI based computer interfaces and their meaning is, for the most part, obvious. For example, to track the location of a mouse pointer inside a widget, the following code might be used:

```
Bind $w <Motion> { puts "Widget %W Mouse Location (%x,%y)" }
```

When the mouse is moved over the widget specified by *$w* then the interactive console would see the message specified in the script.

11.2 User Event Bindings

The following command will bind an event handler to a user defined event that is named *MyEvent*:

```
Bind t.w MyEvent { puts "Event MyEvent occurred in widget %W" }
```

At some point in an application, this event can be raised using a command of the form:

```
Signal t.w MyEvent −x x −y y
```

where *x* and *y* are the values that the event should report as the window relative location where the event occurred. Here the *Signal* command is being used to create a simulated event with the name *MyEvent*. User event names can be any string of characters, however, they must be unique within an application in the event name space. You can not override, or overload, the set of predefined event names.

11.3 Script Expansion

The *script* associated with the event can specify a number of special tokens that are replaced with widget and window manager data before it is evaluated. The tokens are recognized as sub strings of the script that begin with the % sign. The list of supported tokens is as follows:

%%	Replaced with a single % sign
%#	Event serial number
%A	The ASCII value of the keyboard event
%b	The mouse button that caused the event
%k	The key code that caused the event
%K	The key symbol that caused the event
%N	The decimal value of the ASCII key that caused the event
%R	The name of the parent of the widget
%t	The time code of the event
%T	The name of the event
%W	The path name of the widget
%U	The user data associated with the event
%x	The widget relative horizontal location of the event
%X	The screen relative horizontal location of the event
%y	The widget relative vertical location of the event
%Y	The screen relative vertical location of the event

For example, in the *script* associated with the <Motion> event shown above, the items *%W, %x* and *%y* would be replaced with the widget *path name*, and the window relative location of the mouse when the event was generated.

In the Fltk tool kit there is a distinction between a *window* and a *widget*. *Windows* are widgets that are managed by the native window manager on the computing platform in use. *Widgets* are visual objects that are created by the Fltk tool kit. A *Toplevel* widget is a window. Typically, a *Toplevel* widget will contain a collection of widgets that make up the graphical user interface of the application. When events are handled for a widget in the collection, it is important to note that the values of event locations, such as the position of the mouse or where the cursor was when a keystroke happened, are usually relative to the *Toplevel* container, not the widget itself.

To convert from the *window relative* coordinates to *widget relative* coordinates, note that the widget command can be used to retrieve the location of a widget within a window. For example:

```
set x [$w cget −x] ; set y [$w cget −y]
```

will get the location of the widget whose command token is in *w* with respect to the *containing window*. These values can then be subtracted from the event positions to get the *widget relative* location of an event.

11.4 Event Processing

Associated with each widget is a list of tags that specify the order of processing events received by the widget. This list is created by default with 3 or 4 tags in the following order:

● widget name	The path name of the widget
● parent name	The path name of its parent if it is not a top level
● class name	The class name of the widget
● all	Global event handlers

Clearly, *Toplevel* widgets and some types of *Menu* widgets have no parents, so they have only 3 tags in their default bind list. Other widgets will have all four tags.

Events received by the widget are handled by proceeding down the tag list until either all bindings have been successfully processed or until the script associated with a binding returns either TCL_ERROR or TCL_BREAK. A widget may have an event binding associated with each of the tags in its tag list. For example, the script:

```
        Bind Button <Enter> { %W set −background yellow }
        Bind t.b1 <Enter> { %W set −foreground green }
        Bind t.b2 <Enter> { %W set −foreground red}
        Bind Button <Leave> { %W set −foreground black −background gray }
would cause all Button widgets to turn to a yellow background when
the mouse enters the widget, and back to a gray background when the mouse
leaves the widget. The particular buttons t.b1 and t.b2, however, would adopt
green and red foreground colors, respectively.
```

In addition to the default set of tags, widgets may have tags associated with user defined events The order of the bind tags can be established using the *BindTags* command. Using this command an arbitrary list of event bindings can be specified for the widget.

12 BindTags − Manage event processing for a widget

The *BindTags* command is used to specify the content and order of the event binding processing list for a widget can be specified. By default, all widgets inherit a list that orders the event processing sequence as:

```
{ widget ?parent? class all}
```

where *widget* is the path name of the widget, *parent* is the path name of the widget's parent if it has one, *class* is the class name of the widget, and *all* signifies the global event bindings that apply to all widgets.

The format of the BindTags command is:

```
BindTags widget ?list?
```

where *widget* is the path name of the widget and *list* is a list of tag names that specifies the content and order of the event processing tags for a widget. If *list* is not present, the result of the command is the current list of tags for the widget.

The *list* specified can contain both the standard tag names such as the *widget name*, its *class name*, its *parent's name*, and *all*, as well as user defined event binding names. If the *list* is empty, then the tag list for the widget is cleared, and events will not be handled by the widget. Note that an empty *list* is not the same as specifying no list on the command.

For example, to clear the list of tags for a widget, use the command:

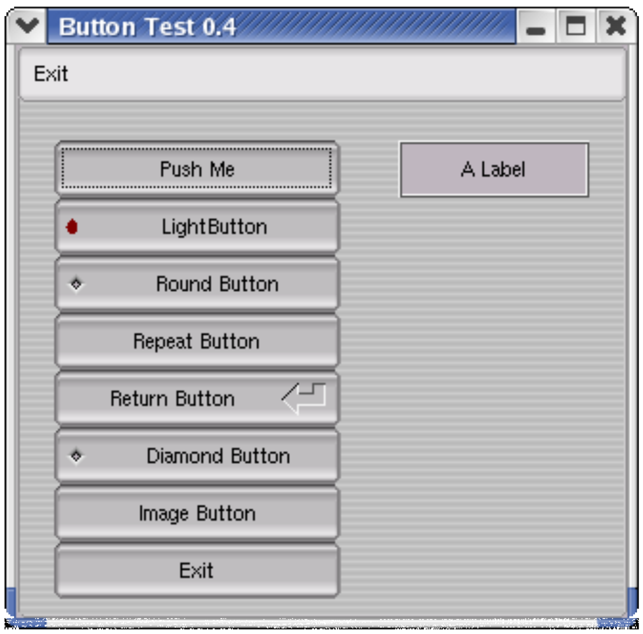
```
BindTags $w {}
```

and to reverse the standard order of processing for a widget, use:

```
BindTags $w { all [Winfo $w class] [Winfo $w parent] $w }
```

13 Button, CheckButton, DiamondButton, LightButton, RepeatButton, ReturnButton, RoundButton, LEDButton – Construct a button

The *Button* command is used to construct a standard button widget that is characterized by a typical rectangular button with text centred in the rectangle. This is the basic widget of the *Button* widget class.



Along with the *standard widget options*, the *Button* widget class supports the following widget specific configurable options:

- type Specifies the behaviour of the button
- value Specifies the value of the button
- onvalue Specifies the on value string
- offvalue Specifies the off value string
- shortcut Specifies the name of the shortcut key
- downrelief Specifies the button relief when pressed

Button *type* can be *invariant*, *toggle*, or *radio*. The default *type* is *invariant* which means that the value of the button does not change when the button is pressed and released. For *toggle* buttons, the value is inverted when the button is pressed, and for a *radio* button, the value is set for the button and cleared for all other buttons in the same container group. (Container groups are determined from the widget path name).

The *value* of a button is internally either *1* or *0*. Externally, the options *onvalue* and *offvalue* can be used to specify strings that will be returned by the widget command depending on the current internal value of the button. For example, a button constructed as:

```
Button root.b1 −onvalue true −offvalue false −value true
```

would result in the widget command:

```
root.b1 cget −value
```

returning either *true* or *false* depending on the internal value of the button. Note that the construction of the button set the initial *value* according to the specified *onvalue* and *offvalue* strings. By default, all buttons have *onvalue* and *offvalue* strings of *1* and *0* respectively.

The *shortcut* for a button specifies the key sequence that can be used to effect a button press and release sequence from the keyboard. Shortcut specifications are just strings of the form:

```
Control−Alt−PgUp
```

that indicate that, in this case, the button is activated when the keyboard key *PgUp* is pressed while the *Ctrl* and *Alt* keys are depressed.

The *downrelief* option is used to specify the type of *relief* that the button will have when it is pressed. The *relief* when released is set using the standard widget option *relief*. By default, buttons have relief of raised and a *downrelief* of *sunken*.

13.1 Typical Button Use

The *Button* widget is a very common component of GUI applications. Typically, when a *Button* is pressed, the result is an action that is executed, usually a Tcl procedure. In addition to the action, GUI developers sometimes prefer to have the visual characteristics of the button change when the mouse moves over the widget to indicate that it has the current input focus. Here is an example of a fairly standard button that is used to terminate an application:

```
Button root.quit −command Exit −text "Dismiss" −tooltip "Click to terminate the application"

Bind root.quit <Enter> { %W configure −fg red }
Bind root.quit <Leave> { %W configure −fg black }
```

Here the *command* script to be executed when the button is pressed is the *Exit* command, which will terminate the application. The *Bind* commands are used to cause the *text* in the button to change from *black* to *red* when the mouse is over the button, then back to *black* when the mouse leaves the area of the button. The *%W* token in the *Bind* scripts is replaced by the *path name* of the button widget which is, in this case, *root.quit*.

All widgets that are in the collection of buttons, including the *ImageButton* widget, are members of the *Button* widget class. By default, these widgets will have the class names *Button* and a class name that is identical to that of the widget constructor command. Using the class name it is possible to implement uniform *Button* behaviour. For example, the commands:

```
Bind Button <Enter> { %W configure −fg red }
Bind Button <Leave> { %W configure −fg black }
```

would make all buttons respond to mouse movement over the widgets in the same way.

13.2 CheckButton − Create a checkbutton

A *CheckButton* is a button that has 2 states, on and off. *CheckButtons* support the same options as the *Button* widget. This button displays a small check box in its client area that displays the current state of the button. Here is a command that controls the value of a Tcl variable named *MyOption* using a CheckButton:

```
CheckButton root.check −variable MyOption −text "My Option" −onvalue TRUE −offvalue FALSE
```

Whenever the button is clicked, the value of the variable *MyOption* will change between TRUE and FALSE.

13.3 DiamondButton − Create a button with a diamond indicator

A *DiamondButton* is a button that uses a diamond shaped indicator that shows the state of the button value. It supports all of the options of the Button widget. The format of the command is:

```
DiamondButton path options
```

where *path* is the path name of the widget to be created and *options* is the list of option and value pairs used to configure the widget. Here is an example of a DiamondButton:

```
DiamondButton root.b −x 100 −y 100 −text "Diamonds?" −command { puts "Diamonds are forever, husbands leave!" }
```

13.4 LEDButton − Create a LED button

A *LEDButton* is a button widget that uses a simulated LED instead of a coloured square. It is otherwise similar to a *LightButton*. This button can be used to control the value of a Tcl Variable in the same manner as the CheckButton example. The only real difference between these buttons is the type of indicator drawn.

13.5 LightButton − Create an illuminating button

A *LightButton* is a button widget that has a light that can be illuminated either by pressing the button of changing its value. It supports all of the options of the *Button* widget.

13.6 RepeatButton − Create a repeat button

A *RepeatButton* is a button that repeatedly issues its command while it is held down. It supports the same options as a *Button* widget. This button is convenient when developing an application that has to loop through a list. By holding down the button, the command will repeat. For example:

```
set idx 0

RepeatButton root.r −text Next −command { puts $idx; incr idx }
```

will produce a nicely incrementing list if integers when it is held down.

13.7 ReturnButton − Create a return button

A *ReturnButton* is a button that generates a key event with the key value for the enter key. It responds to the same option list as the *Button* widget This button is a good one to use as the default response to a dialog box. For example:

```
ReturnButton root.quit −text "Exit Now?" −command Exit
```

This button can be either clicked with the mouse, or the enter key can be pressed to cause the current application to be terminated.

13.8 RoundButton − Create a round button

A *RoundButton* is a button that is useful for creating radio button style dialogs. It has a sub widget that illuminates according to the value of the button. It supports the same widget options as the *Button* widget. This button behaves the same way as the *CheckButton* and the *LightButton*.

14 Call − Call a function from a library

The *Call* command is used to execute a procedure or script module that is contained in a binary format library. The library must be in the current list of library files. The format of the command is:

```
Call options name parameters
```

where *options* is an optional list of option and value pairs that control how the procedure or module is executed, *name* is the name of the procedure or module, and *parameters* is the list of parameters to be passed to the procedure or script module.

The *Call* command supports the following options:

- deleteparams − If the parameters are to be deleted on return
- displacement − The key inset value to use
- exit − If the script should exit on error
- key − The key value to use
- keyfile − The name of the key file to use
- library − Specify a local library list
- mode − The parameter handling mode
- source − If the module is in source format

The *deleteparms*, *displacement*, *key* and *keyfile* options are used to control the decoding of binary modules. Their values depend on the way the binary modules were generated These values should match those used with the *encode* utility to generate the binary module.

The *exit* option determines the behaviour of the *Call* command in the face of a script error. By default, the value of the *exit* option is *false,* and the result of a script error in the module is the return of the error and an exit code of TCL_ERROR. If the value of the *exit* option is *true,* the incidence of an error in the script module or procedure will cause the application to terminate.

The *library* option is used to specify a list of library files that are to be searched for the specified script module or procedure. By default, the *Call* command will search the list of libraries specified using the *Library* command. If the *library* option is specified, the search for the procedure or script module is restricted to the specified list of library names.

The *mode* option specifies how parameters passed to script modules are treated. By default, the value of this option is *append,* and the list of parameters is added to the standard argv parameter list. If the value of the *mode* parameter is *push,* then the specified parameters replace the standard *argv* list while the script module is being executed.

The *deleteparams* option specifies how the specified parameters are handled. By default, the value of this option is *true,* and on return from the script module or procedure, the parameters are deleted. By setting this option to *false,* the specified parameters will remain as part of the standard *argv* list. Where the specified *name* is the name of a procedure, the *deleteparams* option has no effect.

The *source* option is, by default, *false,* and the specified script module or procedure is assumed to be in binary format. If the value of the *source* option is *true,* then the module must be in source format.

Here is an example of the use of the *Call* command:

```
Library add myproclib.lib

Call MyProcedure p1 p2 p3
```

Here, the *Library* command has been used to add a library file to the current library search list. The library file *myproclib.lib* contains a script module that defines the procedure *MyProcedure*, which takes 3 parameters. The *Call* command will find the procedure in the library file, execute the procedure and return the result.

The above example could also be written as follows:

```
Call −library myproclib.lib MyProcedure p1 p2 p3
```

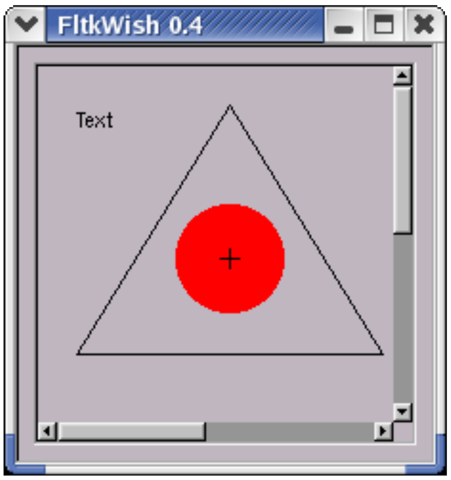
The *name* specified for the *Call* command can also be the name of a script module that is in a library file. The *Call* command will locate the script module by its name, and evaluate the script, in a manner similar to that of the *Run* command, or, in the case of script files, that of the standard *source* command. When a script module is the target of the *Call* command, the supplied parameters are either appended to or replace the standard *argv* list, and the disposition of the parameters is as specified using the *deleteparams* option. For example, a module could be evaluated using a command of the form:

```
Call MyModule.fltk arg1 arg2 arg3
```

Here, the parameters *arg1, arg2* and *arg3* would be appended to the standard *argv* parameter list, and the script module whose name is *MyModule.fltk* would be evaluated. On return from the script, the parameters would be deleted from the *argv* list and the result of the script would be the result of the *Call* command.

15 Canvas − Create a canvas widget

The *Canvas* command creates a widget that implements a general purpose drawing surface. The *Canvas* widget provides a number of drawing primitives that can be used to construct complex drawings and provides support for event bindings to the elements of the *Canvas* drawing.



The format of the command is:

Canvas path ... options ...

where *path* is the path name of the widget and *options* are option and value pairs that are used to configure the widget. The *Canvas* command supports the list of *standard widget options*.

15.1 Widget specific commands

Most of the functionality of the *Canvas* widget is implemented using widget specific commands. The *Canvas* widget supports the standard *cget* and *configure* commands, as well as the following widget specific function commands:

create	Create a canvas item
delete	Delete a canvas item
itembind	Bind events to canvas items
itemcget	Get item attributes
itemconfigure	Set item attributes
itemlist	List the names of canvas items
load	Load a file of canvas items
save	Save canvas items to a file

The general format of a canvas function command is:

$w function options

where *$w* is the path name of the *Canvas* widget, *function* is one of the supported functions, and *options* are either option names or option name and value pairs supported by the command.

15.2 Canvas Items

A canvas item is a geometric primitive that has a collection of properties. Every item supports a set of basic properties and possibly a set of item specific properties. The list of basic properties supported by all canvas items is:

origin	Location of the item origin
rotate	Rotation angle for the item
scale	Scale factors for the item
translate	Translation of the item
color	Drawing color of the item
fillcolor	Filled color of the item
linestyle	Line style of the item
linewidth	Width of the line used for an item
tag	Tag list for the item
activefillcolor	Filled color of an active item
disabledfillcolor	Filled color of a disabled item
activecolor	Drawing color for an active item
disabledcolor	Drawing color for a disabled item
activelinestyle	Line style for an active item
disabledlinestyle	Line style for a disabled item
state	The state of the item
transform	Transformation matrix
x	Horizontal location
y	Vertical location
vertex	A vertex
vertexlist	List of vertices
extent	Extent of the item
width	Width of the item
height	Height of the item

Using the *itemconfigure* command all of the item properties can be specified. Using the *itemcget* command all of the item properties can be queried. Not all items make use of all the properties. Rotation of a *circle*, for example, has no obvious use, so the *circle* item does not make use of the *rotate* value.

15.2.1 The origin of Canvas items

All of the coordinates of location properties of canvas items are specified relative to an *origin*. By changing the *origin* of an item the object can be moved about on the canvas. The *origin* is set by default to be (0.0). Canvas coordinates are relative to the upper left hand corner of the widget. When a canvas item is created it is the usual case that the location of the item on the canvas will be specified using the *x* and *y* properties of the item. Items can be moved by either changing the *x* and *y* values, or by changing the *origin* of the item. An origin is specified as a pair of comma separated values.

For example, here is a *circle* on a canvas with an *origin* set so that the circle will be centred at the point (100,100) on the canvas:

```
set c [$w create circle −x 0 −y 0 −origin 100,100 −fill yes −fillcolor red −radius 20]
```

This command produces a red *circle* on the canvas. The value returned in the Tcl variable c will be a token that identifies the circle. This circle could then be moved about using commands of the form:

```
$w itemconfig $c −x 20 −y 20 −fillcolor green
```

which would move the circle so that its centre would be at canvas location (120,120) and its colour would be green.

15.2.2 The rotate property of Canvas items

The *rotate* property is an angle in degrees that is used to compute a rotation for a geometric object. Because of the implementation used for some canvas objects, this value is ignored. Other objects do however use the rotation angle. By default, the *rotate* value is *0* and objects are not rotated.

15.2.3 The scale property of Canvas items

The *scale* property is two numbers that specify the horizontal and vertical scale factor for the coordinate values used to specify the location and dimensions of geometric objects. By default, the *scale* values are set to (1.0,1.0) so that coordinates and lengths are identical to the values specified for the item. Changing the *scale* will distort, expand or shrink geometric primitives.

For example, here is a command that will change the scale of an existing canvas item whose token identifier is contained in the Tcl variable id:

```
$w itemconfig $id −scale 0.5,0.5
```

This command would shrink the size of the item by a factor of 2.

15.2.4 Canvas item geometry items

15.2.4.1 Positional properties

The location of the canvas item may be specified using the x and y configuration options. These values are measured with respect to the origin specified for the item. A circle, for example, can be moved about by changing the x and y values after it is created.

The position of some items can be specified with a list of vertices. A vertex list is a comma separated list of coordinate values that specify the location of polygon vertices.

15.2.4.2 Extent properties

The height and width properties can be used to specify the dimensions, with respect to an origin, of rectangular items.

15.2.5 The state property of Canvas items

Canvas items can have a state of either *normal*, *active*, *disabled* or *hidden*. By default an item is in the *normal* state. When a mouse moves over an item it becomes *active*, unless it is *disabled* or *hidden*. When *active* the item can respond to user input from mouse actions or keyboard actions. When *disabled* or *hidden* the item will not react to user input.

When building a collection of canvas objects, it is typically desirable that not all of the objects on the canvas be *active*. Things such as static background images and static text labels generally don't need to be responsive to mouse events, and the GUI designer may not want the visual appearance of these static items to change as the mouse moves over them. Setting the state of the items to disabled will prevent user input from affecting the canvas items.

For example, here is a procedure that produces a popup window with a background image and some text over the image:

```
# −−− about.tcl −−− Display an about box with version information

# Display the copyright information

proc About { name width height } {
```

```
# Destroy any previous widget trees beginning with the name prompt.

 catch { Destroy prompt }

 set f [Toplevel prompt]

# Create a canvas with dimensions that surround the image

 set c [Canvas $f.text −w $width −h $height]

# Create an image item as a background

 $c create image −origin 0,0 −state disabled −file $name

# Get the application data for this application

 set data [Application get −name −version −copyright −comment]

# These static text items will not be affected by mouse motion

 $c create text −x 20 −y 10 −text [lindex $data 0] −state disabled −disabledcolor black
 $c create text −x 20 −y 30 −text "Version [lindex $data 1]" −state disabled −disabledcolor black
 $c create text −x 20 −y 50 −text "Copyright (C) [lindex $data 2]" −state disabled −disabledcolor black
 $c create text −x 20 −y 70 −text "[lindex $data 3]" −state disabled −state disabled −disabledcolor black

 Show $f

 Wm title $f "About ..."
 }
```

This procedure makes a good about box for applications. The parameters are the *name* of the image file to use as a background, the *width* of the image and the *height* of the image.

15.2.6 Color properties of Canvas items

The visual appearance of Canvas can be altered by changing the colors used to draw the objects. The following properties affect the color presentation of the items according to their *state.* Unless they are *hidden* or *disabled*, an item is *active* when the mouse is within the boundaries of the item, and is *normal* when the mouse is not within the boundaries of the item.

15.2.6.1 The color property

The *color* property is used to specify the color employed to draw a canvas object when it is in the *normal* state. This color is the line color that draws the outline of the object. By default this *color* is black.

15.2.6.2 The activecolor property

The *activecolor* property is used to specify the drawing color for an object when it is in the *active* state. By default, the activecolor color is red.

15.2.6.3 The disabledcolor property

The *disabledcolor* property is used to specify the drawing color for an item when it is in the *disabled* state. By default, the *disabledcolor* color is gray.

15.2.6.4 The fillcolor property

The *fillcolor* property is used to specify the color used to fill items such as circles and closed polygons when these items are in the *normal* state. By default this color is the same as that of the *color* property.

15.2.6.5 The activefillcolor property

The *activefillcolor* property is used to specify the color that should be used to fill the item when it is in the *active* state. By default, the *activefillcolor* color is white.

15.2.6.6 The disabledfillcolor property

The *disabledfillcolor* property is used to specify the color that should be used to fill the item when it is in the *disabled* state. By default the *disabledfillcolor* color is gray.

15.2.7 Line style properties of Canvas items

The visual appearance of *Canvas* can be altered by changing the line style used to draw the objects. The following properties affect the line style of the items according to their *state*. Unless they are *hidden* or *disabled*, an item is *active* when the mouse is within the boundaries of the item, and is *normal* when the mouse is not within the boundaries of the item.

15.2.7.1 The linestyle property

The *linestyle* property is used to specify the type of line to draw when an item is in the *normal* state. The available line styles are:

solid	Solid lines
dash	Dashed lines
dot	Dotted lines
dashdot	Dashed and dotted lines
dashdotdash	Dash dot dashed lines

Line styles can be qualified with the following additional optional enhancements to the lines as they are drawn. These features can be added to the ends of line segments:

cap−flat
cap−round
cap−square
join−mitre
join−round
join−bevel

By default, the *linestyle* is set to solid.

15.2.7.2 The linewidth property

When an item is drawn on the screen, the drawing functions use a pen with a width specified by the *linewidth* property. By default the value is *1*.

15.2.7.3 The activelinestyle property

The *activelinestyle* property is used to specify the line style used to draw an item when it is in the *active* state. All of the styles specified for the *linestyle* property are available. By default, the *activelinestyle* is set to *solid.*

15.2.7.4 The disabledlinestyle property

The disabledlinestyle property is used to specify the line style used to draw an object when it is in the *disabled* state. All of the styles specified for the *linestyle* property are available. By default, the *disabledinestyle* is set to *solid.*

15.2.8 The tags property of Canvas items

Each item of the *Canvas* can have one or more tags associated with it. When an item is created it gains a tag that is the same as its name, and a tag that specifies its class. For example, all circles will have the tag circle. Tags can be used to configure groups of items that all have the same tags.

For example, the following widget command will create a triangle:

```
set t [$w create triangle −fill yes fillcolor blue −tags blue]
```

Suppose that the item identifier returned by this command is *tri5*. The resulting canvas item will have a tag list that contains the following items:

```
tri5,triangle,blue
```

Tags can be used to manipulate all of the items in a canvas that have the same tag. For example, using the *itembind* function, the following command would attach an event handler script for a mouse button press to all items with the tag *blue* on the canvas:

```
$w itembind withtag blue <ButtonPress> { puts {%w %x %y} }
```

Since the triangle has the tag *blue*, clicking a mouse button while inside the triangle will invoke the script.

15.3 Canvas Item Creation

Geometric primitives supported by the *Canvas* widget are created using the *create* function command. The format of the create function command is:

```
$w create type options
```

where *$w* is the path name of the widget to use, *type* is the type of geometric object to create, and *options* is a list of option and value pairs that are used to configure the attributes of the geometric object. The list of geometric objects supported is:

arc	Create an arc
circle	Create a circle
curve	Create a bezier curve
image	Create an image
line	Create a line
polygon	Create a polygon
point	Create a point
quadrangle	Create a quadrangle
rectangle	Create a rectangle
text	Create a text object
triangle	Create a triangle

The result returned by the *create* command is a token that identifies the canvas item that is created. This token is used to refer to the specific item for the purposes of other widget commands which manage, configure and query the characteristics of the widget. For example, the command:

```
set mytext [$w create text −text "Don't worry! Be happy!"]
```

might return the token *text10* into the variable *mytext*. This token could later be used to modify this text item to change the display color as follows:

```
$w itemconfig $mytext −color orange
```

All canvas items are defined by a set of option values that are initially set to reasonable default values, so items can be created with no options specified and configured at a later time.

15.4 Deleting Canvas Items

Items in a canvas can be deleted using a command of the form

```
$w delete id1 ... idn
```

where $w is the canvas command and the values id1 through idn are the item identifiers that identify the canvas items to be deleted.

15.4.1 Canvas Arc Items

The format of the command that creates an arc is:

```
$w create arc options
```

where *$w* is the path name of the canvas and *options* are the list of option and value pairs that is used to configure the arc. In addition to the set of standard canvas item options, the following item specific options are supported:

from	Start angle in degrees
to	End angle in degrees
fill	If the arc should be filled

The *from* and *to* angles along with the *extent* option values define the arc to be drawn. If the *fill* option is *true,* the canvas item will look like a section of pie. For example:

```
$w create arc −extent 100,100 −from 45 −to 90 −fill yes −fillcolor orange
```

will create an octant of an orange cream pie. The standard item options origin and extent define a rectangular area the encloses the circle that would define an arc of 360 degrees.

15.4.2 Canvas Circle Items

The format of the command used to create the circle canvas item is:

```
$w create circle options
```

where *$w* is the path name of the canvas to use and *options* are the list of option and value pairs used to configure the canvas item. In addition to the set of standard canvas item options, the circle canvas item supports the following item specific options:

radius	The radius of the circle
fill	If the circle is filled

The circle is created at the current *origin* with the specified *radius* and optionally filled with the current *fillcolor*. For example:

```
$w create circle −x 150 −y 150 −radius 200 −fillcolor blue −fill yes
```

will create a circle at location (150,150) and radius 200 filled with the color blue

15.4.3 Canvas Curve Items

The curve is an item that is drawn using 2 end points and 2 control points that are parameters to a Bezier curve tracing function. The format of the command is:

```
$w create curve options
```

where *$w* is the path name of the canvas and *options* is the list of option and value pairs that are used to configure the curve. The curve has no item specific options and supports the list standard canvas item options. To draw a curve, use a command of the form:

```
$w create curve −vertexlist 10,20,40,40,−20,−20,50,80
```

Here the pairs of numbers in the *vertexlist* represent to 4 locations that are used by the Bezier function to draw the curve.

15.4.4 Canvas Image Items

The format of the command used to place images on a canvas is:

```
$w create image options
```

where *$w* is the path name of the canvas to be used and *options* is the list of option and value pairs needed to configure the item. In addition to the standard canvas item options, the image item supports the following options:

file	Name of the image file to use
colormode	How to display the image
flip	If the image is flipped
mirror	If the image is mirrored
center	If the image is centered

The *file* option specifies the name of a file that contains the image to be displayed in a graphic format that is supported by the package. The *Image* command describes all of the file formats that are currently supported.

The *colormode* option may be either *mono* or *rgb*. By default, the *colormode* value is *rgb* and the image is displayed in full color. If *colormode* is specified as *mono*, then the image is displayed in a gray scale format.

The *flip*, *mirror* and *center* options can be used to flip, mirror or center the image. By default, these options are *false*. The following command could be used to display and image on a canvas:

```
$w create image −file images/ashley.gif −x 100 −y 100 −colormode mono
```

Here the image is being placed on the canvas at location (100,100) and will be displayed as a gray scale image. Note that the image will be clipped to the default item dimensions, so if you want to see the entire image, you need to create the item large enough to hold the entire image, or implement some scrolling or panning mechanism.

15.4.5 Canvas Line Items

The format of the command that creates a line is:

```
$w create line options
```

where *$w* is the canvas to be used and *options* is the list of option and value pairs that are used to configure the line. In addition to the list of standard canvas item options, the line supports the following item specific options:

from	Start location of the line
to	End location of the line
x1	X location of the start of the line
y1	Y location of the line start
x2	X location of the line end
y2	Y location of the line end

A line can be created using a command as follows:

```
$w create line −from 20,20 −to 145,90 −color red −linewidth 4 −linestyle dash
```

which will draw a red dashed line between the 2 end points. Alternatively, the same line could be specified as:

```
$w create line −x1 20 −y1 20 0x2 145 −y2 90 −color red −linewidth 4 −linestyle dash
```

15.4.6 Canvas Polygon Items

Polygons are closed convex objects formed by joining a list of vertices together with line segments. There are regular polygons and irregular polygons, the former being objects such as the square, the octagon and the pentagon, the latter being objects whose sides are not all of equal length.

The format of the command that creates a polygon is:

```
$w create polygon options
```

where *$w* is the path name of the canvas and *options* is the list of option and value pairs that are used to configure the polygon item. In addition to the set of standard canvas item options, the polygon supports the following item specific options:

fill	If the item is filled
sides	Number of sides for a regular polygon.

If the *sides* option is specified, then the resulting object will be a regular polygon with the specified number of sides. Otherwise, the vertices of the polygon are specified in a vertex list. In the latter case, the number of vertices specified will be used to draw a closed polygon by joining the last vertex to the first vertex. If the vertices do not actually represent a convex closed polygon, then the fill operation can produce unpredictable results.

For regular polygons, the resulting object is drawn inside of the bounding box defined by the current *origin* end *extent* values. For example, a hexagon might be drawn using the command:

```
$w create polygon −extent 200,200 −sides 6
```

15.4.7 Canvas Point Items

The point is a mark at a single location on the canvas that covers 1 pixel on the display screen. Points use a subset of the standard canvas item options. The following command creates a point:

```
$w create point −x 100 −y 100 −color green
```

which will color the pixel at (100,100) on the canvas green.

15.4.8 Canvas Quadrangle Items

A quadrangle is a closed convex irregular polygon of 4 sides. The format of the command used to create a quadrangle is:

```
$w create quadrangle options
```

where *$w* is the canvas to be used and *options* is the list of option and value pairs used to configure the item. In addition to the set of standard canvas item options, the following widget specific option is supported:

fill	If the quadrangle is filled

Quadrangles are created using a command of the following form:

```
$w create quadrangle −vertexlist 10,10,100,25,125,90,40,20
```

The 4 pairs of coordinates define the corners of the quadrangle.

15.4.9 Canvas Rectangle Items

A rectangle is a regular closed convex polygon of 4 sides. Like the *quadrangle* item, int addition to the list of standard canvas item options, the rectangle supports only 1 item specific option, the *fill* option which is *true* if the rectangle is to be filled. The following command us used to create a rectangle:

```
$w create rectangle −width 200 −height 100 −fill yes −color purple
```

Alternatively, the actual vertices could be specified as in the *quadrangle* item.

15.4.10 Canvas Text Items

The format of the command used to create text items is:

```
$w create text options
```

where *$w* is the canvas to be used and *options* is the list of option and value pairs used to configure the text item. In addition to the set of standard canvas item options, the *text* item supports the following item specific options:

text	The text to be displayed
textfont	The font to use in displaying the text
textsize	The size of the text font to be used.

Here is an example of using the *text* item to display some text on the canvas:

```
$w create text −text "Hello, world" −textfont times,italic,bold −textsize 20 −color tan
```

The *textfont* option value is a comma separated list of items that specify the characteristics of the font to be used. By default, the *textfont* option value is helv and the *textsize* option value is 12.

The following fonts are supplied as part of the Fltk extension package under the Linux and other UNIX operating systems:

helvetica
courier
times
symbol

system
dingbats

Font descriptions can be qualified by appending the following qualifier strings:

bold	Set the text to a bold font
italic	Set the text to an italic font

15.4.11 Canvas Triangle Items

Triangles are closed convex polygons of 3 sides. They are created by specifying the vertices of the triangle in a vertex list, or, they can be created using the polygon item creation command. Amongst the list of standard canvas item options, *triangle* items support the *fill* option for the creation of filled triangles. For example, the command:

```
$w create triangle −vertexlist 100,100,200,200,0,200 −fill yes −fillcolor yellow
```

will produce a yellow triangle on the canvas identified by the contents of *$w* variable.

15.5 The Canvas delete function command

The delete function command is used to delete items from the canvas. The format of the command is:

```
$w delete list
```

where *$w* is the canvas to use and *list* is an optional list of canvas item identifiers to be deleted. If no identifiers are specified, all of the items on the canvas are deleted. Any non−existent items are ignored.

For example, to clear the canvas, use the command:

```
$w delete
```

while the item identified as *circle9* can be removed using the command:

```
$w delete circle9
```

When a canvas is destroyed, either through the use of the *Destroy* command, the destruction of a parent, a call to *Exit* or through the use of the Tcl *rename* command, all of the items in the canvas are automatically destroyed.

15.6 The Canvas itembind function command

Event handlers for mouse and keyboard events, and if so desired, user defined events, can be bound to the items on a canvas using the *itembind* function command. The format of the *itembind* function command is:

```
$w itembind id event script
```

where *$w* is the canvas path name, *id* is the canvas item identifier to use, *event* is the name of the event to use, and *script* is the event handler script to be executed when the event occurs. To bind an event, the specified *id* must refer to an existing canvas item.

The id parameter can take the following special forms in addition to being an item identifier:

all	Bind an event to every item on the canvas
withtag	Bind an event to every item with a specified tag or tags
withouttag	Bind an event to every item that does not have a tag or tags

The *all* keyword is convenient for assigning global event handlers to every item. Each canvas item will have an associated tag list which consists of the *item class* tag and any user applied tags. The *class tag* is a string that describes the item. All *circles*, for instance, will have the class tag of circle. A command of the form:

```
$w itembind withtag circle <Motion> { puts %w %x %y }
```

could be used to display the location of the mouse when it moves over any *circle* item on the canvas. Similarly, the *withouttag* form could be used to apply the event handler to all canvas items that are not circles.

