What debugging tools are available to a Tcl programmer

Updated 2012-12-16 17:17:13 by pooryorick ▲

wiki.tcl.tk

Summary edit

pointers to useful tips, tools, functions, and techniques for Tcl programmers needing to locate and remedy problems in their Tcl (or Tk) programs.

Show Actual Arguments to a Command Invocation edit

RS: Small but useful when put at the beginning of a proc (or further down):

```
puts [info level 0]
```

Interactive mode edit

To see where the error was triggered in interactive mode, use

```
set errorInfo
```

This variable shows index of the faulty line inside the procedure body - **FP** (for the beginners)

```
info body _the_name_of_the_procedure_
```

RS: When debugging interactively, I typed *set errorInfo* quite often - until I considered that writing once

```
interp alias {} ? {} set errorInfo
```

reduces this chore to a single (and quite intuitive) question mark command...

kpv: Alternatively, if you use tkcon, you can use the following alias

```
interp alias {} ? {} tkcon error
```

In-Code Messages edit

As far as I can tell, the primary tool used by most Tcl programmers is the *puts* command. That is to say - most programmers seem satisfied to modify the existing code (or a copy of it) adding in puts commands.

Pros: Provides feedback at appropriate places within the code.

Cons: Requires one to either modify the existing code, or to maintain parallel copies. Adding puts has the potential to altering the code enough that the bug **disappears** or at least moves. Some code may not be available to modify. Some code may in fact be binary applications and not as easily updated.

DKF: Since I tend to put each source file in a separate namespace, I can easily have a separate command (per namespace) that behaves like puts in 'debugging' mode and is a no-op in 'production' mode, avoiding at least part of the problem listed above

One minimal switchable debugger looks like this:

```
proc dputs args {puts stderr $args} ;# on
proc dputs args {}                  ;# off (RS)
```

A slightly more elaborate version renders existing variable names in the caller as "name=value", which is often intended; other words in the arguments are passed through unchanged:

```
proc dputs args {
    set res [list]
    foreach i $args {
        if [uplevel info exists $i] {
            lappend res "$i=[uplevel set $i]"
        } else {
            lappend res $i
        }
    }
    puts stderr $res
}
```

Example output for *hello world, what is this*:

```
hello world, what is {this=C:/WINNT/Profiles/suchenwirth/Eigene Dateien/sep/sep17.tcl}
```

The curlies are added because of the space in the pathname. This will not work with arrays, which cannot be dereferenced by $name.

Another debugging dputs that I've used is:

```
proc dputs {msg} {
    global debugMode
    if { $debugMode } { puts $msg }
}
```

Then, just setting the global variable debugMode switches debugging information on and off-- and can be done at run time through a configuration parameter, rather than prior to wrapping.

[Lectus] 2011-04-22 10:13:49:

I like custom-made debugging tools a lot. Here is a proc I built upon previous examples in this page:

We use the DEBUG variable to tell if we're debugging or not. And with proc we add a new operator !# which outputs debugging messages only if debugging is activated.

```
proc !# {args} {
    global DEBUG
    if {$DEBUG == 1 || $DEBUG == TRUE || $DEBUG == true || $DEBUG == yes || $DEBUG == on || $DEBUG == ON} {
        set res [list]
        foreach i $args {
             if [uplevel info exists $i] {
                 lappend res "$i=[uplevel set $i]"
             } else {
                 lappend res $i
             }
        }
        puts stderr $res
    }
}
```

And here we test it:

```
set DEBUG 1

puts "Enter X value: "
gets stdin x
puts "Enter Y value: "
gets stdin y
puts "The sum is: [expr $x+$y]"
!# x y
```

Result:

```
$ tclsh test.tcl
```

```
Enter X value:
2
Enter Y value:
3
The sum is: 5
x=2 y=3

set DEBUG 0

puts "Enter X value: "
gets stdin x
puts "Enter Y value: "
gets stdin y
puts "The sum is: [expr $x+$y]"
!# x y
```

Result:

```
$ tclsh test.tcl
Enter X value:
2
Enter Y value:
3
The sum is: 5
```

