# the tclsh spot

**by Clif Flynt**

Clif Flynt is president of Noumena Corp., which offers training and consulting services for Tcl/Tk and Internet applications. He is the author of *Tcl/Tk: A Developer's Guide* and the *TclTutor* instruction package. He has been programming computers since 1970 and a Tcl advocate since 1994.

*clif@cflynt.com*



By the mid '80s, it was generally recognized that FORTRAN was a dead language, and all that was left was to rework a few calculation libraries in C and bury the corpse.

This turns out to not quite be the case. FORTRAN has evolved and is still being used. Or, to quote an anonymous source: "We don't know what language engineers will be coding in in the year 2100. However, we do know that it will be called FORTRAN."

In the last month, I've seen three different projects add new functionality (GUI or network support) to a FORTRAN program.

Thanks to Arjen Markus, of WL Delft Hydraulics (a big engineering and FORTRAN shop), it's easy to embed the Tcl/Tk interpreter into the FORTRAN main code, providing yet another face-lift to an old compiler.

So, a brief digression from the articles about firewall validation to discuss embedding the Tcl interpreter into FORTRAN applications.

The first question is probably, "Why?" After all, most scripting applications are written using the pattern of extending the interpreter so you can write your application in the scripting language and call into a compiled library to do the heavy lifting, not embedding an interpreter into a compiled application.

The big reason is that the old code works. It may not do everything we'd like it to do, it may be cranky about input format, and it may have no GUI, but any conceptual flaws were revealed and fixed decades ago. Unfortunately, the act of making the old code work usually means that any pretense of architectural purity has long since been lost, and refactoring it into a library won't be fast or easy.

By adding the Tcl interpreter to a functional FORTRAN program, we can easily extend the program in ways that FORTRAN doesn't normally support. Tcl's clean socket support makes it easy to add client-server support to the program, and Tk makes it simple to add a GUI.

The first FORTRAN program I ever used was the Lunar Lander program, run from cards on the printing console of an IBM 1130. So, when I needed a project for relearning FORTRAN and playing with Markus's package, I decided to reimplement that program with a nice interactive GUI.

Markus has developed a FORTRAN->Tcl interface library that exposes the minimal subset of the Tcl "C" API to FORTRAN and allows the Tcl interpreter to be embedded in a FORTRAN program. It provides entry points to start the Tcl interpreter, evaluate a set of Tcl code, and exchange data between the compiled FORTRAN and interpreted Tcl sections of an application.

The body of the Lunar Lander code loops until the rocket hits the surface. Within that loop, it queries the user for the amount of fuel to burn and calculates the current height and speed. It looks like this:

```
      DO WHILE (fheight .GT. 0)
         IF (fuel .GT. 0) THEN
            WRITE(*,*) 'Enter burn: '
            READ(*,*) burn
         ELSE
            fuel = 0
            burn = 0
         ENDIF
         CALL calcspeed (speed, fuel, gross, burn, impulse, speed)
         i = i + 1
         fuel = fuel - burn
         fheight = fheight - speed
         WRITE(*,100) i, speed, fuel, fheight
 100     FORMAT('TIME: ', I3, ' SPEED: ', F8.2, ' FUEL: ', F8.2, ' HEIGHT: ', F8.2)
      ENDDO
```

A few runs looks something like this:

```
      Enter burn: 0.0
TIME:    1 SPEED:    101.70 FUEL:   1000.00 HEIGHT:       9898.30
      Enter burn: 0.0
TIME:    2 SPEED:    103.40 FUEL:   1000.00 HEIGHT:       9794.90
      Enter burn: 10.0
TIME:    3 SPEED:    103.31 FUEL:    990.00 HEIGHT:       9691.59
      Enter burn: 10.0
TIME:    4 SPEED:    103.20 FUEL:    980.00 HEIGHT:       9588.39
```

The first step in merging the Tcl interpreter into this code is to compile the FORTRAN-Tcl source files and create a library. This requires editing the makefile to reflect your environment, and then make ftcl.a. There are configuration options in ftcl_mod.c to support the C interface of different FORTRAN compilers.

Now that we've got a compiled library, we can start modifying the FORTRAN code.

The ftcl_start subroutine initializes the Tcl interpreter. This merges the Tcl C library Tcl_CreateInterp, Tcl_Init, and Tcl_EvalFile commands into a single subroutine.

The ftcl_start subroutine expects to be able to find the Tcl (and possibly Tk) libraries installed at runtime. The program will run if the libraries aren't installed, but only the core Tcl commands will be available, not the Tk graphics or other extensions.