The *event* can be any of the standard mouse or keyboard events as described for the *Bind* command, or, a user defined event that could be invoked using the *Signal* command. If no *event* name is specified, the result returned by this command is a list of the events currently bound to the item.

The *script* is a Tcl script that is first expanded to fill in any substitutable parameters and then evaluated. If no script is specified, the result of this commend is to delete any event handlers associated with the event. If the script is prepended with a plus sign, then the script is appended to the current event handler script for the specified event, otherwise it replaces the script for the event. The substitutable parameters available are those that can be used with the event mechanism implemented for binding events to widgets as described in the documentation for the *Bind* command.

15.7 The Canvas itemcget function command

The format of the *itemcget* command is:

```
$w itemcget id −name1 ...−namen
```

where *$w* is the canvas, *id* is the identifier of an item on the canvas, and the *names* are the names of item configurable options for which the item is being queried. The result returned by this command is a list of the current values of the configurable parameters for the specified canvas item.

If no names are specified, the result of this command is a list of the names of the configurable options for the item.

15.8 The Canvas itemconfigure function command

The *itemconfigure* command is used to change the values of the configurable options of items on a canvas. The format of the command is:

```
$w itemconfigure id −name value ...
```

where *$w* is the canvas path name, *id* is the name of the canvas item to configure, and the *name* and *value* pairs define the new values of the configurable options of the item.

If no name and value pairs are specified, the result of this command is the list of configurable options for the specified canvas item. Missing option values result in an error message being returned.

For example, the following command could be used to set the *activefillcolor* of a circle with the identifier circle3:

```
$w itemconfigure circle3 −activefillcolor red
```

15.9 The Canvas itemlist function command

The *itemlist* command is used to list the identifiers of the items on a canvas according to some specific criteria. The format of the command is:

```
$w itemlist how parameters
```

where *$w* is the canvas path name, *how* is a keyword that specifies the selection criteria for canvas widgets, and *parameters* are optional parameters that depend on the selection criteria keyword. If no selection criterion is specified, then the result returned by this command is a list the available criteria keywords.

The list of selection criteria is:

all	List all items
withtag	List all items with a specified tag or tags
withouttag	List all items without a specified tag or tags
disabled	List all disabled items
hidden	List all hidden items
visible	List all visible items
type	List all items of a specified type

For the *tag* related criteria, the parameter is a comma separated list of tags to look for with the items. For the *type* criterion, the parameter is the name of an item type, such as *triangle* or *curve*. To produce a list of all items with the tags *circle* and *blue* use a command of the form:

```
set blue_circles [$w itemlist withtag circle,blue]
```

15.10 Canvas initialization from text files

The Canvas widget provides for the storage of its contents in text format to a file and the loading of previously saved canvas drawings from a text file.

15.10.1 The Canvas load function command

The *load* function command is used to load canvas items from a file that was created in a format compatible with that produced by the *save* command. The format of the command is:

```
$w load path
```

where *$w* is the name of the canvas and *path* is the name of the file to load. This command will return an error if the file does not exist.

15.10.2 The Canvas save function command

The *save* function command writes a file that contains all of the widgets in the canvas in a form that can be loaded by the l*oad* command. The format of the command is:

```
$w save path
```

where $w is the canvas path name and *path* is the name of the file to be used.

16 Center − Center a widget on the screen

The center command will center a widget on the screen. The format of the command line is:

```
Center path −width width −height height
```

where *path* is the path name of the window to center, and *width* and *height* are optional values that describe the width and height of the widget being centered.

If the *width* and *height* parameters are not specified, the command will use the values returned by the widget. Because of the way the FLTK tool kit works, these value may not be the values that the widget will have when it is drawn on the screen. This is due to the fact that child widgets can resize their parent widgets, a feature of the tool kit.

The value returned by this command is the path name of the widget being centered. For example, the command:

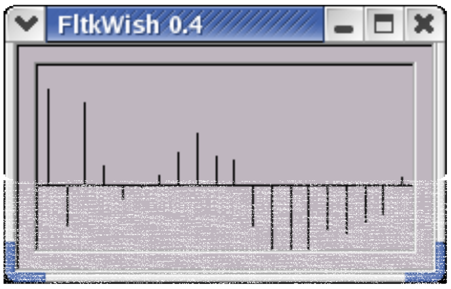
```
Show [Center top]
```

might be used to center a widget whose path name is top.

It is possible to apply this command to any widget, however, operations on widgets that are not *Toplevel* widgets may produce undesirable results.

17 Chart − Create a chart widget

The *Chart* command is used to create a number of different types of charts that can be used for the rapid plotting and display of data. Here is an example of a chart using the *spike* style that plots a series of data points:



The format of the command is:

```
Chart path options
```

where *path* is a valid widget path and *options* are option and value pairs used to configure the widget. In addition to the set of standard widget options, the *Chart* command supports the following widget specific options:

autosize	Automatically scale the plotted data
chartstyle	Specify the type of chart
maxsize	Specify the maximum number of points to plot
size	Query the number of points

By default, *autosize* is *true*, and the plotted data is automatically scaled to the current range of the points in the data buffer for the chart.

The *maxsize* option can be used to set the number of points that the chart will hold for plotting. When this limit is reached, additional plot points cause the removal of those points that are the oldest in the chart data buffer list. This feature is very handy for developing monitoring applications that look at a time window of data points. By default, there is no limit, beyond that posed by available memory, on the number of points.

The *chartstyle* option is used to specify the type of chart to produce. The *Chart* widget provides the following types of charts:

bar	A bar char
filled	A filled line chart
horbar	A horizontal bar chart
line	A line chart
pie	A pie chart
specialpie	Another pie chart with a sector emphasized
spike	Spikes instead of bars.

By default, the *chartstyle* is line.

The *size* option is only used to query the number of points in the chart data buffer. It is used with the widget *cget* command as follows:

```
$w cget −size
```

where *$w* is the widget path name of the chart. The value returned is the number of points currently in the data buffer.

17.1 Chart Widget Function Commands

In addition to the standard widget commands *configure* and *cget*, the *Chart* widget provides the following widget specific commands:

add	Add data points to the chart
bounds	Specify the range of values to use in scaling
clear	Clear the chart data buffer
insert	Insert a data point
replace	Replace a data point

The general format of the widget commands is:

```
$w function ...options ...
```

where *$w* is the path name of the *Chart* widget, *function* is the name of the widget function, and *options* are option and value pairs that are used by the widget commands to manage the data buffer and control the appearance of the chart.

The functions that manage the data buffer support the following options:

color	Specify the color to use for plotting the points
label	Specify a label for the data points
position	Specify the location for the points
values	Specify one or more point values.

Each point in the data buffer has the properties described by the list of options for the functions. Individual function commands make use of the options according to their purpose.

The Chart add function command

The format of the add function command is:

```
$w add −color color −label label −values values
```

where *color* is the name of the color to use when plotting the point values, *label* is a label to use, and *values* is a comma separated list of point values.

This command adds the specified list of point values to the end of the current list of points in the data buffer. The *color* and *label* values specified apply to all of the points added. By default, *color* is *black* and *label* is an empty string. At least 1 value must be supplied, otherwise, this command generates an error message.

For example, the following command will add 3 points to a chart and draw them in green:

```
$w add −values "10.2,−3.7,45.1" −color green
```

The displayed result will depend on the type of chart being used.

17.1.1 The Chart bounds function command

The format of the bounds function command is:

```
$w bounds lower upper
```

where *lowe*r and *upper* are 2 numbers that specify the range that is to be used to scale the plotted points in the data buffer. If no parameters are given, the result of this command is a string that contains the current values of the *lower* and *upper* bounds in use.

For example, to scale data between −50 and 50 a command of the form:

```
$w bounds −50.0 50.0
```

could be used.

17.1.2 The Chart clear function command

The *clear* function empties the data buffer and erases the chart. The format of the command is:

```
$w clear
```

where *$w* is the path name of the *Chart* widget to be cleared.

17.1.3 The Chart insert function command

The *insert* function can be used to insert data into a chart data buffer at a specified location. The format of the *insert* function command is:

```
$w insert −position position −color color −label label −values values
```

where *$w* is the path name of the chart widget, *position* is a number that specifies where to put the point values, *color* and *label* specify the plotting color and label properties for the points, and *values* is a comma separated list of point values to insert.

At least 1 point value must be provided. The specified *position* must be within the range of the number of points currently in the chart data buffer. Chart points are numbered from 0 through size − 1, where *size* is the number of points in the chart data buffer.

Suppose a chart has 20 points in its data buffer. The following command could be used to add 2 points at location 14 and plot them in red:

```
$w insert −position 14 −color red −values 12.35,32.807
```

17.1.4 The Chart replace function command

The *replace* function command can be used to replace a single point at a location in the chart data buffer with a new point value and point attributes. The format of the command is:

```
$w replace −position position −color color −label label −values value
```

where *$w* is the path name of the chart widget, *position* is a location of a point in the chart data buffer, *color* is a color for use in plotting the point, *label* is a label for the point, and *value* is a new value for the point. The new *value* and the *position* must be supplied.

For a chart containing 100 points, the following command could be used to correct the 35th point in the data buffer:

```
$w replace −position 34 −color blue −value −36.4 −label Error
```

Note that the 35th point has a position of 34. This command would change the plotting color to *blue* and add the label *Error* to the point.

18 CheckEvents − Check for pending events

The *CheckEvents* command is used to initiate processing of the Fltk tool kit's event loop. The Fltk extension automatically polls the toolkit event queue, but, because of the way Tcl is constructed, there may be applications for which explicit polling of the event queue is required.

The format of the CheckEvents command is:

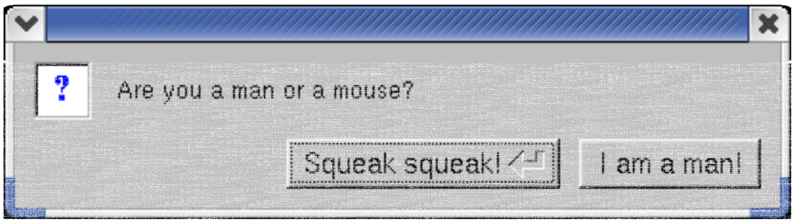
```
CheckEvents ?−active? ??−delay? ?value??
```

CheckEvents will always poll the toolkit event queue. Without any options, *CheckEvents* returns nothing. If *active* is specified, the result is either *true* if there are any active widgets, or *false* if there are no active widgets.

The rate at which the toolkit extension polls the toolkit event loop can be queried or set using the *delay* option. If no value is specified, then the command returns the current delay time in milli−seconds. If *value* is specified, the polling rate is set to the new value. Note that setting a large delay will make responsiveness of the toolkit widgets sluggish!

19 Choose − Choose from some options

The *Choose* command will present the user with a dialog that asks that a choice be made between three options. There are 2 options, the second one being the default option. Note that pressing the *Enter* key will choose the default option, in the following example, "Squeak squeak!".



The format of the command is:

```
Choose question option1 option2
```

where the *question* is the prompt to be used, and the *options* are strings that describe the options available. The value returned is a number that represents the chosen option.

20 Combobox − Create a combobox widget

The *Combobox* command creates a widget that has the features and functionality of a combobox, comprised of a drop down list and a fixed region that displays the current selection. The format of the command is:

```
Combobox path options
```

where *path* is the path name of the widget to be created and *options* are to list of option and value pairs that are used to configure the widget. In addition to the *standard set of widget options*, the widget supports the following widget specific options:

value	The value of the current selection
color	The color used for the widget items
textsize	Size of the font
textfont	Name of the font
length	Length of the selection list
title	Title for the combobox
displayheight	How many lines to display

This particular *Combobox* implementation provides for automatic scrolling of the drop down box and easy management of the selection list. Here is an example of a *Combobox* command:

```
Combobox t.c −variable Choice −command HandleChoice
```

Here the variable *Choice* will be updated whenever the user changes the selection in the *Combobox*. The Tcl procedure *HandleChoice* will be executed whenever a change occurs.

20.1 Widget Specific Commands

In addition to the standard widget commands *configure* and *cget*, the *Combobox* supports the following widget specific commands:

add	Add items to the list
clear	Clear the item list
delete	Delete an item from the list
find	Find an item in the list
insert	Inset an item into the list
load	Load the list from a file
replace	Replace an item in the list
selection	Set the selection in the list
sort	Sort the items in the list

20.1.1 add − Add items to the list

The *add* command is used to add items to the selection list used by the *Combobox*. The format of the command is:

```
t.c add item ?item? ...
```

where *t.c* is the path name of the widget to use and the *items* are strings to be added to the *Combobox* selection list. Any number of items may be specified.

20.1.2 clear − Clear the list

The *clear* command is used to empty the Combobox selection list. The format of the command is:

```
t.c clear
```

where *t.c* is the path name of the widget to use.

20.1.3 delete − Delete items from the list

The *delete* command will remove items from the list. The format of the command is:

```
t.c delete item ?item? ?item? ...
```

where *t.c* is the path name of the *Combobox* widget and the *items* are the ordinals of the items in the list. Items are numbered from 0 through n−1 where n is the number of items in the list.

20.1.4 find − Find an item in the list

The *find* command is used to determine the ordinal of an item in the list. The format of the *find* command is:

```
t.c find item
```

where *t.c* is the path name of the *Combobox* to use and *item* is a string to find in the list. If the string is found, the value returned by this command is the ordinal of the item in the list. If the item is not found, the value returned is −1.

20.1.5 insert − Insert an item into the list

The *insert* command is used to insert an item into the list. The format of the command is:

```
t.c insert where item
```

where *t.c* is the path name of the *Combobox* to use, *where* is the ordinal in the list to place the item after, and *item* is the item to insert. The value of *where* can be *end*, in which case the insert command behaves like the *add* command.

20.1.6 load − Load the list from a file

The *load* command can be used to load a *Combobox* list from the contents of a file. Each line in the file is treated as an item to be added to the list. The format of the command is:

```
t.c load name
```

where *t.c* is the path name of the *Combobox* and *name* is the name of the file to use. If the file is not found, the value returned by this command is an error message, otherwise the command returns the number of items added to the list.

20.1.7 replace − Replace the contents of an item

The *replace* command is used to change the contents of an item in the list. The format of the command is:

```
t.c replace where with
```

where *t.c* is the path name of the *Combobox* to use, *where* is the ordinal of the item to replace and *with* is the value to use for the new item contents.

20.1.8 sort − Sort the list contents

This command causes the current list contents to be sorted in alphabetical order. The format of the command is:

```
t.c sort
```

where *t.c* is the path name of the *Combobox* to use.

20.1.9 selection − Query or set the current selection

The ordinal of the current selection can be set or queried using the *selection* command. The format of the command is :

```
t.c selection ?value?
```

where *t.c* is the path name of the *Combobox* to use. If the *value* parameter is specified, and it is a valid ordinal, then the current selection is set to the item whose ordinal matches the *value*. If the *value* parameter is not specified, the value returned by this command is the ordinal of the current selection.

21 ChooseColor − Choose a color

The *ChooseColor* command presents the user with a color selection dialog that can be used to select colors. The format of the command is:

```
ChooseColor  title
```

where *title* is a title for the dialog window. If no *title* is supplied a default title is used. The value returned by this command is a string of 3 numbers separated by commas that represents the red, green and blue values of the chosen color. If the user cancels the dialog then an empty string is returned.

22 ColorName − Get the name of a color specification

The *ColorName* command can be used to find the name of the color that matches a color specification. The format of the command is:

```
ColorName spec
```

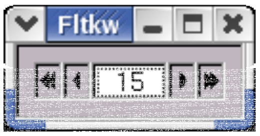
where spec is a color specification in the form r,g,b, where r, g, and b are the red, green and blue color component values of the color. The value returned by this command is the name of the closest color. For example, the command:

```
ColorName 255,0,0
```

will return *red*.

23 Counter − Create a counter widget

A *Counter* is a widget that can be used to control the value of a single number. The *LabeledCounter* widget is also available and is a mega−widget that has the Counter and a Label widget.



The format of the command is:

```
Counter path options
```

where *path* is the path name of the widget to be created and *options* is the list of option and value pairs used to configure the widget. In addition to the set of *standard widget options*, the *Counter* supports the following widget specific options:

value	The current value of the widget
step	The increment value
min	The minimum value
max	The maximum value
faststep	The increment for fast steps
counterstyle	The type of counter

By default, the values of *min* and *max* are *0* and *100*, and the step value is *1*. The default value for the *faststep* option is *10*. The *counterstyle* option can be either *simple* or *normal*. A *simple* Counter has no *faststep* buttons. The default *counterstyle* is *normal*.

Here is a command that uses a *Counter* to change the value of a Tcl variable named *MyCounterVar*:

```
Counter root.c −x 20 −y 20 −variable MyCounterVar −value 0 −min −20 −max 20 −faststep 5
```

24 Destroy − Destroy one or more widgets

The *Destroy* command is the usual method of destroying widgets managed by the Fltk extension. The format of the command is:

```
Destroy name ?...?
```

where *name* is the path name of a widget to be destroyed. The command destroys all of the widgets specified on the command line.

This command always returns the Tcl success indication. For widgets that don't exist, the command does nothing and remains silent. Destroying a container widget will destroy all of the children of the container widget. A typical use of the *Destroy* command is to get rid of all of the widgets in a tree by destroying the root widget. For example, if a widget has been created using the command:

```
Button t.t −text "Dismiss" −command Exit
```

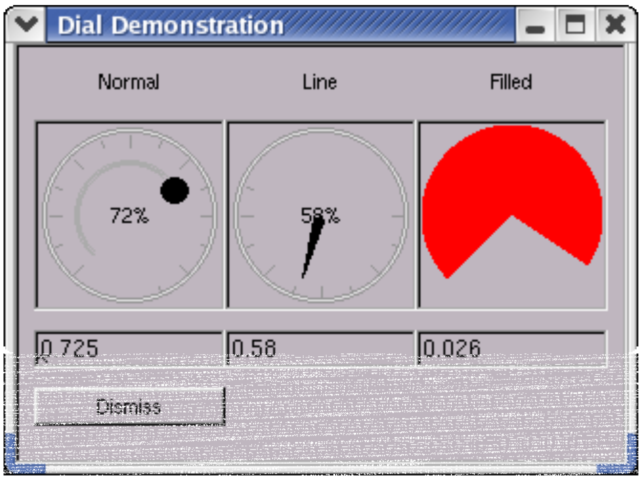
then the command:

```
Destroy t
```

will destroy the tree root named *t* and all of its children, which includes the Button named *t.t*.

25 Dial − Create a dial widget

A *Dial* is a widget that presents a circular object with an indicator that can be dragged about to change the value of the dial.



The format of the *Dial* command is:

```
Dial path options
```

where *path* is the path name of the widget to be created and *options* is the list of option and value pairs that are used to configure the widget. In addition to the set of standard options the *Dial* widget supports the following widget specific options:

value	The current value of the dial
step	The increment amount to use
min	The minimum angle of the dial
max	The maximum angle of the dial
dialstyle	The style of the dial object

The *dialstyle* option can have the values *normal*, *line* and *filled*. By default, the *dialstyle* is *normal* and the dial looks like a circle with a moveable dot that serves as the indicator. A *line* dial has a line as its indicator, and a *filled* dial uses a sector of the circle to indicate its value.

The range of the angles that the *Dial* can move through is set using the *min* and *max* options. By default the range is *45* to *315*, and the *step* is *1*. These numbers are angles that define the limits of the *Dial* on a circle. By querying the *value* option an number between 0 and 1.0 is returned that represents the position of the indicator on the Dial.

Note that when using the *step* option, the step value refers to the amount of the range of the *value* returned by the *Dial*. Since this range is from 0.0 through 1.0, the value of the *step* should normally be set to a value that is a small percentage of this range. The default *step* value is 0.001.

Here is a *Dial* that updates a Tcl variable:

```
set d [Dial root.dial −variable MyDialVariable −variablecommand { puts $MyDialVariable }
```

The *variablecommand* option is adding a script to the widget event handler list that is executed whenever the *Dial* changes its *value*. In this case, the script just prints the current value of the *Dial*. The *variable* option is being used to cause the current *value* of the *Dial* to be loaded into the Tcl variable *MyDialVariable*. If *MyDialVariable* is changed by by the script, the position of the *Dial* is automatically updated to reflect the new *value*.

25 Dial − Create a dial widget

An alternative to the *Dial* widget is the *Knob* widget. The *Knob* widget is a *Dial* rendered using OpenGL.

26 Drawing − Create a Turtle Graphics drawing widget

The *Drawing* command constructs a widget that can be used to draw diagrams using a version of the Turtle Graphics drawing language. The format of the command is:

```
Drawing path options
```

where *path* is the path name of the widget to be created and *options* is the list of option and value pairs used to configure the widget. In addition to the list of *standard widget options* , the widget supports the following widget specific option:

value	A Turtle Graphics drawing script.

Drawings are done on the *Drawing* widget by setting the *value* option to a string that is a set of Turtle Graphics commands. When a new *value* string is supplied, the widget parses the string and adds drawing elements to its list of drawing elements according to the commands in the string. When the widget is rendered on the display it is done by processing the drawing list. This means that, unless the widget receives a *cs* command to clear its drawing list, new command strings are appended to the current drawing.

26.1 The Turtle Graphics Drawing Language

Historically, the Turtle Graphics drawing language was the implementation of a simplified series of commands that provided control of a plotter pen simulation to draw diagrams on a canvas. The language consists of a string of blank separated command and parameter tokens that are processed by an interpreter. The command tokens modify the position, color and state of a drawing pen, while the parameters control how the state of the drawing engine should change. the turtle is the cursor, which, in some implementations aimed at the pre−school market, was represented by a stylized drawing of a turtle which proceeded from location to location across the drawing canvas, leaving a trail of marks as it moved along.

The Turtle Graphics language is a low level language that is unforgiving of errors, but has a compact form that makes it useful for drawing rather complex line drawings. As implemented in this widget, some useful enhancements have been added to the language for drawing text and in order to make certain programming constructs a little easier to write.

A command is a token of 2 characters that may be followed by any number of parameters, or no parameters, according to the needs of the command. In the current implementation all of the commands are aliased to one or more command name aliases that are easier to read and remember than the simple tokens. Command scripts may use any combination of the tokens or the full command name aliases. in the following list of ommands, the various aliases are seperated by a colon character, so, for example, the commands *al*, *align* and *justify* all mean the same thing. Here is a list of the commands:

al:align:justify	Set the text alignment
ar:arc	Draw an arc
bd:bounds	Set the current bounding rectangle
bg:setbg:background	Set the background color
bk:backward:−	Move backwards
cr:circle	Draw a circle
cs:clear:erase	Erase the current drawing and reset the drawing engine
cl:clearscreen	Erase the current drawing and set the background color of the widget
dl:drawline	Draw a line
fd:forward:+	Move forwards
fl:fill	Set the fill state
fs:fontsize	Set the font size

ft:font	Set the text font
hi:hide:hideitem	Hide items with specified tags
hl:help:?	Display some help information
hm:home:!	Move the cursor to the home position
ht:hideturtle	Don't display the current cursor position
im:image	Draw an image at the current position
li:list:listitems	List the current drawing list
ls:style	Set the line style to use
lt:leftturn	Change the current direction counterclockwise
pc:setpc:foreground	Set the drawing color
pd:pendown	Lower the pen (Marking will occur when it is down)
pp:pop:)	Pop the current pen state
ps:push:(	Push the current pen state
pu:penup	Raise the pen (No marks appear when it is up)
pt:point	Draw a point
rc:rect:rectangle	Draw a rectangle
rp:repeat	Repeat a set of commands
rt:rightturn	Change the current direction clockwise
sh:h:seth	Set the current direction
si:show:showitems	Show hidden items with specified tags
sp:setpos:location:=	Set the current cursor position
st:showturtle	Display the current cursor position
sx:x:setx	Set the horizontal position of the cursor
sy:y:sety	Set the vertical position of the cursor
tg:tag:taglist	Specify the tags for draw items
th:thickness	Set the line thickness
tr:trace	Set the command trace state
tx:text:write	Set the text to display
//:/*:*/:#:rem	Quote a comment

Here is a Turtle Graphics script that draws a circle:

```
cs hm pc red th 2 pd cr 10
```

This script will draw a circle of radius 10 pixels using a double width line centered at the current home location of the drawing widget.

26.2 Drawing Concepts

When the drawing engine is initialized, either by constructing a new widget or by using the *cs* command, the drawing engine is set to the state where the cursor is at the home location and the current direction is pointing up towards the top of the Drawing widget. The home location is set to be the center of the Drawing widget, based on its current size. The line width is set to *1*, the line style is set to *solid*, the text font is set to *helvetica* with a point size of *10*, and the drawing color is set to *black*. The background color is, by default, *clear*, which means that whatever color the widget background has will be used as the drawing background color.

Each drawing command will, if the pen is down, cause a mark to be made in along the path of the cursor movement. For example, the command:

```
cs pd fd 20
```

would move the cursor along the current direction by 20 pixels, leaving a black mark on the Drawing widget's client area. After the command has been executed, the current cursor location will be at the end of the newly drawn mark. The following command script would draw a box:

```
fd 20 rt 90 fd 20 rt 90 fd 20 rt 90 fd 20
```

or, more compactly,

```
rp 4 "fd 20 rt 90"
```

In the latter case, the *rp* or *repea*t command is used to tell the drawing engine to execute the quoted command 4 times. At the end of the above sequence of commands, the cursor will be back exactly where it started from, the initial *home* position.

The *fd* command takes only 1 parameter, the number of pixels to move in the current direction. The *rp* command takes 2 parameters, the number of times to repeat the command set, and the command set itself. Tokens in the Turtle Graphics command language can be either a string of characters delimited by spaces, or a character string in quotation marks that may contain blanks.

Where parameters take numeric values, the parameters may be prefixed by a unary arithmetic operator that has the effect of applying the numeric value of the parameter as a relative offset from the current value of the drawing engine state item. For example, the command:

```
sp 10 30
```

will move the current location of the cursor to the window relative widget coordinate (10,30), while the command:

```
sp +10 +30
```

will move the cursor to a location that is at relative offsets +10 and +30 from the current cursor location.

Commands, like the *sp* command, that position the drawing pen can take parameters that are keywords that refer to positions on the drawing surface with respect to the current boundaries of that surface. The horizontal position of the pen can be specified as *left*, *right* or *centered*, and the vertical position of the pen can be specified as either *top*, *bottom* or *centered*. The syntax:

```
sp center top
```

for example, will position the pen along the top boundary of the drawing surface at the center of the horizontal dimension. Optionally, this syntax can include offsets from the specified location using commands of the form:

```
sp right−20 top+20
```

This latter command will put the pen at a position 20 pixels to the right of the right hand boundary and 20 pixels below the top boundary of the drawing surface. There is an additional syntax that may be use to aid the layout of text in the drawing using the positional keywords *center*, *left*, *right*, *top* and *bottom*. If the keyword *x* is added, the current vertical position of the turtle is preserved and the remaing keywords is applied. Similarly, if the keyword *y* is added, the current horizontal position of the turtle is preserved and the remainng keywords are applied. For example, the command:

```
sy +10 al center,x tx "Some Text"
```

would center the text at the current vertical displacement..

26.3 Turtle Graphics Command Reference

26.3.1 al − Set the text alignment

The *al* command can be used to specify an alignment for the text drawn in the Drawing widget. The use of the *al* command is somewhat problematic in that it can defeat the basic concept of the Turtle Graphics language by manipulation of the pen position in ways that make predicting the termnal location difficult. Nevertheless, the *al* command provides some convenience when placing text in centered positions, or aligned in the corners of the widget.

The al command takes 1 parameter that is a text alignment specification in the format supported by the extension package. For example, the command:

```
al centered,top
```

will cause text to be displayed *centered* horizontally at the *top* of the widget client area. By default, all text alignments imply the *inside* property, so this command can not be used to put text outside of the widget client area. To turn off the current alignment, use:

```
al none
```

It is a good idea to enclose aligned text in a push and pop command sequence to prevent alignment actions from leaving the current cursor location in an unknown position. For example:

```
( al bottom,right pc blue tx "I,m blue...o so blue..." )
```

will leave the current pen position back where it was before the text was written.

26.3.2 ar − Draw an arc

The *ar* command takes 3 parameters, the *radius*, the *start* angle in degrees and the *end* angle in degrees. The *angles* are relative to a horizontal line proceeding to the right from the current cursor location. When the arc is drawn, if the state of *fill* is *true* a pie sector will be drawn, filled with the current background color. Otherwise and arc is drawn using the current pen color and thickness.

For example, the command:

```
ar 10 0 180
```

will draw a semi−circular arc, or half of a pie, depending on the *fill* state.

26.3.3 bd − Set the current drawing window limits

The *bd* command is used to specify the dimension of a rectangle that defines the current bounding rectangle of the drawing window. By default, the bounding rectangle is set to the client region of the *Drawing* widget. Since the coordinates of the origin of the drawing are set to the center of the widget, the bounding rectangle will be from (−ClientX()/2,−ClientY()/2) with dimensions ClientWidth() by ClientHeight(). Using the *bd* command, the bounding rectangle can be set to a specified width and height with its upper left hand corner at the current pen position. For example, the command:

```
bd 20 30
```

would set the bounding rectangle to be from the current pen position with width 20 and height 30. For a specific bounding rectangle, the text alignment spcifications are applied with respect to that rectangle. This allows for convenient text positioning with command sequences such as:

```
( sp −10 −10 bd 20 30 al left tx "Hello" }
```

which would align the text at location (−10,5) in the drawing with respect to the drawing origin.

26.3.4 bg − Set the background color

The *bg* command specifies the current background color. Any of the color format specifications supported by the extension package can be used. For example, the command:

```
bg aquamarine
```

will set the current background color to *aquamarine* .

26.3.5 bk − Move backwards

the *bk* command will move the cursor backwards along the current drawing direction by a specified number of pixels. If the pen is down a mark is produced along the track of the motion. For example, the command:

```
bk 40
```

will move the cursor backwards along the current drawing direction by 40 pixels.

26.3.6 cl − Clear the drawing and set the background color

The *cl* command clears the current drawing list and fills the drawing surface with the specified background color. Its behaviour is similar to the cs command except that a color can be specified. For example, the command:

```
cl red
```

will empty the draw list, reset the drawing engine and paint the widget background *red*.

26.3.7 cr − Draw a circle

The *cr* command will draw a circle centered at the current cursor position of the specified radius using the current pen color and thickness. For example, the command:

```
cr 20
```

draws a circle of radius 20 pixels. If the *fill* state is true then the circle will be filled with the current background color.

26.3.8 cs − Clear the drawing

The *cs* command takes no parameters. It deletes all current drawing items and resets the drawing machine state to its default initialization state. It is a good idea to use the *cs* command when updating the drawing on a widget. If the *cs* command is not used and the value of the widget is changed, the new drawing ail be added to the existing drawing.

26.3.9 dl − Draw a line

The *dl* command will draw a line, using the current pen color and pen width, from the current cursor position to the specified location. The command takes 2 parameters, the target x and y locations. These locations can be relative offsets or absolute coordinates of the end point of the line. For example, the command:

```
dl +10 −4
```

will draw a mark from the current cursor location to a position at relative offset (+10,−4) with respect to the start of the line.

26.3.10 fd − Move forward

The *fd* command is used to move the cursor position forward along the current drawing direction. If the pen is down a mark will be drawn along the track of the motion using the current pen color and pen thickness. For example, the command:

```
fd 45
```

would produce a mark 40 pixels long in the current drawing direction.

26.3.11 fl − Set the fill state

The *fl* command turns on or off the fill state of the drawing engine. For fillable objects, such as *circles* , *rectangles* and *pie* sectors, if the fill state is *true* then the object is filled with the current background color. If the fill state is *false*, then the object is not filled. By default, the fill state is *false* and objects are not filled. The following command:

```
fl on
```

will set the fill state to *true*. Any representation of a boolean value can be used to specify the fill state.

26.3.12 fs − set the size of the current font

The *fs* command is used to specify the size of the current text font. By default, the text font is set to *helvetica* with a *10* point font. The following command will change the font to size *14*:

```
fs 14
```

26.3.13 ft − Set the current text font

The ft command is used to specify the current font for text items. By default, the font is set to helvetica with font size 10. The font specification can be any specification supported by the extension package. or example, the command:

```
ft helv,bold,italic
```

could be used to set the font to a bold and italic version of the helvetica font.

26.3.14 hi − Hide draw items

The *hi* command is used to specify the set of draw items in the draw list to mark as hidden. The command takes as a parameter a list of comma seperated strings that are used to match the items in the draw list. If an item in the draw list has one of the strings as one of its tags, the item is hidden: For example, the command:

```
hi red,green,blue
```

would hide all items which had the tag red or *green* or *blue*.

26.3.15 hl − Display help information

The *hl* command displays a list of the Turtle Graphics commands on the current command console. It takes no parameters.

26.3.16 hm − Move the cursor to the home position

The *hm* command moves the cursor to the current *home* position. The *home* position is at the current center of the *Drawing* widget's client area. On resize of the widget, the *home* position is adjusted and the drawing is redrawn. Care should be exercised when using absolute locations in drawings in widgets that may be resized, as the results could be not what is expected.

26.3.17 ht − Hide the cursor

If the cursor position is visible, it is hidden

26.3.18 im − Draw an image

the im command can be used to place an image on the drawing. The image is positiond such that its upper left hand corner is at the current pen position. For example, the command:

```
im images/weather/Cloudy.bmp
```

would draw the image contained in the specified file at the current pen position. The current bounding rectangle will be used to clip and justify the image according to the current alignment specification.

26.3.19 li − List the current draw list

The *draw list* is the list of drawing objects that is used to create the current drawing. The *li* command will display the current *draw list* on the current console. It takes as a parameter a comma seperated list of the tags that are to be matched to identify the items to be listed. For example, the command:

```
li circle,line
```

will produce a list of the items in the draw list that have the tags *circle* or *line*. The special parameter *all* can be useproduce a list of all items in the draw list.

The output from this command describes the origin and type of each of the objects that is used to make up a drawing. Currently, the types of objects defined for the *draw list* include the *line*, c*ircle*, *arc*, *rectangle* and *text* item.

26.3.20 ls − Set the current line style

The *ls* command sets the current line style. By default the line style is *solid*. The following command:

```
ls dashdot
```

would set the line style to *dashdot*. Any of the line style specifications supported by the extension can be used to define the type of line used to draw the objects in the draw list.

26.3.21 lt − Left turn

The *lt* command changes the current drawing direction by a specified angle in the counterclockwise direction. For example, the command:

```
lt 45
```

will change the current drawing direction by 45 degrees in the counterclockwise sense.

26.3.22 pc − Set the pen color

The *pc* command is used to specify the color of the pen used to draw things. By default, the pen color is *black*. The following command:

```
pc 0,0,255
```

will set the pen color to *blue*. Any color specification supported by the extension package can be used.

26.3.23 pd − Pen down

The *pd* command sets the pen state to *down*, meaning that the motion commands will cause a mark to be made using the current pen color and thickness and line style. For example:

```
pc red pd fd 20
```

will draw a *red* line of length *20* pixels from the current cursor location.

26.3.24 pp − Pop the drawing engine state

The *pp* command will pop the state of the drawing engine from an internal state stack, if there is a previously pushed state in the stack.

26.3.25 ps − Push the drawing engine state

The *ps* command will save the state of the drawing engine on an internal stack. The save state includes the current cursor location, the drawing direction, the pen state, pen color, pen width and line style. When popped, the drawing engine will resume the pushed state.

Pushing the state is a convenient method of building up complex diagrams. A segment of the diagram can be enclosed in brackets as follows:

```
( sp +20 −13 pc orange pd rt 30 fd 45 )
```

which will leave the drawing machine state exactly as it was before the segment was drawn. Note that the open and close brackets are command aliases for the push and pop operations respectively.

26.3.26 pt − Draw a point

The *pt* command draws a single pixel point at the current location using the current pen color.

26.3.27 pu − Pen up

The *pu* command places the pen in the up state, hence drawing will not occur when the cursor is moved during a drawing operation. When the pen is up, no drawing objects are entered into the draw list.

