

Notes from Well House Consultants

These notes are written by Well House Consultants and distributed under their Open Training Notes License. If a copy of this license is not supplied at the end of these notes, please visit

*<http://www.wellho.net/net/whcotnl.html>
for details.*

1.1 Well House Consultants

Well House Consultants provides niche training, primarily but not exclusively in Open Source programming languages. We offer public courses at our training centre and private courses at your offices. We also make some of our training notes available under our "Open Training Notes" license, such as we're doing in this document here.

1.2 Open Training Notes License

With an "Open Training Notes License", for which we make no charge, you're allowed to print, use and distribute these notes provided that you retain the complete and unaltered license agreement with them, including our copyright statement. This means that you can learn from the notes, and have others learn from them too.

You are NOT allowed to charge (directly or indirectly) for the copying or distribution of these notes, nor are you allowed to charge for presentations making any use of them.

1.3 Courses presented by the author

If you would like us to attend a course (Java, Perl, Python, PHP, Tcl/Tk, MySQL or Linux) presented by the author of these notes, please see our public course schedule at

<http://www.wellho.net/course/index.html>

If you have a group of 4 or more trainees who require the same course at the same time, it will cost you less to have us run a private course for you. Please visit our onsite training page at

<http://www.wellho.net/course/otc.html>

which will give you details and costing information

1.4 Contact Details

Well House Consultants may be found online at

<http://www.wellho.net>

graham@wellho.net

technical contact

lisa@wellho.net

administration contact

Our full postal address is

404 The Spa

Melksham

Wiltshire

UK SN12 6QL

Phone +44 (0) 1225 708225

Fax +44 (0) 1225 707126

Tix Megawidgets

MegaWidgets are combination widgets (usually in their own frames). The Tix extension to Tcl provides a number of such widgets, such as file selection boxes and standard widgets with scroll bars. In this module, we teach you how to build Tix megawidgets into your GUI and how to set and handle the events they generate.

<i>Introduction</i>	4
<i>What can you do with a Megawidget?</i>	5
<i>Some of the megawidgets provided by Tix</i>	10
<i>Some other megawidgets</i>	15

2.1 Introduction

What is Tix?

The Tk interface extension (Tix) to Tcl/Tk adds an Object Oriented interface for defining new widgets, known as megawidgets, and a few new standard widgets, a new geometry manager, and some new image types. It also adds a considerable number of megawidget types, and this is perhaps most user's primary use of Tix.

Tix is not a part of the standard Tcl/Tk distribution, but it is available via:

- Open source download from www.neosoft.com/tcl
- As a part of the ActiveState release of Tcl
- As a standard part of some Linux distributions, including Red Hat

You can either use an application that has Tix built in,¹ or pull the package into a Tcl/Tk application such as wish using:

```
package require Tix
```

There are more than 40 megawidgets provided by Tix. You'll want to find a good reference book to get full details of them all and how they fit together. Being an optional extension to Tcl/Tk, you'll find that only certain books contain this information. Check out:

Tcl/Tk Tools Mark Harrison
Tcl/Tk in a Nutshell Paul Raines and Jeff Tranter

The appendix at the rear of this folder contains a full listing of all of the Tcl/Tk books on our library shelves, and has information about publishers, ISBN numbers, etc.

Hello Tix world

Let's see a simple example using one of the more straightforward Tix widgets.

Let's say that you want a label, a box showing you a current value that you can edit, and up and down buttons that let you control the value within a range. How many Tk widgets is that? Four visible ones, plus at least one Frame. It's a single Tix widget, known as a `tixControl`.

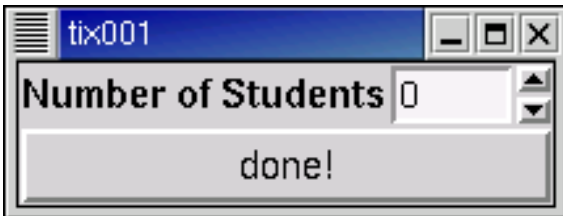


Figure 1 `tixControl`

We have included a regular Tk button to allow you to quit the application. When the user adjusts the value, it's echoed in the widget.

In our simple first example, when the "done!" button is pressed, the value is reported to stdout and the application exits.

