It’s time to wrap up the stock robot and stock graphing package. The last *Tclsh Spot* article showed how to read the stock data, extract data about a single stock, and graph that data. Now it’s time to graph multiple stocks at once, and save our graph as a postscript file.

The procedure we created in the last article to read the stock performs these steps:

- grab each line of data from the file.
- split the line into fields with a `foreach` loop.
- use the `normalize` procedure (discussed in the last article) to convert values like `1 3/4` or `23,456` into computer friendly representations .
- save the data in lists named by the stock symbol and type of data.

The code looks like this:

```
proc readData {infl} {
    global Data

    while {[set len [gets $infl line]] >= 0} {
        set id [lindex $line 1]
        foreach n {date id price change pct dt o high low vol} v $line {
            lappend Data($id.$n) [normalize $v]
        }
    }
}
```

The two dimensional array of lists is a convenient way to save the data for this application. It gives us quick (hash-time) access to each dataset, and presents the data in the list format that `BLT` will use to graph it.

The `makeGraph` procedure we used in the last article is more complex than we want when plotting multiple stocks on a single graph. Trying to squeeze sales volumes, High-Low values and multiple stocks into a graph gets too crowded to be intelligible.

So, `makeGraph` gets trimmed down to just creating the basic graph, and configuring it to display the month/day on the X axis, instead of seconds since the epoch.

```
proc makeGraph {parent} {
  global Data

      # Create the graph

      ::blt::graph $parent.g  -title "Stock Data" -width 600 -height 400
      $parent.g axis configure x -max [clock seconds]

      # Format the x-axis tick labels as Month/Day
      $parent.g  axis configure x -command fmt

  return $parent.g
}
```

At this point, we’ve got a graph, and we’ve got some data, and we’ve got to decide what we want to display.

We could set up some sort of command line parser, and invoke the application as `graphStock SUNW INTC MSFT` to show Sun, Intel and Microsoft. But, this would get old fast.

A better solution is to select stocks from a menu, and interactively remove one or more when we are tired of them.

This brings us to the Tk `menu` commands.

A menu is one of the best widgets for selecting one of several options. Tk includes support for menu buttons that look like normal buttons, menu buttons that sit in a menubar, and menu buttons that display the current choice.

For this application, we'll create menu buttons that look like command buttons. When a user clicks one of these buttons, a list of stock symbols will appear below it to be selected. When the user selects a stock symbol, the data for that symbol will be graphed.

Tk uses two widgets to display a menu: the `menubutton` that appears on the screen, and the container class, `menu`, that is associated with each `menubutton` to contain the menu entries for that `menubutton`.

Syntax: `menubutton` *`buttonName ?options?`*

`menubutton`	Create a `menubutton` widget
`buttonName`	The name for this `menubutton`
`?options?`	The `menubutton` supports many options. This application will use:

`-text` *`displayText`*	The text to display on this button
`-relief` *`STYLE`*	Describes the 3-D effect for the menu button. Options are `raised, sunken, flat, ridge, solid,` and `groove` The default (`flat`) is good for menubars. The value `raised` makes the menu buttons resemble regular buttons.
`-menu` *`menuName`*	The name of the `menu` widget associated with this `menubutton`

A `menu` widget is a container widget that holds the menu entries to be displayed.

Syntax: `menu` *`menuName ?options?`*

`menu`	Create a `menu` widget
`menuName`	The name for this `menu` widget. Note that this name must be a child name to the parent `menubutton`. i.e.: if the `menubutton` is `foo.bar`, the `menu` name must resemble `foo.bar.`*`baz`* .
`?options?`	The `menu` widget supports several options. A couple that are unique to this widget are:

`-postcommand` *`script`*	A script to evaluate just before a menu is posted.
`-tearoff` *`boolean`*	Allows (or disallows) a menu to be removed from the

menu button and displayed in a permanent window. This is enabled by default.

The common procedure for creating a Tk menu is:

1. Create a menubutton using the `menubutton` command.
2. Create a menu using the `menu` command.
3. Add items to the menu with the *`menuName`* `add` command.

As with other Tk widgets, when you've created a `menu` widget, Tk creates a procedure to use to interact with that widget. The menu widget supports many subcommands, but this example will just use the `add` command.