26.3.28 rc − Draw a rectangle

The *rc* command draws a rectangle using the current pen color, thickness and line style. The command takes 2 parameters, the width and the height of the rectangle to draw. For example, the command:

```
rc 40 20
```

will draw a rectangle of width *40* and height *20* with its upper left hand corner at the current cursor position. If the fill state is true, the rectangle will be filled with the current background color.

26.3.29 rp − Repeat a command block

The *rp* command is used to cause a block of commands to be repeated a specified number of times. The *rp* command takes 2 parameters, the repeat *count* and the *command block* . The parser recognizes a *command block* by quoting the list of commands with quotation marks. For example, the command:

```
rp 60 "rc 30 30 rt 6"
```

will draw an interesting rosette by repeating the drawing of a rectangle rotated about the current cursor location.

26.3.30 rt − Right turn

The *rt* command will change the drawing direction in the clockwise sense. For example, the command:

```
rt 30
```

would modify the current drawing direction by adjusting it clockwise by *30* degrees.

26.3.31 sh − Set the drawing direction

The *sh* command will set the current drawing direction. The single parameter is an angle in degrees to be used either as the new drawing direction or as a relative offset, in degrees, from the current drawing direction. When used as a relative value, the *sh* command is the equivalent of the *rt* or *lt* commands. For example, the command:

```
sh 180
```

would set the drawing direction to downwards with respect to the Drawing widget.

26.3.32 si − Show hidden items

The *si* command is used to cause hidden items in the draw list to be redisplayed. The command takes as a parameter a list of comma seperated strings that are used to identify the items to show. For example, the command:

```
si red,green,blue
```

would cause all items with the tags *red* or *green* or *blue* to become visible, if they were previously hidden.

26.3.33 sp − Set the cursor position

The *sp* command is used to set the cursor position. The specified values for the horizontal and vertical location may be either absolute window relative locations or relative offsets from the current cursor position. For example, the command:

```
sp +5 −8
```

will adjust the current cursor position by the offsets *+5* in the horizontal direction and *−8* in the vertical direction.

26.3.34 st − Show the cursor position

The *st* command is used to show the current cursor position, if it is hidden.

26.3.35 sx − Set the horizontal position

The *sx* command is used to set the horizontal offset of the cursor position to a specified value. The offset may be either an absolute window relative location or a relative offset from the current cursor position. For example, the command:

```
sx −30
```

will change the cursor position form its current location to a location offset by *−30* pixels horizontally from the current location.

26.3.36 sy − Set the vertical position

The *sy* command sets the vertical offset of the cursor position to the specified value. The offset may be either an absolute window relative coordinate value or a relative offset from the current cursor position. For example, the command:

```
sy +40
```

will set the vertical position of the cursor to a location offset *40* pixels in the y direction from its current location.

26.3.37 tg − Specify item tags

The *tg* command is used to specify a list of comma seperated strings that are applied to items as tags. The current tag list is always applied to all new draw list items. By default, the current tag list is an empty string, so items have no tags. Each item gets an automatic tag with is its sequence number in the draw list. For eacmple, the command:

```
tg red
```

would cause all items created after this command to have the tag *red*. Item tags can be used with the hi and si commands to control the visibility of al items with a given tag.

26.3.38 th − Set the line thickness

The *th* command sets the current line thickness. By default the line thickness is *1*. The command:

```
th 4
```

would set the line thickness to *4*. Line thickness does not apply to text items.

26.3.39 tr − Set the command trace state.

The *tr* command sets the state of command tracing. By defaule, the state is off, and no tracing occurs. When set to on, a trace of al commands executed by the drawing engine is displayed on the command console. This feature is sometimes helpfull in debugging long drawing scripts. For example, the command:

```
tr on
```

will activate command dracing.

26.3.40 tx − Set the text

The *tx* command specifies the text to be drawn using the current font, pen color, background color and font size. The text is drawn at the current cursor position. For example, the command:

```
tx "Hi, its me!"
```

will cause the greeting to appear at the current cursor location.

26.3.41 // − Comment

The *//* token indicates the beginning of a comment block. Comment blocks are ignored, and the feature is provided to make complex sets of commands more human readable. A comment block is terminated by another comment token. The set of comment tokens will be familiar to users of many types of scripting and programming environments. Here is an example:

```
pc green pd cr 20 /* A green circle */
```

27 Dummy − Do nothing

The *Dummy* command just prints a message on the interactive console indicating that it was called. Its main use is developing skeletons and testing them on phony commands.

28 Exit − Terminate the current application

The *Exit* command is identical to the standard Tcl *exit* command. It destroys the current interpreter and terminates the application.

29 Frame − Construct a frame widget

The *Frame* command creates a container widget that draws a box or frame. It can be used to draw frames around groups of widgets, or, using the event mechanism, it can be used to configure groups of widgets when mouse or keyboard events occur within its boundaries. The *Frame* widget supports, along with the set of *standard widget options*, the following widget specific options:

- auto Control the state of automatic layout
- rows Set the number of widget rows to use
- cols Set the number of widget columns to use
- xpad Set the horizontal pad width
- ypad Set the vertical pad width
- xborder Set the horizontal borger width
- yborder Set the vertical border width

The format of the *Frame* command is:

```
Frame path ... options ...
```

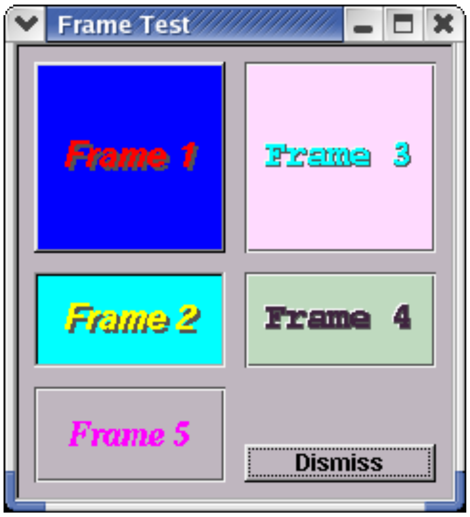
where *path* is a valid path name for the *Frame* and *options* are option and value pairs that configure the widget.

For example, the following command will create a *Frame* with *raised* relief:

```
Frame t.f −relief raisedframe −w 200 −h 200 −x 20 −y 50
```

This *Frame* will be located at (20,50) relative to its parent container window and have dimensions of 200 x 200 pixels. Since the *relief* specifies a *raisedframe* anything inside the *Frame* will not be affected by the drawing of the frame relief.

The *auto* option is used to control the operation of aome automatic child widget layout features of the frame widget. If the value of the *auto* option is *false*, the *Frame* widget does not do any automatic geometry management of the children that it may contain. The widget can then be used to create collections of widgets by arranging the children using their geometry management options, such as *x*, *y*, *w* and *h*.

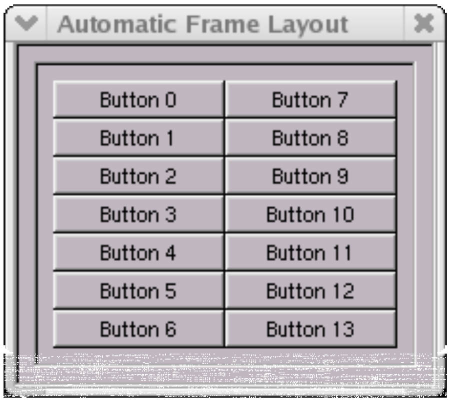


If the value of the *auto* option is *true*, the widget makes use of the values of the options *rows*, *cols*, *xpad*, *ypad*, *xborder* and *yborder* to automatically layout the child widgets. The layout mechanism uses the values of *xborder* and *yborder* to position all of the child widgets inside the specified border area, and the values of the *xpad* and *ypad* options to provide spacing between the widgets. The values of the *rows* and *cols* options are used to compute a size for the child widgets based on the geometry of the *Frame* itself. All of the children are then resizeed to fit inside the Frame and they are automatically arranged in the specified number of rows and columns.

The automatic layout mechanism is useful for rapidly arranging groups of identical widgets, such as *Button* or *Label* widgets, into an orderly array without having to resort to more complex arrangements of child widgets using the *Package* container. By default, the value of *auto* is *true*, the value of *rows* is *7*, *cols* is *2*, *xpad* and *ypad* are both *0*, and *xborder* and *yborder* are both *10*. This configuration provides for the layout of 14 widgets in a 7 by 2 array. Note that changing the geometry of the *Frame* itself results in automatic resize of the array of child widgets.

Here is the code to make use of the automatic layout feature of the Frame widget:

```
Frame t.f

for { set i 0 } { $i < 14 } { incr i } {
   Button t.f.b$i −label "Button $i" −command { puts "Its me! Button %W" }
 }

Show t

Wm title t "Automatic Frame Layout"
```



30 Focus − Set or Query the input focus

The *Focus* command can be used to either set the widget that has the current input focus or to determine which widget holds the input focus. The widget that has input focus is the one that will receive input from the mouse or keyboard.

The format of the command is:

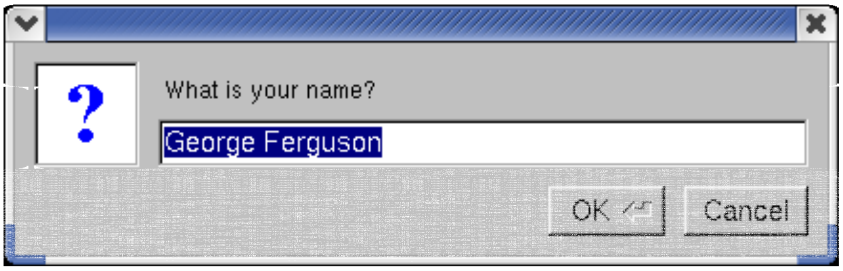
```
Focus path
```

where *path* is the path name of the widget to receive input focus. If *path* is not specified, then the command returns the path name of the widget that currently holds input focus, if any.

On success, the return value of this command is the path name of the widget that holds input focus.

31 GetInput − Get some input from the user

The *GetInput* command will display an input widget that can accept some text input typed by the user.



The format of the command is:

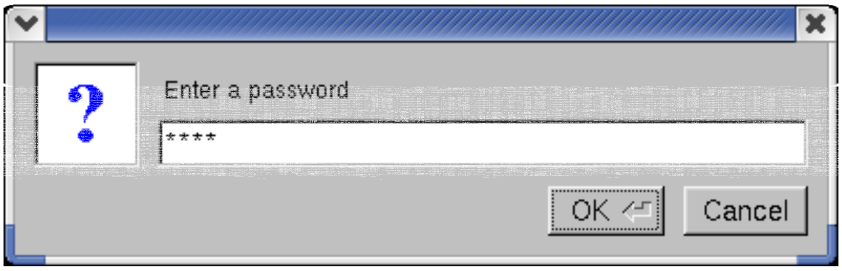
```
GetInput prompt default
```

where *prompt* is a text string to be used as a prompt and *default* is an optional default value for the input. At least a *prompt* must be supplied.

The value returned by this command is the input typed by the user, or an empty string if no default input is supplied.

32 GetPassword − Get a password from the user

The *GetPassword* command will display an input widget that can accept a password from the user.



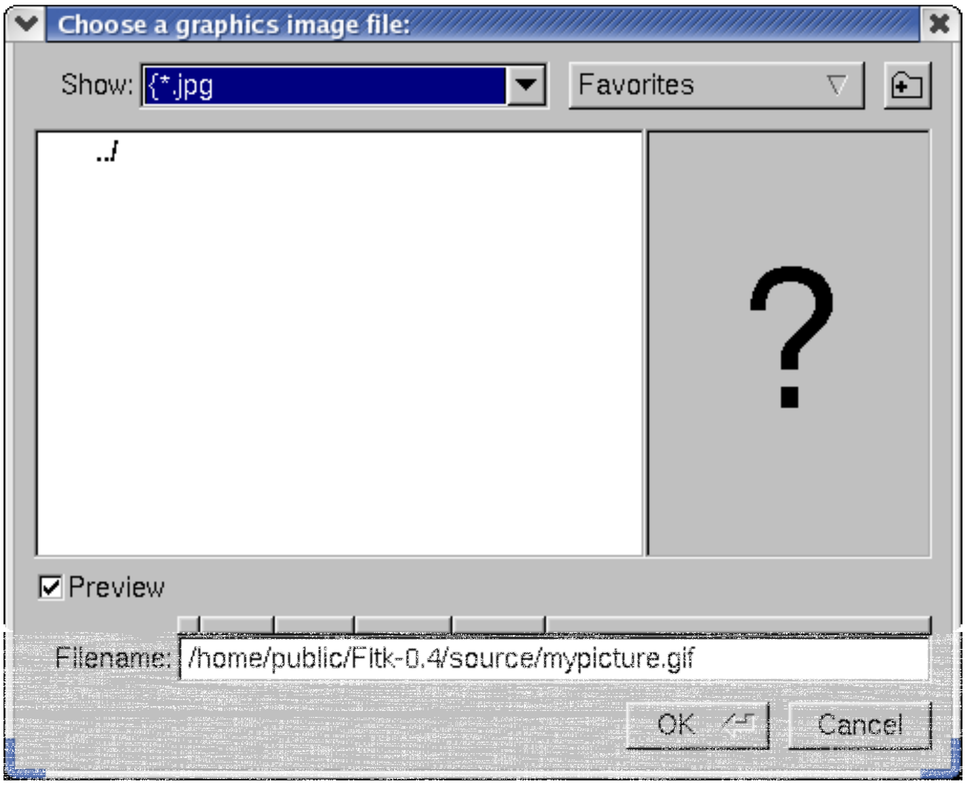
This command hides the characters typed by the user. the format of the command is the same as that of the GetInput command. For example:

```
set pw [GetPassword "Enter a password" "Junk"]
```

The value returend is the the use input or the default value, if one is supplied.

33 GetFileName − Get a file name from the user

The *GetFileName* command will display a file selection dialog box that allows the user to browse directories and to choose a file name.


Choose a graphics image file:
Show:
{*.jpg
Favorites
../
?
Preview
Filename:
/home/public/Fltk-0.4/source/mypicture.gif
OK
Cancel

The format of the command is:

```
GetFileName title pattern default
```

where *title* is the title for the file selection dialog box, *pattern* is a file selection pattern, and *default* is a default value for the file name being selected. At least a *title* must be supplied.

If no *pattern* is provided, the default pattern of "*" is used and all files are visible. *Patterns* can contain the usual wild card specifications. If a *pattern* is specified, then a default file name may also be specified.

Patterns can be formed as a list of wild card specifications separated by commas. For example, the following command will display available files in the specified graphics formats:

```
set name [GetFileName "Choose a graphics image file : " *.jpg,*.bmp,*.gif,*.png my_picture.gif]
```

Here the default selection is *my_picture.gif*. Imagefiles with the extensions *jpg*, *bmp*, *gif*, and *png* will be offered as alternatives.

If the user chooses a file, then the value returned by this command is the chosen file name. If the user cancels the dialog, then an empty string is returned.

34 Group − Create a group container widget

A *Group* is a container widget that is useful for collecting together a set of child widgets that can be subsequently moved about by changing the location of the *Group*. Its appearance is similar to the GroupBox widgets that are part of the Microsoft Windows GUI environment. Groups can also be used to collect together a set of widgets that are of similar dimensions, such as a collection of *Button* widegets used to specify user options. The *Group* widget supports a limited form of automatic child widget placement that makes the positioning of children convenient for some types of application, such as option interfaces. The format of the command is:

```
Group path options
```

where *path* is the path name of the widget to be created and *options* is the list of option and value pairs that is used to configure the widget. The *Group* widget supports, in addition to the list of *standard widget options*, the following widget specific options:

● xborder,yborder	Specify the widget margins
● xpad,ypad	Specify the horizontal and vertical pad widths
● rows,cols	Specify the layout geometry
● auto	Set the state of automatic layout
● value	Set or get the current group box label

The *xborder* and *yborder* options are used to specify the space between the edge of the client area of the *Group* widget and the edges of the child widgets in the *Group*. By default, these values are set to *10*.

The *xpad* and *ypad* options are used to specify the pad width in the horizontal and vertical dimensions used to calculate the position of child widgets added to the *Group*. By default, the value of the *xpad* option is 0, and the value of the *ypad* option is *0*. The pad values are the dimensions of the space left between adjacent widgets.

The *rows* and *cols* values define the order of the array used for layout of the child widgets By default, the value of *rows* is *2* and the value of *cols* is *7*. This results in an array of 14 positions that are used to position child widgets. When automatic layout is used, the child widgets are all resized to the same dimensions and then positioned according to the order of the layout array. The ultimate size of the child widgets is determined by the geometry of the *Group* widget and the values of the *border* and *pad* options.

The *auto* option is used to control the use of automatic child layout. By default, the value of *auto* is *true* and child widgets are automatically positioned according to their order of creation. When *auto* is *false*, child widgets are position according to their specified geometry.

The *value* option sets or gets the label of the *Group* widget. This is a dummy option that is used for variable tracking. Using a constructor of the form:

```
Group t.g −variable GroupLabel
```

will allow the application to set the label text of the widget by changing the value of the variable *GroupLabel*.

To add widgets to a *Group* using manual layout, just create them as children of the *Group*. Here is an example:

```
# Create a group with a label

set g [Group root.g −x 20 −y 50 −width 200 −height 200 −label "My Group" −align top,left,inside]

# Add a few widgets to the group

Scrollbar $g.sb1 −orientation horizontal −x 40 −y 40
Dial $g.d1 −x 40 −y 120 −dialstyle filled
```

The children can then be moved as follows:

```
$g configure −x +40 −y +80
```

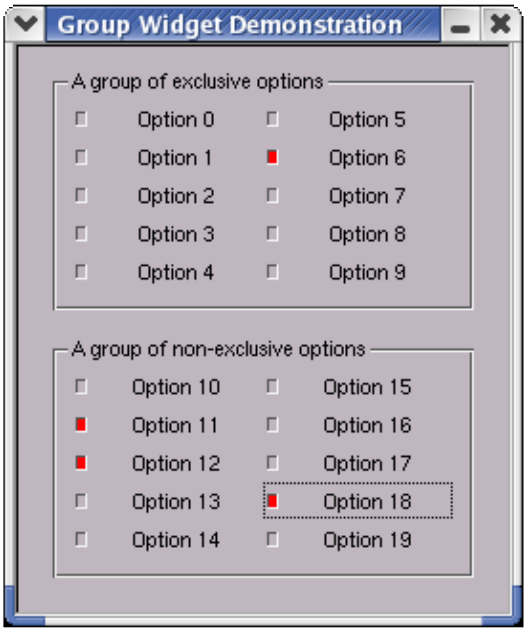
and all the children will maintain their layout inside the *Group*. The *relief* widget option can be used to configure various types of frames that will surround the children in the *Group*.

34.1 Automatic Child Widget Positioning

Automatic positioning of child widgets in a *Group* is accomplished by not supplying any child widget location information on the constructor command for the child. For example, the following script will pack a number of *LightButton* widgets into a Group in a manner useful for configuring user options.

```
#!/bin/sh
# \
exec fltkwish "$0" ${1+"$@"}
#
# −−− group.tcl −−− Test harness for the Group widget
#
# Copyright(C) I.B.Findleton, 2003. All Rights Reserved
#
Destroy t
#
# Configure the LightButton widgets
#
Option add Group.relief flat
Option add LightButton.selectioncolor red
Option add LightButton.relief flat
#
# Pack the groups into a container
#
set f [Package t.all −orientation vertical]
#
# Create a group widget to hold the buttons. Create 5 rows and 2 columns
#
Group $f.g1 −r 5 −c 2 −label "A group of exclusive options"
#
# Add the buttons. These are of the radio type, so all will be exclusive
# within the group.
#
for { set i 0 } { $i < 10 } { incr i } {
        LightButton $f.g1.b$i −text "Option $i" −type radio
        }
#
# Create a group of non−exclusive options
#
Group $f.g2 −r 5 −c 2 −label "A group of non−exclusive options"
#
# Add some buttons of the toggle type for the options. These are non−
# exclusive options, so many can be selected.
#
for { set i 10 } { $i < 20 } { incr i } {
        LightButton $f.g2.b$i −text "Option $i" −type toggle
        }

Show t

Wm title t "Group Widget Demonstration"
```

Here is the result of this script:



Note that the *label* options of the *Group* widgets are used to set the group box titles. The group box titles can be positioned about the top and bottom of the group frame using the available values for the *align* widget option. Label text can not be positioned along the sides of the group frame, but may be either at the *top* or the *bottom*, and may be aligned *left*, *centered* or *right*. Note that the *inside* keyword must be specified along with the desired alignment, otherwise, the label is written outside of the widget. For example, the command:

```
$w set −align bottom,right,inside
```

would place the group frame label at the bottom right of the group frame.

35 Help − Display help information

The *Help* command is used to list the possible values of many of the configurable options. The format of the command is:

```
Help −name
```

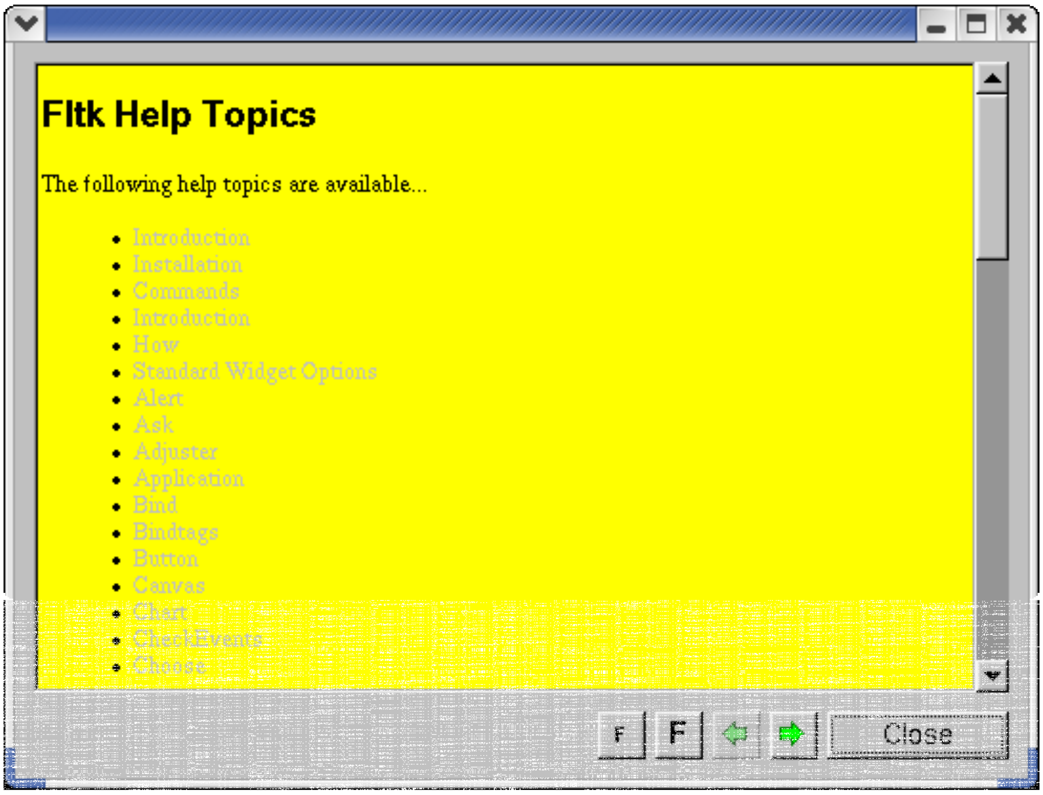
where *name* is the name of a configurable option. For example, to get a list of the available relief values, use the command:

```
Help −relief
```

If no *name* is specified this command returns a list of the possible values for *name.*

36 HelpDialog − Display Help information

The *HelpDialog* command displays a dialog that can be used to page through text files in HTML format. The dialog supports basic HTML markups but is not a full blown browser.



Fltk Help Topics

The following help topics are available...

- Introduction
- Installation
- Commands
- Introduction
- How
- Standard Widget Options
- Alert
- Ask
- Adjuster
- Application
- Bind
- Bindtags
- Button
- Canvas
- Chart
- CheckEvents
- Choose

The format of the command is:

```
HelpDialog file
```

where *file* is the name of the root file to be loaded. This command is a quick way of implementing help display dialogs. When creating the HTML files to be used, links should be placed in the document to ease navigation throughout the help information. The browsing widget does not support frames, but will do most of the non frames related things.

37 HelpViewer − Create a HTML viewing widget

The *HtmlViewer* widget is a widget that can be loaded with a block of data in HTML format. The widget supports many of the capabilities of the HTML 2.0 standard. The format of the the command line is:

```
HtmlViewer path options
```

where *path* is the path name of the widget and *options* is the list o option and value pairs that is used to configure the widget. In addition to the list of *standard widget options*, the widget supports the following widget specific options:

- value The contents of the widget, which is a block of HTML data.
- texttcolor The foreground color of the HTML text
- textfont The font to use for display of text
- textsize The size of the font to use
- length The size, in pixels, of the text
- doctitle The current document title
- directory The current directory
- filename The complete path name of the current file
- topline The current top line in the widget
- linkproc The URL handler
- url The url to use

37.1 Loading HTML Data

The *HtmlViewer* widget is designed to process HTML format text files, although the *value* option can be used to load a block of formatted data directly into the widget, or to retrieve the currently loaded block of data. For the purpose of loading data from a file in HTML format, the widget supports the *load* command. The format of the command is:

```
$w load file
```

where *w* is the token that represents the widget command for the *HtmlViewer* widget and *file* is either the path name to a file or a Universal Resource locator (URL).

The *HtmlViewer* widget itself cannot resolve URL references and does not include the communications support to directly access web servers connected via a network interface. Where the *HtmlWidget* recognizes a URL reference, it calls a function that must invoke the required communications functionality, download the requested data to a local file, and return the name of the file to the widget. The *linkproc* option is provided so that a script can be specified that will perform the required functions.

37.2 value

The *value* option can be used to get or set the data block that is the HTML data currently displayed in the widget. Typically the data would have been loaded from an HTML format text file, but, the following command could be used to set the data:

```
$w configure −value $data
```

where *$w* is the token that represents the widget command and *$data* is a reference to a variable containing a block of text in HTML format.

37.3 textcolor,textfont, and textsize

The *textcolor*, *textfont* and *textsize* options are used to set the default characteristics of the data displayed in the *HtmlViewer* widget. Text files in HTML format typically contain formatting information that will override the default specifications provided by these options. Where the HTML text does not provide any specific font and color information, the current values of these options are used to draw the text.

37.4 length

The *length* option is a read only option that returns the size in pixels of the block of HTML data in the widget buffer.

37.5 doctitle

The *doctitle* option can be used to query the title of the currently displayed document.

37.6 directory and filename

When a file is loaded into the *HtmlViewer* widget, these 2 options are available for query. The value of the *filename* option is the full path name to the file that contains the HTML format data currently being displayed by the widget, and the value of the *directory* option is the directory path to the HTML data being displayed. When links in the displayed text are activated, the values of *filename* and *directory* will change to reflect the file location of the HTML page being displayed.

37.7 topline

The *topline* option can be used to query or set the number of the top line being displayed in the widget. If used to set the top line, the value specified needs to be within the range of line numbers in the text data.

37.8 linkproc

The *linkproc* option is used to specify the Tcl script that is responsible for resolving URL references by downloading the requested HTML data into a local file. The value specified for the *linkproc* option should be a Tcl script that returns a result that is the name of the local file that contains the requested data.

By default, the value of the *linkproc* option is an empty string, and URL references that are not local file names will result in nothing being displayed in the *HtmlViewer* widget. If a script is provided as a value for the *linkproc* option, it is first expanded to replace embedded keywords, then it is evaluated. The keywords that are recognized are:

- %u The URL to be resolved
- %w The name of the widget making the request
- %% A percent sign

Typically, the script will parse the URL to determine the location of the requested data, contact a server and download the data to a local file, and return the name of the local file.

Here is an example of a Tcl procedure that will get pages from the World Wide Web:

```
package require http 2.1

# Load a URL into a local file

proc UrlProc { url } {

    global env

    puts "Converting $url"

    set fd [open [set name $env(PWD)/url_temp.html] w]

    ::http::geturl $url −channel $fd −blocksize 4096

    close $fd
return $name
}
```

An *HtmlViewer* widget created with the following command would call the above procedure to resolve the passed URL references, download the HTML data to a file named *url_temp.html*, then display the data in the widget:

```
HtmlWidget t.h −linkproc { UrlProc %u }
```

Note that the *http* package is part of the standard Tcl distribution.

37.9 url

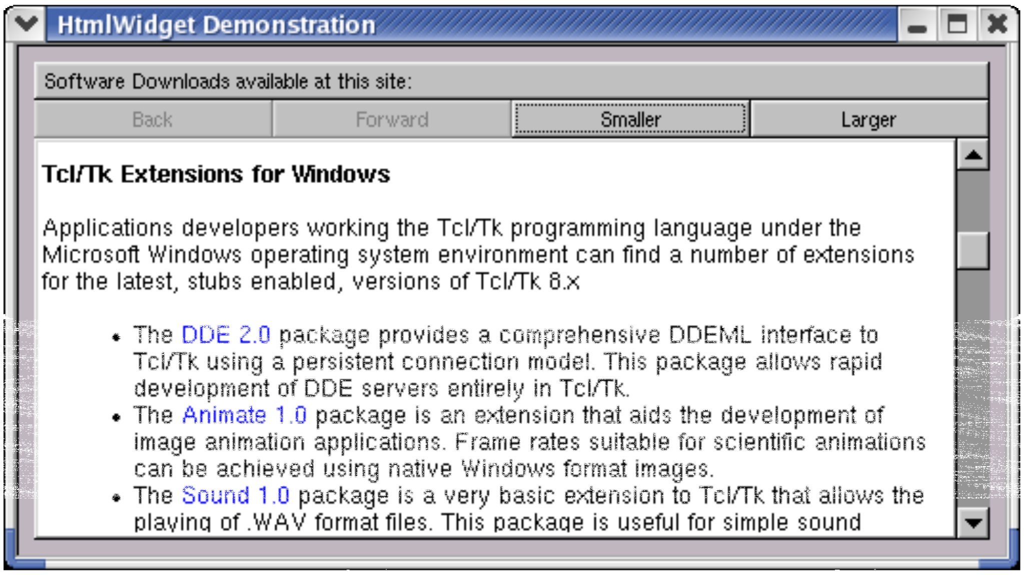
The *url* option can be used to specify the file or URL to be loaded either during the construction of the *HtmlViewer* widget or with the use of the widget command. Here is an example:

```
$w configure −url url_temp.html
```

where *$w* is the token that represents the widget command and *url_temp.html* is a local file name.

38 HtmlWidget − Construct an HTML Display Widget

The *HtmlWidget* is a mega−widget that adds navigation and font control features to an *HtmlViewer* widget. This widget provides the functionality of the *HelpDialog* widget as a normal, non dialog, embedable widget. It is useful for the inclusion of an HTML display widget directly into a GUI which has some navigation capabilities of its own.



Tcl/Tk Extensions for Windows

Applications developers working the Tcl/Tk programming language under the Microsoft Windows operating system environment can find a number of extensions for the latest, stubs enabled, versions of Tcl/Tk 8.x

- The DDE 2.0 package provides a comprehensive DDEML interface to Tcl/Tk using a persistent connection model. This package allows rapid development of DDE servers entirely in Tcl/Tk.
- The Animate 1.0 package is an extension that aids the development of image animation applications. Frame rates suitable for scientific animations can be achieved using native Windows format images.
- The Sound 1.0 package is a very basic extension to Tcl/Tk that allows the playing of .WAV format files. This package is useful for simple sound

The above image is an example of an HtmlWidget displaying a URL from the internet. The page displayed is http://pages.infinit.net/cclients/software.htm, the main download page for the Fltk extension.

The format of the the widget command is:

    HtmlWidget path options

where *path* is the path name of the widget to be constructed and *options* is the list of option and value pairs that is used to configure the widget. In addition to the list of *standard widget options*, the *HtmlWidget* also supports all of the widget specific options of the *HtmlViewer* widget, and these additional widget specific options:

- htmlrelief The relief of the HTML sub window
- configuration Configuration options for the widget
- labelfont The font for the decorations
- labelcolor The foreground color for the decorations
- labelsize The font size for the decorations

The *htmlrelief* option is used to set the relief that the sub widget that displays the HTML text uses. By default this relief is *flat.*

The *configuration* option is used to specify which decorations the widget implements. The decorations are buttons and labels that implement navigation, font control and display document titles when they are present. The default value of the *configuration* option is *default*, and the resulting widget has a title widget and 4 button widgets that are displayed along with the widget that displays the HTML text.

The *configuration* of the widget is specified as a series of comma separated strings that specify which decorations to implement. The list of supported *configuration* strings is:

- title Creates a title bar that displays document titles
- navigation Creates the forward and back navigation buttons
- font Creates the larger and smaller font size control buttons.
- default Creates all available decorations

Here is an example of a basic *HtmlWidget* that has only the HTML window:

```
HtmlWidget t.h −configuration ""
```

whereas the following command will create a widget with a title and navigation buttons but no font control:

```
HtmlWidget t.h −configuration "title,navigtion"
```

Note that the *configuration* option is somewhat unusual in the Fltk extension package in that it can only be used with effect when the widget is constructed. If you want to change the *configuration* of the widget, you must first use the *Destroy* command to get rid of the existing widget and then create a new one with the desired configuration.

38.1 Widget Specific Commands

In addition to the standard *cget* and *configure* commands, the *HtmlWidget* supports the following widget specific commands:

- load   Load a file name
- page   Show a page in the page stack
- font   Manage the displayed font

38.1.1 load

The *load* command has the following format:

```
$w load url
```

where *$w* is the token that represents the widget command and *url* is either a local file name or a Universal Resource Locator (URL) that can be used to download the data to a local file. If a URL is supplied, the widget must have a valid script as its *linkproc*. The script must resolve any URLs , download the data to a local file, and return the name of the local file. See the example script described for the *HtmlViewer* widget.

38.1.2 page

The page command is used to display a page in the page stack or to query the widget about the pages in the page stack. The format of the command is:

```
$w page action
```

where $w is the token that represents the widget command and action is the action to take. If no action is specified then the result of the command is the current index of the page being displayed in the widget. An index value of −1 indicates that there is no page in the page stack.

The values that action may have are:

- back   Display the previous page in the stack
- clear   Empty the page stack
- count   Return the number of pages in the page stack
- forward   Display the next page in the stack
- home   Display the first page in the stack
- list   Return a list of the pages in the stack

The value of *action* may also be an integet in the range from 0 through the number of pages currently loaded. If the value of *action* is a valid page number then the corresponding page will be loaded.

38.1.3 font

The format of the *font* command is:

```
$w font action
```

where *$w* is the token that represents the widget command and *action* is either *larger* or *smaller* or not specified. The font size will be changed according to the *action* with the range of font sizes supported by the current font. The command has no effect if the current font size is at the end of its range, and will not change the size of font specifications embedded in the HTML data itself.

The value returned by this command is the current font size. If the action is not specified, then the value returned is the current font size.

39 Hide − Make one or more windows invisible

The *Hide* command can be used to make one or more visible windows invisible. If a window is made invisible, all of the children of the window are also made invisible.

The format of the command is:

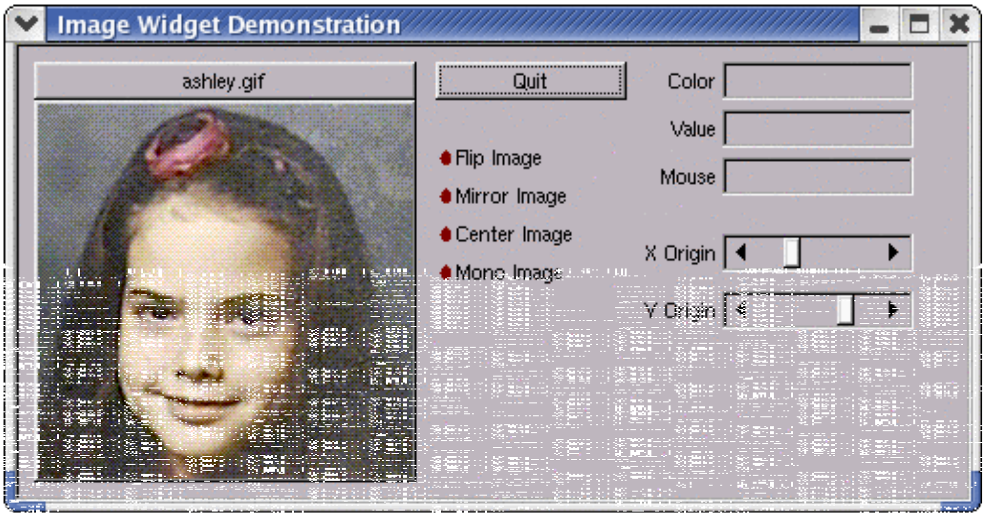
```
Hide path names
```

where *path names* is a list of one or more valid widget path names. If a window does not exist, the *path name* is ignored. Hidden widgets can be made visible using the *Show* command.