**Syntax:** ftcl_start scriptname

scriptname The name of a Tcl script to load into the interpreter.
This line will initialize the Tcl interpreter.

```
CALL ftcl_start('config.tcl')
```

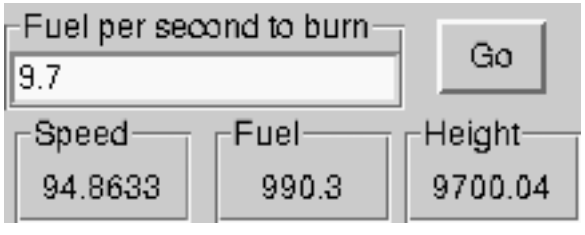
At runtime the config.tcl file is loaded and evaluated. This config.tcl script loads Tk and the GUI code and then builds the GUI to start the application.

```
package require Tk
source GUI-1.tcl
buildGUI 10000
```

The buildGUI procedure can generate any Tk GUI we'd like.

A GUI to duplicate the original Lunar Lander might have an entry widget to accept the amount of fuel to burn, a button to let the program know when we're ready to proceed, and a few labels to display the current height, speed, and remaining fuel.

It would look like this:




The entry and label widgets are contained within labelframe widgets. The labelframe widget is a container widget that can draw an outline around its edge and place a label on that outline. A labelframe can be used to group a related set of widgets, or identify a single widget. The labelframe creates a cleaner-looking GUI than the older technique of putting a label widget next to an associated data widget.

**Syntax:** labelframe .widgetName ?-option value?

widgetName The name for this labelframe.

-option value An option/value pair to configure the labelframe. Common options include:

-text *label text*
The text to display as a label for this frame.

-labelanchor *anchor*
Defines how to place the label. May be one or two of the letters n, s, e, w.

A single letter describes which side to put the label on, with the top being n, the right being e, etc.

If the anchor is two letters, the first defines the side of the labelframe for the text, while the second letter defines which side to anchor the label to, with the default being to center the text.

The default value for this is nw.

The first labelwidget contains a Tk entry widget. The entry widget is used to allow a user to enter any sort of textual information, such as login ID, password, or the amount of fuel to burn.

**Syntax**: entry .entryName ?-option value?

entryName The name for this entry.

-option value An option/value pair to configure the entry. Common options include:

-textvariable *variableName*
The contents of the entry widget will be automatically placed in the variable variableName.

-width *number*
Set the entry widget to be number characters wide. The default value is 20.

The -textvariable option simplifies creating a GUI. You don't need to write code to query the widgets; just use the variable name.

Like all Tk widgets, the labelframe command returns the name of the widget that was created. Using this name with commands related to this widget (grid, or creating child widgets) makes your code more robust if the GUI needs to be modified. For instance, in the code below, the frame and entry widget can be moved to another frame or top-level window by changing only the labelframe command.

The code to create the label frame and entry widget looks like this:

```
set w [labelframe .lfb -text "Fuel per second to burn"]
grid $w -row 1 -column 1 -sticky ew -columnspan 2
entry $w.burn -textvariable burn
grid $w.burn
```

The modern GUI could not exist without the button. The Tk button command creates a button that can contain an image, text, or both, and will invoke a command when the button is clicked.

**Syntax:** button .buttonName ?arguments?

A couple of commonly used arguments are:

-text string The text to display on the button.

-command body The body of a command to evaluate when the button is activated.

The code that defines this button is:

```
button .b -text "Go" -command "set ready 1"
grid .b -row 1 -column 3
```

Finally, there are the three labels showing the current height, speed, and remaining fuel.

The Tk label widget displays a string. Similar to the entry widget, a Tk label can be linked to a variable and will automatically update itself to display the contents of that variable when the variable is modified.

**Syntax:** label .labelName ?-option value?

-textvariable variableName This label will display the contents of the named variable.

-text string This label will display a particular string.

These widgets can be created and displayed in a foreach loop:

```
set col 0

foreach txt {Speed Fuel Height} var {speed fuel ht}  {
    set w [labelframe .lf$var -text $txt]
    grid $w -row 2 -column [incr col]
    set w [label $w.l-$var -textvariable $var -width 8]
    grid $w
}
```

Now, we need to move data to and from this GUI from the FORTRAN code.

