

# The vw+ Debugging system

## Introduction

This describes a lightweight tool for debugging a tcl/tk program. It has several features and benefits.

- ◆ Simple to install and use – designed for the new programmer to aid understanding tcl/tk
- ◆ Inserted into a program at any time with only a single *source* statement
- ◆ Can dynamically instrument code for single stepping a procedure and remove it later
- ◆ Can view global, namespace, and local variables in a procedure or TclOO method
- ◆ Can tailor debugging to a specific set of procedure(s).
- ◆ Animation effects to run slowly and watch variables and code progress

This pdf describes a smallish self contained variable browser and debugger with breakpoints and stepping.

It is implemented with a single file that is sourced into a program, usually at startup, but can also be attached to a running program as well.

The single file defines 6 procedures:

- ◆ vw+            The viewer for global/namespace variables and arrays
- ◆ bp+            A low level breakpoint command that is called by the user's program
- ◆ lbp+           The same but for procedures and methods with local variables
- ◆ go+            The command to resume from a breakpoint
- ◆ util+           Utility commands
- ◆ instrument+   Command to insert breakpoints into a procedure or TclOO method

To minimize impact, there is only one global (array) variable used, and the file includes a section at the top where this array is initialized, as discussed later.

There are also 3 optional aliases that can be used (which can be changed or commented out to not use) that provide 3 single letter abbreviations (v, g, and u, for vw+, go+, and util+) for the 3 commands often typed in by the user.

When the file is sourced, it only sets values in its global array variable and defines the 6 procedures. It takes no further actions until some of the above commands are used. It is thus safe to source anytime.

In TclOO class variables are stored in namespaces. If you know the namespace for an object, you can use the vw+ procedure to monitor those variables. When you create a new object, the TclOO system returns to you the namespace that the new object will use. The lbp+ break-point can be used in OO methods as well (and single stepped using instrument+). lbp+ breakpoints can also be used inside of a constructor.

The data viewing procedure can be used independently of the code stepping and breakpoints. It can provide a quick way to monitor (and change) variable values.

## Viewing Variable Data

There are a few ways to use this tool. The first is for small programs with code that is simply down the page outside of all procedures that makes use of global variables, perhaps including some in a namespace.

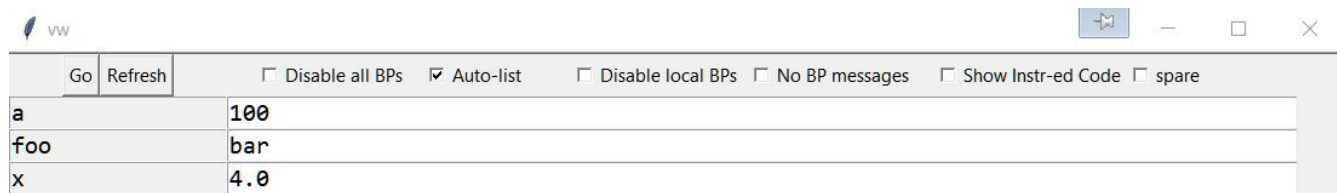
Variables are viewed in toplevel windows that are created by the `vw+` command (aliased as `v`). This command creates a 2 column, spreadsheet like grid, with the variables and their values in the 2 columns. The values in these windows can be changed while the program is running.

The tool is designed to work with the console (including the console for linux) where commands may be entered and data is output to it by the debugger (and the user program). Values can be modified by entering data in the values column.

To get started, suppose we have just a few lines of code at the beginning of a tcl file. To use the `vw+` system, we would add the debugging code shown in **red**: The debugger source file can be anywhere the user installed it. Install is nothing more than placing the `vw+` system's 1 tcl file in any chosen directory.

```
source {path/to/vw+source.tcl} ;# loads the vw+ debugger
console show
set a 100 ;# sets up some variables
set foo bar
set x 4.0
vw+ ;# show the global variables window
```

When we run this with `wish` (or a `telkit` with the `tk` gui) we will get the following window plus a console (I suggest using one even on linux, where there's a wiki page on this and a TIP to include one on linux in the future) and one empty `tk` main window (the one called `.` or `dot`)



The screenshot shows a window titled 'vw' with a toolbar containing 'Go', 'Refresh', and several checkboxes: 'Disable all BPs', 'Auto-list' (checked), 'Disable local BPs', 'No BP messages', 'Show Instr-ed Code', and 'spare'. Below the toolbar is a table with two columns: variable names and their values.

a	100
foo	bar
x	4.0

Above we see our 3 global variables. There are, of course, several others used by the system which did not show up in the window. This is by design and is accomplished by one of the statements near the top of the source'd file which saves a list of all the initial global variables and so only shows the ones added since then. If you want a window to include all of them, you would enter a (string match like) pattern:

```
vw+ *
```

This parameter always has another `*` added at the end, so if you want only the global variables that begin with "err" you could enter,

```
vw+ err
```

Note that in these forms of the command, each `vw+` command would reuse the window with the default name of `.vw` and you can see that in the title (less the dot). If you want multiple windows with different

names one adds a second parameter.