When widgets are first constructed, they are hidden. In order for widgets to be visible, the *Show* command must be invoked on the root widget of a widget tree.

40 Image − Construct an image widget

The *Image* command creates a widget that displays a picture. The source of the picture can be a file in one of the image formats that is supported by the Fltk extension package.



The format of the command is:

Image path options

where *path* is the path name of the image widget to be created and *options* are the option and value pairs that are used to configure the widget. In addition to the set of *standard widget options*, the image widget supports the following widget specific options:

centered	If the image is centered in the widget
file	Name of the image file to use
flip	If the image should be flipped vertically
imagedepth	Color depth of the image
imageheight	Height of the image in pixels
imagewidth	Width of the image in pixels
imagex	Horizontal location of the image in the widget
imagey	Vertical location of the image in the widget
mirror	If the image should be flipped horizontally
monochrome	If the image is displayed as a monochrome image
shrinkwrap	If the image is shrink−wrapped
nodisabledimage	If a disabled image should not be generated

By default, the image will be displayed centered in the widget. If the image is larger than the widget, the center of the image will be at the center of the widget.

40.1 Supported File Formats

The file used to load the image can be in one of the following file formats:

bmp	Windows format device independent bitmaps
gif	Graphics Interchange Form
jpeg	Independent JPEG format
tiff	Tag image file format

png	Portable network graphics format
ico	Windows icon file format
raw	A rectangular array of pixel intensities
xbm	X Windows Bitmap
xpm	X Windows Pixmap

Most image formats will automatically set the *imageheight*, *imagewidth* and *imagedepth* properties when the image is loaded. The *raw* format, however, has no header in the image file, so loading this type of image requires that the image dimensions and color depth be set so that the image will load correctly.

40.2 Configuration Options

The *imagedepth* option can be either *3* for full RGB format colors, or *1* for 8 bit intensity images. By default, *imagedepth* is *3*. Note that with the exception of the *raw* format images, the *imagedepth* value is used only for the display of the image, not for the loading of the image. It can not be used to specify color space reduction operations on images during loading.

The *imagex* and *imagey* values are by default set to 0. This causes the image to be placed at the upper left hand corner of the widget, unless the valued of the *centered* option is *true*. By manipulation of the *imagex* and *imagey* values, the image can be moved about the client area of the widget.

By default, the *flip* and *mirror* options are *false*. Some images may need to be flipped or mirrored when loaded to get them to display correctly, particularly if the coordinate system used by the application that created the image files is different from that being used by the Fltk tool kit.

The *centered* option is set to *true* by default. This causes the loaded images to be centered in the widget.

The *shrinkwrap* option is *false* by default. Setting *shrinkwrap* to true will cause the widget to resize itself around the image. The standard widget options *padx*, *pady* and *borderwidth* can be used to adjust the space surrounding the image if so desired.

The *nodisabledimage* option is used to specify whether a disabled form of an image should be generated. By default, the value of this option is *false*, and when an image is loaded its disabled form is automatically generated. When set to *true* this option prevents generation and display of a disabled format of the image. When loading very large images, suppression of the disabled format of the image can speed up image display.

For large images, placing the Image widget inside of a *Scroll* widget will generate automatic scroll bars that can be used to pan across large images. Alternatively, the application can use one of the other input widgets, such as the *Scrollbar*, *Roller*, *Slider* or *Adjuster* to change the values of the *imagex* and *imagey* option values to implement a panning feature.

40.3 Image Markup

Images can be marked up using a set of drawing primitives and a set of pre−defined symbols. Image marks are kept in a list that is displayed after the image itself is drawn in the widget window. The following drawing primitives are available for marking up images:

line	Draw a line between 2 points
circle	Draw a circle at a specified point
arc	Draw an arc
rectangle	Draw a rectangle
text	Draw text
polyline	Draw a line through a set of vertices
polygon	Draw a polygon
bezier	Draw a bezier curve

In addition to the drawing primitives, the following pre−defined marks can be used:

plus	Draw a plus sign
cross	Draw an x symbol
position	Draw a circular position mark
box	Draw a small rectangle
low	Draw a low symbol
high	Draw a high symbol

A special mark, called a *drawing*, can also be specified. The *drawing* mark is used to create compound line drawings that behave like any of the other marks for the purpose of location. *Drawing*s make use of a small drawing language that resembles the old Logo Turtle Graphics language.

The set of marks defined above are useful when marking up weather maps, and may be useful for marking up other types of technical drawings.

40.4 Mark Attributes

A mark is a geometrical object that has a set of attributes that are used to control the location and appearance of the mark. Configuring a mark depends on the particular geometry of the mark. For example, a circle is configured by specifying its center, radius, color and fill attributes, while a rectangle requires 2 pairs of coordinates that specify the location of the upper left hand corner and the height and width of the rectangle. All of the marks supported by the Image widget are configured using a selection of the following list of attribute names:

at	A pair of coordinates to specify a location for the object
to	A pair of coordinates to specify a destination location
color	The name of a color to use in drawing the object
fill	A boolean value specifying whether the object should be filled
width	A value that is used as a horizontal extent
size	A value that is used to specify the size of an object, such as the radius of a circle
fillcolor	The color to use to fill an object
foreground	The foreground color of text
background	The background color of text
font	A font description for a text font
points	A list of point coordinates
start	A starting angle
end	An ending angle
bbox	A bounding box
rounded	A boolean value specifying whether the object is rounded
text	A string to be used as the text to display
name	The name of the object
tags	A list of tags for the object
borderwidth	The width of the border around an object
bordercolor	The color to use when drawing a border
state	The state of the object determines its visibility
penstyle	The type of pen used to draw the object
x,y	The location of the object
data	A text string that can hold user data or a *drawing* specification.

Marks are drawn by using the *add* widget command to add new marks to the image mark list. Other widget commands allow for the management of the mark list and the management of the values of the attributes of the marks.

40.5 Widget Commands

In addition to the standard cget and configure commands, the Image widget supports the following widget specific commands:

add	Add a mark to the mark list
clear	Clear the list of marks
closest	Get the closest mark to a location
getpixel	Get the color of a pixel
itemcget	Get the attributes of a mark
itemconfigure	Set the attributes of a mark
setpixel	Set the color of a pixel in the image
save	Save the image to a file

40.5.1 add Add a mark to the mark list

The *add* command creates a new mark and adds it to the mark list. The format of the *add* command is:

```
$w add mark options
```

where *$w* is the path name of the *Image* widget, *mark* is the name of the mark to be drawn, and *options* is the list of option and value pairs that are needed to define the mark. For example, the command:

```
$w add line −at 10,30 −to 100,120 −color red −penstyle dash
```

will draw a red line from image location 10,30 to image location 100,120. The line will be drawn as a series of dashes.

The value returned by the *add* command is a token that can be used to identify the item in the mark list. This token can be used to manage the values of mark attributes using the *itemconfigure* and *itemcget* widget commands.

40.5.2 clear Clear the mark list

The *clear* command is used to remove items from the mark list. The format of the command is:

```
$w clear mark mark ...
```

where *$w* is the path name of the *Image* widget and the *marks* are an optional list of mark identifiers that are to be deleted from the list. If no *marks* are specified, the entire list is cleared.

The *clear* command does not return a value.

40.5.3 closest Get the closest mark to a location

The *closest* command takes a set of one or more coordinate pairs and returns the list of marks that are those closest to the coordinates specified. The format of the command is:

```
$w closest x1,y1 x2,y2 ...
```

where *$w* is the path name of the widget to use and the *x,y* pairs are the locations to use. The value returned by this command is a list of the tokens of the marks that correspond to the coordinate pairs. If there are no marks in the list, the result of this command is an empty list.

The coordinates supplied should be widget relative values that represent actual locations in the image. Other values, such as window relative coordinates, will not necessarily find the intended targets.

40.5.4 getpixel Get the color of a pixel

The *getpixel* command is used to get the color of pixels in the image displayed by the widget. The format of the command is:

```
$w getpixel x1,y1 x2,y2 ...
```

where *$w* is the path name of the widget to use and the *x,y* pairs are image locations to query. The result of this command is a list of elements each of which is a list of 3 numbers that represent the red, green and blue components of the color of the image at the corresponding location. Coordinate locations that are not within the displayed image bounds result in an error message.

40.5.5 itemcget Query the attributes of a mark

The *itemcget* command is used to query the current values of mark attributes. The format of the command is:

```
$w itemcget mark list
```

where *$w* is the path name of the widget, *mark* is the token identifier of the mark to query, and *list* is the list of mark attributes to query. The result of this command is a list of the current values of the queried attributes. For example, the command:

```
$w itemcget $id −at −to −color −penstyle
```

could be used to determine the current values of the attributes of a *line* mark whose identifier is represented by *$id*.

40.5.6 itemconfigure Configure mark attributes

The *itemconfigure* command is used to set the values of mark attributes for marks already in the mark list. The format of the command is:

```
$w itemconfigure mark options
```

where *$w* is the path name of the widget to use, *mark* is the token identifier of the mark to manage, and *options* is the list of option and value pairs that defines the new values of the mark attributes to be set. For example, the command:

```
$w itemconfigure $circle −at 120,40
```

could be used to move a circle center from its current location to 120,40. Here the token for the circle is represented by the *$circle* reference.

40.5.7 setpixel Set the color of a pixel

The *setpixel* command can set one or more pixels of an image to a specified color. The format of the command is:

```
$w setpixel x1,y1 color1 x2,y2 color2 ...
```

where *$w* is the path name of the widget, the *x,y* pairs are pixel coordinates in the image, and the *colors* are color specifications for the new pixel colors. There must be a coordinate pair and a color specification for each pixel to be set.

The color specification can be any of the valid color descriptions supported by the extension package. For example, the command:

```
$w setpixel 100,100 pink 140,20 128,30,45 35,50 192
```

sets the pixel at 100,100 to pink while the pixel at 140,20 will be set to the color whose red, green and blue components are 128, 30 and 45 respectively. The last pixel at 35,50 will be set to a red, green and blue color value of 192, 192 and 192.

40.5.8 save   Save the image to a file

The *save* command will write the current image to a file. The format of the command is:

```
$w save −file name −depth bits
```

where *$w* is the path name of the widget to use, *name* is the name of the file to use, and *depth* is an optional specification of the color depth, in bits per pixel, to use.

The format of the file is determined by the file name extension that is employed in the file *name*. Any of the file formats supported by the extension package can be used. The specification of a bit depth can result in automatic color space reduction, which may result in degradation of picture quality.

40.6 Drawings

A drawing is a mark that is created by the specification of a series pf pen actions using a drawing specification language. The drawing specification language uses a syntax that has the form of a list of blank separated tokens which are drawing commands and their parameters. The drawing language drives a drawing engine that can produce 3 types of drawing element, the line, the circle and the text item.

A new *drawing* is created with a command of the form:

```
$w add drawing options
```

where $w is the path name of the *Image* widget being used and *options* is the list of option and value pairs that is used to configure the *drawing*. The *drawing* has the same list of options that are recognized by the other *marks* available for the Image widget, however, only the *data* and the *at*, *x* and *y* options are generally useful. The *data* option contains a text string that describes the actual drawing to be done, while the location values set the origin of the drawing in coordinates of the image being displayed.

The drawing engine has a pen that draws things which can have a color, a width and a style. The pen may be either up, in which case no drawing will occur, or it can be down, in which case drawing will occur. The set of commands that can be used to manipulate the pen are the following:

pu	Raise the pen
pd	Lower the pen
fd	Move the pen a specified amount in the current direction
bk	Move the pen a specified amount in the reverse of the current direction
rt	Change the current direction clockwise by a specified angle in degrees
lt	Change the current direction counterclockwise by a specified angle in degrees
cs	Clear the current drawing
hm	Move to the current drawing origin
dl	Draw a line without moving the current position in the current direction
cr	Draw a circle of a specified radius at the current location
tx	Draw a text string at the current location
pc	Set the current pen color
bg	Set the background color
ft	Specify the current font
fs	Specify the current font size
sp	Set the current position of the pen
sx	Set the horizontal position of the pen
sy	Set the vertical position of the pen
sh	Set the current horizontal angle
th	Set the current pen thickness

ls	Set the current pen line style
li	List the current drawing components
rp	Repeat a set of commands a specified number of times

The drawing description is a string with blank separated commands and parameters that the drawing engine will interpret. When a *drawing* is created the initial pen location is at the specified origin and the initial drawing direction is vertical toward the top of the *Image* widget. The *fd*, *bk* and *tx* commands will leave the current position of the pen at the end of the line or text block being drawn. The default color for the pen is *black*, with thickness of 1 and a line style of *solid.* Here is an example of a drawing the will produce a blue box:

```
$w add drawing −at 100,100 −data { cs hm pc blue pd rp 4 "fd 40 rt 90" }
```

Here the blue square will begin at location *100,100*, then, using a *blue* pen will draw the 4 sides using the *rp* command. Aside from the *sp* command, which takes 2 parameters, one for each dimension of the image, and the *rp* command which takes a repeat count and a command string, the other commands take either 1 parameter or no parameters. The parameter values can have a sign prefix that is interpreted as an adjustment to the current value. For example, the string:

```
sp +20 −14
```

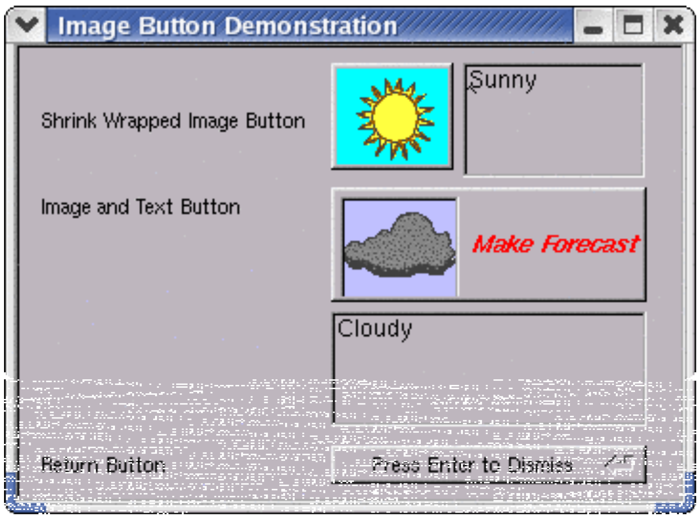
means change the current value of the horizontal position by adding 20, and the current value of the vertical position bu decreasing it by 14. If the sign prefix is not present, the actual value of the parameter becomes the new value of the item.

Note that the elements of a drawing will not be cleared unless a *cs* command is encountered. Drawings can be built up by using the itemconfigure command of the Image widget to append new elements to the current drawing.

For a detailed description of the Turtle Graphics command set supported by this extension package see the chapter on the *Drawing* widget.

41 ImageButton − Construct an image button widget

The *ImageButton* command creates a button that is drawn using images instead of the text format typical of the other *Button* widgets in the package.



The format of the command is:

```
ImageButton path options
```

where *path* is the path name of the widget to be created and *options* are the option and value pairs that are used to configure the widget. In addition to the *standard set of widget options*, the *ImageButton* widget supports the following widget specific options:

upimage	Image to use for the unpressed state
downimage	Image to use for the pressed state
onvalue	String to return as the button on value
offvalue	String to return as the button off value
downrelief	Button relief when pressed
value	Value of the button
type	Type of the button
monochrome	If the image is displayed as gray scale
imagex	Horizontal location of the image
imagey	Vertical location of the image
imageheight	Height in pixels of the image
imagewidth	Width in pixels of the image
imagedepth	Color depth of the image
centered	If the image is centered
shrinkwrap	If the image is shrink−wrapped
state	The state of the button

All of the options have default values, so the button can be created without any options and then configured later using the widget *configure* function.

The *upimage* and *downimage* options take the name of a file that is in one of the file formats supported by the *Image* widget. If only an *upimage* is supplied, it will be used for both the pressed and not pressed states of the button. If the button is in the *disabled* state, a modified rendering of the *upimage* is used to draw the button.

The *onvalue* and *offvalue* options default to the strings *"1"* and *"0"*. These are the values of the button that will be returned if the widget is queried, or if the button is attached to a Tcl variable using the *variable* option.

The *downrelief* option is by default a *sunkenframe* relief. This relief is used when the button is in the pressed state.

The *value* option is used to set the initial value of the button. This option takes a boolean name to indicate whether the value should be the *onvalue* or the *offvalue*.

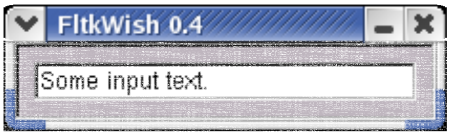
The *type* option can be invariant, *toggle* or *radio*. The default *type* is *toggle*, so each time the button is pressed and released it changes its *value* from *on* to *off* and back. An *invariant* button does not change its *value*, while a *radio* button changes its value when pressed and remains in the new state until changed using the widget *configure* function.

The *state* option can have the values *normal* or *disabled*. When *disabled* the button will not process any mouse or keyboard input. By default, the state is *normal*.

The remainder of the widget specific options implement features of the image display functionality of the extension package. These options have the same meaning as those described for the *Image* widget.

42 Input − Create an input widget

An *Input* is a widget that can accept user input via the keyboard. It may be single line or multi−line in form.



The format of the command is:

Input path options

where *path* is the path name of the widget to be created and *options* is the list of option and value pairs used to configure the widget. In addition to the set of *standard widget options*, the *Input* widget supports the following widget specific options:

value	The contents of the widget
color	The color of the text
textfont	The font in use
textsize	The size of the text font
length	Query the amount of text in the widget
format	The type of the input widget
mark	Location of the current mark
position	Location of the current input position

The *value* option is used to either get or set the contents of the *Input* widget. A string with embedded newline characters is either the result of a query or the way to set the contents of the widget.

The *color*, *textfont* and *textsize* options can be used to configure the display of the text in the widget. Only a single rendering specification can be applied to the entire widget.

The *length* option can be queried to discover the number of characters in the widget. The *mark* and *position* options can be set or queried to move about the *mark* and the *input location pointer* respectively. The relationship between the *mark* and the *position* values is that the characters between these two values are considered to be the *selection range* for editing operations.

The *format* option can have the following values:

normal	Standard single line input
float	Input of a floating point number
integer	Input of an integer
multi−line	Multiline text input box
secret	A password entry input box

The default format is *normal*. The other formats affect the justification of the displayed input values.

42.1 Using Input Widgets

The *Input* widget is used to allow the application user to enter information that can be used by the application for some purpose. Typically an *Input*widget is bound to a Tcl variable that is to receive the user input. Optionally, the *Input* widget can also have a command that is executed when the user presses the *Enter* key. For example, the command:

```
Input t.t −variable MyTclVariable −command { HandleInput %W }
```

will create an *Input*widget bound to the variable *MyTclVariable*. If the contents of the widget are changed by user keyboard activity, then the Tcl procedure *HandleInput* is executed when the user presses the *Enter* key. The input handler function might look like this:

```
proc HandleInput { w } {
    global MyTclVariable
    puts "The user entered $MyTclVariable"
    set MyTclVariable ""
    }
```

Here, the handler will print out the contents of the information entered by the user and then clear the *Input*widget. By initializing the contents of *MyTclVariable* to some value, the *Input*widget will display this initial value when it first appears. If *MyTclVariable* does not exist when the *Input* widget is created, then the variable will be created within the scope of the widget constructor command and initialized to the value 0.

42.2 Input Widget Commands

In addition to the standard widget commands *configure* and *cget*, the *Input* widget supports the following widget specific commands:

insert	Insert text into the widget
cut	Cut the selection from the widget to the clipboard
copy	Copy the selection to the clipboard
replace	Replace the selection with other text
copycuts	Copy cuts from the undo stack
undo	Undo previous operations
load	Load text from a file
position	Set or get the input position
mark	Set or get the mark

42.2.1 The insert command

The *insert* function command inserts text into the widget at the current insertion position. The format of the command is:

```
$w insert text
```

where *$w* is the path name of the *Input* widget to use and *text* is the text to insert.

42.2.2 The cut command

The *cut* function cuts the currently selected text from the widget and places it on the clipboard. The *cut* function command has the following format:

```
$w cut from to
```

where *$w* is the path name of the *Input* widget to use, *from* is the starting location and *to* is the ending location of the text to cut. The location values are zero based character indices into the data contained in the widget.

42.2.3 The copy command

The *copy* function will copy text from the widget to the clipboard. The format of the function command is:

```
$w copy
```

This command transfers the contents of the current selection to the clipboard.

42.2.4 The replace command

The *replace* function will replace the text in the widget with new text. The format of the function command is:

```
$w replace from to with
```

where *$w* is the path name of the widget to use, *from* is the starting location, *to* is the ending location and with is the text to be used as a replacement.

42.2.5 The copycuts command

The *copycuts* function copies previous cuts in the undo stack back into the widget. The format of the function command is:

```
$w copycuts
```

where *$w* is the widget to use.

42.2.6 The undo command

This command will undo the results of previous editing operations. The format of the command is:

```
$w undo
```

where *$w* is the path name of the widget.

42.2.7 The load command

The l*oad* function can be used to initialize an *Input* widget with the contents of a text file. The format of the command is:

```
$w load filename
```

where *$w* is the path name of the widget to be used and *filename* is the name of the text file to load. The contents of the file are read into the widget.

42.2.8 The mark command

This function is used to set the mark location in the widget. The format of the command is:

```
$w mark location
```

where *$w* is the path name of the widget to use and *location* is the place to put the mark. If location is not specified the current mark location is returned.

42.2.9 The position command

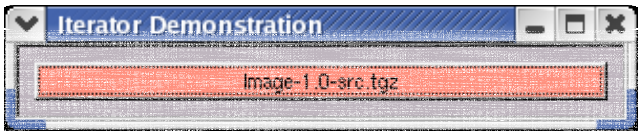
The *position* function is used to set the current input location pointer. The format of the command is:

```
$w position location
```

where *$w* is the path name of the widget to use and *location* is the location in the widget text to put the location pointer. If *location* is not specified then the current value of the input location pointer is returned.

43 Iterator − Construct a list iterator button

The *Iterator* is button that iterates through the elements of a Tcl list as it is pressed. This widget looks identical to a normal *Button* widget and implements some additional internal functionality that makes list iteration convenient. An Iterator can be configured to automatically cycle through a Tcl list, issuing associated widget command scripts at a programmable delay rate. Because the *Iterator* is a button widget, it is a member of the *Button* class.



The format of the command line that constructs an *Iterator* is:

```
Iterator path options
```

where *path* is the path name of the widget to be constructed, and *options* is the list of option and value pairs that is used to configure the widget. In addition to the list of *standard widget options*, the *Iterator* widget supports the following widget specific options:

- value − The current element of the list
- forward − If the iteration is in the forward direction
- increment − The stride to use when iterating
- first − The first element to start the iteration with
- list − The name of the Tcl list to iterate over
- length − The length of the Tcl list
- rate − The auto repeat delay in milli−second
- type − The type of the button
- indicator − If the repeat indicator is shown
- autorepeat − If auto repeat is active

The *value* option can be used to specify the current position of the iterator or to retrieve the current element of the list. When used to set the current element, the *value* option takes a string that must match an element in the list.

The *forward* option is a boolean value that determines whether the *Iterator* proceeds forward over the list or backwards over the list. By default, the value of the *forward* option is *true*, and the iteration proceeds in the direction of ascending list indices. By setting the *forward* option to *false*, the iteration will proceed in the direction of descending list indices.

The *increment* option is used to define *Iterator* increment value. By default the value of the *increment* option is *1*. The value of the *increment* option must be less than the length of the list.

The *first* option is used to specify the starting element in the list for iteration. By default the value of the *first* option is *0* and the iteration begins at th first element of the list.

The *list* option is used to specify the name of a Tcl variable that contains the list over which to iterate. By default the value of *list* is an empty string and no iteration occurs.

The *length* option is used to query the length of the list in use.

The rate option is used to specify the delay, in milli−seconds, for the repeat of a command in auto−repeat mode If there is no command associated with the Iterator widget, this option has no effect. The default value of the rate option is 0, and, when in auto−repeat mode, commands are repeated as fast as the command script can be evaluated.

The type option is used to specify the type of the button. Buttons can be of type invariant or toggle, depending on the desired behaviour of the value of the widget when the button is pressed. The default type is toggle, and each time the Iterator is pressed, the value of the value option changes between 0 and 1.

The indicator option, when true, causes an indicator to be drawn on the button that will flash when in auto−repeat mode. By default, the value of the indicator option is false, and no indicator is drawn.

The autorepeat option is used to invoke the auto−repeat mode of the the Iterator. In auto−repeat mode, pressing the button will cause the command associated with the Iterator to be invoked at an interval specified by the current value of the rate option. This is useful when, for example, building an image animation application that will loop continuously through a set of images. When the value of autorepeat is true, pressing the Iterator will cause the associated command to be invoked repeatedly until the button is pressed a second time. By default, the value of the autorepeat option is false.

Here is an example of an *Iterator* widget:

```
set list [glob *]

Iterator t.i −w 300 −list list −command "%W set −label %value" −bg tan

Show t

Wm title t "Iterator demonstration"
```

Assuming that there are files in the current directory, pressing the Iterator button will cause each of the file names in the directory to be displayed, in turn, in the widget.

43.1 Widget Specific Commands

In addition to the standard configure and cget widget commands, the Iterator widget supports the following widget specific commands:

- next − Move the next item in the list
- previous − Move to the previous item in the list
- stop − Stop an auto−repeat cycle
- current − Get or set the current loop position

The next command will cause the Iterator widget to set its position in the current list to the one following the current position. The widget command, if any, will be invoked. If the current position is the end of the list, then the command will positon to the beginning of the list. The amount of motion in the list is defined by the current value of the increment option. The format of the next command is:

```
$w next
```

where $w is the path name of the Iterator widget. The value returned by this command is the current position of the Iterator in the current list.

The previous command will cause the Iterator widget to set its position in the current list to the one previous to the current position. The widget command, if any, will be invoked. If the current position is the beginning of the list, then the command will position to the end of the list. The amount of motion is determined by the current value of the increment option. The format of the previous command is:

```
$w previous
```

where $w is the path name of the Iterator widget. The value returned by this command is the current position of the Iterator in the current list.

The stop command will stop an auto−repeat cycle. When the Iterator widget is in auto−repeat mode, pressing the widget or invoking the next or previous commands will begin an automatic repeat cycle that continuously invokes the widget command at a delay rate specified by the current value of the rate option. The auto−repeat cycle will proceed through the list in increments specified by the current value of the increment option in the appropriate direction. Invoking the stop command will arrest the auto−repeat cycle. The format of the stop command is:

```
$w stop
```

where $w is the path name of the Iterator widget. The value returned by the stop command is the current position of the Iterator in the current list.

The current command will either set or get the current position of the Iterator in the current list. The format of the current

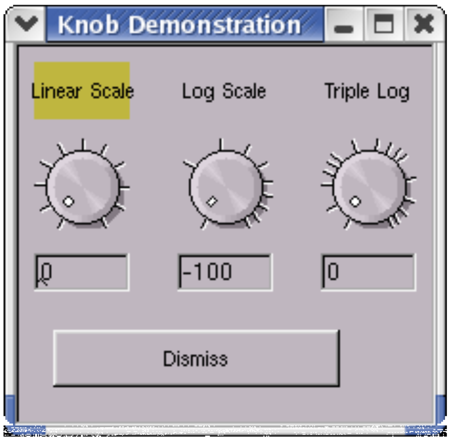
```
$w current value
```

where $w is the path name of the Iterator widget and value is an optional value that must be within the range of 0 through the length of the list minus 1. If value is specified, the current position of the Iterator is set to the specified value. If no value is specified, the command returns the current position.

command is:

44 Knob − Create a knob widget

The *Knob* command creates a widget that looks like a knob. This OpenGL based widget can be manipulated by turning it with the mouse to change its value. Several types of scales are supported.



The format of the *Knob* command is:

Knob path options

Where *path* is the path name of the widget to be created and *options* are the name and value pairs that are used to configure the widget.

In addition to the set of *standard widget options*, the *Knob* supports the following widget specific options:

value	Set or get the current value of the knob
step	The value of the step
min	The minimum angle
max	The maximum angle
knobstyle	The style of the knob
ticks	Number of ticks to draw
scale	Range of the scale to use
zero	Zero value

The *value* of the widget is the value represented by the current position of the knob indicator. The *step* value is the amount the value of the *Knob* will change when it is turned. By adjusting the *step*, the sensitivity of the *Knob* is changed.

The *min* and *max* values are angles, specified in degrees, that indicate the indicator position at which the value of the *Knob* will be minimum and maximum respectively. By default these values are *45* and *315* degrees. You can turn the *Knob* between these two positions.

The *knobstyle* specifies the type of knob indicator (*dot* or *line*) and the type of scale to be used for the knob (*linear* or *logarithmic*). By default, the *knobstyle* is set to use a *dot* for the indicator and a *linear* scale for the variation of the value. *Knobstyle* is specified by a list of comma separated elements that specify the style in the following format:

indicator,scale,range

where *indicator* can be *dot* or *line*, scale can be l*inear* or *logarithmic*, and *range* is an optional value that is used to specify the range of the *logarithmic* scale. For example, the specification:

```
−knobstyle line,log,3
```

would produce a knob with a *line* indicator and a triple *logarithmic* scale.

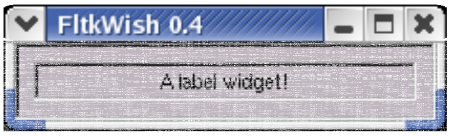
The *tick* option specifies the number of tick marks to draw around the knob. By default this value is *10*.

The *scale* value is the range of values that span the range that the *Knob* can cover. By default, the value of *scale* is *100*. The *zero* value specifies the value of the *Knob* at the zero position of the indicator. By default, the value of *zero* is *0.* Here is a *Knob* that uses a linear scale to cover the range −50 to 50:

```
Knob root.knob −zero −50
```

45 Label − Create a label widget

The *Label* widget is a simple rectangular widget that displays some text.



The format of the command is:

```
Label path options
```

where *path* is the path name of the widget to be created and *options* is the list of option and value pairs that is used to configure the widget. The widget supports the set of *standard widget options*.

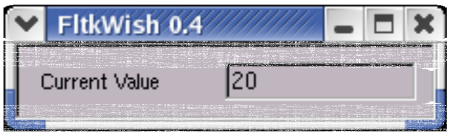
Here is an example of a Label widget:

```
Label root.label −x 20 −y 100 −text "This is a label" −width 150 −background yellow −relief raised
```

In this case, the default value of the *align* widget option is *centered*, and the value of the widget *relief* is *raised*, so the resulting widget will look like a button but without adding some other functionality, the widget will not react to mouse events. A more typical use for a *Label* widget is demonstrated by the following command:

```
set p [Package root.pack −orientation horizontal −height 20]

Label $p.label −text "Current Value" −relief flat −align left,inside
Output $p.output −relief sunken −variable MyOutputVariable
set MyOutputVariable 20
```

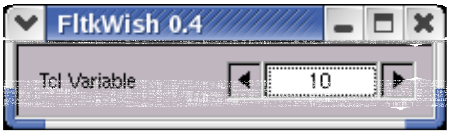
This *Label* will have the text left aligned inside of the widget and have no evident relief. It looks just like a text string. It is positioned to the left of the *Output* widget that is monitoring the Tcl variable *MyOutputVariable*. The two widgets in the *Package* combine to make a mega−widget that has a text label and a widget with a sunken relief that displays the current value of the *MyOutputVariable* variable.



See also the *LabeledText* widget, which is a compound widget that does the same sort of thing as this example.

46 LabeledCounter − Construct a labeled counter widget

The *LabeledCounter* widget contains a *Label* widget and a *Counter* widget. The counter widget can be used to adjust the values of a Tcl variable.



The format of the command line is:

LabeledCounter path options

where *path* is the path name of the widget and *options* are the option name and value pairs that are used to configure the widget. In addition to the set of *standard widget options*, the *LabeledCounter* widget supports the following list of widget specific option:

- countertextforeground Color of the counter text
- countertextbackground Background color of the counter
- countertextfont The counter font
- countertextsize Size of the counter text
- countertextrelief Relief of the counter
- countertextjustify How to justify the counter value
- counterstyle Type of counter
- orientation Orientation of the widget stack
- order Order of drawing the label and the counter
- ratio Amount of the widget window used for the counter
- pad Padding between the label and the counter
- labelrelief Relief of the label component
- step Increment value
- min Minimum value
- max Maximum value
- faststep Increment to use with the fast buttons
- value Query or set the current value of the counter

The *LabeledCounter* widget can be constructed using an *orientation* that is either *vertical* or *horizontal*. The default *orientation* is *horizontal*, so the components of the widget are side by side. The *order* option deternimes which of the widgets is drawn first, and is by default, *label,value*. This combination of default values results in a widget that has the label on the left of the counter. By default the value of *ratio* is *.5*, so both widgets have the same size. Changing the *orientation* parameter to *vertical* would cause the label component to appear above the counter component.

Changing the *order* option to *value,label* will cause the counter to be drawn before the label component. In the *horizontal* orientation, this results in the counter being drawn to the left of the label. In the *vertical* orientation, this results in the label being drawn under the counter component.

Note that when using the *vertical* orientation, the default widget *height* should be adjusted to accomodate the two widgets. By default, the value of the widget *height* is the standard height of a *Label* widget.

The *pad* option can be used to insert some padding between the label and the counter omponents. By default, the value of *pad* is *0*, so the widgets are places adjacent to each other.

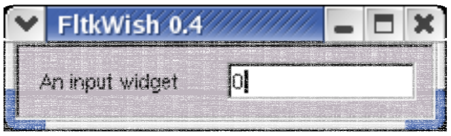
The counter appearance can be controlled using the *countertextforeground*, *countertextbackground*, *countertextfont*, *countertextsize*, *countertextrelief*, and *countertextjustify* options. The *counterstyle* option can be *simple* or *normal*. A *simple* style has only the single set of increment controls, whereas the *normal* style has additional controls for fast changing the value. The default style is *simple*.

The behaviour of the counter is defined by the *step*, *min*, *max* and *faststep* options. These options have default values of *1*, *0*, *100* and *10* respectively.

The *value* option may be used to set the current value of the counter or to query the current value. The value supplied when setting the counter must lie within the current valid range defined by the min and max values.

47 LabeledInput − Create an input box with a configurable label

The *LabeledInput* widget is a compound widget that displays a label and an input box. The input box is typically used to get some value that may become the contents of a bound Tcl variable. The label of the widget can be configured as to its justification, font, size, foreground and background color and relative position with respect to the input box.



The format of the command is:

```
LabeledInput path options
```

where *path* is the path name of the widget to be constructed and *options* is the list of option and value pairs that is used to configure the widget.

In addition to the list of *standard widget options*, the *LabeledInput* widget supports the following list of widget specific options:

value	The current value of the text box
textforeground	The foreground color of the input box
textsize	The size of the font used for the input box
textfont	The font used for the input box
textbackground	The background color used by the input box
textrelief	The relief of the input box
orientation	How the widget is oriented
spacing	The spacing between components
order	How the components are ordered
ratio	The relative sizes of the components
labelrelief	The relief of the label component

The label component of the widget will inherit its text display characteristics from the parent widget that wraps the label and input components. The text attributes, with the exception of the relief of the label component, are configured using the *standard widget options*. For example, the command:

```
LabeledInput t.t −text Hello −foreground red −background blue −value world
```

would create a widget with the label *Hello* and the value *world*. The label would be displayed in *red* on *blue*.