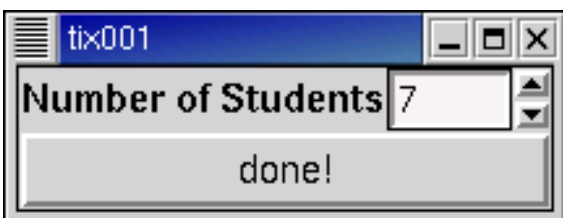


Figure 2 `tixControl` with a value filled in

¹ typically, the application supplied with Linux distributions is known as `tixwish`

```
$ ./tix001
You selected the value 7
$
```

Let's see what the source of this application looks like:

```
#!/usr/bin/wish

package require Tix

proc finish {} {
    set num [.trainees cget -value]
    puts "You selected the value $num"
    exit
}

tixControl .trainees -label "Number of Students" \
    -min 1 -max 12
button .quit -text done! -command finish

pack .trainees .quit -fill both -expand true
```

We've chosen to use "wish" and load in the Tix package through a `package require` command for this initial example. We've put the `package require` at the top of our source code as an obvious flag to anyone reading the source code that Tix is needed, although we could have left the `require` to later in the application. In any case, the extra code of Tix is only actually loaded when the first command from within it is loaded.

Creation, display and control of megawidgets uses exactly the same principles as the creation, display and control of the standard Tk widgets. Remember how we created a Tk widget?

```
button .quit -text done! -command finish
```

Well ... we use a similar command¹ to create our megawidget.

```
tixControl .trainees -label "Number of Students" \
    -min 1 -max 12
```

Widget naming uses the same "folder and file" analogy the Tk uses, so starting with a "." character but with no embedded "."s tells us that this is a widget that will be in the top-level window.

For megawidgets and also for regular widgets, the command that creates the widget does not display it. It just defines it. You have to use a call to a geometry manager to add the widget in to the display. We'll use the `pack` manager for this example:

```
pack .trainees .quit -fill both -expand true
```

Once the widgets are displayed, they'll run callback commands as specified by the `-command` option, and also be accessible via widget subcommands such as `cget` to get a value from a widget. It's going to `cget` that we happen to use with a `tixControl`, since it doesn't have anything that would logically provide a "done" button.

```
.trainees cget -value
```

2.2 What can you do with a Megawidget?

Standard widget functionality

Remember that a megawidget is a widget. You can set standard attributes to change the look, feel and performance of a widget. And they inherit options from the frame widget, as shown in Figure 3, which is coded as follows:

¹ although with a lot of different options for each widget

```
#!/usr/bin/wish
package require Tix
proc finish {} {
    set num [.trainees cget -value]
    puts "You selected the value $num"
    exit
}

# Demonstrate standard widget options
# Options not shown are -cursor and -takefocus

tixControl .trainees -label "Number of Students" \
    -min 1 -max 12 \
    -background red \
    -highlightbackground blue \
    -highlightcolor green \
    -relief sunken \
    -borderwidth 5 \
    -highlightthickness 2 \
    -height 100 \
    -width 400
button .quit -text done! -command finish

pack .trainees .quit -fill both -expand true
```

They can track variables directly (if you don't want to write a command specially to handle the widget, or you want to have the basic variable updating automated and only write the extras) as shown in Figure 3.

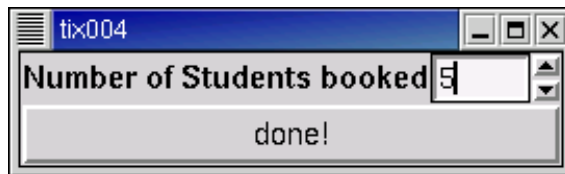


Figure 3 Automatically update the basic variable

```
#!/usr/bin/wish

package require Tix

proc finish {} {
    global have
    puts "You have $have students"
    exit
}

set have 2
tixControl .booked -label "Number of Students booked" \
    -min 0 -max 8 \
    -variable have
button .quit -text done! -command finish

pack .booked .quit -fill both -expand true
```

And in operation:

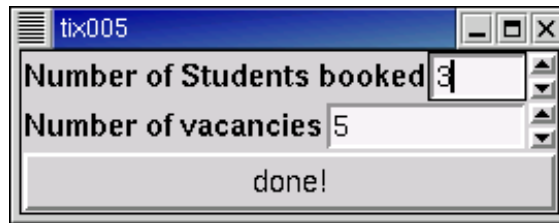
```
$ ./tix004
```

```
You have 5 students
```

```
$
```

You can use multiple megawidgets¹ and have one affect the other as you interact with them. In this example, when we set the number of trainees, the number of vacancies changes automatically to reflect the total availability of eight places (see Figure 4).