Verbose Evaluation edit

RWT: This proc will evaluate a script "verbosely," printing each command, then executing it, then printing the result. It isn't quite as general purpose as */bin/sh* 's **set -v** because it won't step into procedures.

```
proc verbose_eval {script} {
    set cmd ""
    foreach line [split $script \n] {
        if {$line eq ""} {continue}
        append cmd $line\n
        if { [info complete $cmd] } {
            puts -nonewline $cmd
            puts -nonewline [uplevel 1 $cmd]
            set cmd ""
        }
    }
}
```

Test it out with a little script.

```
set script {
    puts hello
    expr 2.2*3
}
verbose_eval $script
```

I've also used a function like:

```
proc debugMe {{args {}}} {
    global debug

    if { [info exists debug] && $debug} {
        set level1 [expr {[info level] - 1}]
        set level2 [expr {[info level] - 2}]
        if { $level1 > 0 } {
            set name1 [lindex [info level $level1] 0]
            if { $level2 > 0 } {
                set name2 [lindex [info level $level2] 0]
            } else {
                set name2 Startup
```

```
            }
        } else {
            set name1 Startup
            set name2 User
        }
        puts "Proc: $name1; Called by $name2; [info cmdcount] commands at [clock format [clock seconds]]"
        if { $args != {} } { puts "$args" }
    }
}
```

which is based loosely on one of the procs in the BOOK ActiveState Tcl Cookbook. It's useful to figure out how you got into a procedure and how long different procedures are taking to run (DMS).

AM 2008-12-30: Here is another variant on this theme: a simple approximation to the type of tracing "bash -x" offers:

```
set code ""
set infile [open [lindex $argv 0]]
set argv [lrange $argv 1 end]
while {[gets $infile line] >= 0} {
    puts $line
    if { [info complete $line] } {
        puts [eval $line]
    } else {
        append code "$line\n"
        if { [info complete $code] } {
            puts [eval $code]
            set code ""
        }
    }
}
close $infile
```

Note: it is a very rough script, as it makes no attempt to hide the local variables and it will only print the code outside any proc. But with some work it can be put into a very decent shape.

The use is simple (save it as "trace.tcl"):

```
tclsh trace.tcl myscript.tcl ...
```

Wrapping Proc edit

NEM - Well, I've got to have a go at this. You can easily wrap proc to do lots of debugging stuff:

```
if {$debug} {
    rename proc _proc
    _proc proc {name arglist body} {
        _proc $name $arglist [concat "proc_start;" $body ";proc_end"]
    }
    _proc proc_start {} {
        puts stderr ">>> ENTER PROC [lindex [info level -1] 0]"
        for {set level [expr [info level] -1]} {$level > 0} {incr level -1} {
            puts stderr "  LEVEL $level: [info level $level]"
        }
        puts stderr ""
    }
    _proc proc_end {} {
        puts stderr ">>> LEAVE PROC [lindex [info level -1] 0]\n"
    }
}
```

Just bung that at the top of any file you want to instrument, before defining any procs, and bingo. You can build arbitrarily complex debugging levels on top of that. Of course, you may want to be a bit more selective and just redefine individual procs, which is pretty easy to do with [info body] and friends.

XoTcl Interceptors edit

A more general approach are the interceptors (filters and mixin classes) in XoTcl. With the interceptor, one can add an arbitrary code in front of a method and call the original method via next

```
className instproc someFilter args {
    #### pre-part
    set r [next]
    ### post-part
    return $r
}
```

For more details, see the "filter" message interception technique of XOTcl

Trace edit

Also, check out the new trace subcommands in 8.4 - trace add execution. Here is a powerful debugging aid built on top of them (a more fine grained version of the above):

```
rename proc _proc
_proc proc {name arglist body} {
    uplevel 1 [list _proc $name $arglist $body]
    uplevel 1 [list trace add execution $name enterstep [list ::proc_start $name]]
}
_proc proc_start {name command op} {
    puts "$name >> $command"
}
```

Now Tcl will helpfully print out every command that executes in every proc from this point onwards (this could be a *lot* of output!)