47.1 Input Box Configuration

The input box component is configured with the relevant options described above. By default, the input box is created with a *sunken* relief using the default font characteristics and text is always left justified in the input box. Here is an example of a widget command that would right align the contents of the input box and modify the relief to *raised*:

```
$w config −textrelief raised
```

47.2 Widget Configuration Options

The *LabeledInput* widget can be laid out according to the values of the *orientation*, *order* and *spacing* options. The default *orientation* is *horizontal*, the default *order* is *label,input*, and the default *spacing* is *2*. This results in a widget with the label to the left of the input box and a 2 pixel spacing between the components.

The *order* option determines which of the components is laid out first. By changing the *order* to *input,label*, the input box will appear to the left of the label.

The *orientation* option can be set to *vertical* as opposed to the default value of *horizontal*. This will cause the widget to be laid out with the label and input components stacked on top of each other. The default *order* will result in a widget with the label above the input box, while setting the *order* to *text,label* will result in a widget with the input box on top of the label.

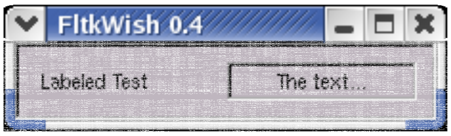
Here is an example of a *LabeledInput* widget configured to put the label on top of the input box:

```
LabeledInput t.t  −h 40 −w 100 −orientation vertical −justify centered −label Hello −value world
```

A final option of interest is the *ratio* value. By default, the *ratio* value is *0.5*, which means that the relative dimensions of the two widget components are equal. The *ratio* value actually specifies the proportion of the widget, along its *orientation* dimension, that is assigned to the input box. The value of the *ratio* can range from *0.0* to *1.0*, however, the extremes do not produce useful widgets.

48 LabeledText − Create a text box with a configurable label

The *LabeledText* widget is a compound widget that displays a label and a text box. The text box is typically used to display some value that may be the contents of a bound Tcl variable. The label of the widget can be configured as to its justification, font, size, foreground and background color and relative position with respect to the text box.



The format of the command is:

```
LabeledText path options
```

where *path* is the path name of the widget to be constructed and *options* is the list of option and value pairs that is used to configure the widget.

In addition to the list of *standard widget options*, the *LabeledText* widget supports the following list of widget specific options:

value	The current value of the text box
textforeground	The foreground color of the text box
textsize	The size of the font used for the text box
textfont	The font used for the text box
textbackground	The background color used by the text box
textrelief	The relief of the test box
textjustify	The justification used for the text box
orientation	How the widget is oriented
spacing	The spacing between components
order	How the components are ordered
ratio	The relative sizes of the components
labelrelief	The relief of the label component

The label component of the widget will inherit its text display characteristics from the parent widget that wraps the label and text components. The text attributes, with the exception of the relief of the label component, are configured using the *standard widget options*. For example, the command:

```
LabeledText t.t −text Hello −foreground red −background blue −value world
```

would create a widget with the label *Hello* and the value *world*. The label would be displayed in *red* on *blue*.

48.1 Text Box Configuration

The text box component is configured with the relevant options described above. By default, the text box is created with a *sunken* relief using the default font characteristics and text is *centered* in the text box. Here is an example of a widget command that would right align the contents of the text box and modify the relief to *raised*:

```
$w config −textrelief raised −textjustify right,inside
```

Note that since the text box is a standard widget, the contents can be displayed outside the text box, but for this particular widget such a usage is not suggested.

48.2 Widget Configuration Options

The *LabeledText* widget can be laid out according to the values of the *orientation*, *order* and *spacing* options. The default *orientation* is *horizontal*, the default *order* is *label,text*, and the default *spacing* is *2*. This results in a widget with the label to the left of the text box and a 2 pixel spacing between the components.

The *order* option determines which of the components is laid out first. By changing the *order* to *text,label*, the text box will appear to the left of the label.

The *orientation* option can be set to *vertical* as opposed to the default value of *horizontal*. This will cause the widget to be laid out with the label and text components stacked on top of each other. The default *order* will result in a widget with the label above the text box, while setting the *order* to *text,label* will result in a widget with the text box on top of the label.

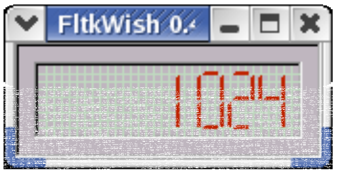
Here is an example of a *LabeledText* widget configured to put the label on top of the text box:

```
LabeledText t.t  −h 40 −w 100 −orientation vertical −justify centered −label Hello −value world
```

A final option of interest is the *ratio* value. By default, the *ratio* value is *0.5*, which means that the relative dimensions of the two widget components are equal. The *ratio* value actually specifies the proportion of the widget, along its *orientation* dimension, that is assigned to the text box. The value of the *ratio* can range from *0.0* to *1.0*, however, the extremes do not produce useful widgets.

49 Lcd − Create a Liquid Crystal Display Widget

The *Lcd* widget has the appearance of a seven segment digit display gadget. It is typically used to display the value of some integer variable in an eye catching manner.



The format of the command that constructs the *Lcd* widget is:

```
Lcd path options
```

where *path* is the path name of the widget to be constructed, and *options* is the list of option name and value pairs that is used to configure the widget.

In addition to the list of *standard widget options*, the *Lcd* widget supports the following widget specific options:

- value − The value to display
- lcdcolor − The color of the digit segments
- decimalpoint − The position of the decimal point
- barwidth − The width of the segment lines
- characters − The number of characters to display after the decimal point
- grid − If the background grid is to be displayed
- gridcolor − The color for the background grid

The *value* option is used to set the string of characters to be displayed or to get the current string being displayed.

The *lcdcolor* option is, by default, *black*. Setting this options will cause the color of the bars used to create the display digets to change to the specified color.

The *decimalpoint* option is a boolean value that specifies whether or not a decimal point is to be shown. By default, the value of the *decimalpoint* option is *false*.

The *barwidth* option is, by default, *3*. This value is the number of pixels wide the bars are drawn.

The *chracters* option is used to specify the number of decimal position that are shown. By default, the value of this option is *auto*, and the number is determined by the input value. The *decimalpoint* and *characters* options are depreciated as this version of the widget displays whatever characters are found in the *value* string, regardless of whether they are numeric or not.

The *grid* option is a boolean value that determines whether the background grid is drawn. Drawing the background grid has the effect of making the elements of the liquid crystal visible. By default, the value of this option is *true*.

The *gridcolor* option is used to set the color used for drawing the background grid. By default, this color is *gray80*.

Here is an example of the construction of the *Lcd* widget:

```
Lcd t.l −value 1024 −lcdcolor orangered3
Show t
```

50 Library − Manage the library search list

The *Library* command is used to maintain the list of binary library files that contain script modules and procedures. The format of the *Library* command is:

```
Library function file1 file2 ... filen
```

where the *files* are the names of library files and function is one of the following:

- add − Add files to the library list
- clear − Clear the library list
- delete − Delete files from the library list
- list − Get the current library list
- modules − List the modules in a library file
- procedures − List the procedures in a library file
- provides − Find a procedure or module
- source − Find the source of a procedure or module

The library list is a list of library files that will be searched by the *Call* command for script modules and procedures that are being executed. By default, the library list is empty.

When a library file is added to the library list, the index table for the library is constructed and added to the list of index tables for other libraries in the library list. When a procedure or a script module is called using the *Call* command, it is loaded from the library file and evaluated. The first time a script module or procedure is called, there is an overhead associated with loading the module or procedure and evaluating it. On subsequent calls, the module need not be reloaded ad the interpreter will have compiled its contents in the memory resident byte code representation.

50.1 Add Library Files

The format of the *add* function command is:

```
Library add file1 ... filen
```

where the *files* are the path names of the library files to be added. The current contents of the library list are extended to include the list of specified files. This command will return an error if any of the specified files can not be found, or if the file format is incorrect, such as when the *file* is not a valid library file.

50.2 Clear the Library List

The *clear* function command removes all of the files in the current library list from the list. The format of the command is:

```
Library clear
```

50.3 Delete Files from the Library List

The *delete* function command can be used to remove library files from the library list. The format of the command is:

```
Library delete file1 ... filen
```

where the *files* are the names of library files in the library list to be deleted. This command will remove any of the specified files from the current library list. If a specified *file* is not in the list, it is ignored.

50.4 List the Contents of the Library List

The *list* function command is used to display the list of library files currently in the library list. The format of the command is:

```
Library list file1 ... filen
```

where the *files* are optional library file names. If no *file* names are specified, the value returned by this command is a list that contains the names of all of the library files in the current library list. If *file* names are specified, the value returned by this command is the list of all of the specified files that are in the current library list.

50.5 List the Modules in the Library List

The *modules* function command is used to list the names of the script modules in the current library list. The format of the command is:

```
Library modules file1 ... filen
```

where the *files* are an optional list of library file names. If no *files* are specified, the result of this command is a list of all of the module names in the library files in the current library list. If *files* are specified, the result of this command is a list of all of the module names in the library files that are specified in the list of files which are in the current library list.

50.6 List the Procedures in the Library List

The *procedures* function command is used to list the names of procedures that can be found in the libraries in the current library list. The format of the command is:

```
Library procedures file1 ... filen
```

where the *files* are an optional list of library file names. If no *files* are specified, then the result of this command is a list of lists of all of the module names and the procedures that they contain in the the library files in the current library list.The elements of the list consist of the module name and the list of procedure names that the modules define.

If *files* are specified, then the result of the command is a list of lists of the module names and the procedures that they define for the library files that are specfied which are found in the current library list.

50.7 Locate a Procedure or Module

The *provides* function command is used to determine the name of the library file that contains a procedure or module. The format of the command is:

```
Library provides name1 ... namen
```

where the *names* are the name of procedures of script modules. If the procedure or script module is in one of the libraries in the current library list, then the result of this command will be the name of the library file that contains the procedure or script module. If the name is not found in the files of the current library list, the result of this command is an empty string.

50.8 Locate the Source of a Procedure or Module

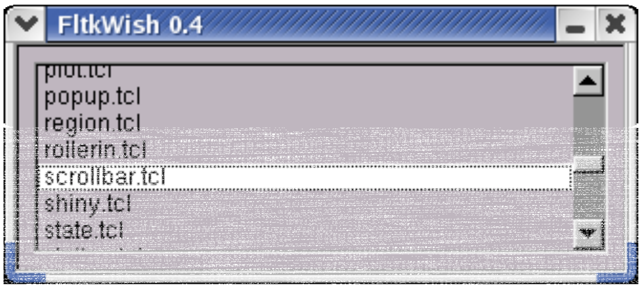
The *source* function command can be used to locate the source of a procedure or module. The format of the command is:

```
Library source name1 ... namen
```

where the *names* are the names of the procedures or modules to be located. The result of this command is a list of the original source file path names used to build the library files that contain the procedures or modules. The format of the list is a list of 2 element lists. Each element contains the library file name and a list that contains the details of the original file name used to include the specified module or procedure in the library.

51 Listbox − Create a listbox widget

The *Listbox* widget creates an object that can be used to present selections of a list of objects. The particular widget implemented in this package has a number of additional features that make possible its use for multi−column list presentation and provides for some interesting text formatting options for the list elements. Read the Fltk tool kit documentation if you want to make use of the advanced features of the widget.



The format of the command is:

Listbox path options

where *path* is the path name of the widget to be created and *options* is the list of option and value pairs used to configure the widget. In addition to the set of *standard widget options*, the *Listbox* supports the following widget specific options:

columnchar	The column separator character
columnwidth	Width of the columns
formatchar	The formatting escape character
size	Query the number of lines in the widget
separator	Characters to use for splitting items and data
topline	The display line at the top of the list
value	The currently selected item value
textcolor	Color for the text
textfont	Font for the text
textsize	Size of the font

The *columnchar* option gets or sets the current column separation character, which is by default the tab character. The *columnwidth* option gets or sets the width of the columns. It is a list of pixel widths for the columns. The *formatchar* is a character that can be used to signal embedded formatting. The default *formatchar* is the @ symbol. It is important to remember that when using embedded formatting characters with the contents of the *Listbox* widget that these formatting characters are embedded in the actual strings, so they have to be considered when doing things like searching for specific elements.

The *size* option is used to query the number of lines in the *Listbox*. The *topline* option can be used to set or get the ordinal of the line in the *Listbox* that is currently at the top of the displayed subset of list elements. The *Listbox* features automatic scrollbar management. Adjusting the topline value will cause the selected line to appear at the top of the displayed lines and to adjust the scroll position, if active, appropriately.

The *separator* option is used to set the string of characters that are used to split items and their associated data. By default, the value of this option is the equal sign (=). A string of the form *item=data* will be split at the equal sign into the item name and data components.. See the *add* function command for an example of how this is used.

The *value* option will return the current text of the selection when queried. The selection is a zero based index into the list of items in the *Listbox*. By setting the *value* you are also setting the current selection. By default, the value of *value* is the entry in the listbox whose index is *0*. When the *value* option is used to set the current selection, the *Listbox* is searched for an entry that matches the supplied option value. If such a matching entry is found, it is made to be the current selection, otherwise, the current selection remains unchanged.

If the *variable* option is specified for a *Listbox*, changing the selection with the mouse will cause the Tcl variable being used to be modified to contain the text of the new selection.

The *textcolor*, *textfont* and *textsize* options control the color of the displayed text, the font and the size of the font used. By default, the text is displayed in *black* using the *helvetica* font with size *10* characters.

51.1 Using Listbox Widgets

The *Listbox* widget can serve many purposes, common applications being the presentation of a list of items that allows the application user to select an item from the list, and the simple presentation of lines of text from a file just to allow the user to read the information using the automatic scrollbar management features of the widget.

Where the application is using a *Listbox* to present a choice to the user, it is typically bound to a Tcl variable that is to receive the currently chosen item. Optionally, the widget can also have a command that is executed each time a selection is made. For example, the command:

```
Listbox t.t −variable Choice −command { HandleChoice %W }
```

will construct a widget that is bound to the Tcl variable *Choice*. Each time the user clicks on an item in the Listbox, the value of *Choice* will be updated to reflect the current selection, and the Tcl procedure *HandleChoice* will be invoked. Here is a simple version of the *HandleChoice* procedure:

```
proc HandleChoice { w } {

     global Choice

     puts "$Choice has been chosen!"
     }
```

This procedure will just print the current choice. If the variable *Choice* does not exist when the *Listbox* is created, it will be automatically created within the scope of the widget constructor command and initialized to 0. It is a good idea to make a global variable declaration and to initialize it to an empty string when using these features of the *Listbox* widget.

51.2 Listbox Widget Commands

In addition to the standard widget commands *cget* and *configure*, the *Listbox* widget supports the following widget specific commands:

add	Add items to the listbox
clear	Empty the listbox
contains	Find an item in the listbox
count	Get the number of items in the listbox
data	Specify item data
deselect	Deselect items
hide	Hide items
insert	Insert an item into the list
load	Load the widget from a file
move	Move items in the widget
position	Set the current position
remove	Remove items from the widget
select	Select items in the widget
selected	Get the selected items
scroll	Scroll the widget

show	Show an item
text	Set the text for an item
visible	Test the visibility of an item

A possibly useful feature of the *Listbox* is the ability to associate user data with the items of the *Listbox*. In this manner, the *Listbox* can be set up to select from a collection of items based on some text labels that have no particular relationship with the data items other than to label them. What the user sees in the *Listbox* is the labels, not the data itself. The widget commands provide for the setting and retrieving data items by item index. In the implementation employed for the Fltk extension, the data items are Tcl objects of any type.

51.2.1 The Listbox add function command

This command adds items to the *Listbox* at the current insertion position. Typically the position will point to the end of the current contents of the *Listbox*. The format of the command is:

```
$w add item=data
```

where *$w* is the widget to use, *item* is a string that describes the item, and *data* is the actual data associated with the item. Supplying the *data* parameter is optional. The format of the parameter for the add function determines the contents of the Listbox. By default, the equal sign (=) is used to indicate that an item has an associated data element. By using the separator option, the default indicator can be changed according to application requirements. For example, the command:

```
$w add −separator : item:data
```

uses the colon (:) as an indicator.

The result returned by this command is the ordinal of the item in the *Listbox* list. For example, the command:

```
$w add "Item 1"="Data for Item 1"
```

will add the item at the current position and return the value of the 0 if this is the first item in the *Listbox*.

Any number of parameter may be specified on the command line. A Listbox can be loaded from a Tcl list of items using the following syntax:

```
eval { $w add } list
```

where *list* is a Tcl list that contains the elements to be added. Simple items contain no associated data, so a command of the form:

```
$w add item1 item2 item3 ... itemn
```

can be used to initialize the contents of the listbox.

51.2.2 The Listbox clear function command

The *clear* function command empties the *Listbox*. The format of the command is:

```
$w clear
```

where *$w* is the path name of the *Listbox* to clear.

51.2.3 The Listbox contains function command

The *contains* function command is used to determine the indices of items in the *Listbox* that contain text that matches the specified parameters on the command line. The format of the command is:

```
$w contains string
```

where *string* is the string to search for. The value returned by this function is a list of indices of the items in the *Listbox* that match the string. For example, the command:

```
$w contains help
```

would return the list of items in the Listbox that contain the string help. Items have indices that range from 1 through the number of items in the widget.

51.2.4 The Listbox count function command

The *count* function command returns the number of items in the *Listbox* . The format of the command is:

```
$w count
```

51.2.5 The Listbox data function command

The data function command can be used to set or to get the data associated with an item in the list. The format of the command is:

```
$w data location data
```

where *$w* is the path name of the *Listbox* to use, *location* is the ordinal of the item in the *Listbox*, and *data* is the data to associate with the item.

If *data* is not supplied, then the current data associated with the item is returned. The value of *location* must be in the range of 1 through the number of items in the *Listbox*.

51.2.6 The Listbox deselect function command

The *deselect* function command is used to clear any selection in the Listbox. The command has no parameters and has the following form:

```
$w deselect
```

where *$w* is the path name of the *Listbox* to use.

51.2.7 The Listbox hide function command

The items in a *Listbox* can be either visible or invisible. By default the items are visible. The *hide* function can be used to hide items in the *Listbox*. The form of the command is:

```
$w hide location ...
```

where *$w* is the path name of the *Listbox* to use and *location* is the ordinal of an item in the *Listbox* list. Any number of *locations* can be supplied on the command line. The values of the *locations* must be within the range valid for the contents of the *Listbox*. Once hidden, items can be made visible again using the *show* command.

51.2.8 The Listbox insert function command

The *insert* function command will insert a new item into the *Listbox* at a specified location. The format of the command is:

```
$w insert location item data
```

where *$w* is the path name of the *Listbox* to use, *location* is the ordinal at which to insert the item, *item* is the string that describes the item and *data* is the item data. The *data* argument is optional.

51.2.9 The Listbox load function command

The *load* function command will read a file and insert each line of the file into the *Listbox* as an item. The format of the command is:

```
$w load filename
```

where *$w* is the path name of the *Listbox* to use and *filename* is the name of the text file to load. The text file can contain multi−column data and embedded formatting information. Each line is added to the *Listbox* as an item.

51.2.10 The Listbox move function command

The *move* function command is used to change the location of items in the *Listbox*. The format of the command is:

```
$w move from to
```

where *$w* is the path name of the *Listbox* to be used, *from* is the current location of the item and *to* is the new location of the item. The f*rom* and *to* arguments must be within the valid range of locations for the current contents of the *Listbox*.

51.2.11 The Listbox position function command

The *position* function command gets or sets the current *position marker* in the *Listbox*. The format of the command is:

```
$w position location
```

where *$w* is the path name of the *Listbox* and *location* is the location to set the *position marker*. If the *location* argument is not specified this command returns the current location of the marker.

51.2.12 The Listbox remove function command

The *remove* function command is used to remove items from the *Listbox* list. The format of the command is:

```
$w remove location ...
```

where *$w* is the path name of the *Listbox* and *location* is one or more item ordinals to be removed. As many *locations* as desired can be supplied. The *location* values must be within the valid range of locations for the current contents of the widget. Practically this last statement means that if more than one *location* is supplied, the locations must be in descending numerical order. For example:

```
$w remove 20 14 7 2
```

will work as expected, while

```
$w remove 2 7 14 20
```

would actually remove items 2 6 12 and 17 from the *Listbox*. Only really confident script writers should use this latter approach.

51.2.13 The Listbox scroll function command

The scroll function command is used to cause the listboz to scroll so that a specified index is displayed. The format of the command is:

```
$w scroll location
```

where $w is the path name of the Listbox, and location is the location to scroll to.

51.2.14 The Listbox select function command

The *select* function command is used to mark an item or items in the *Listbox* as being selected. The format of the command is:

```
$w select location count
```

where *$w* is the path name of the *Listbox* to use and *location* is the ordinal of the item to be marked selected. If the *count* parameter is supplied it represents the number of adjacent lines to select. By default, only 1 line is selected. The *Listbox* supports multiple selections.

51.2.15 The Listbox selected function command

The *selected* function command is used to check if an item in the *Listbox* is currently selected. The format of the command is:

```
$w selected location
```

where *$w* is the path name of the *Listbox* to use and *location* is the ordinal of the item to be tested. The value returned by this command is *1* if the item is selected and *0* if it is not selected.

51.2.16 The Listbox show function command

The *show* function command is used to make hidden items visible. The format of the command is:

```
$w show location ...
```

where *$w* is the path name of the *Listbox* to use and the *locations* are the ordinals of the items to be made visible. As many *locations* as desired can be supplied, however, the values must be within the valid range for the current contents of the *Listbox*.

51.2.17 The Listbox text function command

The *text* function command is used to replace the text and data of an item in the *Listbox*. The format of the command is:

```
$w text location item data
```

where *$w* is the path name of the *Listbox* to use, *location* is the ordinal of the item to use, *item* is the new string that describes the item, and *data* is an optional data item to be associated with the item.

The location must be within the valid range for the current *Listbox* contents.

51.2.18 The Listbox visible function command

The *visible* function command is used to determine if a particular item is currently visible to the user. The format of the command is:

```
$w visible location
```

where *$w* is the path name of the *Listbox* to use and *location* is the ordinal of the item to query. If the item can be seen by the user the value returned is *1*, otherwise the value returned is *0*.

52 Menu − Create a Menu

A *Menu* is an association of labels with commands. Menus are of three types, a *menu*, a *button* or a *menubar*. The *menu* is a widget that when activated presents a list of choices. A *button* is a menu that can exist outside of any specific window. It is also known as a popup menu. A *menubar* is a menu that arranges and manages other menus. It is a container menu.

The format of the Menu command is:

```
Menu path options
```

where *path* is the path name of the widget to be created and *options* is the list of option and value pairs that are used to configure the widget. *Menus* are somewhat different from the other widgets in the Fltk tool kit in that they can be both children and top level windows. For this reason, while the path name of a menu follows the same format and structure as that of other widgets, it does not necessarily imply any parent and child relationship to the higher level components of its path.

In addition to the list of *standard widget options*, the *Menu* widget supports the following widget specific options:

disabledforeground	Foreground color when disabled
takefocus	If the menu takes the focus on activation
postcommand	Script to execute when activates
type	The type of the menu widget

The default *type* is *menubar*, and can take the alternate values *button* and *menu*. The *postcommand* option specifies a script to be executed when the menu is invoked. Note that for a *menubar*, since this is a container for other menu items, the *postcommand* script has no meaning since you can not select a *menubar* itself. The *disabledforeground* option specifies the color of the text in the menu when the menu is disabled. By default, the *disabledforeground* color is *gray*. The *takefocus* option is a boolean value that specifies whether the menu takes the focus when it is activated.

Here is an example of a standard menubar with a few items that implement some standard types of things:

```
Toplevel root −title "Menu Demonstration" −autosize false

# Create a menu that looks like a typical application menu

set Data(Menu) [set m [Menu root.menu]]

# The File menu. Note the use of the "terminator" menu item which is used
# to signal the end of a cascaded sub menu. The & signs generate the usual
# underlined letter form of a shortcut hint, along with the keyboard shortcut.

$m add cascade −label &File
$m add command −label &New −command { FileNew }
$m add command −label &Open −command { FileOpen }
$m add command −label &Save −state inactive −command { FileSave }
$m add command −label Save&As −state inactive −command { FileSaveAs }
$m add separator
$m add command −label E&xit −command { exit }
$m add terminator

# The Edit menu

$m add cascade −label &Edit
$m add command −label &Copy −command { MenuProc %W }
$m add command −label Cu&t −command { MenuProc %W }
```

```
$m add command −label &Paste −command { MenuProc %W }
$m add terminator

# A Help button

$m add command −label &Help −command { MenuProc %W }

# Display the main window

Show root
```

The above code is taken from the demonstration script *menu.tcl*. The result is a window with a standard style of menu bar across the top. The various command scripts invoke functions that are defined in the demonstration script.

52.1 Types of Menu Widgets

A *menubar* is a horizontal bar that appears at the top of its parent window and is a container for menu *items*. Each menu *item* is possibly a container for other menu items, or is a single entity which may have some associated command that is invoked when the item is selected. The standard *menubar* that appears at the top of main UI based applications has items with labels such as **File**, **Edit** and **Help**, each of which will drop down additional menu items that implement the functionality of the menu.

A *button* is a menu container that will drop down a set of menu *items* when it is pressed. *Buttons* are typically used as popup style menus, but can be used in any context. Here is an example of a menu widget that appears as a *button*:

```
    set m [Menu float.menu −type button −w 100 −h 20 −label "Floating Menu" postcommand { puts "Floating %W menu
activated" }
    $m add command −label &Continue
    $m add command −label E&xit −command Exit

    $m post
```

The above script will cause a button to appear. When the button is pressed, the drop down menu will contain 2 items, labeled Continue and Exit. The Continue item will will do nothing, while the Exit item will cause the application running the script to terminate.

A menu is a container that will drop down a set of menu items when it is selected. As with a button, it is typically used as either a popup menu or as an embedded menu in a GUI independent of the standard type of menubar at the top of the main window of the application.

52.2 Menu Widget Commands

In addition to the standard widget commands *cget* and *configure*, the *Menu* widget supports the following widget specific commands:

activate	Activate the menu
add	Add an item to the menu
clone	Duplicate an item
delete	Delete an item
entrycget	Query a menu entry
entryconfigure	Configure a menu entry
index	Get the index for a menu entry
insert	Insert a menu item

invoke	Invoke a menu item
listitems	Get a list of the items in a menu
post	Post a menu
postcascade	Post a cascaded menu
type	Get the menu type
unpost	Unpost a menu
yposition	Get the vertical location of a menu item

52.3 Script Expansion

When the script associated with a menu item is invoked, the script is first scanned for substitutable keywords that are replaced with the current values of the configurable menu item properties. The list of keywords supported is:

%W	The name of the menu entry
%E	The menu entry identifier
%s	The state of the menu entry
%v	The value of the entry
%P	The path name of the menu
%T	The type of the menu entry
%%	Expands to a % sign

52.4 Initialization of Menus

Because menus and menu entries are a just specialized type of widget, the configurable properties of the menus and the menu entries can be initialized from the Option Database in the same fashion as is done for regular widgets. The *class* option defines the class or classes associated with the menu or entry being configured.

52.4.1 The Menu activate function command

This command will activate a menu item. The format of the command is:

```
$m activate index
```

where *$m* is the token that represents the menu widget command and *index* is the menu item to activate. If no *index* is specified the current active item is deactivated, otherwise the item specified by *index* is activated. If *index* is invalid then an error message is the result.

52.4.2 The Menu delete function command

The *delete* function command is used to delete a menu item from a menu. The format of the command is:

```
$m delete index
```

where *$m* is the token that represents the menu command and *index* is the identifier of the menu item to delete. If the *index* value is valid for the menu, the matching menu item is removed from the menu. An invalid *index* results in an error message.

52.4.3 The Menu index function command

The *index* function command will search a menu for one or more text strings and return the corresponding list of menu item indices for the items that match the text strings. The indices are useful with other menu function commands, such as the *post* function and the *invoke* function.

The format of the command line is:

```
$m index str ...
```

where *$m* is a token that represents a menu widget command and *str* is a list of one or more strings that are to be located in the menu. The strings are the text that the user sees when the menu items are visible, as set by the *label* option. The result of this command is a list of menu item indices that match the strings. If there is a string in the list that is not in the menu, the result of this command is an error message.

Menu item *label* strings can contain the ampersand (&) character that is used to specify keyboard shortcuts. When searching menus for indices do not include the ampersands in the search strings. Here is an example of how to find the index of the **Edit** sub menu in the main *menubar* of an application:

```
set idx [root.menu index Edit]
```

Here, the main menu is a widget named *root.menu*

52.4.4 The Menu invoke function command

The *invoke* function command causes the action of a menu item to be invoked. Typically the menu item will be of type *command* and the result of this function command would be the execution of the associated command script. The format of the command is:

```
$m invoke index
```

where *$m* is the token that identifies the menu widget command and *index* is the index of the menu item in the menu. The result of this function command is the result of the evaluation of the associated command script, or an error message if there is an error in the command format or parameters.

52.4.5 The Menu listitems function command

The *listitems* function command is used to create a list of the items in a menu or a sub menu. The format of the command is:

```
$m listitems option parameters
```

where $*m* is the token that represents the widget command for the menu, *option* specifies the type of list being requested, and the *parameters* are values that may be required for a specific option. The value of option can be *−all*, *−index* or *−items*.

The *−all* option results in a list of all of the items in the menu. The list elements contain the names and characteristics of the items in the menu. The format of the names includes the widget name and the menu index value. Each element of the list describes one menu item.

The *−index* option can be used to list the items in a sub menu whose indices are specified as the parameters. The format of the resulting list is the same as that for the −all option but the list will only include the items in the sub menu matching the specified indices. For example, the command:

```
$m listitems −index 2 5 7
```

will produce a list of elements that describes the current configuration of menu items 2, 5 and 7. If an index is specified that is not valid for the menu then an error message is produced.

The *−items* option is used to produce a list of all of the items in a menu. Only the menu item names are returned in the list.

52.4.6 The Menu add function command

The *add* function command is used to add an item to a menu. The format of the command is:

```
$w add type options
```

where *$w* is the path name of the menu to use, *type* is the type of menu item to add and *options* is the list of option and value pairs used to configure the menu item.

The *type* parameter can have the following values:

cascade	Add a cascaded menu
checkbutton	Add a checkbutton menu entry
command	Add a command menu entry
radiobutton	Add a radiobutton menu entry
separator	Add a separator menu entry
spacer	Add a spacer menu entry
terminator	Add a terminator menu entry
invisible	Add an invisible menu entry

52.4.6.1 Cascade Items

A *cascade* menu item is the root of a sub menu that, when selected, will post the sub menu cascaded from the location of the item. *Cascade* entries are familiar from the standard menu bar that appears along the top of many GUI based application main windows. A standard *cascade* menu item in a menu bar is the **File** menu item.

52.4.6.2 Checkbutton Items

A *checkbutton* menu item appears as a text label with a checkbox that is alternatively checked or unchecked when the menu item is posted. Typically this item is associated with a Tcl variable that contains a boolean value. If the value is non zero, the item is checked, otherwise the item is unchecked.

52.4.6.3 Command Items

A *command* item has a script associated with it which will be executed when the item is selected. Command scripts are first expanded to substitute any embedded key symbols, then are evaluated. A typical *command* item is the **Help** menu entry, which usually invokes a script that displays help information about the application.

52.4.6.4 Radiobutton Items

A *radiobutton* item has an indicator that shows whether the state specific to the item is selected or not. Typically a menu will have 2 or more radio buttons. Selecting one button will cancel the selection on all of the other buttons in the menu.

52.4.6.5 Separator Items

The *separator* item is a spacing item. It creates a horizontal line between adjacent menu items. *Separator* items can not be selected. They are just a visual aid for grouping menu items.

52.4.6.6 Spacer Items

The *spacer* item is an inactive item that can be inserted into a menu to spread out the menu items. Typically the *spacer* item is used in horizontal menubars to put some distance between groups of unrelated menu items.

52.4.6.7 Terminator Items

The *terminator* item is used to indicate that a sub menu is to be terminated. When creating a menu hierarchy, the *terminator* item acts like a close brace and pops the current menu context up one level. The last sub menu in a hierarchy is automatically terminated, but good form dictates that *terminator* items should always be used.

52.4.6.8 Invisible Items

The *invisible* item is used to hide an item from the user. The invisibility of items can be changed at any time, so menus can be reconfigured to respond to application requirements.

52.4.7 Configuration of Menu Items

The menu entries in a menu can be configured using the itemconfigure function of the widget command. The format of the command is:

```
$w itemconfigure index options
```

where the path name of the *Menu* widget is contained in the variable *w*, *index* is the index that identifies the menu item to be configured, and *options* is the list of option and value pairs that are being used to configure the menu item. The *index* of a menu item is determined by the order in which the item was constructed, and ranges from 0 through the number of items in the menu. The *listitems* function of the *Menu* widget can be used to obtain a list of the menu items in a *Menu* widget.

Menu items can be configured using the following list of options:

activebackground	Background color when active
activeforeground	Foreground color when active
accelerator	Accelerator key
background	Background color
bitmap	Image to use
class	Class name of the item
command	Script to use when selected
font	Font to use
fontstype	Style of the font
fontsize	Size of the font
foreground	Foreground color
hidemargin	If margins suppressed
image	An image to use
indicatoron	If the indicator is on
label	Text to display
menu	Menu identifier
offvalue	Off value
onvalue	On value
option	Option value
selectcolor	Selection color
selectimage	Selection image
state	State of the menu item
type	Type of the menu item
underline	If text is underlined
value	Value of the item
variable	The variable associated with a menu item
width	Width of the item

Not all of the options are supported by all of the menu item types. As a general rule, menu items will inherit their characteristics from the menu widget container. For example, if the *relief* of the parent widget is *raised*, then the menu items will also display using the *raised* relief.

The itemcget function can be used to query the state of any of the configurable options of a menu item. The format of the command is:

```
$w itemcget index options
```

where w is the Tcl variable that contains the path name of the Menu widget, index is the index of the item to be queried, and options is the list of option names that are to be queried.

When menu items are constructed, the result returned by their constructor commands, if successful, is the name of a command that can also be used to configure the menu item. For example, the commands:

```
Menu t.m
set item [t.m add command]
```

will result in the Tcl variable *item* containing the command *t.m:0*, which is the command that represents the interface to item 0 of the *Menu* whose path name is *t.m*. The item command can be used to configure the item using the standard *configure* and *cget* commands typical of other widgets. For example, the background color of the above created menu item could be set with a command of the form:

```
$item set −background green
```

52.4.8 The variable option

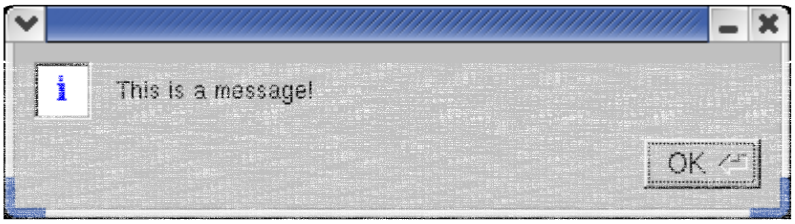
Menu items such as *checkbuttons* and *radiobuttons* can be bound to Tcl variables using the *variable* option. By default, a menu item will not have a variable binding. When a *variable* is specified, the Tcl variable should contain a value that corresponds to either the *onvalue* or the *offvalue* specified for the menu item. By default, *onvalue* is 1 and *offvalue* is 0.

When a *checkbutton* or a *radiobutton* item is selected, its indicator will change state between being logically *on* and logically *off*, depending on its state before selection. If a variable has been specified then the contents of the associated Tcl variable will be changed to reflect the state of the indicator.

For menu items that are not *checkbutton* or *radiobutton* types have no effect on their bound variables.

53 Message − Display a message

The *Message* command can be used to display a message box to the user. This command is used to present information messages which the user can acknowledge by pressing a button.


This is a message!

The format of the command is:

```
Message message
```

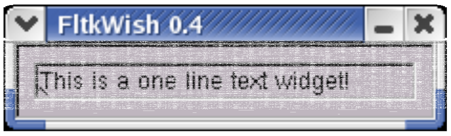
where *message* is a string to be used as the message. For example,

```
Message "Hello, world!"
```

is a way of implementing a very traditional demonstration application in a single command.