Figure 4 Multiple megawidgets interacting



We only changed the number of students. The vacancies track it in the following code:

```
#!/usr/bin/wish

package require Tix

proc finish {} {
    global have
    puts "You have $have students"
    exit
}

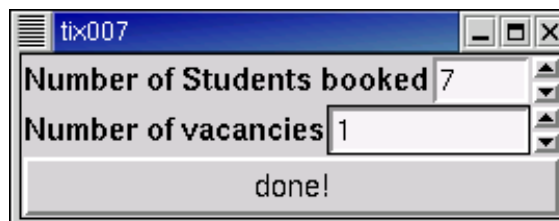
proc setvacant {newval} {
    global spare
    set spare [expr 8 - $newval]
}

set have 2
set spare 6
tixControl .booked -label "Number of Students booked" \
    -min 0 -max 8 \
    -variable have \
    -command setvacant
tixControl .vacant -label "Number of vacancies" \
    -min 0 -max 8 \
    -variable spare
button .quit -text done! -command finish

pack .booked .vacant .quit -fill both -expand true
```

Here's a final example using the functionality that you'll see with both standard and megawidgets. We've used two megawidgets to allow the user to adjust the number of trainees and/or the number of vacancies on a course. Whenever either value is changed, a callback `proc` adjusts the other. This is illustrated in Figure 5.

Figure 5 Multiple megawidgets interacting



The source code is as follows:

¹ of the same or differing types

```
#!/usr/bin/wish

package require Tix

proc finish {} {
    global have
    puts "You have $have students"
    exit
}

proc others {where newval} {
    upvar $where basket
    set basket [expr 8 - $newval]
}

set have 2
set spare 6
tixControl .booked -label "Number of Students booked" \
    -min 0 -max 8 \
    -variable have \
    -command {others spare}
tixControl .vacant -label "Number of vacancies" \
    -variable spare \
    -command {others have}
button .quit -text done! -command finish

pack .booked .vacant .quit -fill both -expand true
```

You'll notice that in this example, we have used both **-value** and **-command** options on the megawidgets. We're having the variables "have" and "spare" track the contents of the entry boxes automatically. Additionally, we're running a command when either changes in order to change the second one. Run our example and you'll see that as one value is increased, the other is decreased automatically.

Exercise caution when you write widgets to track a variable and also to run a command. It's all too easy to write code that calls itself in a never-ending loop. A command changes a variable, which causes the command to be called to change the variable, which ... you get the idea. Such situations can also arise between two or more widgets. Widget A alters a value in Widget B, so Widget B runs a callback that alters the value in Widget A again which ... oops, another infinite loop.

Where you have a stable system in which the values naturally take on an equilibrium,¹ the code will work without any specific action. It's only when a variable has its value changed that a callback is invoked, and if a variable is reset to its existing value the callback won't be run. Where there's a chance that your system won't be stable, you should use the **-disablecallback** option to break the cycle within your callback routine. Thus

```
proc something {} {
    .wid configure -disablecallback true
    # do the work of the something callback
    .wid configure -disablecallback false
}
```

Our example is also a good example of how a single callback routine can be used to handle multiple widgets, and how parameters are passed in to callbacks. You'll notice that our command options both call a single proc and appear to pass in one parameter:

¹ we have such a system in our example

```
-command {others spare}
-command {others have}
```

In fact, they've passed in two parameters. One is provided automatically by the widget implementation, and that's the new value of the variable:

```
proc others {where newval} {
    upvar $where basket
```

We have used `upvar` to collect the name of the variable that we're updating from the first parameter of the command. That saves us the need to write a whole series of near-duplicate procs, and also saves us the need to start declaring a large number of global variables.

Although this module is describing the megawidgets provided by Tix, you'll be seeing that a good, clear understanding of the fundamentals of Tcl, and the widget sets that are provided by the underlying Tk, is vital.

One of the other issues that often catches newcomers¹ is the difference between each of the following:

```
-command {others have}           ; # 1
-command {others $have}         ; # 2
-command "others $have"         ; # 3
-command [others $have]         ; # 4
```

Potentially, all four of these are valid in a widget or megawidget constructor.

The fourth example² instructs Tcl to run the `others` command prior to building the widget, passing it a single parameter which is the contents of the `$have` variable prior to the construction of the widget. The value returned by the `others` command is the name of the proc which is to be run whenever the value in the widget is altered by the user or some other piece of program. Note that this example, whilst quite valid, is the least likely of our four examples to be used.

The third example uses double quotes around the command. The value of `$have` is substituted in at the time the widget is constructed. When the widget's `-command` is run, it will run "other" with two parameters, the first being the value that was in the `$have` variable at the time that the widget was built. The second being the value selected on the widget. Note that this second parameter only applies to a `tixControl` widget, a `scale` widget and certain others that append the return value to the command automatically. This example is likely to be how you want to write your code if you're using a `foreach` loop to build your widgets and want each callback to include a unique parameter indicating which widget was selected.