The ftcl package supports a family of subroutines to copy data between the Tcl interpreter and the FORTRAN variables. These include:

ftcl_get_int(TclVariableName, FortranVariableName)
Copy an integer value from a Tcl variable to a FORTRAN variable.

ftcl_get_real(TclVariableName, FortranVariableName)
Copy a real value from a Tcl variable to a FORTRAN variable.

ftcl_get_double(TclVariableName, FortranVariableName)
Copy a double value from a Tcl variable to a FORTRAN variable.

ftcl_get_string(TclVariableName, FortranVariableName)
Copy a string value from a Tcl variable to a FORTRAN variable.

ftcl_put_int(TclVariableName, FortranVariableName)
Copy an integer value from a FORTRAN variable to a Tcl variable.

ftcl_put_real(TclVariableName, FortranVariableName)
Copy a real value from a FORTRAN variable to a Tcl variable.

ftcl_put_double(TclVariableName, FortranVariableName)
Copy a double value from a FORTRAN variable to a Tcl variable.

ftcl_put_string(TclVariableName, FortranVariableName)
Copy a string value from a FORTRAN variable to a Tcl variable.

In each case, the TclVariableName is passed as a string, while the FortranVariableName is just the name of the FORTRAN variable.

The command associated with the Go button sets the variable ready to 1 when the user clicks it. We can set and read that variable into a FORTRAN variable named irdy with these lines:

```
irdy = 0
CALL ftcl_put_int('ready', irdy)
! Pass control to the Tcl event loop
CALL ftcl_get_int('ready', irdy)
```

Like other interactive systems, Tcl has an event loop that collects inputs from the outside world (timer events, mouse movements, keyboard events, etc.) and processes them. While the FORTRAN code is being executed, the event loop isn't being processed, and the Tk GUI is inactive. The next trick is to transfer control to the Tcl interpreter to run the event loop and make the GUI active.

The ftcl_script subroutine passes a string to be evaluated to the Tcl interpreter. This string can be a single command or a longer script. We can process the event loop with the update command, which causes the Tcl interpreter to run a single pass through the event loop, and process a single event. This code is a round-robin polling loop that waits for the user to press the Go button:

```
irdy = 0
CALL ftcl_put_int('ready', irdy)
DO WHILE (irdy == 0)
   CALL ftcl_script('update')
   CALL ftcl_get_int('ready', irdy)
ENDDO
```

Polling loops are simple, but they eat up all available CPU cycles.

The vwait command described a few "Tclsh Spot" articles ago will cause a script to pause until a variable changes value. The interpreter stops at the vwait command and enters the event loop, processing events until the variable is assigned a new value. After this, the interpreter continues evaluating the commands in the script.

Internally, Tcl uses the select system library call to wait for events, rather than polling. Using the vwait command reduced the CPU usage of the Lander program from 50–80% to under 1%.

**Syntax:** vwait varName

varName The variable name to watch. The script following the vwait command will be evaluated after the variable's value is modified.

A simple procedure to wait for the button to be pressed looks like this:

```
proc wait4click {} {
    global ready
    vwait ready
}
```

And the FORTRAN main loop code looks like this:

```
DO WHILE (fheight .GT. 0)
    irdy = 0
    CALL ftcl_put_int('ready', irdy)
    CALL ftcl_script('wait4click')

    ! The Tcl burn variable now contains the amount of
    ! fuel to burn.

    CALL ftcl_get_real('burn', burn)


    ! If we're out of fuel, no burn.
    IF (fuel .LE. 0) THEN
       fuel = 0
       burn = 0
       CALL ftcl_put_real('burn', burn)
    ENDIF

    ! Calculate the speed.
    CALL CALCSPEED (speed, fuel, gross, burn,
          impulse, speed)

    ! Update the FORTRAN variables.
    i = i + 1
    fuel = fuel - burn
    fheight = fheight - speed

    ! Update the Tcl variables.
    CALL ftcl_put_int('time', i)
    CALL ftcl_put_real('speed', speed)
    CALL ftcl_put_real('fuel', fuel)
    CALL ftcl_put_real('ht', fheight)
ENDDO
```

The last step is to compile the new FORTRAN code and link with the ftcl and Tcl libraries. Using the GNU FORTRAN compiler, it looks like this:

```
g95 -o lander lander.f90 ftcl.a -L/usr/local/lib -ltcl8.4
          -ltk8.4 -lm
```

At this point, we've used 21st-century technology to duplicate the behavior of a 1960s program. It feels like it should have exposed rivets and be the size of a walk-in freezer.

The next "Tclsh Spot" article will look at using a Tk GUI to display the information graphically and run in realtime.

As usual, this code (including a version of Arjen Marcus's ftcl library) is available at *http://www.noucorp.com*.