Syntax: *`menuName`* `add` *`type ?options?`*

`menuName` `add`	Add a new menu entry to a menu.
`type`	The type for this entry.
`?options?`	The options that will be used in this example are:
	`-command` *`script`* — A script to evaluate when this entry is selected
	`-label` *`string`* — The text to display in this menu entry.

The types of items that can be added to a menu include:

`separator`	A line that separates one set of menu entries from another
`cascade`	Defines this entry as one that has another menu associated with it, to provide cascading menus.
`command`	Same as the standalone button widget.

Here's a small example of building a menu. It gives the user a choice of colors, and reconfigures the menubar to be that color.

```
menubutton .mb -menu .mb.menu -text "Color" -relief raised
menu .mb.menu

foreach color [list red orange yellow green blue] {
  .mb.menu add command -label $color \
      -command ".mb configure -background $color"
}
```

This code creates a menu that resembles this:

.

Which brings us back to deciding what to do with our stock data.

We could put all the stock symbols under a single menu button, labeled something like "Select one stock out of a huge list". One feature of the Tk menu is that if you add a hundred items to the menu, it will still work, but most of the menu will fall off the bottom of your screen.

Perhaps we should split the stocks into several smaller menus.

Tk supports as many cascading menus as an application might need. In an application where you have a lot of buttons at the top of the application, this is a good way to hide complexity from the user. In this application, we'll only have our stock symbol buttons, and a `Files` menu, so we can use a simpler approach.

We can arrange our stock symbols within alphabetic ranges under menu buttons with names like **A-E**, **F-L**, etc.

A quick way to build these buttons is:

```
set b [frame .buttons]
foreach range [list "A-E" "F-L" "M-R" "S-Z"] {
    set mb [menubutton $b.m_$range -text $range \
        -menu $b.m_$range.menu -relief raised]
    menu $b.m_$range.menu
    pack $mb -side left -padx 5
}
```

The things to note in this code fragment are:

- The names are generated on the fly from the displayed text, with a lowercase `m_` prepended.

  Tk widget names must start with a lowercase letter. If you are building widgets on the fly, it's safest to prepend a known lowercase letter to avoid getting caught later by something that starts with a uppercase or numeric value.

- The buttons are put into a separate frame, rather than being put onto the main frame.

  Using frames for the major sections of a GUI makes life a bit easier if someone wants the buttons to go down the left margin, or along the bottom, or whatever later.

- The buttons are put into their frame with `pack` instead of grid.

> You can use either `grid` or `pack` within a frame, but you can not use both geometry managers in the same frame. Using pack is a bit simpler when you are just going to put widgets in a row.

The next step is to add the stock symbols to the menus, and the first step there is to find out just what stock symbols were defined in our data file.

We could have kept a list of stock symbols as we read in the data, checked for duplicates, etc, but the introspective nature of Tcl makes our life a bit easier.

There are several `array` commands that can be used to find out details about an associative array. One of the most useful is `array names`

Syntax: `array names` *`arrayName ?pattern?`*

`array`	Defines this as an array command
`names`	Returns a list of the indices used in $
`arrayName`	The name of the array.
`pattern`	If this option is present, array names ret$ all the array indices.

We could use `array names Data` to get a list of all the indices in the Data array. But, that would give us *ALL* the indices : 9 items per stock symbol. We really just want one entry per stock symbol. So, we use the *`pattern`* option, and select only the indices that end in `.date` with this line:

```
array names Data *.date
```

One trick with array names is that it returns the indices in the order they appear in the hash table. This is not a random order, but it can look like one.

The whole idea of putting the stock symbols under alphabetic menu buttons is to make it easier to find the one you want, so we need to place the data into the menus in a sorted order. We could look at the data that has already been placed in a menu, and use the *`menuWidget`* `insert` subcommand to insert the new item in the proper place, but it's simpler to just use the `lsort` command to presort the results of the `array names` command, and then just use the `add` subcommand to put the items into the menu.

So, our loop code looks like this:

```
foreach id [lsort [array names Data *.date]] {
 # Do stuff
}
```

The next step is to figure out which menu to add the item to.