The second example, using curly braces around the command, requests deferred execution. In other words, the "other" command will be run at the time the callback is activated, and it's at that time that the value of `$have` will be substituted. This is a major difference to the two cases above, where the variable `$have`'s value at setup time is used, rather than its dynamic value as the program has been running.

The first (and final) example doesn't refer to a variable called "have" directly; it passes the text "have" across to the callback proc. Notice that there is no `$` character. You'll write callback code like this if you want to pass a variable name into the callback routine. If you look at the source code of the callback routine you'll probably find that it starts straight away with an `upvar` command. Although the code may appear obtuse, this is a very common structure and indeed it's the one that we used in the example that lead to this discussion.

Summarising. As you write callback procs, start by asking yourself "Do I want to pass in a variable name, its value now, its value later, or the result of running a command on it?"

¹ and it's healthy to have a reminder

² starting at the end!

2.3 Some of the megawidgets provided by Tix

There are more than 40 megawidgets provided by Tix – a rich selection which includes all the widgets that you wish were in the standard library, but are not. In this section, we'll take you on a tour that uses the major groups of megawidgets and introduces you to the use and facilities of some of them. Please refer to reference material for full details of every Tix widget.

Our sample application

We're going to use Tix to create a tree showing which trainee workstations are running optional software that we only use on certain courses.¹ The initial display we want is shown in Figure 6.



Figure 6 Creating a tree with Tix

We'll select a system name in the first column, then click on each of the pieces of optional software that machine is running in turn by using the buttons in the second column. The results will be displayed in the third column.

This is what it will look like (see Figure 7).

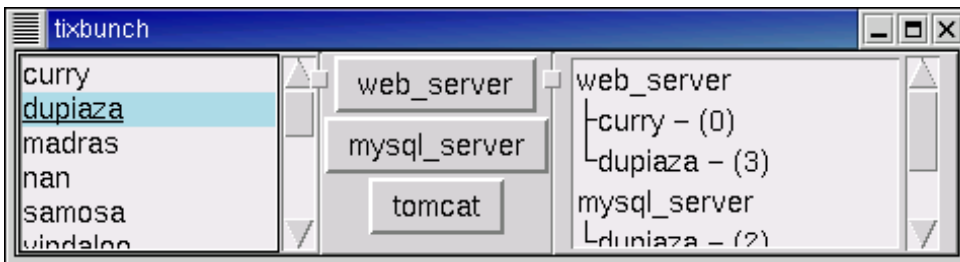


Figure 7 Creating a tree with Tix

The source code of the application is as follows:

```
#!/usr/bin/wish

package require Tix

set trainee {curry dupiaza madras nan samosa vindaloo
  ceylon balti bhuna pillau dhansak papadom bhajee}
set server {web_server mysql_server tomcat}

proc action {chosenserver} {
  global rlist
  global clients
  global kidno
  global trainee
  set chosen [$clients curselection]
  set cli [lindex $trainee $chosen]
  $rlist add $chosenserver.$kidno -text "$cli - ($kidno)"
  incr kidno
}
```

¹ Tcl and Tix are loaded on all of our systems!

```

    }

set kidno 0

# Create a paned window .... we'll have 3 panes:

tixPanedWindow .overall -orientation horizontal -height 100
.overall add source -min 100 -max 150
.overall add target -min 100
.overall add result -size 200

# Collect the second pane, and put in a series of labels

set tname [.overall subwidget target]
foreach tar $server {
    button $tname.$star -text $star -command \
        "action $star"
    pack $tname.$star
}

# Collect the first pane, and put in a scrolled list box

set sname [.overall subwidget source]
tixScrolledListBox $sname.tsys
set clients [$sname.tsys subwidget listbox]
foreach sou $straineer {
    $clients insert end $sou
}
pack $sname.tsys

# Collect the third pane, and put in a hierarchical widget
# Seed that widget (top level only) with server names

set rname [.overall subwidget result]
tixScrolledHList $rname.log -options {
    hlist.itemType text
    hlist.indebt 5 }
set rlist [$rname.log subwidget hlist]
foreach host $server {
    $rlist add $host -text $host
}
pack $rname.log

pack .overall -side left -expand yes -fill both

```

Container Widgets

Tix provides a number of container widgets such as the `tixNoteBook`, the `tixScrolledWindow` and the `tixPanedWindow`. For our application, we require a number of window panes side-by-side; therefore, we have elected to use a `tixPanedWindow`:

```

tixPanedWindow .overall -orientation horizontal -height 100
.overall add source -min 100 -max 150
.overall add target -min 100
.overall add result -size 200

```

The window is created via a command named after the widget type as usual.¹ We pass in the name we want to give the widget; in this case starting with a "." and then a single name representing a widget in the top level of our wish. Numerous options are available; for this tour we've used just an illustrative one or two:

-orientation by default, the panes are stacked vertically above one another
-height by default, the widget is sized to suit the largest component that we put in it, but we've chosen to be specific as we'll have a number of large components that we'll want to scroll in their panes

Having created the `PanedWindow`, we add our panes to it using the `add` subcommand. All that we're specifying at this point is the name of the thing we're adding into each pane, and an indication of how it is to be sized. We're not saying what sort of widget it is here, nor does the widget named have to exist at this point.