54 Output − Create a text output widget

The *Output* widget is used to display one or more lines of text. The contents of the widget can not be changed using editing operations.



This is a one line text widget!

The format of the command is:

```
Output path options
```

where *path* is the path name of the widget to be created and *options* is the list of option and value pairs used to configure the widget. In addition to the set of *standard widget options*, the following widget specific options are supported:

value	The text in the widget
textfont	The font to use
textsize	The size of the font
length	Query the amount of text in the widget

The *Output* widget is typically used to present the user with some multi−line text that has informative value but which the user is not expected to change. The widget can be initialized by sending a string of characters with embedded newline characters. For example:

```
Output root.o1 −value "Hello\nWorld!\n"
```

will produce two lines of text in the widget.

Another common use of the *Output* widget is to monitor the contents of a Tcl variable. The command:

```
Output root.o2 −variable MyOutputVar
```

will display whatever happens to be in the Tcl variable *MyOutputVar*.

55 Option − Manage the contents of the option database

The *Option* command is used to manage option values in the option database. The functions available through this command are:

add	Add an option string to the database
clear	Remove option strings from the database
get	Get option values from the database
list	List options in the database
readfile	Load options from a file
writefile	Write options to a file

The option database consists of entries that have the form:

```
pattern.name value
```

where *pattern* is a hierarchical pattern that is used to describe how an option is to be applied, *name* is the name of a configurable widget option, and *value* is the value to be used for the configurable widget option.

The *pattern* can be a widget path name, a widget class name, a global reference, or an application reference, or a combination of any of these elements. For example, an entry in the option database like:

```
Button.relief flat
```

means that all widgets of class *Button* should have a relief of *flat*.. When widgets of class *Button* are created, the option database is read and the above entry would be applied to the new widget *before* the options of the widget command are processed. If the option database has an entry:

```
button4.relief raised
```

and a widget gets created with the name *button4*, its relief will be set to *raised*, even although its class is *Button*. This is because the name of the widget is a more restrictive description than the class name, and the most restrictive description is use to initialize the default values for the applicable widgets. Where an application name is being used by setting the appropriate value as part of the application data, then the application name can be an element of the pattern. For example, the entry:

```
MyApplication.Button.relief ridge
```

will apply the ridge relief to all widgets of the *Button* class within a script that has set the application name to *MyApplication*.

The entries in the option database are applied using a scheme that proceeds from the global to the specific. The most specific formulation is the entry that gets applied to the widget. After all of the relevant options in the database have been applied, the options on the widget command line are applied.

55.1 Adding Option Database Entries

The format of the add function command is:

```
Option add key value priority
```

where *key* is the key string, or *pattern* and *name* string, for the option to be added, and *value* is the associated value expression. A *priority* value may be specified that will be used to determine the precedence of the specified value when more than one initialization possibility exists. See the discussion on the option database to understand the implications of the priority value.

55.2 Removing Option Database Entries

The format of the clear function command is:

```
Option clear −exact pattern ...
```

where the parameters are all optional. If no parameters are supplied, the contents of the option database is cleared. Any number of *pattern* strings may be supplied. If the −exact switch has been supplied then any options in the database whose keys exactly match the pattern will be removed. If the −exact option is not specified, then wild card matching is used to identify keys for deletion.

55.3 Retrieving Option Values

The format of the *get* function command is:

```
Option get pattern name
```

or

```
Option get pattern name
```

where the *pattern* values are key string patterns and the *name* is the name of the option value to retrieve. This command concatenates the patterns and name into a key using the current database key separator character which is, by default, the period. The reason for the two forms of the command is to allow the use of more exotic key formats, such as keys of the form:

```
Class*root.button*name
```

as opposed to the blander default version of:

```
Class.root.button.name
```

55.4 Listing the contents of the option database

The format of the option list function is:

```
Option list key ...
```

where zero or more *key* strings can be specified. If no *key* strings are supplied, the result of this command is a list of all of the records in the option database. If *key* strings are supplied, the result of this command is a list of the values of the database records that match the keys.

55.5 Loading the option database from a file

The readfile function command has the format:

```
Option readfile path priority
```

where *path* is the name of the file to be read and *priority* is an optional priority value that can be assigned to the options loaded from the file.

This function reads a file of option key and value pairs and inserts them into the database. If a *priority* value is specified, the added options are given the specified priority. Otherwise, the priority assigned is the same as if the options had been loaded from a script. Usually, the default priority is the one you want.

Files suitable for use with the readfile function can be created using the *writefile* function. They are text files, so you can create them by hand if you so desire.

55.6 Creating an Option File

The format of the writefile function command is:

```
Option writefile path mode
```

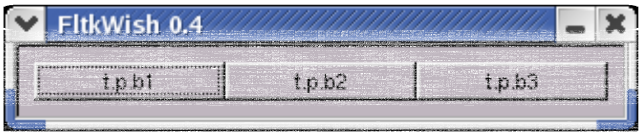
where *path* is the name of the file to be used and *mode* is an optional file mode specification. By default, if no *mode* is specified the output file overwrites any existing file, otherwise, if a *mode* is specified, the output is appended to any existing file.

The files created by this function have a format that includes a comment line, identified by the presence of a # sign in the first column of each line, and blank separated *key* and value *pairs*, one to each line. The standard comment generated contains information about when the file was generated. Files created using the *writefile* function can be read using the *readfile* function.

56 Package − Manage the geometry of widgets

The *Package* command is used to create a container widget that is helpful in the layout of collections of widgets. A *Package* is a widget that wraps a list of child widgets into a bundle, resizing the children so that they all have the same size in one of the specified dimensions, and optionally expanding the other dimension so that the *Package* is completely filled by the widgets.

A *Package* will automatically resize itself when new widgets are added. *Packages* come in 2 flavours, *horizontal* and *vertical.* A *horizontal* package packs children side by side, while a *vertical* package packs children one on top of the other.



The format of the *Package* command is:

```
Package path options
```

where *path* is the path name of the *Package* to be created and *options* are the option and value pairs used to configure the widget. In addition to the set of standard widget options, the *Package* command supports the following widget specific options:

orientation	How to layout the children
padding	How much padding between the widgets

The *orientation* is by default *horizontal*, so child widgets will be resized to the *height* of the package widget and packed side by side horizontally. By default the *padding* value is *zero*, so no additional space is inserted between the widgets.

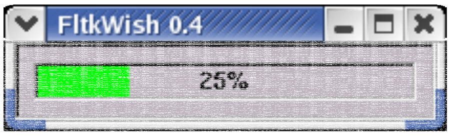
A *Package* is used to layout a set of widgets by first packing the widgets, then either packing the *Package* into another *Package*, or by using the geometry configuration options for the *Package* to put the packed widgets in the desired position. For example, the following code shows how to pack a list of buttons into a package horizontally:

```
# Create a top level widget

set root [Toplevel root]

# Create a package to hold the buttons.

Package $root.p −width 400 −height 20

# Create child buttons. They are automatically packed horizontally

Button $root.p.b1; Button $root.p.b2; Button $root.p.b3

# Place the package in the root window

$root.p configure −x 20 −y 30
```

Alternatively, another *Package* with a *vertical* orientation could have been used to pack the *Package* into a collection of *Packages* stacked on top of the other.

57 ProgressBar − Create a progress bar widget

The *ProgressBar* is a widget that is used to display the progress of some activity.



The format of the command is:

```
ProgressBar path options
```

where *path* is the path name of the widget to be created and *options* is the list of option and value pairs used to configure the widget. In addition to the set of *standard widget options*, the *ProgressBar* supports the following widget specific options:

value	The current value of the widget
indicator	The color of the indicator
troughcolor	The color of the trough
limit	The upper limit on progress
percent	If the current progress value should be displayed.

The usual case is that the *value* is initialized to *0* and that the limit is *100*. The *ProgressBar* is updated to show the progress by setting the *value* to the current amount of progress. The *indicator* is *gray* by default, and the *troughcolor* is *green* by default. The *percent* option is a binary value that determines whether the current percentage of progress is displayed. By default the value of the *percent* option is *true.*

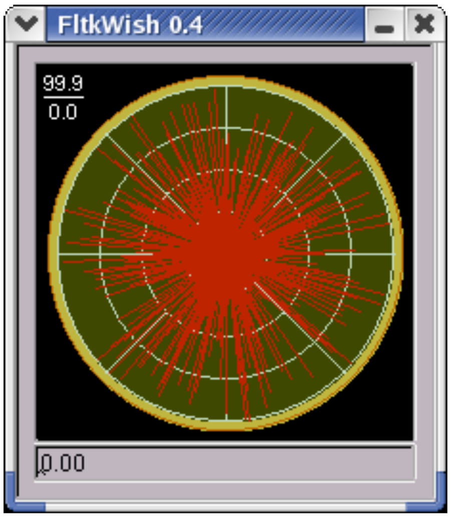
Here is a progress bar that automatically updates itself based on the contents of a Tcl variable called *MyProgress*:

```
ProgressBar −x 20 −y 50 −variable MyProgress
```

When the variable changes in the range 0 through 100, the *ProgressBar* shows what is happening.

58 RadialPlot − Create a widget to plot radial diagrams

The *RadialPlot* widget is a widget that is used to display data that is to be ploted in a polar coordinate representation. The widget displays a graph in the form of a circle on which values are prepresented in the form of radial spikes.



The format of the command that constructs the RadialPlot widget is:

```
RadialPlot path options
```

where *path* is the path name of the widget and *options* is the list of keyword and value pairs that is used to configure the widget. In addition to the set of *standard widget options*, the *RadialPlot* widget supports the following widget specific options:

- auto If automatic angle assignment is active
- autoscale If automatic range normalization is active
- drawing The drawing script to use
- grid If the background grid is plotted
- gridbackground If the grid background is drawn
- gridcolor The color of the background grid
- gridlines If the grid lines are drawn
- gridlinecolor Color for the grid lines
- gridradii If the grid radii are drawn
- logscale If the scale is logarithmic
- max The largest value to use for range normalization
- min The lowest value to use for range normalization
- plotcolor The color of the plotted radii
- sticky If range normalization is sticky
- style The style of the lines used for the plot
- value The value of the currently selected point

58.1 Automatic Plotting

The *auto* and *autoscale* options control the behaviour of the widget when automatic plotting is being used. By default, the values of these 2 options are *true*, and a series of points added to the widget will be automatically assigned an angle based on the number of points plotted, and the lengths of the plotted radii will be automatically scaled to the size of the widget's display area. When the *auto* option is *false*, points must be supplied with both an *angle* and a *value* to be plotted. When the *autoscale* option is *false*, the lengths of the plotted radii are scaled using the current values of the *max* and *min* options.

Note that setting either the *max* or the *min* option will also automatically set the *autoscale* option to *false*.

The *sticky* option controls the behaviour of the automatic range normalization function. By default, the value of the *sticky* option is *true* and as points are added to the widget, the values of the *max* and *min* options are automatically changed upwards and downwards, respectively, according to the limits of the values of the points plottted. This has the effect of maintaining a range normalization that always covers the range from the highest to the lowest values of all of the points entered for plotting.

If the *sticky* option is *false*, then the range for normalization is computed from the list of points currently being plotted, without regard for the values of any points that may have previously been plotted but which are not longer in the list of plotted points. In this mode, the length of the plotted radii will vary according to the range of values displayed in the widget.

Note that if the value of the *autoscale* option is *false*, the setting of the *sticky* option has no effect.

The *logscale* option is by default *false*. Setting this option to *true* causes the plotting of the base 10 logarithm of the plotted values instead of the values themselves.

58.2 The Background Grid

The value of the *grid* option controls whether or not a background grid is plotted by the widget. By default the value of the *grid* option is *true*, and a background grid is plotted according to the value of the other grid related options. If the value of the *grid* option is *false*, no background grid is plotted.

The background grid is a filled circle drawn using the value of the *gridcolor* option. Whether or not this filled circle is drawn is controlled by the *gridbackground* option. By default, the value of the *gridcolor* option is *darkolivegreen*, and the value of the *gridbackground* option is *true*.

The *grid* may optionally have a set of circular grid range lines and radial sector marks plotted. These features are controlled by the values of the *gridlines*, *gridradii* and *gridlinecolor* options. By default, the value of the *gridlinestrue*, and the default *gridlinecolor* value is *gray*. The default value of the *gridradii* option is *8*, which produces a grid with the octants of the circle delimited by radii. This choice of options produces something like a view of a radar screen. By setting the value of the *gridlines* option to *false*, drawing of the gridlines is suppressed. The value of the *gridradii* option may be set to any integer that defines the desired partition of the circle of the grid.

58.3 Adding Annotations

The *drawing* option has a default value of *"ht"*. The value of the *drawing* option can be set to any value Turtle Graphics script. This feature can be used to add annotations to the plot.

58.4 Displayed Values

The *plotcolor* option has the default value *orangered3*. This is the color used to display the radii on the plot that represent the values of the plotted points. The *style* option can be used to set the style of the line used to draw the radii. By default, the value of *style* is *solid*.

58.5 Selections

A single selection can be made from the currently ploted list of points in the widget. The selected point is shown as a radial spike plotted in the current *selectioncolor*. The selection can be made by using the widget *selectvalue* option is the value of the selected point. The *value* option is a read−only option, and setting the value of this option has no effect.

58.6 Widget Specific Commands

In addition to the standard widget commands *configure* and *cget*, the *RadialPlot* widget supports the following widget specific commands:

- add Add a point to the widget
- clear Clear all points from the widget
- color Set the color of points in the widget
- count Get the number of points in the widget
- hide Hide selected points in the widget
- list List points in the widget
- replace Replace the value and attributes of a point in the widget
- select Get or set the currently selected point
- show Show hidden points
- statistics Get some statistics on point values

58.6.1 Point Attributes

A point plotted in the *RadialPlot* widget has attributes which determine its location and appearance when plotted. Each point has the following attributes:

- angle The angle at which it is plotted
- color The color used to draw the point
- style The style of the line used to draw the point
- tags The list of tags associated with the point
- value The value of the point
- visible If the point is visible

The *angle* and *value* attributes define the location that a point occupies on the plot. The *angle* is specified in degrees and can have a value between *0.0* and *359.0* degrees. The *value* attribute can be any real number. To cause a plot to appear in the widget, the *value* and the *angle* must be specified. When automatic angle assignment is active, the *angle* is computed according to the order that the point is entered into the widget. The angle is computed using the formula:

```
angle = order * 360.0 / points
```

where *points* is the number of points in the widget point list. *Order* is just the ordinal of the arrival of the point in the point list.

The *color* attribute is a color used to plot the point. Its default value is the color that is the current value of the widget's *plotcolor* option. The *style* is the style of the line used to plot the point. By default, the value of *style* is the same as the current value of the widget's *style* option.

If the value of the *visible* attribute is *false*, the point will not appear on the plot. By default, the value of the *visible* attribute is *true*.

The *tags* attribute is a list of comma separated strings that are associated with a point. All points automatically acquire a *tag* that is the string representation of their *value* attribute. Additional tags can be assigned when points are inserted into the plot widget. *Tags* can be useful in searching and selecting specific points within the list of points.

58.6.2 add − Add a point to the widget

The *add* widget command is used to add points to the widget point list. The format of the *add* command is:

```
$w add −value value −angle angle −color color −style style −tags taglist −visible boolean
```

where *$w* is the widget path name to the *RadialPlot* widget, *value* is the value of the point, *angle* is the angle at which to plot the point, *color* is the color to use when drawing the point, *style* is the line style to use for the point, *taglist* is a list of comma separated strings to add to the point tag list, and *boolean* the name of a boolean value that determines whether the point will be in the visible state.

For example, the command:

```
$w add −value 100.0 −color blue −angle 45.0
```

will add a point with the value *100.0* which will be plotted at *45.0* degrees using the color *blue*.

58.6.3 clear − Clear the point list

The *clear* command empties the current list of points. This will also clear the widget display. For example, the command:

```
$w clear
```

empties the current point list for the *RadialPlot* widget whose path name is in the variable *$w*

58.6.4 color − Set the color of points

The *color* command can be used to set the color used to plot points in the point list. The general form of the *color*

```
$w color −color color options
```

where *$w* is the path name of the *RadialPlot* widget to use, *color* is the new color to be set for the affected points, and *options* are optional keyword and value pairs that can be specified to select from the list of points those to be affected.

If no *options* are specified, all of the points in the point list will be set to use the specified *color*. Alternatively, the command may specify the *−tags* option followed by a comma separated list of strings that are used to identify points in the point list with matching *tags*. The *−unique* option may also be specified to cause the selection of points to stop at the first match it finds.

58.6.5 count − Get the count of points in the point list

The *count* command returns the number of points currently in the point list. For example,

```
$w count
```

will return the point count for a widget whose path name is in the variable *$w*.

58.6.6 hide − Hide points in the point list

The *hide* command is used to slecetively hide points in the current list of plotted points. The format of the command is:

```
$w hide taglist
```

where *$w* is the path name of the *RadialPlot* widget to use and *taglist* is a comma separated list of strings that specify the tags to use in searching for points to hide. The list of points in the *RadialPlot* widget is searched for points which have tags that match one of the strings in the *taglist.* Points with a matching tag are marked hidden and they are not displayed on the widget plot.

The value of *taglist* may be *all,* in which case all of the points in the point list are hidden.

58.6.7 list − List points in the point list

The *list* command is used to display the attributes of points in the point list. The format of the command is:

```
$w list
```

where *$w* is the path name of the *RadialPlot* widget to use. The result of this command is list whose elements are lists of the attributes of the points in the point list.

58.6.8 replace − Replace points in the point list

The *replace* command is used to replace the attributes of a point in the point list with a new set of attributes. The point to be replaced is identified by either a matching *tag* list or by a matching *angle,* if no *tag* list is specified. If no matching point is found in the point list, a new point is added to the list with the specified attributes.

The format of the replace command is:

```
$w replace −value value −angle angle −color color −style style −visible boolean −tags taglist
```

where *$w* is the path name of the widget, *value* is the new value for the point, *angle,* if specified, is the angle of the point to be replaced, *style* is the new style for the line used to plot the point, *color* is the new color for the line used to plot the point, *boolean* determines the state of visibility of the point, and *taglist,* if specified, is the list of comma separated strings used to identify the point t o be replaced.

Either an *angle* or a *taglist* must be specified, along with a *value,* for the command to be successful. If a *taglist* is specified, the *angle* is ignored.

58.6.9 select − Select a point in the point list

The *select* command is used to set the selection of the *RadialPlot* widget. One or more of the points in the point list may be selected, in which case they will be displayed using the current value of the *selectioncolor*. The format of the command is:

```
$w select taglist
```

where *$w* is the path name of the widget and *taglist* is an optional list of comma separated strings that is used to identify the point or points to be selected.

If no *taglist* is specified, the result of the command is the tag list of the currently selected point or points. If a *taglist* is specified, then all points with matching tags will be selected and the remaining points marked not selected.

58.6.10 show − Show hidden points in the point list

The *show* command is used to make hidden points in the point list visible. The format of the command is:

```
$w show taglist
```

where *$w* is the path name of the widget, and *taglist* is the list of comma separated strings that are used to identify the points to be made visible on the basis of matching tags. The value of *taglist* may be *all,* in which case all of the points in the point list are made visible.

58.6.11 statistics − Get basic statistics on point values

The *statistics* command is used to retrieve some basic statistics on the list of values plotted in the *RadialPlot* widget. The format of the command is:

```
$w statistics
```

where *$w* is the path name of the widget to use. The result of this command is a list of elements in the form of *keyword=value* that hold the basic statistics of the list of values of the plotted points. The list of *keywords* returned is:

- Count The number of points in the list
- Min The lowest value in the list
- Max The highest value in the list
- Total The sum of all the values in the list
- Mean The average value of all the values in the list
- Variance The variance of values in the list.

59 Region − Create a region widget

The *Region* widget is an invisible widget used to manage events. The *Region* widget is typically constructed to cover the same area as some other widget. Events are then bound to the *Region* widget. The events are only delivered to the user script if they occur within one of the defined regions of the *Region* widget.

The *Region* widget supports all of the *standard widget options* and has no widget specific options. The format of the command is:

```
Region path options
```

Where *path* is the path name of the widget to be created and *options* is the list of option and value pairs used to configure the widget.

The *Region* widget maintains a list of regions that define the areas of the widget for which events are to be allowed. The regions can be of the following types:

box	A rectangular area
circle	A circular area.

The management of the list of regions is effected using the following widget specific commands:

add	Add a region to the region list
delete	Remove one or more regions from the list
itemcget	Retrieve the properties of a region in the list
itemconfigure	Modify the properties of a region in the region list
list	List the regions in the region list.

59.1 The Add Function

New regions are addred to the region list using a command of the form:

```
$w add type options
```

where *$w* is the path name of the *Region* to use, type is one of the valid region types, and *options* is the list of option and value pairs used to configure the region. The following option names are supported:

x	Horizontal location of the origin
y	Vertical location of the origin
width	Horizontal extent
height	Vertical extent
radius	Radius of a circle

The default values of all of these properties is 0. The value returned by the *add* function is a token which can be used to identify the region for the purposes of the other functions supported by the widget commands.

59.2 The Delete Function

The *delete* function will remove one or more regions from the region list. The format of the command is:

```
$w delete name ...
```

where *$w* is the path name of the *Region* to use and the *name* parameters are the names of the regions in the region list to delete. If no names are supplied, all of the regions in the list are deleted.

59.3 The ItemCGet Function

The *itemcget* function is used to query the properties of a region in the region list. The format of the command is:

```
$w itemcget name options
```

where *$w* is the path name of the *Region* to use, name is the token that identifies the region in the region list to be queried, and *options* is a list of the properties of the reqion to query. The value returned by this function is a list of elements that are the current values of the properties that have been queried.

59.4 The ItemConfigure Function

The *itemconfigure* function is used to set the values of the properties of regions in the region list. The format of the command is:

```
$w itemconfigure name options
```

where *$w* is the path name of the *Region* to use, name is the token that identifies the region to be configured, and options is a list of option and value pairs used to configure the region.

59.5 The List Function

The *list* function is used to produce a list of the regions that are in the region list. The format of the command is:

```
$w list
```

where *$w* is the path name of the *Region* to be used. The result of this function is a list of the tokens that identify the regions in the region list.

59.6 Box Regions

A *box* region is a rectangular area defined by its origin and extent. The following example shows how to create a box region:

```
$w add box −x 50 −y 50 −width 100 −height 30
```

which will create a region at (50,50) of dimensions 100 x 30. Events which occur insize this box will result in the invocation of any event bindings for the Region widget identified by the path name $w.

59.7 Circle Regions

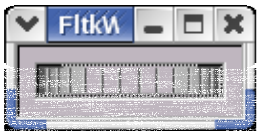
A *circle* region is a circular area defined by an *origin* and a *radius*. The following command will add a circle region to the region list:

```
$w add circle −x 100 −y 100 −radius 25
```

which will create a circular region at (100,100) of radius 25. When the mouse is inside this area and an event occurs, any event handlers bound to the *Region* with the path name $w will be invoked.

60 Roller − Create a roller widget

A *Roller* is a widget that can be used to adjust a value using a widget that looks like a thumb wheel.



The format of the command is:

```
Roller path options
```

where *path* is the path name of the widget to be created and *options* is the list of option and value pairs used to configure the widget. In addition to the set of *standard widget options* the *Roller* supports the following widget specific options:

value	The current value of the widget
step	The increment value
min	The minimum value
max	The maximum value
orientation	The orientation of the widget

The *orientation* can be *horizontal* or *vertical*. By default the *orientation* is *horizontal*. The *value* range of the Roller is set using the *min* and *max* options and is by default the range from *0* to *100*. The *step* option sets the amount of change that is made to the *value* when the *Roller* is moved. By default this value is *1*.

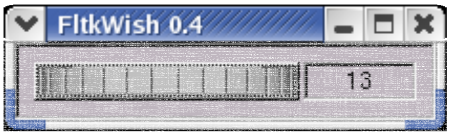
Here is an example of a *Roller* that does fine adjustments to the value of a Tcl variable named *MyRollerVar*:

```
Roller root.roller −variable MyRollerVar −min −1.0 −max 1.0 −step 0.001 −orientation vertical
```

This is a very slow moving Roller.

61 RollerInput − Create a roller input widget

A *RollerInput* is a *Roller* with a text box widget that displays the current value of the *Roller*.



The command format is:

RollerInput path options

where path is the path name of the widget to be created and *options* is the list of option and value pairs that are used to configure the widget. The *RollerInput* widget supports the set of *standard widget options*, the widget specific options that apply to a *Roller*, and the following additional widget specific options:

valuecolor	The color of the value text
textfont	The font used to display the value
textsize	The size of the display font
textbackground	The background color for the text display
textformat	The format string to use
textrelief	The relief to use for the text display
proportion	A ratio used for computing the step value
order	How the components are arranged

The *proportion* option is an advanced feature that allows refinement of the *step* value for cases where very precise stepping is required. Usually using the *step* option is sufficient.

The remaining options are used to configure how the value is displayed. The values of these options are similar to those used by other widgets to display text. All these values have reasonable defaults which you may want to change for specific applications.

The *order* option controls the layout of the components of the widget. By default, the value of the *order* option is *roller,value*. This *order* produces a widget that, when the *orientation* is *horizontal*, has the roller to the left of the text box. If the *order* is set to *value,roller*, the text box will appear to the right of the roller. Where the orientation is *vertical*, the default *order* places the roller on top of the text box, otherwise the roller will be below the text box.

Here is an example of a RollerInput widget that has the text box to the left of the roller widget:

```
RollerInput t.r −order value,roller −min 0 −max 200 −step 2 −textbackground yellow −width 200 −proportion .8
```

Here the values will range from 0 through 200 and change in a *step* of 2. The *value* will be in a box with *sunken* relief displayed in *black* on *yellow*.

62 Run − Run a binary module

The *Run* command is used to load and evaluate a file that contains a binary module created from a Tcl script file. Binary modules are created by encoding a script file using a special encoding program. Modules in binary format can not be read or modified using a standard text editor. Binary modules are typically used to create applications that are to be distributed into user environments where it is not desirable that the users modify the code.

The format of the *Run* command is:

```
Run file options −− user options
```

where *file* is the name of a file containing a binary module, *options* is the list of option and value pairs that is used to configure the decoding algorithm used to decode the binary module, and *user options* is the list of option and value pairs that is to be passed to the binary module. Note the presence of the −− in the command line. This marks the end of the option list for the decoder and the start of the option list for *user options*. When the command executes, the decoder *options* are processed to configure the decoder, and the *user options* are passed to the script module in the standard Tcl *argv* array. When execution of the module terminates, the *user options* are optionally stripped from the standard Tcl *argv* list.

The *fltkwish* interpreter uses a default file name extension of *.fltk* for binary modules. If the *file* parameter is specified without a file name extension, the *Run* command will add the extension when looking for the file in the specified file path.

The value returned by the *Run* command, if it executes successfully, is the value returned by the module when it returns. The return value will be an error message if there is an error found in the options specified for the decoder. If there are no problems with the options, then the result return is the result returned by the binary module. Here is a shell script that will execute a binary module using the interpreter:

```
#!/bin/sh
#
# −−− start.sh −−− Execute a binary module
#
echo Run $0 −exit true  −− ${1+"$@"} |  fltkwish
```

This script can be renamed to the application name, and then executed as a command line. The shell will pass the appropriate parameters from the command line to the script.

62.1 Decoder Options

The *Run* command uses a decoder that can be configured using the following options:

- deleteparms − Specifies whether to delete the user parameters on module return
- displacement − Specifies the key offset to use when decoding files
- file − Specifies the file name to load
- key − Specifies the key string to use
- keyfile − Specifies a file whose contents are to be used as a key
- mode − Specify the handling of user options
- source − Specifies whether the input file is encoded.

The *deleteparms* option is a boolean value that determines how the list of *user options* is handled. By default, the value of the *deleteparms* option is *true*, and the specified *user options* are removed from the *argv* array when the binary module returns. By setting the *deleteparms* option to *false*, the list of *user options* supplied is added to the standard *argv* list and these options remain following the return of the module.

The *displacement* option is used to specify an initial inset into the key string that is being used to decode the binary module. The default value is *0*, and the first character of the key string or key file is used. The value of the *displacement*option can be set to any positive integer. The *displacement* is useful when the same key is used repeatedly. Using different values of the *displacement* results in different decoding of the binary data.

The *file* option specifies the name of the file that contains the binary module. This option has no default, and a file name must be specified for the *Run* command to execute successfully. The file name may also be specified as the first parameter of the command line that does not begin with a minus sign (−).

For example, the following commands are equivalent:

```
        Run mymodule.fltk ....
```
and
```
        Run −file mymodule.fltk ...
```

Only one of these forms should normally be used to specify the module file name. The second form is useful where the module needs to know the name of the module file.

The *key* option is used to specify the key string to be used to decode the binary module. The key string must match the one used to encode the module. If neither the key nor the *keyfile* option is specified, the decoder will use the default key string that was used to build the interpreter.

The *keyfile* option is used to specify the name of a file whose contents are to be used to decode the binary module. This mechanism provides a means for distributing modules that are readable by only those who have access to the key file used to encode the module.

The *mode* option is used to specify how the *user options*, if any, are handled. By default, the value of the *mode* option is *append* and any *user options* are appened to the current *argv* list. If the value of the *mode* option is set to *push*, then the *user options* completely replace the standard *argv* list while the module is running. When the module returns, the standard options are restored.

The *source* option is a binary value that specifies whether the module file should be decoded, or processed directly. By default, the value of the *source* option is *false*, and the module file will be decoded. By setting the value of the *source* to *true*, the Run command can be used to load and evaluate normal Tcl script files. This will make the *Run* command identical to the Tcl *source* command, with the additional feature of being able to specify user parameters when the script is invoked. Here is an example:

```
        Run myfile.tcl −source true −− −p1 myvalue1 ...
```

Here, the script myfile.tcl is an un−encoded Tcl script file which can access the user parameter list.

62.2 Encoding Binary Module Files

The distribution contains 2 programs, *key* and *encode*. The *key* program can be used to generate keys and key files of arbitrary length using a command of the form:

```
        key 1024 >mykey.txt
```

This command will produce a 1024 byte key string and write it to the file *mykey.txt*. The default key length is 256 bytes.

A Tcl script file can be encoded using the command:

```
        cat script.tcl | encode −key KEY −c 1 >mymodule.fltk
```

where *script.tcl* is the Tcl script file to encode, *KEY* is the key to use, and *mymodule.fltk* is the resulting binary module file. The *−c* option indicates that the encoding program should generate an embedded check sum value. This value is required when using encoded files with the *Run* command.

Note that the use of binary modules will not protect source code from undesired use. The encoding mechanism is designed to simply make it inconvenient to modify applications by users. The nature of the design of the Tcl interpreter, and its extensive introspection tools, means that the source is easily available to knowledgeable users. The encoding mechanism used for this implementation is not a serious encryption technique, and should not be relied upon for the safeguard of sensitive information.

63 Scheme − Specify the widget rendering scheme

The rendering scheme for the widgets can be set to one of the supported schemes. For the current release of the package, the following schemes are provided:

normal	Standard Fltk widget rendering scheme
shiny	A rendering scheme based on OpenGL
gradient	A rendering scheme that uses various color gradient effects.
image	A scheme that uses an image for widget backgrounds
plastic	A scheme that provides a modern plastic look to widgets
modern	A scheme that is similar to the plastic scheme without a background image pattern
skins	A scheme that applies patterns as skins to widgets

The default scheme is *normal*, and widgets are rendered in a manner that gives them a traditional look, similar to the widgets that appear as part of, for example, the Windows operating system. The *shiny* scheme uses OpenGL to produce widgets that have the appearance of being rendered in a shiny chrome material. It is visually interesting, but can be slower in its responsiveness. The *gradient* scheme implements a number of shading methods to produce a variety of visual effects. The *image* scheme makes use of a background image, which may be tiled, to render the backgrounds of all of the widgets in use. The *plastic* scheme is part of the standard Fltk release package and delivers a novel appearance to widgets. The *skins* scheme uses pattern generation to produce a wide variety of different widget appearances.

The desired scheme can also be specified using the FLTK_SCHEME environment variable. This variable can be set to the name of one of the provided schemes and that scheme will be invoked with default parameter settings. After the scheme is invoked, adjustments can be made within scripts by using the *configure* sub−command. Note that the *image* scheme requires the specification of a background image, and there is no default image.

The format of the command is:

```
Scheme scheme options
```

where *scheme* is the name of a scheme and *options* are scheme specific options that can be used to configure the scheme. The following options are supported by all schemes is:

foreground	The foreground color
background	The background color
selectioncolor	The selection color
name	The name of the scheme

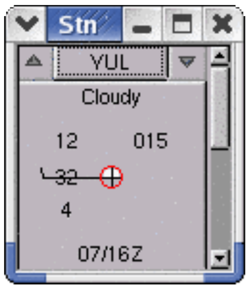
When the *Scheme* command is used to establish the current rendering scheme, all widgets will automatically inherit the scheme properties. Generally speaking, the scheme should be set before the first widget is created. Individual widgets can then be *configured* according to taste afterwards using the widget *configure* function. If the scheme is set after widgets are created, any widget specific configuration done on existing widgets will be lost and the properties specified for the scheme will be propagated to the widgets.

The *background* color specified for a scheme is used to control the rendering of widgets. If a widget background color matches the scheme background color, the scheme is used to render the widget. Otherwise, the widget is rendered according to its configured properties. This allows applications to suppress scheme rendering for individual widgets by simply specifying a different background color for the widget. For example,

```
Scheme gradient −bg blue

Label t.label −bg green
Label t.another ....

Show t
```

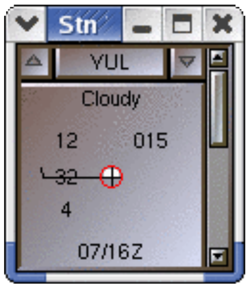
This set of commands will cause the widget *t.label* to be displayed using the normal scheme with a *green* background, while the widget *t.another* will inherit the scheme properties.

63.1 The normal scheme



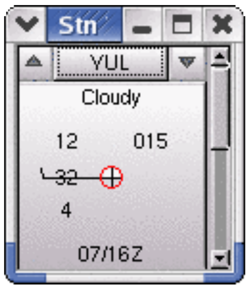
The *normal* scheme will also process the *borderwidth* option, however, the effect of this option is to change the style of the *raised* and *sunken* relief on widgets. The default *borderwidth* is 2. A value of 3 will result in the original FLTK widget look, while a value of 1 will produce an effect similar to using thin relief styles.

63.2 The shiny scheme



The *shiny* scheme is used to produce widgets that are rendered using OpenGL. In addition to the standard scheme options, the *shiny* scheme supports the *borderwidth* option. By default the *borderwidth* option has a value of 2. Widgets rendered using the *shiny* scheme have the appearance of polished metal.

63.3 The gradient scheme



The *gradient* scheme uses a number of algortihms to blend 2 colors to create the background for widgets. In addition to the standard list of scheme parameters, the *gradient* scheme supports the following options:

borderwidth	Sets the width of the border
primary	The primary color to use
secondary	The secondary color to use
type	The type of gradient function to use
scatter	A boolean value indicating whether random scattering should be used
ratio	A scale factor

The default *borderwidth* is 2. The *gradient* scheme works by drawing a widget background by blending the *primary* and *secondary* colors of the scheme according to some function of the location of a pixel in the widget. The type of function is specified by the *type* parameter. The *scatter* and *ratio* parameters control how random purturbations are applied to the blending function

For most widgets, the value of the *primary* color is determined by the widget's *background* color. A default value of *gray* is provided for widgets that do not use the *background* option of the *standard set of widget options*. The default value of the *secondary* color is *white*.

By default, the value of *scatter* is *false*, and the computed blending factor is not scattered by applying a random purturbation. If the value of *scatter* is true, the blending factor is purturbed. By default, the value of *ratio* is 0.1. The *ratio* parameter specifies the percentage of the range of a value that should be used for purturbations. Depending on the operation applied to compute the blending factor, the value that is scaled by the *ratio* parameter will be distance, color separation or some other suitable value.