Again, the introspective nature of Tcl and a clever naming scheme saves us from having to keep track of what menu buttons have been created.

The pack command can tell us the names of the widgets that have been packed in a frame. The command for this is pack slaves *master*, which returns a list of the window names that are contained in the *master* frame.

We can use the split command to extract the start and end letters of a range from the menu button name with this piece of line noise:

```
  foreach ch [pack slaves $b] {
      foreach {start end} [split [lindex [split $ch _] end] -] {}
```

The pack slaves $b command returns names resembling .buttons.m_A-E.

The innermost lindex and split commands ([lindex [split $ch _] end] separates the A-E from the rest of the name. The outer split command splits A-E into a list, and the empty-bodied foreach loop puts **A** into the start variable, and **E** into the end variable.

Finally, we can use the string compare command to figure out if a symbol is within the range for this menu button.

Syntax: string compare *string1 string2*

Returns a value of 1, 0, or -1 if *string1* is alphabeticly greater, equal of less than *string2*.

After all that discussion, here is the code that will add the stock symbols to the menus.

```
foreach id [lsort [array names Data *.date]] {

  # extract the stock symbol from the id - id resembles SUNW.date
  set name [lindex [split $id .] 0]

  # Look at the menu buttons, and figure out which range this is in.
  foreach ch [pack slaves $b] {
      foreach {start end} [split [lindex [split $ch _] end] -] {}
       if {([string compare $name $start] >= 0) &&
           ([string compare $name $end ] <= 0)} {
          $ch.menu add command -label $name -command "makeElement $graph $name"
      }
  }
}
```

The last item in the menu discussion is that -command option.

The previous version of the stock graphing package the makeGraph procedure also created the graph elements. In this version, we've split creating elements from generating the graph.

When we are putting several lines on a graph, it can be difficult to tell one from the other. Using different colors for each line, and showing the colors in the legend makes it easy to tell them apart.

The graph command supports a -color option to define the color of a line. To make your life simpler, the default behavior of creating a graph element automaticly creates a legend showing the

color and style of the line.

One way to create lines in different colors is to define a list of colors, and iterate through the list as you create graph elements.

```
set Graph(colors) {blue pink green orange yellow black gray}
set Graph(current) 0

proc makeElement {graph name} {
  global Data Graph

  # Create a line showing the stock price when the robot ran.

  set el [$graph  element create "$name Price" -xdata $Data($name.date) \
        -ydata $Data($name.price) -symbol none \
        -color [lindex $Graph(colors) $Graph(current)]]

  # Step to the next color, and loop to the beginning when we reach
  #  the end.
  incr Graph(current)

  if {$Graph(current) >= [llength $Graph(colors)]} {
      set Graph(current) 0
  }
```

Using the colors is good, but, we can still do more with this graph.

For instance. Now that we've got a nice easy way to add lines to the graph, it might be nice to be able to get rid of lines when we get tired of them.

One of the features of the BLT graph widget is that you can bind actions to the graph lines and legends. You could use this to highlight a particular line or range, bring up a popup with more information, or even to delete a line.

The BTL graph widget bind command looks like this.

Syntax: *graphName itemType* bind *ID eventType script*

graphName	The BLT graph to have an action bound to it.
itemType	The type of component of the graph that will have an action bound to it. May be element, legend, or marker.
ID	The identifier for this component.
eventType	The event to trigger this action. Events can be defined in one of three formats:

alphanumeric	A single printable (alphanumeric or punctuation) character defines a KeyPress event for that character.
<<virtualEvent>>	A virtual event defined by your script with the event command.
<modifier-type-detail>	This format precisely defines any event that can occur. The fields of an event descriptor are described below. Where two names are together, they are

synonyms for each other. For example, either `Button1` or `B1` can be used to the left mouse button click.

script The script to evaluate when this event occurs.

This code will cause a line to be deleted when you right-mouse-button-click either the line, or the legend:

```
  $graph element bind $el  [list $graph element delete $el]
  $graph legend bind $el  [list $graph element delete $el]
```

That code lets us add and delete lines on the graph, so lets take one more look at menu tricks.