Sizing options we have used:

-min to set a minimum size; if not specified, the pane can be sized down to 0
-max to set a maximum size; if not specified, the pane can be sized as large as you like
-size the INITIAL size

The controls to allow the user to resize each of the panes are provided automatically by the megawidget; after all, it's to help the programmer easily provide such facilities that the Tix extension was written.

It remains only for us to pack our `tixPanedWindow` into our wish shell, which we do just before our application enters its event handling loop:

```
pack .overall -side left -expand yes -fill both
```

We could probably have done without the options here, as this example has just the one megawidget in the main wish shell window. We do, though, need to add in content to each pane.

Adding content to a pane

You can add content to a container widget or megawidget using more widgets or more megawidgets. It's your choice. The middle column of our sample application adds in a series of simple Tk buttons that the user can select to perform an action. The buttons are described in a list at the top of our code:

```
set server {web_server mysql_server tomcat}
```

Once we have created our container widget, with its various panes, we need to find out the name of the pane on which we're going to now work. We'll do this using the subwidget subcommand:

```
set tname [.overall subwidget target]
```

We can then write a loop to generate our ordinary Tk widgets within the pane, using the conventional (and required) `.Style` notation to indicate that the buttons live in this particular pane.

```
foreach tar $server {
  button $tname.$star -text $star -command \
    "action $star"
  pack $tname.$star
}
```

Each button has been packed as it is created, and each has a call to an action command that we'll come back to later, with the name of the server to which the button applies being passed in to the action command. Note the necessity to use `". . . ."` rather than `{ . . . }` in specifying the command.

¹ it's in essence a constructor method in OO terms

Creating a scrolled megawidget

Look back to the GUI that we're writing for this application. In the leftmost box, we require a scrolled list. We could roll our own using Tk widgets, but it's much quicker to use a Tix megawidget.

Scrolling megawidgets provide one or two scroll bars, as appropriate. There are a number of them including:

```
tixScrolledGrid
tixScrolledHList
tixScrolledListBox
tixScrolledTlist
tixScrolledText
tixScrolledWidget
```

As we need to generate a list from which a selection can be made, we'll use a **tixScrolledListBox** here:

```
set sname [.overall subwidget source]
tixScrolledListBox $sname.tsys
set clients [$sname.tsys subwidget listbox]
foreach sou $straineer {
    $clients insert end $sou
}
pack $sname.tsys
```

Yes, it really is that easy. This is a wrapper around the Tk listbox widget so all we have to do is:

- Get the name of the widget container
- Create the widget using an appropriate name so that our Tcl/Tk knows where to place it in the hierarchy
- Find the name of the listbox subwidget created by the previous step
- Add in our list box elements as if we were doing so to a regular listbox¹
- Pack the **tixScrolledListBox** widget

Creating a hierarchical widget

We've now created widgets in two of the three panes for our application – just one to go. This is going to display the results, and we want something that will show our information in a directory or folder or tree-like structure. Tix includes a number of widgets for us to choose from, including **tixTree**, **tixVTree** and **tixHList**.

We've chosen to use a **tixScrolledHList** – a scrolled hierarchical list:

```
set rname [.overall subwidget result]
tixScrolledHList $rname.log -options {
    hlist.itemType text
    hlist.indent 5 }
set rlist [$rname.log subwidget hlist]
foreach host $server {
    $rlist add $host -text $host
}
pack $rname.log
```

We collect the pane name just as we have done for the first two panes in our example, and then we construct our hierarchical list. Such lists can be much more complex than some of the other types of widgets, and we bunch our options together into a **-option** parameter. Amongst the options supported are:

¹ hey - it IS a regular listbox - it's just that it's embedded in the scrolling widget

```

-columns
-command
-indent
-indicator
-selectmode

```

Once we have created the widget, we can find the name of the **hlist** that's within the scrolling megawidget, and methods such as `add` can be used to add items to it. We use a dot-separated hierarchy tree in just the same way that the widgets themselves use such a tree. In our example, all the initial entries are at the top level so we've used just plain text words as the names.