---

trace all command calls into a single file edit

Sarnold It can help to find what command crashes the interpreter (for example 8.5a3)

```
# overwrite at each invocatoion of this script
set __XXXfd [open c:/WINDOWS/Temp/tcltrace.tmp w]
proc __XXXtrace {cmd type} {
    puts $::__XXXfd $cmd
}

rename proc _proc
_proc proc {name arglist body} {
    uplevel [list _proc $name $arglist $body]
    uplevel trace add execution $name enter __XXXtrace
}
```

Global variable watch & control edit

the following little goody displays the values of global variables (no arrays - you'd use a modified version for those), but you can also change them from the window:

```
proc varwindow {w args} {
    catch {destroy $w}
    toplevel $w
    set n 0
    foreach i $args {
        label $w.l$n -text $i -anchor w
        entry $w.e$n -textvariable $i -bg white
        grid $w.l$n $w.e$n
```

```
        incr n
    }
} ;#RS
#usage example:
varwindow .vw foo bar baz
```

Just make sure your other code calls update in long-running loops, so you get a chance to change. Limit: you can't scroll the thing, so better add no more variables that your screen can hold ;-)

Wrapping Set edit

You can even write your own set command (make sure its effects are like the original, otherwise most Tcl code might break). For instance, this version says what it does, on stdout:

```
rename set _set
proc set {var args} {
    puts [list set $var $args]
    uplevel _set $var $args
} ;#RS
```

Might help in finding what was going on before a crash. When sourced in a wish app, shows what's up on the Tcl side of Tk. Pretty educative!

Minimal Stepper edit

For inspecting what a Tk application does in a loop, you may insert the following line at the position in question:

```
puts -nonewline >;flush stdout;update;gets stdin ;#RS
```

Expect edit

Csan: how about `expect -c 'strace N' script.tcl`, where N is the level which you want the trace dig into .

LV: This is a wonderful tip - it can show you each line of tcl as it is interpreted.

Csan: For tracing variable access (setting, reading, unsetting) see Tcl's built-in trace command - wonderful one also. Combine it with expect's strace and you are in Tcl Debug Heaven ;)

Csan also mentions that expect supports:

```
expect -D 1 script.tcl
```

which then invokes an interactive debugger that allows to you single step, set breakpoints, examine variables, manipulate procs, etc (type 'h' at the dbg> prompt to get a short list and explanation of the available commands).

Finding Source Code edit

LV On the chat today, I was asking how to determine what package and dynamic libraries a package require might have loaded. kbk suggested info loaded - which after a package require lists what things were loaded.

See where libs get loaded from:

```
#!/usr/bin/env tclsh
package require Tk
foreach item [info loaded] {
    foreach {lib pkg} $item break
    puts "lib = $lib"
    puts "pkg = $pkg"
```

```
    exec ldd $lib
}
exit
```

Note there is a problem here - at least when LV runs this script, lib shows 2 *words* and pkg shows none.

Also note that if you change the tclsh above to wish, you get something different. The info loaded command reports {} in place of the library name.

AK: Fixed the script. info loaded returns a list of 2-element lists, not a dictionary.

tcllib's profiler edit

EKB: I put together the following code, which can be placed at the top of a Tk app (requires tcllib's profiler). With this:

- Putting "DEBUG *msg*" will open a message box (OK, this is pretty basic, but handy)
- Pressing control-p will show selected profiling info (exclusive runtime by proc as a % of total)
- Pressing control-h will show a list of opened file channels (the "h" is for "handle")
- Pressing control-l will show a list of windows, with the window class and, for text widgets, a list of tags

```
set DEVEL_VER true

proc DEBUG {msg} {
    global DEVEL_VER

    if {$DEVEL_VER} {
        tk_messageBox -icon info -message "Debugging information:\n\n$msg"
    }
}

proc LONGDEBUG {msg} {
    # This gives a scrollable window for a long message
    set tlvl_num [clock seconds]
    while [winfo exists .debug_$tlvl_num] {
        incr tlvl_num
    }

    set tlvl ".debug_$tlvl_num"
    toplevel $tlvl

    ScrolledWindow $tlvl.sw

    text $tlvl.sw.t -font "Courier 8"

    $tlvl.sw setwidget $tlvl.sw.t

    $tlvl.sw.t insert end $msg
    $tlvl.sw.t config -state disabled

    pack $tlvl.sw -fill both -expand yes
}

proc GET_PROFINFO {} {
    set procinfo [::profiler::sortFunctions exclusiveRuntime]
    set numprocs [llength $procinfo]
    set tottime 0
    set pnames {}
    set ptimes {}
    for {set i [expr $numprocs - 1]} {$i >= 0} {incr i -1} {
        set item [lindex $procinfo $i]
        set ptime [lindex $item 1]
        incr tottime $ptime
        lappend pnames [lindex $item 0]
        lappend ptimes $ptime
    }
```

```
    set msg {}
    foreach pname $pnames ptime $ptimes {
        if {$ptime == 0} {break}
        set frac [expr (100 * $ptime) / $tottime]
        if {$frac == 0} {set frac "< 1"}
        lappend msg "$frac%\t$pname"
    }
    return [join $msg "\n"]
}

proc GET_WINDOWS_recurse {window listvar} {
    upvar $listvar wlist

    set class [winfo class $window]
    set item "$window ($class)"
    if {$class eq "Text"} {
        set item "$item\n\t[join [$window tag names] \n\t]"
    }
    lappend wlist $item
    foreach child [winfo children $window] {
       GET_WINDOWS_recurse $child wlist
    }
}

proc GET_WINDOWS {} {
    set wlist {}
    GET_WINDOWS_recurse . wlist
    return [join $wlist "\n"]
}

if {$DEVEL_VER} {
    package require profiler
    ::profiler::init
    bind . <Control-p> {LONGDEBUG [GET_PROFINFO]}
    bind . <Control-l> {LONGDEBUG [GET_WINDOWS]}
    bind . <Control-h> {LONGDEBUG [join [file channels] "\n"]}
}
```

PED edit

etdxc 2006-05-09: I really like Mark Smith's runtime procedure editor, PED. I add the statement; bind all <Key-F12> "::ped::create" as the last line so I can simply drop it into an app and it self initialises when (auto) sourced.

Misc edit

Anyone have pointers to help debugging Starkits? - RS: That's like asking: "How to repair a watch wrapped in shrink-wrap plastic?" Unwrap it, debug it, then wrap again if needed :)

LV Unless, of course, the bug has to do with running from within a starkit...

AMG: A coworker asked me: "Do you have a recommendation for a Tcl/Tk debugger?" Here is my response.

> Do you have a recommendation for a Tcl/Tk debugger?

*I usually just insert [puts] lines, etc., into my code. Sometimes I do this by editing the *.tcl files on disk, and sometimes I insert them at runtime using Tkcon. Tkcon supports attaching to a remote interpreter via X11 and other protocols.*

> I've been looking at the ones listed on the Tclers Wiki, just wondered what your thoughts/experiences are.

For a long time now I've been meaning to look at debuggers to find one with good support for breakpoints,

watchpoints, and single-stepping. The Tcl [trace] command makes these possible, plus [info] provides tons of introspection capabilities. A debugger can be written using these commands. I just never looked into it, with one exception. I once played around with TclPro, which had these features and worked just fine, only that it hasn't been updated since Tcl 8.3.2.

http://wiki.tcl.tk/3875#pagetoc9ec85e31

http://wiki.tcl.tk/8637

Tkcon has a command called [idebug] which you can insert into your code to create breakpoints and stack dumps.
http://wiki.tcl.tk/1878

I think most long-time Tcl programmers don't feel the need for a dedicated debugger utility, because it's usually sufficient to edit the program at runtime (use [puts] to print program state information, or put this information in a widget or titlebar) and to interactively invoke code (add a command line to the GUI, attach with Tkcon, or run pieces of the program in a separate interpreter). I've used these techniques for years and have never needed anything more.

I try to embed Tkcon into every major GUI program I write. That way it's easily scriptable and debuggable, even when run on MS-Windows.

LV Note that the ideas in TclPro evolved into Tcl Dev Kit.

Arts and crafts of Tcl-Tk programming | Category Debugging