For example, if you want 3 windows,

```
vw+  
vw+ * allglobals  
vw+ err errorvars
```

Note, that you can either put these commands in your program file or type them into the console. The alias comes in handy, and the command reduces to “v”.

This vw+ command takes several forms. Interactively, from a console, one can get help:

```
vw+ ?      (or v ? with the alias)
```

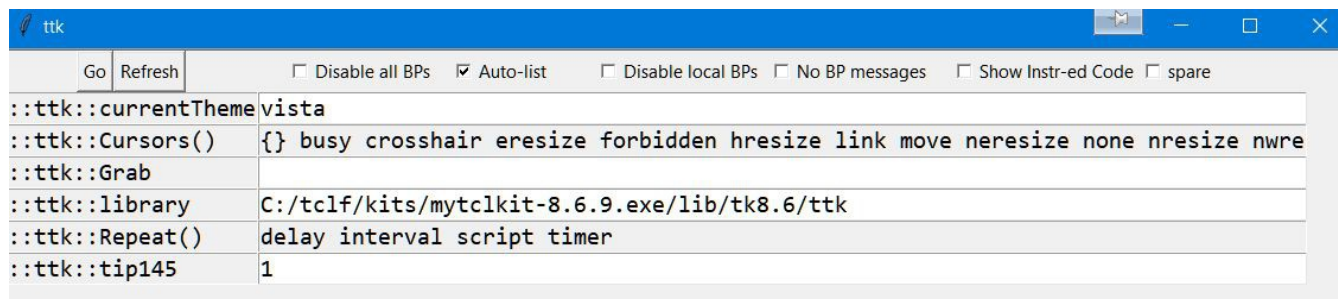
vw+ can take a list of variables or a pattern in the first argument. If a list, it must be 2 or more and be placed in {}'s.

```
v {foo bar baz a_global_array(x)} fooey
```

If you want to work with variables defined in a namespace, you enter the namespace name followed by two colons. For example, if you want to see the variables in the ::ttk namespace you would enter,

```
v ttk:: ttk
```

and you might get this: (you can also supply a pattern after the :: to see fewer variables, default is \*):



The screenshot shows a window titled 'ttk' with a toolbar containing 'Go', 'Refresh', and several checkboxes: 'Disable all BPs' (unchecked), 'Auto-list' (checked), 'Disable local BPs' (unchecked), 'No BP messages' (unchecked), 'Show Instr-ed Code' (unchecked), and 'spare' (unchecked). Below the toolbar is a table of variables and their values:

::ttk::currentTheme	vista
::ttk::Cursors()	{ } busy crosshair eresize forbidden hresize link move neresize none nresize nwre
::ttk::Grab	
::ttk::library	C:/tcltk/kits/mytclkit-8.6.9.exe/lib/tk8.6/ttk
::ttk::Repeat()	delay interval script timer
::ttk::tip145	1

With the variable names on the left and their values on the right we can see the variable *currentTheme* has the value *vista*. If the variable is an array, then the value (s) shown above are not the data, but rather

```
-----  
The array ::ttk::Cursors()  
-----  
:::ttk::Cursors() = wait  
:::ttk::Cursors(busy) = wait  
:::ttk::Cursors(crosshair) = crosshair  
:::ttk::Cursors(eresize) = size_we  
:::ttk::Cursors(forbidden) = no  
:::ttk::Cursors(hresize) = size_we  
:::ttk::Cursors(link) = hand2  
:::ttk::Cursors(move) = fleur  
:::ttk::Cursors(neresize) = size_ne_sw  
:::ttk::Cursors(none) =  
:::ttk::Cursors(nresize) = size_ns  
:::ttk::Cursors(nwresize) = size_nw_se  
:::ttk::Cursors(sresize) = size_nw_se  
:::ttk::Cursors(sresize) = size_ns  
:::ttk::Cursors(standard) = arrow  
:::ttk::Cursors(swresize) = size_ne_sw  
:::ttk::Cursors(text) = ibeam  
:::ttk::Cursors(vresize) = size_ns  
:::ttk::Cursors(wresize) = size_we
```

the indices. If you want to see the values of an array, you would *left click on the array name*. For example, the *Cursors* array produced the above on my windows 10 system (onto the console). A single array element is viewed just like a simple variable.