The *type* value specifies the type of function applied to compute the blending factor. Here is a list of the currently available type names:

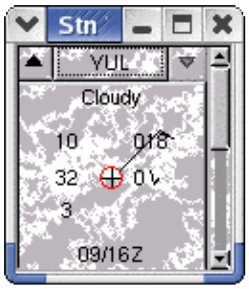
diagonal	Square law from top left to bottom right
slope	Square law from bottom left to top right
down	Linear from top to bottom
up	Linear from bottom to top
left	Linear from left to right
right	Linear from right to left
random	Completely random noise
convex	Downward bleeding effect
concave	Upward bleeding effect
inside	Quadratic
outside	Quadratic
mound	Radial square law yielding a raised impression
pothole	Radial square law yielding a sunken impression
walk	Random walk between colors
marble	Simulated marble surface effect

The default value of the *type* parameter is *convex*. This scheme will produce an appearance of a convex surface on a CRT, while producing the appearance of fine quality note paper on a flat panel display.

63.4 The skins scheme

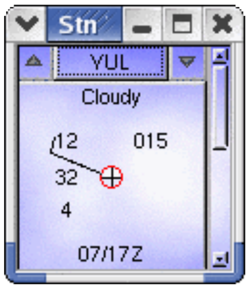
The *skins* scheme is similar to the *gradient* scheme but is implemented by generating an image of the gradient pattern to be used and then applying it to the widgets in the same fashion as the *image* scheme works. This scheme can be somewhat faster in rendering the display on slower computers as the pattern is computed only once and then reused for each widget.

The *skins* scheme takes the same parametes as does the *gradient* scheme, and also supports the *mode* parameter used by the *image* scheme to determine how the generated image is applied to widgets.



Here is an example of the *skins* scheme being used to produce an effect that looks like the widgets are made of marble. The nature of the marble effect can be controlled through the use of the *thump* and *scatter* scheme parameters. Different marble effects can be achieved through the appropriate selection of the *primary* and *secondary* colors used.

63.5 The image scheme



The *image* scheme makes use of an image as the background for the widgets in a widget tree. In addition to the standard list of scheme parameters, the *image* scheme supports the following options:

file	The file name of the image to use
borderwidth	The width of the border to use
mode	The mode of the scheme
x_inset,y_inset	The insets into the image to use.

The *file* can have an image in any of the file formats supported by the extension package.

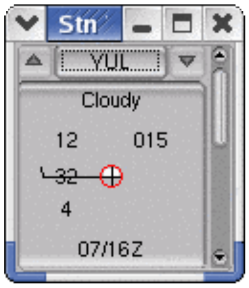
If no *file* name is specified, the *image* scheme will produce widgets that have a background according to the color specified for the background of the widgets in question. If a *file* is specified, the area of the image that covers the area of a widget will be used to draw the background of the widget. This causes the appearance of widgets to be as if the image was wallpapered over them.

The application of the image to the widgets can be done in one of two ways as determined by the setting of the *mode* parameter. By default the *mode* parameter is *widget*, and the image is applied to each widget independantly. If the mode is set to *window*, then the image is applied such that the area of the image that corresponds to the area of the widget within its containing window is used. The *window* mode is suitable for GUIs that are
not built up inside of *Scrolls*. The *widget* mode is suitable for GUIs that are built up inside of a *Scroll*.

By default, the value of the *borderwidth* is 2.

The *x_inset* and *y_inset* options can be used to select the origin in the image to use when drawing the widget backgrounds. By default, the values of the *x_inset* and *y_inset* options are *0*, and the upper left hand corner of the image is used as the origin. It may be useful to adjust the values of the *x_inset* and *y_inset* options when centering a portion of the image inside a widget.

63.6 The plastic and modern schemes



The *plastic* scheme is a scheme that is provided as part of the Fltk 1.1.x release. It delivers a modern look to the widgets that is somewhat reminiscent of plastic objects. It has no scheme specific configurable parameters. The *modern* scheme is a slightly modified version of the *plastic* scheme which renders a little faster as it does not use a background pattern image. Both schemes have the plastic widget rendering theme..

63.7 Configuration of schemes

The *Scheme* command supports the following additional functions:

configure	Used to set scheme properties
cget	Used to query scheme properties
set	Used to change the scheme.

The format of these commands is:

```
Scheme function options
```

where *function* is one of the functions, and *options* is either a list of option and value pairs to be set, or a list of option names to be queried. The *configure* and *cget* function commands can be used to change the configurable options of the currently installed scheme. When the options for an installed scheme are changed, all of the currently displayed widgets will change to reflect the new configuration. The *set* function command is used to change the current scheme. For example, the type of gradient scheme could be changed using a command like:

```
Scheme configure −type diagonal −primary red −secondary blue
```

while the current type of gradient scheme can be discovers using a command of the form:

```
Scheme cget −type
```

If the FLTK_SCHEME environment variable has been set, then the specified scheme could be loaded using a command of the form:

```
Scheme $env(FLTK_SCHEME)
```

or

```
Scheme set −name $env(FLTK_SCHEME)
```

64 Screen − Get the current screen geometry

The Screen command will return a list of values that describes the geometry of the current display screen. The format of the command is:

```
Screen
```

The returned value is a list of 4 elements that have the following meanings:

```
x y width height
```

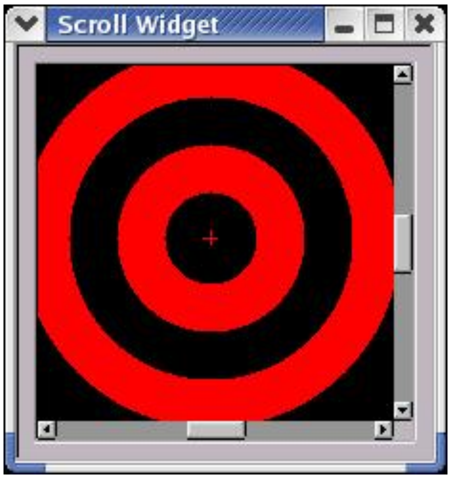
where:

- x − The current screen x offset
- y − The curren screen y offset
- width − The current screen width in pixels
- height − The current screen height in pixels

Typically, the values of x and y are zero.

65 Scroll − Create a scrollable container widget

The *Scroll* widget is a container that provides automatic scrolling of its client area. Scrollbars are created automatically according to the relationship between the size of the *Scroll* widget and the size of the items contained in the *Scroll* container.



The format of the *Scroll* command is:

```
Scroll path options
```

where *path* is the path name of the widget to be created and *options* is the list of option and value pairs that is used to configure the widget. In addition to the standard set of widget options, the *Scroll* supports the following widget specific options:

configuration	Specify the widget scrollbar configuration
scrollbars	Specify the state of the scrollbars
xposition	Specify the position of the horizontal scrollbar
yposition	Specify the position of the vertical scrollbar
position	Query the position of the scrollbars
xstep	Specifies the amount to move the horizontal scroll bar
ystep	Specifies the amount to move the vertical scroll bar

The *configuration* option allows the positioning of the scrollbars either on the traditional right and bottom locations of the widget or on the top and right of the widget. The *configuration* is specified by a set of comma delimited keywords from the list *top*, *left*, *bottom* and *right*. For example, to set up a *Scroll* with scrollbars along the top and right, use the command:

```
Scroll root.scroll −configuration top,right ...
```

Only 2 scrollbars are available, so the configuration options will only select a valid configuration. By default, the configuration is *right,bottom*.

The *scrollbars* option defines how and when scrollbars are displayed. The possible option values are:

horizontal	Show the horizontal scrollbar
vertical	Show the vertical scrollbar
both	Show both scrollbars
always_horizontal	Always show the horizontal scrollbar
always_vertical	Always show the vertical scrollbar
always_both	Always show both scrollbars
none	Never show scrollbars

The default value is *both*, which means that the scrollbars appear on both axes as needed. The widget will still scroll if *none* is specified by dragging the mouse in the direction of desired panning.

The *xposition* and *yposition* options can be used to adjust the scroll position of the axes. The default value for these options is (0,0). The *position* option can be used to query the current scrollbar positions.

By default, the horizontal and vertical scroll bars will move the scroll position by 1 unit when the buttons on the ends of the scrollbars are pressed using the mouse. If the *xstep* and *ystep* values are specified, these values become the size of the scroll position motion when a button is pressed.

65.1 Adding widgets to a Scroll

Widgets are added to a *Scroll* container by creating children of the container widget. For example, consider a *Scroll* widget created with the following command:

```
set s [Scroll root.scroll −width 200 −height 200]
```

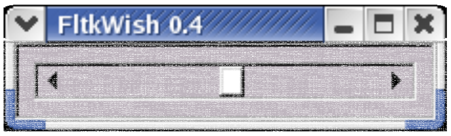
which results in a widget with a client area of 200 x 200 pixels. Now add an *Image* widget that has a larger image displayed in it:

```
set i [Image $s.i −file images/ashley.gif −width 400 −height 400 −centered yes]
```

The result will be a *Scroll* with scrollbar along the bottom and right which can be used to pan across the larger *Image* widget which has a picture of Ashley at its center. Any number of children could be added to the *Scroll*. Complex mega−widgets can be created using *Scroll* widgets that contain *Package* widgets that contain collections of other types of widgets.

66 Scrollbar − Create a scroll bar widget

The *Scrollbar* command creates a standard scrollbar widget. Using this widget one can construct mega−widgets that have scrollable client areas, however, the typical use is to adjust the value of a Tcl variable. The *Scoll* container provides fully automatic scrollable client areas, and is a simpler approach than using the *Scrollbar*.



The format of the command is:

```
Scrollbar path options
```

where *path* is the path name of the widget to be created and *options* is the list of option and value pairs used to configure the widget. In addition to the set of *standard widget options*, the *Scrollbar* supports the following set of widget specific options:

value	Specify the current value of the widget
step	Specify the step increment
min	Specify the minimum value
max	Specify the maximum value
orientation	Specify the orientation of the widget
sliderstyle	Specify the style of the slider
size	Specify the size of the widget
sliderrelief	Specify the relief of the slider

The *value* is the current position of the slider in relation to the *min* and *max* values. The *step* value is the size of the change caused by pressing the buttons at the end of the scrollbar, as opposed to dragging the slider. By default the *min* and *max* values are *0* and *100*, and the *step* value is *5*. Floating point values can be specified for the range and the step.

The *orientation* can be *vertical* or *horizontal*. The default is *horizontal*.

The *sliderstyle* option can be *normal*, *filled* or *nice*. The default is *normal*. The different styles present some fancy visual effects.

The *sliderrelief* is the relief used for the slider when it is active. Any relief can be specified. The *size* option specifies the pixel size of the slider along the specified orientation.

Scrollbars are usually needed to adjust values, and are typically tied to some Tcl variable. Here is a command that creates a slider that will change the contents of the variable *myvar*:

```
Scrollbar root.sb1 −variable myvar −min 0.1 −max 1.0 −step 0.05 −value 0.5
```

If you want to scroll some other widget or collection of widgets, use the *Scroll* container instead of building your own facility.

h

67 Show − Show one or more windows

The *Show* command is used to make one or more windows visible. There is a distinction between a window and a widget in Fltk. Windows are a special case of a widget and can be managed by the windows manager on the target computer system.

The format of the command is:

```
Show options path names ,,,
```

where *options* is a list of options for the window display mode and *path names* are the path names of the widgets to be made visible. Only windows support display mode options. If a container widget is made visible, all of its children are also made visible.

The list of options available is:

center − Center the window in the current display
display The display to use for the window
dnd − Allow drag and drop
nodnd − Diable drag and drop
kbd − Allow keyboard input
nokbd − Disable keyboard input
tooltips − Allow tool tips
notooltips − Disable tool tips
title The window title to use
fg The foreground color
bg The background color
geometry The window geometry
bg2 The second background color
name The window name
iconic If the window is iconic
x,y Specify the location of the window
w,h Specify the dimensions of the window

These *options* are related to the equivalent options implemented by the X Windows toolkit and by the FLTK library. The X Windows options are mostly useful for applications running on UNIX systems that support the X Windows GUI interface. The color options *fg*,*bg* and *bg2* are supported on all platforms, as are the *geometry*, *title* and i*conic* options. The FLTK library options *dnd*,*nodnd*,*kbd*,*nokbd*,*tooltips* and *notooltips* toggle features of the library. The *center* option, by default *false*, will center the window on the current display screen. The *x*,*y*,*w* and *h* options are just an easier way to specify the geometry of the widtget. The *geometry* option can also be used, but its format is slightly obscure to non−UNIX users.

Where multiple windows are being processed, the geometry options can be applied severally or uniquely to the windows. For example, the command:

```
Show −x 10 −y 30 t −center u −x 500 −y 30 v
```

would put the window whose path is t at (10,30), while the window whose path is u will be centered on the screen, and the window whose path is v will be placed at (500,30) on the screen. The window can be centered at a specific height on the screen by using a command of the form:

```
Show −y 200 −centered true t
```

or centered vertically at a particular inset using a command of the form:

```
Show −x 300 −centered true t
```

The *scheme* option is, in this instance, related to the FLTK library, and is not the same as the Scheme command implemented as part of this extension. Instead of using the *scheme* option, users should use the *Scheme* command of the extension. The FLTK library may implement schemes not supported by the Tcl extension, and vice versa.

Usually, the only widgets that need to make use of the Show command are the *Toplevel* widgets that are the containers of all other application widgets. When widgets are constructed, they are invisible. Typically an application will construct its widgets, then use *Show* to display them all at once. Any widget that is constructed inside a visible container is automatically made visible.

The *Hide* command can be used to hide existing widgets. Hiding a widget also hides all of its children.

68 Signal − Signal an Event

The *Signal* command is used to construct events and cause the event handlers associated with a widget to be invoked. *Signal* can be used to simulate standard mouse and keyboard events, and to cause event handlers bound to user defined events to be invoked.

The format of the command is:

```
Signal path event options
```

where *path* is the path name of the widget to receive the event, *event* is the name of an event and *options* is a list of option and value pairs that are used to configure the properties of the *event*. Events have the following list of configurable properties:

x	Window relative horizontal location of the event
y	Window relative vertical location of the event
sx	Screen relative horizontal location of the event
sy	Screen relative vertical location of the event
button	Name of the mouse button
buttonstate	State of the mouse button
key	Name of the key
keystate	State of the keyboard

Depending on the desired results, the configurable event properties generally must all be set.

The *x* and *y* properties are window relative locations for the event. The window in question is the window that is the containing parent of the widget. You can discover this window using the *Winfo* command. The *sx* and *sy* values are the screen relative, or absolute screen location coordinates of the event.

The *button* name can be *left*, *middle* or *right*. The *buttonstate* may be *up* or *down*. The *key* property is the name of the key. For the alphanumeric keys, the name is just the letter or number of the key. The usual names of the extended keys, such as *Pg Up*, *Home* and *Escape*, apply to the non alphanumeric keys. The *keystate* property describes the state of the keyboard when the key is pressed. The state is a combination of flags that have the names:

shift	Shift key presses
control	Control key pressed
alt	Alt key pressed
caps	Capitals
numlock	Number lock set
scrolllock	Scroll lock set

The state of the keyboard is defined by a comma separated list of *keystate* names.

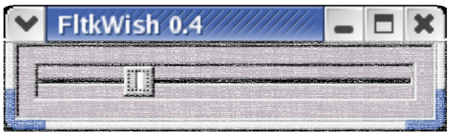
Here is an example of how to generate the <Motion> event for a widget:

```
Signal $w <Motion> −x 20 −y 40
```

where *$w* is the path name of the widget and (20,40) is the window relative coordinate of the event. If the widget has an event handler bound to the <Motion> event, the above command will invoke it.

69 Slider − Create a slider widget

A *Slider* is a scrollbar style of widget that can be used to change the value of a variable. *Sliders* have a somewhat less elaborate appearance than do scrollbars.



The format of the command is:

```
Slider path options
```

where *path* is the path name of the widget to be created and *options* is the list of option and value pairs used to configure the widget. In addition to the set of *standard widget options*, the following widget specific options are supported:

value	The current value of the slider
step	The increment size to use
min	The minimum value
max	The maximum value
orientation	The widget orientation
sliderstyle	Style options for the slider
size	Pixel length of the slider
sliderrelief	Relief of the slider
format	A format code for the displayed value

The *orientation* can have the values *vertical* or *horizontal*. By default, the *orientation* is *horizontal*. The *sliderstyle* option can take the values *normal, filled* or *nice*. By default, the *sliderstyle* is normal.

The range of the slider is set by the *min* and *max* options which have default values of *0* and *100*. The slider *step* value is by default *1*. Changing the *step* value affects the resolution of slider movements. The *sliderrelief* is the relief of the actual slider itself and can accept any of the relief values supported by the extension package.

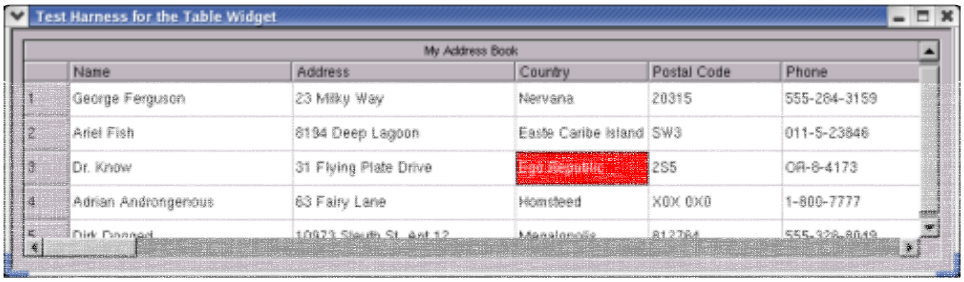
The *format* option is used to set the type of format specifier used to display the slider value for sliders that have this feature. The option can be *integer*, *float* or *general*. The default format is *integer.*

Here is the command that creates a slider that controls the value of a Tcl variable names *MySliderVar*:

```
Slider root.s −min 10 −max 40 −step 0.1 −variable MySliderVar −format float −size 150
```

70 Table − Create a table of items

The *Table* widget is used to data in tabular form. *Table* widgets look like spreadsheet pages.



My Address Book

	Name	Address	Country	Postal Code	Phone
1	George Ferguson	23 Milky Way	Nervana	20315	555-284-3159
2	Ariel Fish	8194 Deep Lagoon	Easte Caribe Island	SW3	011-5-23846
3	Dr. Know	31 Flying Plate Drive	Ego Republic	2S5	OR-8-4173
4	Adrian Androngenous	63 Fairy Lane	Homsteed	X0X 0X0	1-800-7777
5	Dirk Dagged	10973 Sleuth St. Apt 12	Megalopolis	812784	555-328-8049

The format of the command that creates a *Table* widget is:

Table path options

where *path* is the path name of the widget to be created and *options* is the list of option and value pairs that is used to configure the widget. In addition to the set of *standard widget options*, the *Table* widget supports the following widget specific options:

columns	Number of columns in the table
column_widths	Widths of the columns in the table
columns_resizeable	If the columns can be resized
features	List of table features
rows	Number of rows in the table
row_heights	Height of the rows
rows_resizeable	If the rows are resizeable
value	A write only option!

70.1 Features

Table widgets can be configured using a comma separated list of features that describe the appearance of the table. Here is the list of feature names:

none	A plain table that displays the data
row_header	The rows have a prefix title
row_footer	The rows have a suffix title
row_divider	The rows have a divider line
column_header	The columns have a header
column_footer	The columnd have a footer
column_divider	The columns are divided
multi−select	Rows and column selection available
row_select	Row selection available
col_select	Column selection available
persist_select	Persistent selection
full_resize	Resize allowed on rows and columns
dividers	Full grid dividers
headers	Headers on

footers	Footers on
column_ends	Headers and footers on columns
row_ends	Headers and footers on rows
row_all	Everything for rows
column_all	Everything for columns

Some experimentation with the features is warranted in order to gain familiarity with their effects on the appearance and behaviour of the *Table* widget. A specific configuration is established by using a command of the form:

```
$w configure −features list
```

where *$w* is the path name of the *Table* widget and *list* is a comma separated list of the feature names. For example, the command:

```
$w configure −features full_resize,headers,footers,dividers
```

will produce a widget with row and column headers and footers, rows and columns resizeable using the mouse, and a set of grid lines dividing the rows and columns of the table. The default configuration of the *Table* widget is *dividers,row_header,column_header*.

70.2 Cell Styles

There are a number of features of the table cells than can be configured to change the appearance of the cells. Cells typically are used to hold the contents of elements of a Tcl array variable, and as such, they are text strings. The appearance of the cells is governed by the following list of style options:

background	The background color
foreround	The foreground colot
relief	The cell relief
alignment	How text is justified
font	The font being used
fontsize	The font size being used
fontstyle	The style of the font
locked	If a cell is locked
resizeable	If a cell is resizable
bordercolor	Color of the border
borderspacing	Space for the border
width	Cell width
height	Cell height
value	Contents of the cell
padx	Internal horizontal padding
pady	Internal vertical padding

The style of a cell or a group of cells is managed with the *getstyle* and *setstyle* widget commands.

70.3 Tcl Variables and the Table Widget

The Table widget is useful for the display of 2 dimensional arrays of values. Tcl provides the array type variable that is a convenient way of arranging data in 2 dimensional arrays through the use of indices. The Table widget can be bound to a Tcl array that uses a specific index format, causing the Table widget to display the contents of the Tcl array.

The appropriate index format for the Tcl array is that of a pair of integer indices separated with a comma. For example, the statement:

```
set MyArray(10,4) "Something to display"
```

could be used to cause the Table widget to display the string "Something to display" in the cell located at row 10 and column 4. The Table widget associates with the Tcl variable through the mechanism of the variable widget configuration option. Here is one method of forming the association:

```
$w configure −variable MyArray
```

where $w is the path name of the Table widget and MyArray is the name of the Tcl array to use for the cell contents. The association created is symmetric, so any changes to the Table widget caused by, for example, the editing of a cell, will be reflected in the contents of the associated Tcl array element, and any changes in the contents of the array element will be reflected in the displayed cells of the Table widget.

The *Table* widget handles the end cases of, for example, a cell having an association with a Tcl array element that does not exist, by displaying nothing, or by creating the needed variable should the cell be modified through user editing operations.

Cells are specified by a pair of integers that range from −2 to rows + 1 for the row index and −2 to cols + 1 for the column index. The ranges 0 through rows − 1 and 0 through cols − 1 refer to the actual cells, while the indices −2 and either row + 1 or col + 1 refer to the header and footer titles of the table, while the values −1 and either rows or cols refer to specific row and column header and footers. The use of these extended range indices depends on the configured features of the Table widget. Where the appropriate feature is enabled, the widget will make use of the contents of the bound Tcl variable to fill in the appropriate feature, otherwise, these Tcl array elements will be ignored.

70.4 Widget Commands

In addition to the standard *cget* and *configure* widget commands, the *Table* widget supports the following widget specific commands:

getstyle	Get the style of a Table element
setstyle	Set the style of a Table element

The format of the commands is:

```
$w function type options
```

where $w is the path name of the *Table* widget, *function* is either *getstyle* or *setstyle*, *type* is the name of the style specification to act on, and *options* is the list of option names to be either set or queried for the style. The available style types are as follows:

global	Style elements that affect all cells
row	Style elements that affect cells in a row
column	Style elements that affect cells in a column
header	Style elements that affect header cells
footer	Style elements that affect footer cells
cell	Style elements that affect specific cells

For the *global* style, the command format looks like this:

```
$w setstyle global −foreground blue −background white −relief sunken −align centered
```

while for the *cell* style the command looks like this:

```
$w setstyle cell 10 5 −foreground red −background blue −relief raised −align left
```

The commands for the other style management functions take parameters appropriate to their scope. Where the getstyle function is used, the result of the command is a list that contains the current values of the style options. For example, the current *global* style can be queried using a command like:

```
$w getstyle global −background −relief −align
```

71 Tabs − Create a notebook tabs widget

The *Tabs* widget is a container widget that presents a number of notebook style tabs that can be used to select the currently active child widget. The format of the command is:

```
Tabs path options
```

where *path* is the path name of the widget to be created and *options* is the list of option and value pairs used to configure the widget. In addition to the set of *standard set of widget options* the *Tabs* widget supports the following option:

activetab	Set the currently active tab
count	Get the number of tabs in the widget
tabstyle	Set the type of tabs to draw

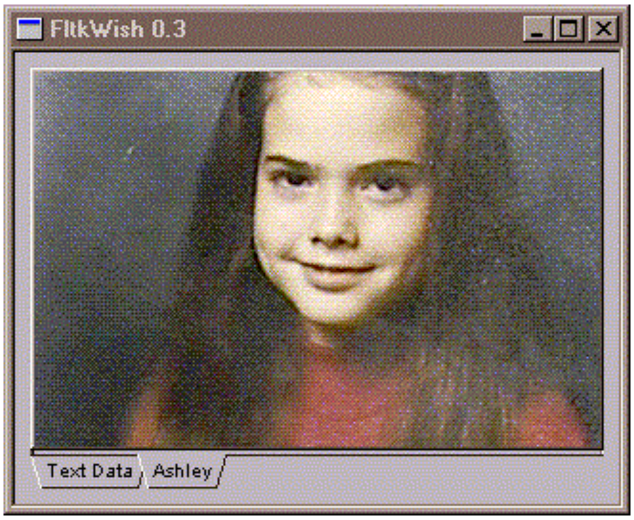
The *activetab* option takes a value that is a number that ranges from 1 through the number of child widgets in the container. Querying the *activetab* option will return the current tab ordinal. The number of available tabs in the widget can be determined by querying the *count* option.

The *tabstyle* option may have the value *old* or *new*. New style tabs are the square tabs used by more recent releases of the FLTK toolkit, while old style tabs are the angled tabs draw my releases prior to the 1.1 series of FLTK releases. The default tab style is *new*.

The *Tabs* container creates a tab for each child that is added to the container, and uses the *label* of the widget as the text written on the tab. The default *label* for a widget is its path name, so it is usually a good idea to configure the child widgets to have *labels* that are useful in identifying the contents of the child. All of the layout and features of the *Tabs* widget are done automatically, so some practice is needed to get something looking pleasant to the eye.

Children are added to the widget simply by creating them. Fairly complex mega−widgets can be constructed by packing interesting combinations of things into *Package*, *Scroll*, *Group* or *Tile* containers and then arranging for these containers to themselves be children of a *Tabs* container.

Here is a simple example of a *Tabs* widget:

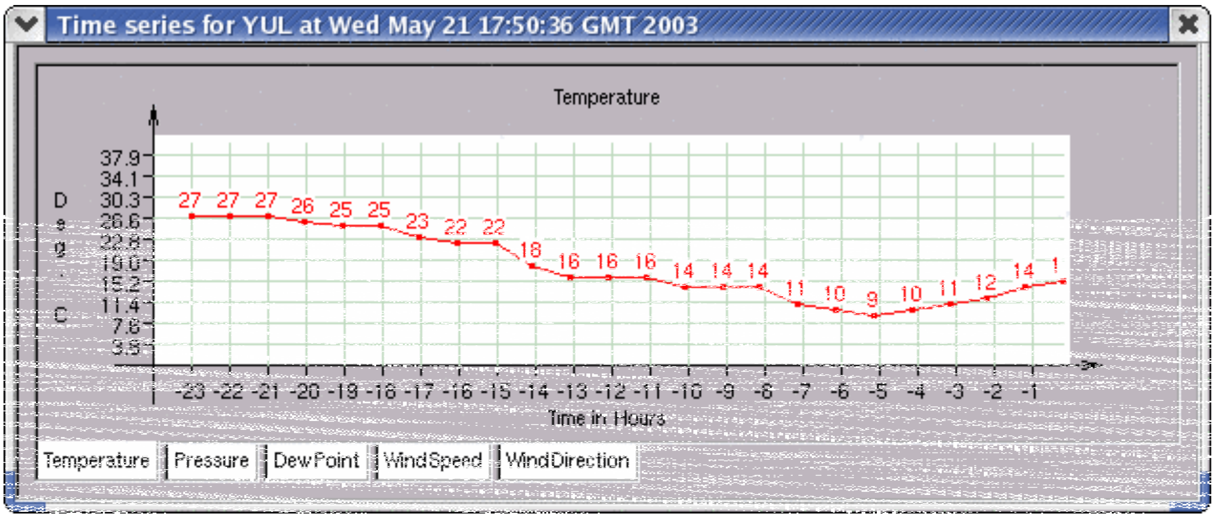


The above is produced using the following code fragment:

```
Tabs root.t −w 300 −h 220 −tabstyle old
```

```
Text root.t.text −label "Text Data" −w 300 −h 200 −value "This is some text for the widget!"
Image root.t.image −label Ashley −file images/ashley.gif −w 300 −h 200
```

These commands result in a tab notebook that has 2 tabs, one labeled "Text Data" and the other labeled "Ashley". Clicking on the appropriate tab will activate the appropriate child. Note that the widgets packed into the *Tabs* container are smaller than the container itself. This is to provide space for the tabs themselves. If no space is left, the tabs will get squashed! The *Tabs* container decides how to place the tabs based on the distance between the edges of the child windows and the edge of the container window. The largest distance determines the tab location.



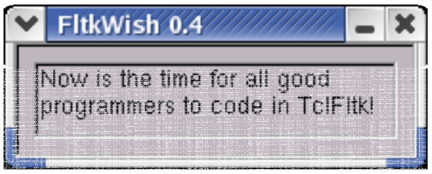
The above *Tabs* container holds a number of time series graphs produced using the *XYPlot* widget. Each of the tabs will bring to the foreground the relevant time series. See the file *timesubs.tcl* in the *scripts* directory of the distribution for the details of the construction of this display.

72 TestWidget − Create a test widget

This command is a place holder for the development of new widgets. Its command format and options depend on the nature of the widget being developed.

73 Text − Create a text widget

The *Text* widget is used to edit multiple lines of text. The widget supports the usual set of basic editing features and text display features, but it is not by any means a highly evolved text editing widget such as one might find in other tool kits.



Now is the time for all good
programmers to code in Tcl!Fltk!

The format of the command is:

```
Text path options
```

where *path* is the path name of the widget to be created and *options* is the list of option and value pairs used to configure the widget. In addition to the set of *standard widget options*, the following widget specific options are supported:

value	The text in the widget
textfont	The font to use
textsize	The size of the font
length	Query the amount of text in the widget

The *Text* widget is typically used to present the user with some multi−line text that needs changing. The user does standard editing operations on the text, then the result is retrieved and used for whatever purpose by the application. The widget can be initialized by sending a string of characters with embedded newline characters. For example:

```
Text root.t −value "Hello\nWorld!\n"
```

will produce two lines of text in the widget.

74 Tile − Create a tiled widget

A *Tile* widget is a container that allows the resizing of its child widgets by the dragging of the internal borders of the widgets. Usually a *Tile* container has a number of widgets that are placed beside each other. The relative sizes of the widgets can then be adjusted by dragging the adjacent borders.

The format of the command is:

```
Tile path options
```

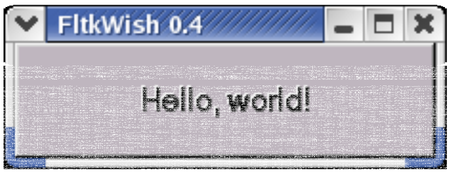
where *path* is the path name of the widget to be created and *options* is the list of option and value pairs used to configure the widget. In addition to the set of *standard widget options*, the *Tile* widget supports the following option:

```
position
```

Children are added to the *Tile* container simply by creating them. You need to do your own geometry management when laying out the children in the container. Once they are in place, just drag the internal borders to resize the children. They remain packed in the container.

75 Toplevel − Construct a top level widget

The *Toplevel* command creates a window container widget that is also a top level window for the window manager. This means it is a window with a border, title and system menu, and the usual maximize, minimize and close buttons. A *Toplevel* window is usually either the root window of an application, or one of several container windows for an application.



The format of the command is:

```
Toplevel path options
```

where *path* is a valid path name for the widget and *options* are the option and value pairs used to configure the window. In addition to the set of *standard widget options*, the *Toplevel* command provides the following list of widget specific options:

fullscreen	Make a full screen window
hidden	Make a hidden window
iconic	Make an iconic window
modal	If the window is system modal
shrinkwrap	Shrink the window to the background image size
tile	Tile the background image
autosize	If windows should autosize
border	Set the border width

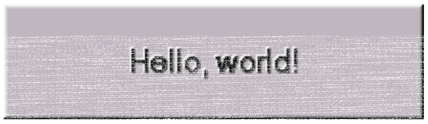
A top level window will normally be created with default attributes which results in a window of the current default dimensions and position in the normal (i.e not full screen and not iconic) mode.

If the window has a background image, then by default it will be displayed centered in the window client area. If the *shrinkwrap* option is set to *true*, then the window is resized to wrap the image, and the user will not be able to resize the window using the standard resize frame or by using the *maximize* button. If the *tile* option is *true*, then the window image will be tiled across the client area.

Modal windows are typically used by dialogs. A *modal* window will capture all mouse and keyboard input until it either is closed or becomes non−modal.

The *autosize* option is by default *true*, and the window will try to resize itself to surround any children. If either a *height* or a *width* is supplied for the window, *autosize* will be set to *false*, and no attempt to resize the window will occur.

The *border* option can be used to supress the presentation of the window title and system menus. By default, *border* is *true*, and the window is drawn with its title and system menus. Setting the value of the *border* option to *false* will suppress these decorations.

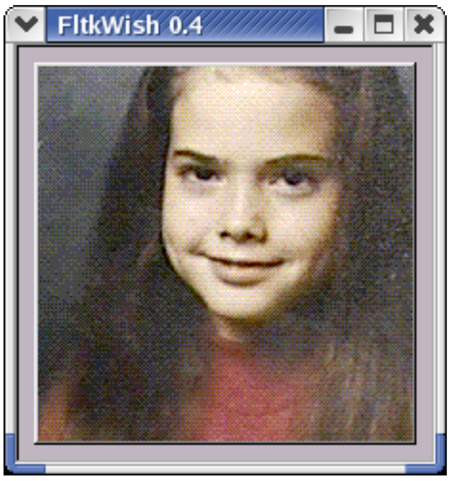


Toplevel windows can be created automatically by creating a child widget for a Toplevel window that does not already exist. For example, the following command will create a *Toplevel* window that contains an *Image* widget:

```
Image root.image −file images/ashley.gif
```

75 Toplevel − Construct a top level widget

If there is not already a *Toplevel* named *root*, then it is created and automatically wrapped around the *Image* widget named *root.image*.



76 Update − Redraw widgets

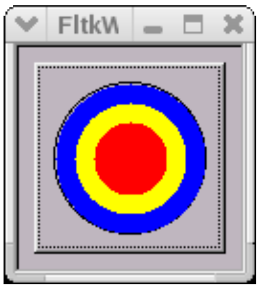
The *Update* command is used to cause the redrawing of the contents of one or more widgets. The format of the command line is:

```
Update widget1 ... widget n
```

where the optional parameters *widget1* to *widgetn* are the path names of the widgets to be redrawn. If no parameters are provided, all of the curent widgets are redrawn. This command is used in scripts to make certain that the displayed contents of windows is not unduely delayed by other computational activity in the script. Tcl/Tk can postpone redrawing of widgets while loops are being processed, for example.

77 UserButton − Create a custom button

The *UserButton* command constructs a button that has a face that can be drawn on using the turtle graphics command language.



The format of the command is:

```
UserButton path options
```

where *path* is the path name of the button and *options* is the list of option and value pairs that is used to configure the widget. In addition to the list of *standard widget options*, the *UserButton* widget supports the following widget specific options:

drawing	The drawing script for the button face
value	The current value of the button
type	The type of the button
downrelief	The relief of the button when pressed
onvalue	The value of the button when it is on
offvalue	The value of the button when it is off
shortcut	The name of the shortcut key for the button

Aside from the *drawing* option, the other widget specific options implement behaviour identical to that implemented by these options for the other widgets in the *Button* class. The *UserButton* widget is also a member of the *Button* class.

The *drawing* option is, by default, an empty string, and the face of the widget is blank. Any set of drawing commands supported by the *turtle graphics drawing language* may be passed as a script to the *UserButton*. A complete description of the turtle graphics language is given in the chapter on the *Drawing* widget. For example, the command:

```
UserButton t.t −drawing "cs ht fl on bg blue cr 40 bg yellow cr 30 bg red cr 20"
Show t
```

will produce a button that has a target displayed on its face.