Finally, we pack our final megawidget.

Event and action handling

Job completed? Not quite. The previous sections have taken you through the creation of a paned Window into which we have placed a variety of widgets of both the simple and mega flavours. But we need to also provide to code to do something with this GUI.

There was already a clue in the

```
-command "action $star"
```

that we provided on each of the buttons in the second column. It points to a `proc` which is going to be selected whenever a button is pressed:

```

proc action {chosenserver} {
    global rlist
    global clients
    global kidno
    global trainee
    set chosen [$clients curselection]
    set cli [lindex $trainee $chosen]
    $rlist add $chosenserver.$kidno -text "$cli - ($kidno)"
    incr kidno
}

```

The incoming "chosenserver" variable contains the name of the server whose button our user has selected to call the action routine, which is one of the two pieces of the jigsaw that we need to add another entry into our hierarchy. The second piece we need is to know which system is currently selected in our Listbox.

The `curselection` method when run on our listbox tells us the position number¹ that is selected. We take that position number and use `lindex` to find out the system name based on the list originally used to set up the listbox.

With the jigsaw complete, we can now add an item to our hierarchical widget. The item is going to be within the `$chosenserver`-th top-level entry, and it's going to display the text in `$cli`. Entry names need to be unique, and in our example that wouldn't necessarily be the case if we used the system names. They could crop up many times, so we've chosen to use a variable called "kidno" to number each of the children that we add.

There you go – a completed demonstration.

Aspects of going live

Although our demonstration is now complete, you would wish to provide some further capabilities and code if you were going to use it in real life. Here are some things to consider:

¹ and not the actual text

- You'll want to read back the results of your user's entries so that when he exits from the application his work will be saved as appropriate. Buttons such as "save to file" and "quit" would need to be provided and implemented.
- You would want to title and explain the various columns (panes) on your GUI, possibly providing balloons, megawidgets, etc. You might want to clear your selection in the first column when you add things to the third column.
- The application as provided lacks error checking. This is vital for a live system. Can you find ways to make our test code crash, or do something very silly? Find each of these and program for them.
- You may wish to enhance the output hierarchy box to allow sections to be expanded and collapsed, and to allow it to be sorted.
- Paradoxically, you might want to simplify some of the aspects of our application too. We've used different widgets in each section because this is a teaching example, but a regular user will feel much more comfortable with a number of similar looking widgets.
- Finally on this list of quick considerations, you would want to generalise out your code. At very least, you would want to read your system names and services from a data file. The resulting code might even turn out to be shorter, but let us warn you that it will be much harder for a newcomer to Tcl to follow. That's why our sample uses ugly, fixed lists of system names.

2.4 Some other megawidgets

File selection boxes and dialog boxes

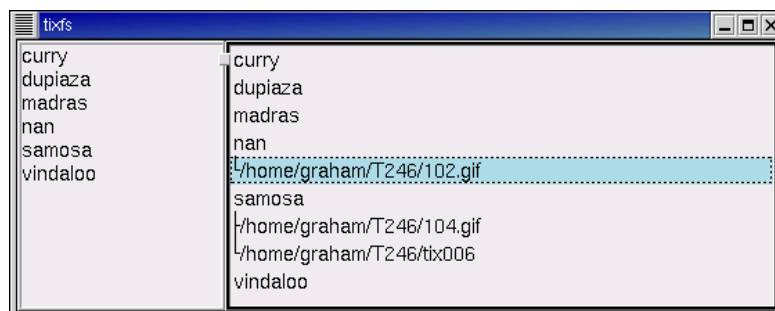
One of the major megawidget groups are the dialogue boxes. Unlike the megawidgets that you've seen so far, they are not displayed in the main window but as additional popup menus. Dialogue boxes that are available include: `tixDialogShell` and `tixStdDialogShell`.

A further major group of megawidgets deal with file and directory selections. Some of these are dialogue boxes, and others can be included within the main window, where you might use a `tixFileSelectBox`. But we'll look at an example that is a popup using a `tixFileSelectDialog`.

In this example program, we set up a scrolled list of trainee computers. When our user selects one of the host computers, we bring up a file selection dialogue and let the user choose a file which we then add to the scrolled hierarchical list in the second pane of the main window.

The resulting display is shown in Figure 8.

Figure 8 A scrolled list of trainee computers



Our file dialogue might look like Figure 9.