One of the Tcl strong suits is how easy it is to build loops to handle what would be many lines of repetitive coding in other languages.

It's pretty common to have a menu that contains several options, each of which evaluates some command. This is easy to construct as a loop.

For example, Microsoft made us all familiar with the ubiquitous **Files** menu that has so few items dealing with files. If we have several commands that we want to place under the files menu, we can define procedures with the same name as the button prompt, and enter them into the menu with a loop like this:

```
foreach selection {Clear Postscript Exit } {
    $menu add command -label $selection -command "$selection $graph"
}
```

The procedures for these commands range from a one liner for `Exit`:

```
proc Exit {g} {
    exit
    }
```

to procedures that introduce some more BLT features.

The `Clear` procedure will remove all the lines from a graph with a single menu click. Again, our code could maintain a list of all the lines on a graph, and remove items from the list when we remove them from the graph, etc. But, BLT will do that for us. The fewer lines of code I write to accomplish a task, the fewer I need to debug later.

Like other Tcl widgets, the BLT widget is introspective, the `element names` command, like the `array names` command will return a list of elements.

```
proc Clear {g} {
    foreach e [$g element names] {
       $g element delete $e
    }
```

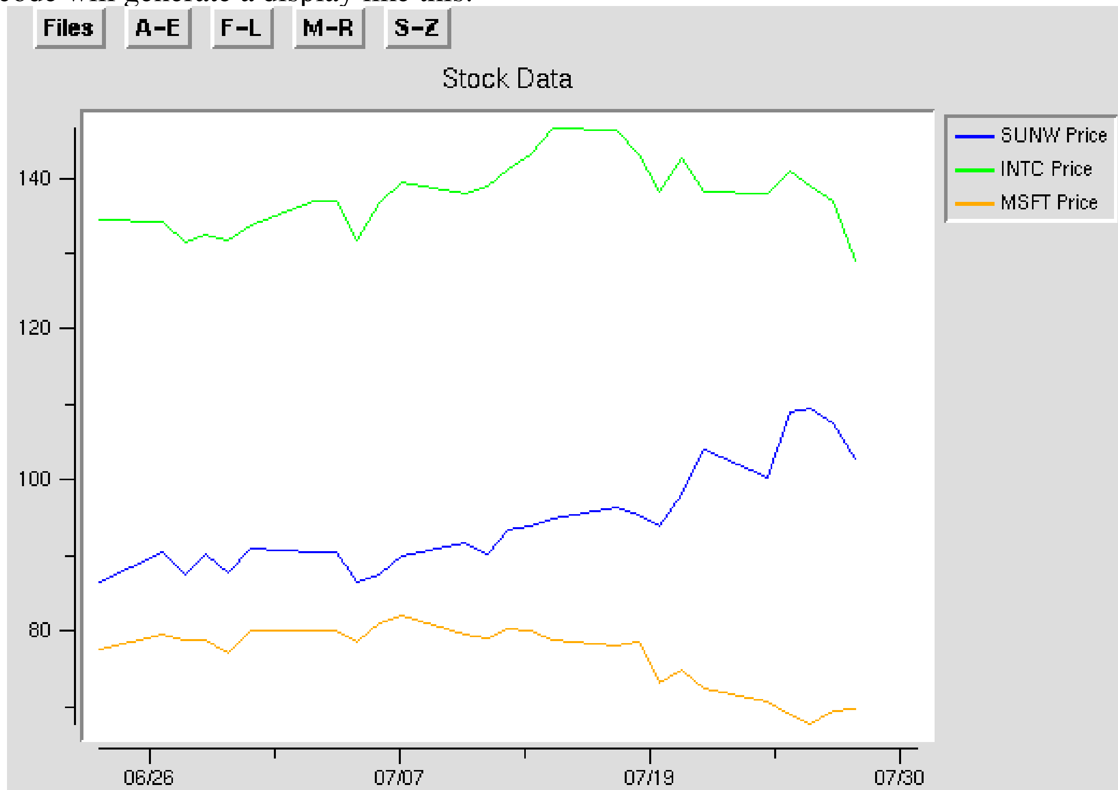
```
}
```

Finally, it's nice to have a hardcopy of graphs. BLT supports a `postscript` command that emulates the behavior of the Tk canvas `postscript` command. This procedure will convert the displayed data into a postscript file named `stockGraph.ps`.