78 Windows − Interrogate the list of widgets

The *Windows* command gets information about the current list of widgets being managed by the Fltk extension. The functions supported by this command are:

list	Return the list of all widgets
count	Return the count of the managed widgets
toplevels	Return the list of top level widgets
class	Return a list of windows by class name
group	Return a list of windows in a group

78.1 list − Get a list of windows

The *list* function will return a list of all of the widgets currently registered to an application, or optionally, a subset of this list based on a set of selection strings. The format of the the command is:

```
Windows list ?patterns?
```

where *patterns* is a comma seperated list of strings used to filter the window list. If the *patterns* option is provided, the returned list of widgets contains all of the widgets whose path names contain one of the substrings in *patterns*. For example, the command:

```
Windows list YUL,plot
```

could be used to select all of the widget with the substrings *YUL* or *plot* in their path names. The command:

```
Windows list
```

results in a list of all widget names for the widgets that are currently in existence and managed by the Fltk extension.

78.2 count − Get the widget count of toplevels

The *count* function simply gets the number of *Toplevel* widget in the current application.

78.3 toplevels − Get the count of container windows

The *toplevel* function gets the number of container widgets in the widget list. Container widgets may be *Toplevel* widgets, but can also be other types of containers.

78.4 class − Get the list of widgets in a class

The format of this function command is:

```
Windows class name
```

where *name* is a class name. The result of this function is a list of all widget that are members of the specified class name.

78.5 group − Get the widgets in a group

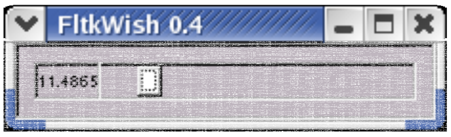
Some widgets act in groups, such as *RadioButtons*. This function will list all of the widgets in a group. The format of the command is:

```
Windows group name
```

where name is the name of the group. The name of the group is the path name of the group container being used to group the widgets.

79 ValueSlider − Create a slider with a value display

The *ValueSlider* widget is similar to a *Slider* widget except that it has an attached text box that continuously displays the current value of the widget.



The format of the command is:

```
ValueSlider path options
```

where *path* is the path name of the widget to be created and *options* is the list of option and value pairs used to configure the widget. In addition to the set of *standard widget options*, the *ValueSlider* supports all of the widget specific options of the *Slider* widget and the following list of widget specific options:

color	The color of the text display
textfont	The font used to display the text
textsize	The size of the font

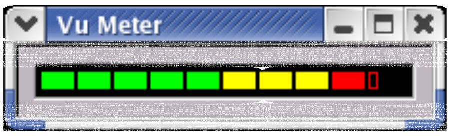
Here is a *ValueSlider* that shows the value in red:

```
ValueSlider root.vs −color red −variable MySliderVar −orientation vertical
```

This widget will set the value of the Tcl variable *MySliderVar* as the slider is moved.

80 Vu − Construct a digital volume units widget

The *Vu* command is used to construct a widget that resembles a digital volume units display typical of some LED based displays on audio equipment.



The format of the command is:

Vu path options

where *path* is the pat name of the widget to be constructed and *options* is the list of keyword and value pairs that is used to configure the widget. In addition to the set of *standard widget options*, the *Vu* widget supports the following widget specific options:

- value The current value of the widget
- orientation The orientation of the widget
- maximum The largest value to display
- minimum The lowest value to display
- logscale If the scale is logarithmic
- autoscale If automatic range normalization is used

The *value* options is used to get or set the current value of the widget. This value is used to create the display presentation.

The *orientation* option can have the values *horizontal* or *vertical.* By default, the value of the *orientation* option is *horizontal* and the widget is displayed as a horizontal oblong with the minimum value at the left of the rectangle. If the *orientation* is set to *vertical,* the widget is displayed as a vertical oblong with the minimum value at the bottom of the rectangle.

The *maximum* and *minimum* options are used to set the range of the values being displayed by the widget. By default, the values of the *maximum* and *minimum* options are *100.0* and *0.0* respectively. When the *autoscale* option is *true,* the values of the *maximum* and *minimum* options are computed automatically based on a series of *values* sent to the widget. The widget will automatically adjust the *maximum* and *minimum* values to accomodate the range of *values* presented. By default, the value of the *autoscale* option is *false.*

The *logscale* option is by default *false.* If set to *true,* the base 10 logarithm of the value is used to construct the plot instead of the *value* itself.

81 Winfo − Get information about a widget

The *Winfo* command is used to retrieve information about a widget window. The functions supported by the command are:

exists	Returns 1 if the widget exists, otherwise, returns 0
geometry	Returns the current widget geometry in standard X windows format
x	Returns the current horizontal position of the widget
y	Returns the current vertical position of the widget
width	Returns the current width of the widget
height	Returns the current height of the widget
id	Returns the system dependant widget identifier
childcount	Returns the number of child widgets of this widget
root	Returns the root widget for this widget
children	Returns the list of children of this widget
class	Returns the class name of this widget
parent	Returns the parent of this widget

The general format of the command is:

```
Winfo function path
```

where *function* is one of the functions from the list of supported functions and *path* is the path name of the widget.

For example, the command:

```
Winfo geometry root
```

would return the current geometry of the window identified by *root* in the form:

```
wxh+x+y
```

where *w* and *h* are the width and height of the widget, and *x* and *y* represent the screen or widget relative locations of the widget with respect to its parent. Since the parent of a top level widget is the screen, in this case the location is screen relative.

82 Wm − Interact with the window manager

The *Wm* command interacts with the window manager to control the behaviour of top level widgets. The functions supported are:

title	Set the title of the widget window
iconname	Set the name of the widget window icon
maxsize	Set the maximum dimensions of a widget window
minsize	Set the minimum size of a widget window
deiconize	Restore a hidden or minimized window to its normal state
withdraw	Hide the widget window
geometry	Set the widget window initial geometry
position	Set the position of the widget window on the screen

The format of the command is:

```
Wm function widget ?data?
```

where *function* is the function to perform from the list of supported functions, *widget* is the path name of the widget to act upon, and *data* is any data needed for the function.

For example, the command:

```
Wm title root "My Root Window"
```

would set the title of the window that is named *root* to the string "*My Root Window*"

83 Wizard − Create a wizard widget

Wizard widgets are containers that can hold a number of child widgets that overlay each other. This widget is similar to the *Tabs* widget, although the no tabs are drawn and the currently visible child is controlled by the application, as opposed to being controlled by the user's mouse input. This widget is useful for stepping the user through a series of actions and option selections.

The format of the command used to construct a *Wizard* is:

Wizard path options

where *path* is the path name of the Wizard and *options* are the option and value pairs used to configure the widget. In addition to the *standard set of widget options*, the *Wizard* accepts the following widget specific options:

activechild	The index of the currently active child
count	The number of children in the container

The *activechild* option takes a number that must be from 1 through the number of child widgets in the container. If the *activechild* option is queried, then the value returned is the index of the currently active child widget, or −1 if there are no children in the widget. The *count* option can be queried to determine the number of children in the container.

83.1 Widget Specific Commands

In addition to the standard widget commands *configure* and *cget*, the *Wizard* supports the commands *next* and *previous*. The *next* command will cause the next child in the container to become the active widget, while the *previous* command will cause the previous widget to become active. Clearly, these commands have no effect at the oposite ends of the child list, respectively.

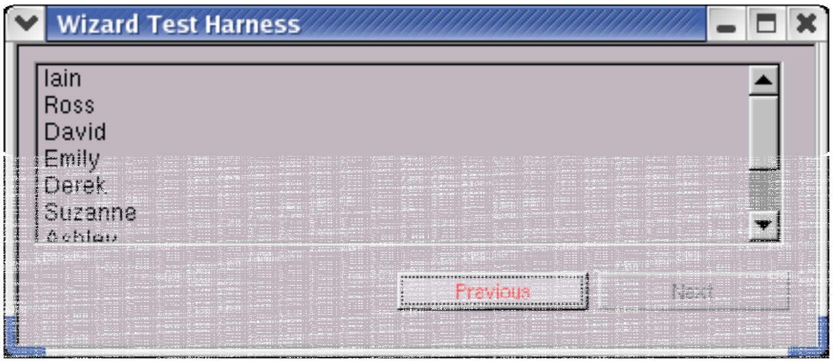
83.2 Adding Children to a Wizard

Child widgets are added to the *Wizard* simply by creating them as children. The order of creation determines the order they are displayed in the container. Here is some sample code that creates a typical wizard:

```
#!/bin/sh
# \
exec fltkwish "$0" ${1+"$@"}
#
# --- wizard.tcl --- Test harness for the Wizard container
#
# Copyright(C) I.B.Findleton, 2003. All Rights Reserved
#
# Move to the next item in the wizard

proc Next { w next prev } {

    global status

    $w next

    SetState $w $next $prev

    }

# Move to the previous item in the wizard
```

```
proc Prev { w next prev } {

    $w previous

    SetState $w $next $prev

    }

# Set the state of the wizard buttons

proc SetState { w next prev } {

    set count [$w get −count]
    set current [$w get −activechild]

    if { $current == 1 } {
        $prev set −state disabled
        $next set −state normal
    } elseif { $current == $count } {
        $prev set −state normal
        $next set −state disabled
    } else {
        $prev set −state normal
        $next set −state normal
        }

    }


# Create a GUI for the wizard

Destroy t

set f [Frame t.g −w 400 −h 140 −relief flat −auto false]

set w [Wizard $f.w −w width −h 100 −relief flat]

set f0 [Package $f.actions −x right−8 −y 120 −pad 5 −orientation horizontal −w 205]
Button $f0.previous −command "Prev $w $f0.next $f0.previous" −label Previous −state disa
Button $f0.next −command "Next $w $f0.next $f0.previous" −label Next

proc ScrolledImage { w args } {


    eval { Scroll $w −nocomplain true } $args
    eval { Image $w.image −nocomplain true } $args

    return $w
    }

# The first child

ScrolledImage $w.c1 −Image.f $Fltk(Library)/images/ashley.gif −Scroll.w width  −Scroll.h

# The second child

ScrolledImage $w.c2 −Image.f $Fltk(Library)/images/clouds.jpg −Scroll.w width −Scroll.h
```

```
# The third child

Button $w.c3 −x centered −y centered −label "Centered Button"

# The fourth child

Button $w.c4 −x center −y top −label "Top Center"

# The fifth child

Listbox $w.c5 −h height −w width

$w.c5 add Iain Ross David Emily Derek Suzanne Ashley Amber Julie Anne−Marie

SetState $w $f0.next $f0.previous

Show t

Wm title t "Wizard Test Harness"
```



84 XYPlot Create a 2 dimensional plot widget

The *XYPlot* widget is a data graphing widget that can be used to display data points in a 2 dimensional space. The widget can also perform some basic linear regression calculations and display a linear fit to the data points along with the standard error bounds for the regression line. The format of the the command line is:

```
XYPlot path options
```

where path is the path name of the widget to be created and options is the list of option and value pairs that is used to configure the widget. In addition to the set of *standard widget options* , the *XYPlot* widget supports the following widget specific options:

textfont	Font used for point labels
textsize	Size of text used for point labels
textcolor	Color of the text
textbackground	Color of the text background
xlabel,ylabel	Labels for the X and Y axes
xlabelcommand	Command to create a label on the abscissa
ylabelcommand	Command to create a label on the ordinate
xformat,yformat	Formats for the X and Y axes labels
xrange,yrange,zrange	Set the normalization range for values
valuegradient	If the color of the points is scaled to values
line	If the points are joined by a line
linestyle	The default line style
fit	If a regression fit is shown
fitcolor	Color of the fit lines
fitlinestyle	Line style of the fit lines
grid	If a grid is shown
gridcolor	Color used to draw the grid
gridlines	Number of grid lines to draw
plotbackground	Color of the plot background
autolabel	If the points are labeled with their values
autolabelformat	Format for the point labels
value	A write only option!
zerox	If the X=0 line is drawn
zeroy	If the Y=0 line is drawn
zerolinestyle	How to draw zero lines
zerolinecolor	Color of zero lines
pagegeometry	Get the dimensions of the plot diagram
pagex	Get the X cordinate of a plotted value
pagey	Get the Y coordinate of a plotted value
drawing	Supply a turtle graphics script

84.1 Configurable Options

84.1.1 Text Options

The options *textfont*, *textsize* , *textbackground* and *textcolor* apply to the optional labels that may be displayed on the graph associated with the plotted points. These values default to *helvetica*, *10*,*clear* and *black*. Note that the text that appears for the titles, axes labels and axes tick marks is controlled by the appropriate *standard widget options*, which just happen to have the same default values.

84.1.2 xlabel and ylabel

These 2 options are the text strings that appear as the labels for the X and Y axes of the plot. Their default values are *X Axis* and *Y Axis* respectively.

84.1.3 xlabelcommand and ylabelcommand

These 2 options allow the user to provide a Tcl script to be executed whenever a value along one of the axes is displayed. The script should format the value to be displayed along the axis and return that value appropriately formatted for display as its result. This is useful when there is a non−ordinal relationship between the coordinates of the values being plotted, such as when the displayed label is some function of the actual coordinate value.

The script is first expanded by converting the following tokens based on the widget and the data being plotted:

%W	The path name of the widget
%a	The axis being drawn (Either X or Y)
%v	The value of along the relevant axis

The above tokens are first replaced by their actual values, the resulting script is then evaluated, and the returned result is drawn along the appropriate axis. For example, the command:

```
$w set −xlabelcommand "XLabel %W %v"
```

would cause the Tcl procedure *XLabel* to be executed with parameters that are the widget path name and the value along the X axis to be formatted. The *XLabel* procedure needs to return the properly formated value to be displayed.

84.1.4 xformat,yformat

These options are used to set the format specifiers used to display values for the axis tick marks. The widget will automatically scale the ranges of the values of the plot point coordinates and label the tick marks at locations that represent about 10 percent of the scale range. The default format specification is %6.1f. Any valid format specification acceptable to the standard C library function sprintf is acceptable. The range values are floating point numbers, so a floating point specification is good unless you want to cause a program crash.

84.1.5 xrange,yrange,zrange

These options set the range that is used for the normalization of the values used for the x, y and z (value) coordinates in the plotted graphs. By default, the normalization is computed automatically from the range of values provided for the components. If a normalization range is set, then the values are scaled within the specified range. For example, the command:

```
XYPlot t.t −xrange 92,105
```

would cause the range of the X axis to be from 92 to 105, and all points whose X coordinate fall within this range will be plotted. Points with an X coordinate outside of this range will not appear on the chart.

By setting one of these options to an empty string, range normalization is reset to automatic, and the axis range is determined automatically from the range of values for the relevant cordinate value.

84.1.6 valuegradient

This boolean option determines whether the points plotted on the graph have colors that are adjusted according to the value of the point. By default, the value of this option is false.

84.1.7 line

This is a boolean option that determines whether a line joining the plotted points is drawn. By default, the value of this option is *false* and no line is drawn. When the value of the option is *true*, adjacent points that meet certain selection criteria are joined by a line. Each point on the graph can be drawn using a user defined *symbol*. For a line to be drawn joining 2 points, each of the points must be adjacent in the list of points being plotted, and they must have the same *symbol*. By default, all points have the same *symbol*. The add function details the method of changing the *symbol* used to plot a point.

84.1.8 linestyle

The *linestyle* option is used to set the style of the line used to join adjacent points, if the value of the *line* option is *true*. By default, the value of the *linestyle* option is *solid*.

84.1.9 fit

The value of the *fit* option is a boolean value that determines whether or not a regression analysis is performed on the list of data points being plotted. By default, the value of the *fit* option is *false*, and no regression analysis is carried out. If the value of the *fit* option is *true*, the linear regression line, based on a best least squares error fit, that describes the relationship between the x and y coordinate values of the points is computed. The regression line, regression equation and the standard error estimates of the predicted values of the dependent variable are all displayed on the graph.

84.1.10 fitcolor

The *fitcolor* option sets the color used to display the regression lines on a plot for which the *fit* option is *true*. By default, the value of this option is *orange* .

84.1.11 fitlinestyle

The *fitlinestyle* option is used to specify the line style used to draw the regression line and standard error bounds lines of the regression model. By default, the value of the *fitlinestyle* option is *dash*.

84.1.12 grid

The *grid* option determines whether a background grid is displayed as part of the graph. By default, the value of the *grid* option is *false*. If the value of the *grid* option is *true*, the background grid will appear using the same spacing as that used by the access ticks.

84.1.13 gridcolor

The *gridcolor* option is used to define the color of the grid lines displayed when the *grid* option is *true*. By default, the value of this option is *gray80*, which displays a very pale gray color.

84.1.14 gridlines

The *gridlines* option is used to query or specify a value that determines the number of horizontal or vertical lines that are drawn when the value of the *grid* option is *true*. The number of lines to draw is specified by a floating point value for the horizontal and vertical dimensions that is used to compute the interval between ticks on the graph along the respective axes. By default, the values are 11.0 for both axes. This results in 11 lines being drawn, depending on the actual dimensions of the widget.

The *gridlines* values are set using a command of the form:

```
$w set −gridlines horizontal,vertical
```

where *$w* specifies the widget path name of the *XYPlot* widget, and *horizontal* and *vertical* are floating point numbers that are used to determine the number of lines drawn. The determination is carried out by computing an interval value for the relevant axis by dividing the number of pixels available along the axis by the relevant factor. Because of rounding and varying resolutions and widget dimensions, the values of *horizontal* and *vertical* are only approximately equal to the number of grid lines actually drawn.

84.1.15 plotbackground

The *plotbackground* option is used to set the color of the background for the area of the widget window that is used to display plotted data. By default the value of the *plotbackground* option is *white*.

84.1.16 autolabel

The value of the *autolabel* option determines whether a label is automatically generated for plotted points. The automatic label is based on the value assigned to a point. By default, the value of the *autolabel* option is *false*.

84.1.17 autolabelformat

The value of the *autolabelformat* option is a format specifier that is used to display automatically generated labels. By default, the value of this option is %g. Any specification acceptable to the standard C library *printf* function may be used. Note that the value of a point is a floating point number, so it is useful to used a floating point format specifier.

84.1.18 value

The *value* option is a write only option and is used by the variable binding functions. It has no effect on the command line.

84.1.19 zerox

The *zerox* option is a boolean option that is by default *false*. If set to *true*, then the line that represents the location of the vertical axis at a value of x = 0 is drawn, if this value lies within the current range of plotted values for the independant variable.

84.1.20 zeroy

The *zeroy* option is a bolean option that is by default *false*. If set to *true*, then the line that represents the location of the horizontal axis at the value y = 0 is drawn, if this value lies within the current range of plotted values for the dependant variable.

84.1.21 zerolinestyle

The *zerolinestyle* option is used to specify the style of the line used to draw the axes at the zero value locations of the X and Y variables on the plot. By default the line style is *dash*.

84.1.22 zerolinecolor

The *zerolinecolor* option is used to specify the color used to draw the axes at the zero value locations of the X and Y variables on the plot. By default the color used is *black* .

84.1.23 pagegeometry

The pagegeometry option is read only and returns a list of 4 numbers that represent the location of the origin and the extent of the area of the widget being used to display the plotted values. The first 2 numbers are the location of the upper left hand corner of the plotting area while the second 2 numbers are the width and height of the plotting area.

84.1.24 pagex

The pagex option, when set, returns the location X location of the value supplied. When read, the value returned is the X location of the most recently set value. The values returned can be used directly as coordinatate by the drawing script, if one is supplied.

84.1.25 pagey

The pagey option, when set, returns the Y location of the value supplied. When read, the value returned is the Y location of the most recently set value. The values returned can be used directly as coordinates in the drawing scripts, if one is supplied.

84.1.26 drawing

The *drawing* option can be used to supply a turtle graphics script that will be drawn on the region of the widget that is being used for displaying the plotted data. This feature can be used to annotate the plotted data, or to enhance the plot with additional text or graphics. The drawing area used is the same region of the widget that is used to display the plotted data.

The coordinate system that is used by the turtle graphics engine is that of the displayed values of the horizontal and vertical axes of the plot. The *pagex* and *pagey* options can be used to translate between values in the space of the plotted data to display coordinates used to draw items. For example, here is a command that will draw a circle at the point in the data space with coordinates of (0.0, 0.0):

```
$w set −drawing "cs pc red sp [$w set −pagex 0.0] [$w set −pagey 0.0] cr 20"
```

The *XYPlot* widget path name is in the variable *w*. The turtle graphics script draws a red circle centered about the point at data coordinates (0.0,0.0) with a radius of 20 pixels.

Any of the commands and features of the turtle graphics drawing engine may be used to create annotations and graphics on the plot. The documentation on the *Drawing* widget describes how to develop turtle graphics scripts.

84.2 Points and their attributes

A point as defined for use by the *XYPlot* widget consists of a set of 2 coordinate values representing the location of the point with respect to the abscissa and the ordinate of the graph, and a value that is the value of the point. For example, if a set of points is stored as a Tcl array, an entry in the array can be set using the following Tcl command:

```
set Data(4.3,35.2) 103,5
```

In this example, the coordinates of the point are 4.3 and 35.2, while the value of the point is 103.5. Points may also have additional attributes that define how the point is to appear on the graph, as well as attributes useful for managing points or series of points.

When a point is created, or for a point already contained in the list of points being plotted, the following attributes can be set or modified:

x	Location with respect to the abscissa
y	Location with respect to the ordinate
value	Value of the point
symbol	The name of the symbol to use to plot the point
color	Color used to plot the point

label	Label used for the point
tags	List of tags for the point
linestyle	Style of the line used to join points
labelcolor	Color used for label text
labelalign	Where to put the label
labelbackground	Background color for the label

Points are indexed in the point list using a number between 0 and 1 minus the number of points in the list. Using this index, the attributes of a point may be modified. When a new point is added to the list, the attributes of the point can be set using the above option names.

When points are plotted on the graph they appear as one of a set of available *symbols*. By default, all points are plotted using the point symbol. Here is the list of available *symbols*:

point	A small point
cross	An x symbol
plus	A plus sign symbol
circle	A small unfilled circle
triangle	A small triangle
square	A small box
blob	A small filled circle

Labels for points are just text strings. They can be of any length, however, plots can become crowded if long text labels are specified.

Tags for points are strings of comma separated tag names that are associated with points. These tag lists are used to manage the characteristics of the displayed points using the widget commands. By default, points have no *tag*.

84.3 Using Tcl Arrays

The *XYPlot* widget can be bound to a Tcl variable that is an array of points that are to be plotted. Using the *variable* widget option, the name of a Tcl array can be supplied to the widget as the source of the data to be plotted. For example, if a global Tcl array has elements of the form:

```
Data(x,y)
```

where the x and y components of the array index are numerical values, then the command:

```
$w configure −variable Data
```

will cause the plot widget to collect all of the members of the array Data and use the values stored in the array to plot the points. This provides a convenient way of plotting data directly from a Tcl script. See the *example script* below for the details of how to implement variable binding.

The indices of Tcl arrays can optionally be used to specify the color and label attributes of the points to be plotted. The general form of the indices being used is:

```
x,y,color,label
```

where *x* and *y* are numerical values that specify the coordinates of the point to be plotted, *color* is a color name to be used to plot the point, and *label* is a text string that is to be used as the label for the plotted point. The *color* and *label* components are optional, and if they are not present, the default values will be used. The following array element:

```
Data(4.3,19.7,purple,Special)
```

would be plotted at the graph location (4.3,19.7) in *purple* and with the label *Special*.

84.4 Widget Commands

In addition to the standard widget commands *cget* and *configure*, the *XYPlot* widget supports the following list of widget specific commands:

add	Add a point to the list of points
bounds	Specify the normalization range for the axes
clear	Clear the list of points
closest	Get the point closest to a location
color	Specify the color for the list of points
count	Get the point count
hide	Hide points
labelbackground	Set the label background color for points
labelcolor	Set the label color for points
labelalign	Set the location of labels
linestyle	Set the line style used to join points
statistics	Get the statistical information about the points
show	Show points in the list
symbol	Set the symbol for a list of points

84.4.1 add Add points to the list

The format of the *add* command is:

```
$w add options
```

where *$w* is the path name of the widget and *options* is a list of option and value pairs that is used to configure the point. The names of the options are the names of the attributes of the points. For example, the command:

```
$w add −x 100 −y 30 −value −18 −color blue −tags blue,special −symbol square −label "Blue is Special"
```

will add a point with coordinates (100,30) and value 18 to the list of points being plotted by the widget. When displayed, this point will be represented by a small square box colored blue and it will have the tags blue and special associated with it.

84.4.2 bounds Set the normalization range for the axes

By default, the *XYPlot* widget will automatically set the range of the axes labels based on the ranges of the values of the the coordinates of the plotted points. It may be preferable in some cases to be able to specify the range of values for the axes and to plot the points on the graph according to these values.

The format of the *bounds* command is:

```
$w bounds  −x min,max −y min,max −value min,max
```

where any or all of the options may be specified. Here *$w* is the path name of the widget and *min* and *max* refer to the desired minimum and maximum values of the ranges for the respective coordinate axes. For example, it is often the case where the value of the dependent variable ranges between 0 and 1, as in the case where the points being plotted represent a percentage type of value. The following command will set the bounds of the Y axis appropriately:

```
$w bounds −y 0,1
```

Using the *bounds* command to set the range of normalization for a coordinate or the point values turns off automatic scaling. While this is certainly useful in some cases, should any points be added that have values or coordinate locations outside of the specified ranges, they may not be plotted on the graph.

84.4.3 clear Clear a set of points

The *clear* command can be used to remove all of the points from the widget, or to selectively remove points from the widget. The format of the *clear* command is:

```
$w clear tag tag ...
```

where *$w* is the path name of the widget and the optional *tag* strings are the tags that identify the points to be removed from the list. If no *tags* are specified, then all points in the list are deleted. If any *tags* are specified, all points that have the specified tags in their tag lists will be deleted.

84.4.4 closest Get the point closest to a location

The *closest* command returns the attributes of the point that is closest to the coordinates specified on the command line. This function is provided to support the mapping between locations generated by the mouse over the widget window and the coordinates used to plot the points on the graph.

The format of the *closest* command is:

```
$w closest x y
```

where *$w* is the path name of the widget being queried, and *x* and *y* are the window relative coordinates of the location to query. The window relative location is the location that is returned by a mouse event when the use moves or clicks the mouse over the widget window.

The value returned by this command depends on whether or not a point is in the point list and which of the points in the list is closest to the window location. If there are no points in the point list, then the result returned by this command is simply the 2 input values.

If there is a point found in the point list, the result returned is a Tcl list that contains 2 elements. The first element of the list is a list that has the 2 numbers that are the input location values, and the second element of the list is a list of 3 numbers that represent the plot coordinates of the point and its value.

Here is a script that shows how to use the closest command:

```
Bind $w <ButtonPress> { puts { [%W closest %x %y] } }
```

84.4.5 color Set the color of a list of points

The *color* command will set the color of the point symbol and the color of any line joining the points for all of the points in the point list that match the selection criterion. The format of thhe the *color* command is:

```
$w color color_name tag tag ...
```

where *$w* is the path name of the widget and *color_name* is the color to be set. If no *tags* are specified, all points in the list are affected. If any *tags* are specified, only those points with matching entries in their tag lists will be affected.

The following command will set all of the points in the current point list to *green:*

```
$w color green
```

84.4.6 count Get the number of points in the point list

The *count* command takes no parameters and returns the number of points in the point list. The format of the command is:

```
$w count
```

where *$w* is the path name of the widget to use.

84.4.7 hide Hide points

The *hide* command is used to render invisible points in the point list. The format of the command is:

```
$w hide tag tag ...
```

where *$w* is the path name of the widget to use and the optional *tag* values are tags that identify the points to be hidden. If no *tags* are specified, all of the points in the list are made invisible. If any *tags* are specified, only those points which contain matching tags in their point tag lists will be hidden.

Points which have been hidden using the *hide* command can be made visible using the *show* command.

84.4.8 labelbackground Set the label background color

The *labelbackground* command is used to set the background color for labels of points in the point list. The format of the command is:

```
$w labelbackground color tag tag ...
```

where *$w* is the path name of the widget to use, *color* is the color to use and the optional *tag* values are tags that identify the points. If no *tags* are specified, all of the points in the list are set to use the specified background color. If any *tags* are specified, only those points which contain matching tags in their point tag lists will be affected.

84.4.9 labelcolor Set the label text color

The *labelcolor* command is used to set the foreground color for labels of points in the point list. The format of the command is:

```
$w labelcolor color tag tag ...
```

where *$w* is the path name of the widget to use, *color* is the color to use and the optional *tag* values are tags that identify the points. If no *tags* are specified, all of the points in the list are set to use the specified foreground color. If any *tags* are specified, only those points which contain matching tags in their point tag lists will be affected.

84.4.10 labelalign Set the label position

The *labelalign* command is used to set the location for labels of points in the point list. The format of the command is:

```
$w labelalign alignment tag tag ...
```

where *$w* is the path name of the widget to use, *alignment* is the location to use and the optional *tag* values are tags that identify the points. If no *tags* are specified, all of the points in the list are set to use the specified alignment. If any *tags* are specified, only those points which contain matching tags in their point tag lists will be affected.

Labels can be aligned using the following location names:

top	Above the point
bottom	Below the point
left	Left of the point
right	Right of the point

When a label is displayed, the background is always erased in the area used to display the label text.

84.4.11 linestyle   Set the line style of points

The *linestyle* command is used to set the line style used for the lines that join points in the point list. The format of the command is:

```
$w llinestyle style tag tag ...
```

where *$w* is the path name of the widget to use, *style* is the style to use and the optional *tag* values are tags that identify the points. If no *tags* are specified, all of the points in the list are set to use the specified background color. If any *tags* are specified, only those points which contain matching tags in their point tag lists will be affected.

Line styles may be any of the line styles specified by the extension package. The default line style is *solid.* Other common styles are *dash*, *dot* and *dashdot*.

84.4.12 show   Show points

The *show* command is used to make visible points in the point list that are hidden. The format of the command is:

```
$w show tag tag ...
```

where *$w* is the path name of the widget to use and the optional *tag* values are tags that identify the points. If no *tags* are specified, all of the points in the list are set to be visible. If any *tags* are specified, only those points which contain matching tags in their point tag lists will be affected.

Points can be made invisible using the *hide* command.

84.4.13 statistics   Get the model statistics

The *statistics* command will return a list of elements that contains the various statistical values computed for the variables and used to construct the regression model, if the fit option has been set to true.

84.4.14 symbol   Set the symbols used to plot points

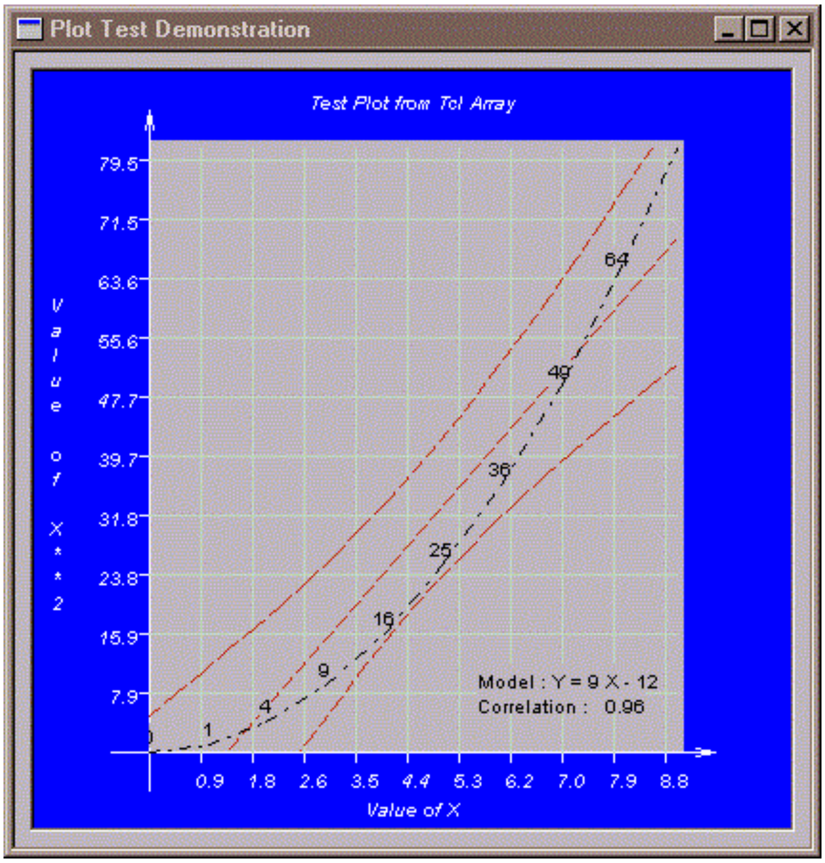
The *symbol* command is used to set the symbols of points in the point list. The format of the command is:

```
$w symbol name tag tag ...
```

where *$w* is the path name of the widget to use, name is the name of the symbol to use and the optional *tag* values are tags that identify the points. If no *tags* are specified, all of the points in the list are set to use the specified background color. If any *tags* are specified, only those points which contain matching tags in their point tag lists will be affected.

The default symbol for a point is *point*.

84.5 Example of the use of the XYPlot Widget



The following script can be found in the *plotdata.tcl* file in the distribution *scripts* directory. It demonstrates how to display a set of points on a 2 dimensional graph, along with some linear correlation statistics on the relationship between the X and Y values of the points. In this example, the points themselves have labels which are the values of the ordinate.

```
#!/bin/sh
# \
exec fltkwish "$0" -- ${1+"$@"}
#
# --- plot.tcl --- Test harness for the XYPlot Widget
#
# Copyright(C) I.B.Findleton, 2001. All Rights Reserved
#
# This script shows how to use the xyplot widget to generate a
# graph from a set of data in a Tcl array variable. The widget
# will scan the named variable for values and plot the data.
#
catch { Destroy t }

# Generate some data in a global array

for { set i 0 } { $i < 10 } { incr i } {
        set Data($i,[expr $i * $i]) [expr $i * $i]
        }
#
# Create a plot widget and bind a variable to it. This plot will also
# compute the linear regression fit to the plotted points and display
# both the fit and the standard error bounds for the regression.
#
XYPlot t.t -fit true -align top,inside -label "Test Plot from Tcl Array" \
        -line true -variable Data -autolabel true -linestyle dashdot \
        -xlabel "Value of X" -ylabel "Value of X**2" -grid true \
        −bg blue −fg white −font helv,italic −plotbackground gray
#
```

85 Relief − Specify the type of relief for a widget

The *relief* of a widget determines how its border pixels are drawn. The Fltk extension supports relief names that are provided by the Fltk tool kit being used to draw the widgets. For the Fltk tool kit, the following relief names are valid:

none
flat
raised
sunken
raisedframe
sunkenframe
raisedthin
sunkenthin
raisedthinframe
sunkenthinframe
engraved
engravedframe
embossed
embossedframe
border
borderframe
shadow
shadowframe
round
roundframe
roundshadow
roundflat
roundraised
roundsunken
raiseddiamond
sunkendiamond
oval
ovalframe
ovalshadow
ovalflat

The relief names are used differently amongst the widgets, and can be used to describe different states of the widgets. For example, *Buttons* use the relief option for the unpressed state, and a *downrelief* option for the pressed state.

There are 2 classes of relief, the *frame* relief types and the *non−frame* relief types. The *frame* relief types just draw the frame around the widget, and do not draw the widget client area. The *non−frame* relief types draw both the frame and the widget client area. When building up a complex compound widget there can be some efficiencies of drawing obtained by using only the frame style to draw relief.

Some schemes make use of the *borderwidth* option to set the number of pixels used to draw relief. Various effects can be achieved by varying the *borderwidth* value, particularly with schemes that make use of OpenGl for widget drawing.

86 Copyright Notice

86.1 Miscellaneous Contributions

Some few of the widgets provided as part of this distribution are based on the copyrighted work of other contributors. Where such software is included in this distribution, the license conditions of the original authors apply. While all of the miscellaneous software used to create the package is available under some version of the GNU Public License or under other Open Source license arrangements, the rights of the original authors respecting their software remain in force. Before you make use of this software for any purpose you should consult the relevant license materials. All relevant license documents are distributed as part of the source release of this package which is available at:

http://pages.infinit.net/cclients/software.htm

Contact: ifindleton@videotron.ca