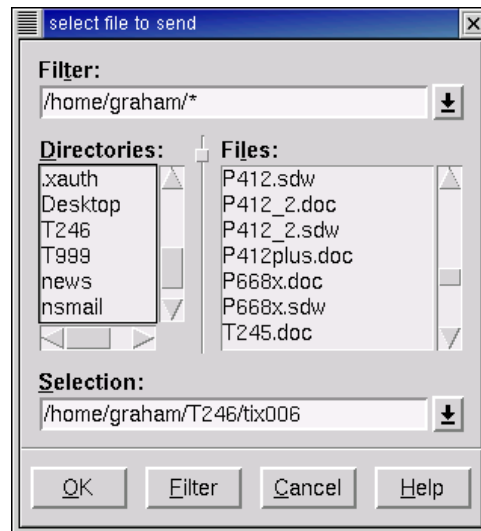


Figure 9 The file dialogue

We could use an application such as this to plan out a series of files we're going to distribute from the machine on which the application is running, to a series of trainee systems.

```
#!/usr/bin/wish

package require Tix

set trainee {curry dupiaza madras nan samosa vindaloo}

proc action {} {
    global clients
    global trainee
    global cli
    set chosen [$clients curselection]
    set cli [lindex $trainee $chosen]
    .whichfile popup
}

proc getname {filecalled} {
    global rlist
    global kidno
    global cli
    $rlist add $cli.$kidno -text "$filecalled"
    incr kidno
}

set kidno 0

# Create a paned window .... we'll have 2 panes:

tixPanedWindow .overall -orientation horizontal
.overall add source -min 100 -max 150
.overall add result -size 400

# Collect the first pane, and put in a scrolled list box

set sname [.overall subwidget source]
tixScrolledListBox $sname.tsys -command action
set clients [$sname.tsys subwidget listbox]
foreach sou $trainee {
    $clients insert end $sou
}
}
```

```

pack $sname.tsys

# Collect the second pane, and put in a tree widget
# Seed that widget (top level only) with trainee names

set rname [.overall subwidget result]
tixScrolledHList $rname.log -options {
    hlist.itemType text
    hlist.indent 5 }
set rlist [$rname.log subwidget hlist]
foreach host $trainee {
    $rlist add $host -text $host
}
pack $rname.log -expand yes -fill both

pack .overall -side left -expand yes -fill both

# Create a File Selection dialogue

tixFileSelectDialog .whichfile -title "select file to send" \
    -command getname

if [regexp "." $argv] {
    .whichfile subwidget fsbox configure -directory $argv
}

```

Let's look at how the text dialogue is set up. Its definition is just like the definition of any other widget, with a title and a command name.

```

tixFileSelectDialog .whichfile -title "select file to send" \
    -command getname

```

By default, it offers the user all files in the current directory but this can be changed using options such as **-directory** on the **fsbox** subwidget. You can give a pattern if you want to list only files with certain extensions. We'll give it a directory name to work in if one was specified on the command line:

```

if [regexp "." $argv] {
    .whichfile subwidget fsbox configure -directory $argv
}

```

We don't want to pack our dialogue! Instead, we'll run its popup method when we want it to appear as a fresh menu:

```

.whichfile popup

```

Once a selection is made, the specified command is called – "getname" in our example – with the selected file name as its parameter. The popup menu will be taken down upon acceptance of the selection without any further program requirements. Here's the code that handled the file selected in our example:

```

proc getname {filecalled} {
    global rlist
    global kidno
    global cli
    $rlist add $cli.$kidno -text "$filecalled"
    incr kidno
}

```

Tree Widgets

With the hierarchical widgets that you've seen already, you specify your own hierarchy using the names of the elements that are added into the megawidget to indicate the position. For example, we've just looked seen:

```
$rlist add $cli.$skidno -text "$filecalled"
```

The Tix megawidget package also provides tree widgets, where the position of an item within the tree is governed by its name. The classic example of the use of such a scheme is in the display of a file system tree.

The following example uses Tcl commands to parse a directory tree, looking for files that are over a certain size. The returned information is stored in a tree widget, with the full capability of expanding and contracting directories at any level. Figure 10 is what it looks like with some of the directories collapsed:

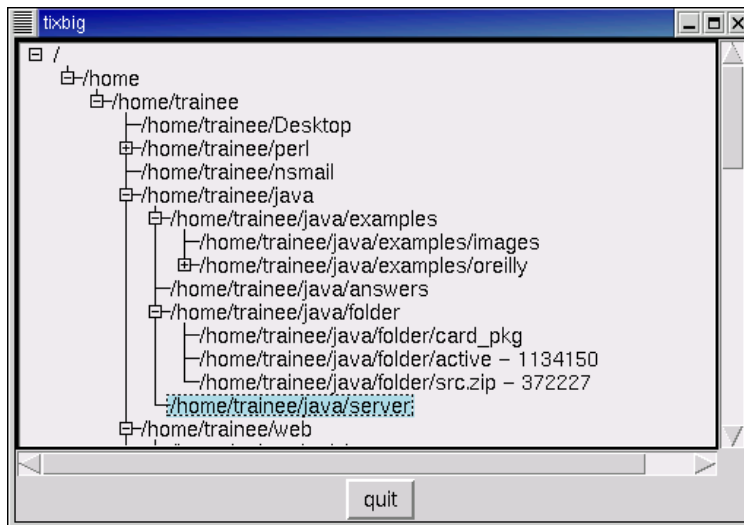


Figure 10 Parsing a directory tree

The source code:

```
#!/usr/bin/wish

package require Tix;

# Tree of big files below ....

set minsize 90000
set queue /home/trainee

tixTree .large -command doit -width 500 -height 300 -options {
    hlist.separator /
    hlist.indent 20
    hlist.drawBranch true
}

pack .large -expand yes -fill both
set hlist [.large subwidget hlist]
$hlist add / -text "/"
$hlist add /home -text /home
$hlist add /home/trainee -text /home/trainee

while {[llength $queue] > 0} {

    set current [lindex $queue 0]
    set queue [lreplace $queue 0 0]
```

```

if [catch {set files [glob "$current/*"]} oops] {} else {
    foreach f $files {
        if {[file isdirectory $f]} {
            $hlist add $f -text "$f"
            lappend queue $f
        } else {
            set size [file size $f]
            if {$size > $minsize} {
                $hlist add $f -text "$f - $size"
            }
        }
    }
}

.large autosetmode
button .quit -text quit -command exit
pack .quit

```

The tree is created as follows:

```

tixTree .large -command doit -width 500 -height 300 -options {
hlist.separator /
hlist.indent 20
hlist.drawBranch true
}

```

The separator is important – the megawidget uses it to calculate the tree position of each element that's added.

When we pack the tree, we find out the name of the list that's embedded within it, and we add the root of our tree¹ in too:

```

pack .large -expand yes -fill both
set hlist [.large subwidget hlist]
$hlist add / -text "/"
$hlist add /home -text /home
$hlist add /home/trainee -text /home/trainee

```

Other branches and leaves are added into our tree as we traverse the directory structure with Tcl, looking for large files:

```

$hlist add $f -text "$f" ; # add directory
$hlist add $f -text "$f - $size" ; # add file

```

You'll notice that we don't have to specify which element is a branch and which a leaf – our widget works that as it's generated.

If we want the ability to collapse and expand trees, we can write our own code to do so, but it's usually much easier to let the system do it automatically. Our example turns this capability on for us:

```

.large autosetmode

```

¹ and the first couple of branches in this instance

 **Exercise**

License

*These notes are distributed under the **Well House Consultants Open Training Notes License**. Basically, if you distribute it and use it for free, we'll let you have it for free. If you charge for its distribution of use, we'll charge.*

3.1 Open Training Notes License

Training notes distributed under the **Well House Consultants Open Training Notes License** (WHCOTNL) may be reproduced for any purpose PROVIDE THAT:

- This License statement is retained, unaltered (save for additions to the change log) and complete.
- No charge is made for the distribution, nor for the use or application thereof. This means that you can use them to run training sessions or as support material for those sessions, but you cannot then make a charge for those training sessions.
- Alterations to the content of the document are clearly marked as being such, and a log of amendments is added below this notice.
- These notes are provided "as is" with no warranty of fitness for purpose. Whilst every attempt has been made to ensure their accuracy, no liability can be accepted for any errors of the consequences thereof.

Copyright is retained by Well House Consultants Ltd, of 404, The Spa, Melksham, Wiltshire, UK, SN12 6QL - phone number +44 (1) 1225 708225. Email contact - Graham Ellis (graham@wellho.net).

Please send any amendments and corrections to these notes to the Copyright holder - under the spirit of the Open Distribution license, we will incorporate suitable changes into future releases for the use of the community.

If you are charged for this material, or for presentation of a course (Other than by Well House Consultants) using this material, please let us know. It is a violation of the license under which this notes are distributed for such a charge to be made, except by the Copyright Holder.

If you would like Well House Consultants to use this material to present a training course for your organisation, or if you wish to attend a public course is one is available, please contact us or see our web site - <http://www.wellho.net> - for further details.

Change log
Original Version, Well House Consultants, 2004

Updated by: _____ on _____
Updated by: _____ on _____
Updated by: _____ on _____
Updated by: _____ on _____
Updated by: _____ on _____
Updated by: _____ on _____

License Ends.