If you are not using a console on linux, this output would go to the terminal window (stdout). It uses the tcl [parray] command to do the output. Note, some output goes to *stderr*, since with a console, this produces output in *red text*.

## List and dictionary display

A *shifted left click* on a variable will attempt to format the output as a dictionary variable. For example

The screenshot shows a Tcl IDE with a variable named `websters`. The variable is defined as an array containing 18 elements, each representing a word and its definition. The IDE has a menu bar with 'Help' and 'Extra2', and a toolbar with 'Go', 'Refresh', and checkboxes for 'Disable all BPs', 'Auto-list', and 'Disable local BPs'. The variable `websters` is highlighted in the toolbar, and its value is displayed in a separate window.

The variable `websters` is defined as follows:

```
websters =
  Abacination {(n.) The act of abacinating.}
  Abaciscus {(n.) One of the tiles or squares of a tessellated pavement; an abaculus.}
  Abacist {(n.) One who uses an abacus in casting accounts; a calculator.}
  Aback-1 {(adv.) Toward the back or rear; backward.}
  Aback-2 {(adv.) Behind; in the rear.}
  Aback-3 {(n.) An abacus.}
  Abaser {(n.) He who, or that which, abases.}
  Abashed {(imp. & p. p.) of Abash}
  Abashing {(p. pr. & vb. n.) of Abash}
```

The Dictionary `websters` is displayed as follows:

```
Abacination => |(n.) The act of abacinating.|
Abaciscus   => |(n.) One of the tiles or squares of a tessellated pavement; an abaculus.|
Abacist     => |(n.) One who uses an abacus in casting accounts; a calculator.|
Aback-1     => |(adv.) Toward the back or rear; backward.|
Aback-2     => |(adv.) Behind; in the rear.|
Aback-3     => |(n.) An abacus.|
Abaser      => |(n.) He who, or that which, abases.|
Abashed     => |(imp. & p. p.) of Abash|
Abashing    => |(p. pr. & vb. n.) of Abash|
```

The List `websters` is displayed as follows:

```
0 => |(adv.) Behind; in the rear.|
1 => |(adv.) Toward the back or rear; backward.|
2 => |(imp. & p. p.) of Abash|
3 => |(n.) An abacus.|
4 => |(n.) He who, or that which, abases.|
5 => |(n.) One of the tiles or squares of a tessellated pavement; an abaculus.|
6 => |(n.) One who uses an abacus in casting accounts; a calculator.|
7 => |(n.) The act of abacinating.|
8 => |(p. pr. & vb. n.) of Abash|
9 => |Abacination|
10 => |Abaciscus|
11 => |Abacist|
12 => |Aback-1|
13 => |Aback-2|
14 => |Aback-3|
15 => |Abaser|
16 => |Abashed|
17 => |Abashing|
```

Also *a control left click* will format as a simple list, but in a single column and unsorted, so in the order in the list, while *an alt left click* will output the list sorted using -dictionary as the lsort option. | bars | are added so if there are any trailing or leading spaces, they are easier to see.

## Break-points

The data windows also have a few buttons and check-boxes. They are for use with break-points. The two procedures used for break-points are **bp+** and **lbp+**. The bp+ command uses the vwait tcl command to wait until a variable is set which is part of the event system. While the program is paused at the point where the bp+ or lbp+ commands are made one can view the several windows that might be open, or enter commands into the console (or the 2 console like entry widgets described below).

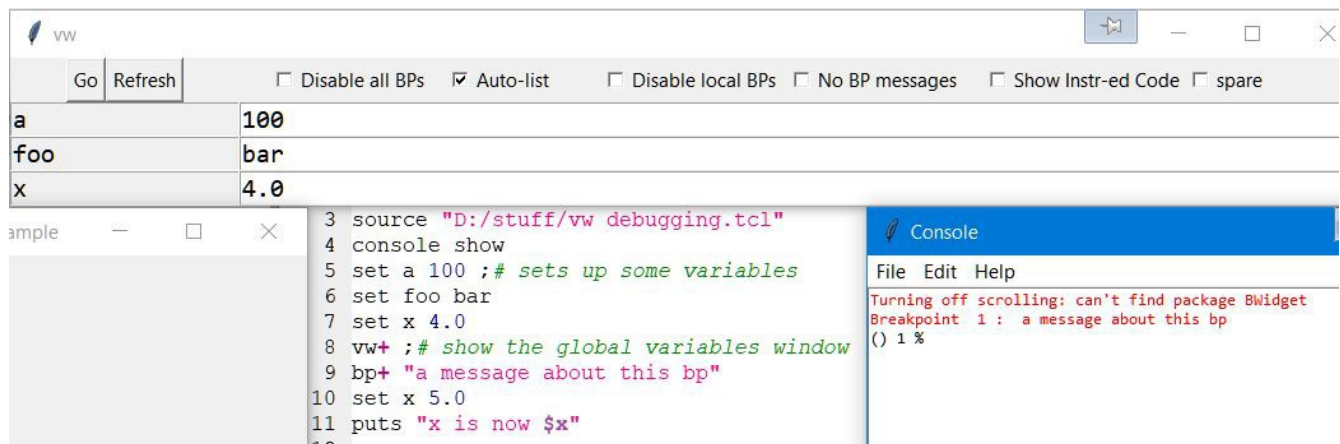
Globals and Namespace variables can be changed by typing into the section on the right, except for arrays. They are grey'd out and are readonly. You can copy the indices, but you can't change their values. Of course, you can change the array's values in the console with a set statement or any thing else that you might run, such as calling some command or a procedure you have written. Individual array elements can be changed however, if they were included using the {...} variable list.