Tk supports widgets for selecting file names, but for this example, a hardcoded name will suffice.

```
proc Postscript {g} {
   $g postscript output stockGraph.ps
}
```

This code will generate a display like this:



The next *Tclsh Spot* article will get start looking at some Tcl tools for system monitoring.

This code, and code from other *Tclsh Spot* articles, is available at `http://www.noucorp.com`

NOTE TO EDITORS: This article is already long. Feel free to delete this code listing if you need space.

```
#!/usr/local/bin/bltwish
namespace import ::blt::*

set Graph(dataName) stock.data

set Graph(colors) {blue pink green orange yellow black gray}
```

```
set Graph(current) 0

proc normalize {val} {

    # Only do fraction to decimal conversion if a fraction exists.

    if {[regexp {(([0-9]* +)*([0-9]*)/([0-9]*)} $val m whole num denom]} {
        set val [expr $whole + ($num / $denom.0)]
    }

    # Delete any commas that might be in the value
    regsub -all "," $val "" val

    return $val
}


proc readData {infl} {
    global Data

    while {[set len [gets $infl line]] >= 0} {
        if {[string first "N/A" $line] > 0} {continue}
        set id [lindex $line 1]
        foreach n {date id price change pct dt o high low vol} v $line {
            lappend Data($id.$n) [normalize $v]
        }
    }
}

proc fmt {graph sec} {
   return [clock format $sec -format {%m/%d}]
}

proc makeElement {graph name} {
  global Data Graph

  # Create a line showing the stock price when the robot ran.

  set el [$graph  element create "$name Price" -xdata $Data($name.date) \
        -ydata $Data($name.price) -symbol none \
        -color [lindex $Graph(colors) $Graph(current)]]

  incr Graph(current)

  if {$Graph(current) >= [llength $Graph(colors)]} {
      set Graph(current) 0
  }

  $graph element bind $el <Button-3> [list $graph element delete $el]
  $graph legend bind $el <Button-3> [list $graph element delete $el]

  return $graph
}

proc makeGraph {parent} {
  global Data

      # Create the graph

      ::blt::graph $parent.g  -title "Stock Data" -width 600 -height 400
      $parent.g axis configure x -max [clock seconds]
```

```
      # Format the x-axis tick labels as Month/Day
      $parent.g  axis configure x -command fmt

  return $parent.g
}

set infl [open $Graph(dataName) r]
readData $infl
close $infl

set b [frame .buttons]
set w [frame .graphs]

grid $b -row 0 -column 0
grid $w -row 1 -column 0  -columnspan 20
set graph [makeGraph $w ]
grid $graph

set mb [menubutton $b.m_file -text Files -menu $b.m_file.menu -relief raised]
set menu [menu $b.m_file.menu]
pack $mb -side left -padx 5


foreach range [list "A-E" "F-L" "M-R" "S-Z"] {
    set mb [menubutton $b.m_$range -text $range \
        -menu $b.m_$range.menu -relief raised]
    menu $b.m_$range.menu
    pack $mb -side left -padx 5
}

foreach id [lsort [array names Data *.date]] {

#  # extract the stock symbol from the id - id resembles SUNW.date
  set name [lindex [split $id .] 0]

  # Look at the menu buttons, and figure out which range this is in.
  foreach ch [pack slaves $b] {
      foreach {start end} [split [lindex [split $ch _] end] -] {}
       if {([string compare $name $start] >= 0) &&
           ([string compare $name $end ] <= 0)} {
          $ch.menu add command -label $name -command "makeElement $graph $name"
      }
  }
}

proc Exit {g} {exit}

proc Clear {g} {
    foreach e [$g element names] {
       $g element delete $e
    }
}

proc Postscript {g} {
   set outputfile [tk_getSaveFile]
   if {![string match $outputfile ""]} {
       $g postscript output $outputfile
   }
}
```

```
foreach selection {Clear Postscript Exit } {
    $menu add command -label $selection -command "$selection $graph"
}
```