The “Go” button is used to continue from a break-point.

Next, let's add a break-point to our example program (shown in red), which is followed by a change to the variable x.

```
source {pathto/vw+source.tcl} ;# loads the vw+ debugger
set a 100 ;# sets up some variables
set foo bar
set x 4.0
vw+ ;# show the global variables window
bp+ "a message about this bp"
set x 5.0
puts "x is now $x"
```

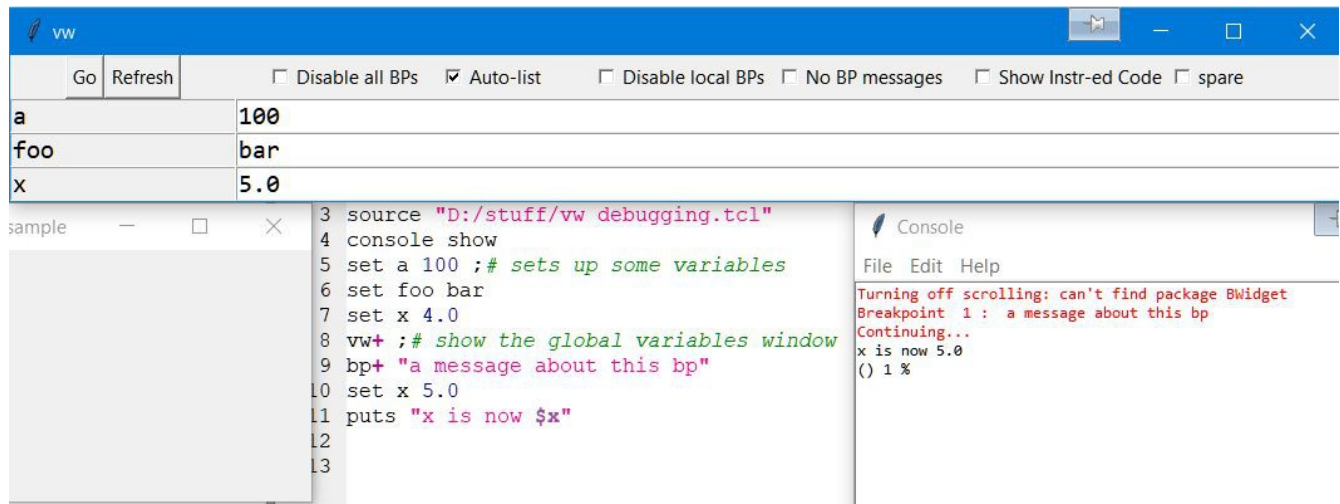
Running this program will result in a pause and the message in the first parameter (optional) would be shown on stdout (or the console). It might look like so (shown on top of a text editor window):



At this point you can continue by clicking the go button. This would set the value of x to 5.0 which you



will see in the variable window called vw since the vw+ command did not enter a window name. The puts statement will also have run, outputting its data to the console:



At this point, the program will have reached the end of the file, but will not exit, since it's a tk event program and it will pause in the event loop. You can check the variable windows to see the final values of the variables. The only other actions at this time would be to exit by closing the tk main window or you could type in commands to the console. You could also type exit to quit. **Update note:** the spare checkbox is now a keep window on-top control, plus there's another to turn off auto sizing of the window.

That is the simple method where there are no procedures. For procedures, with dynamic data it gets a bit more interesting.

## Working with Procedures or TclOO methods

Procedures and methods are handled a bit differently. In that case, you would use the **lbp+** command for a breakpoint instead. Suppose we now have defined a procedure “hello” like so,

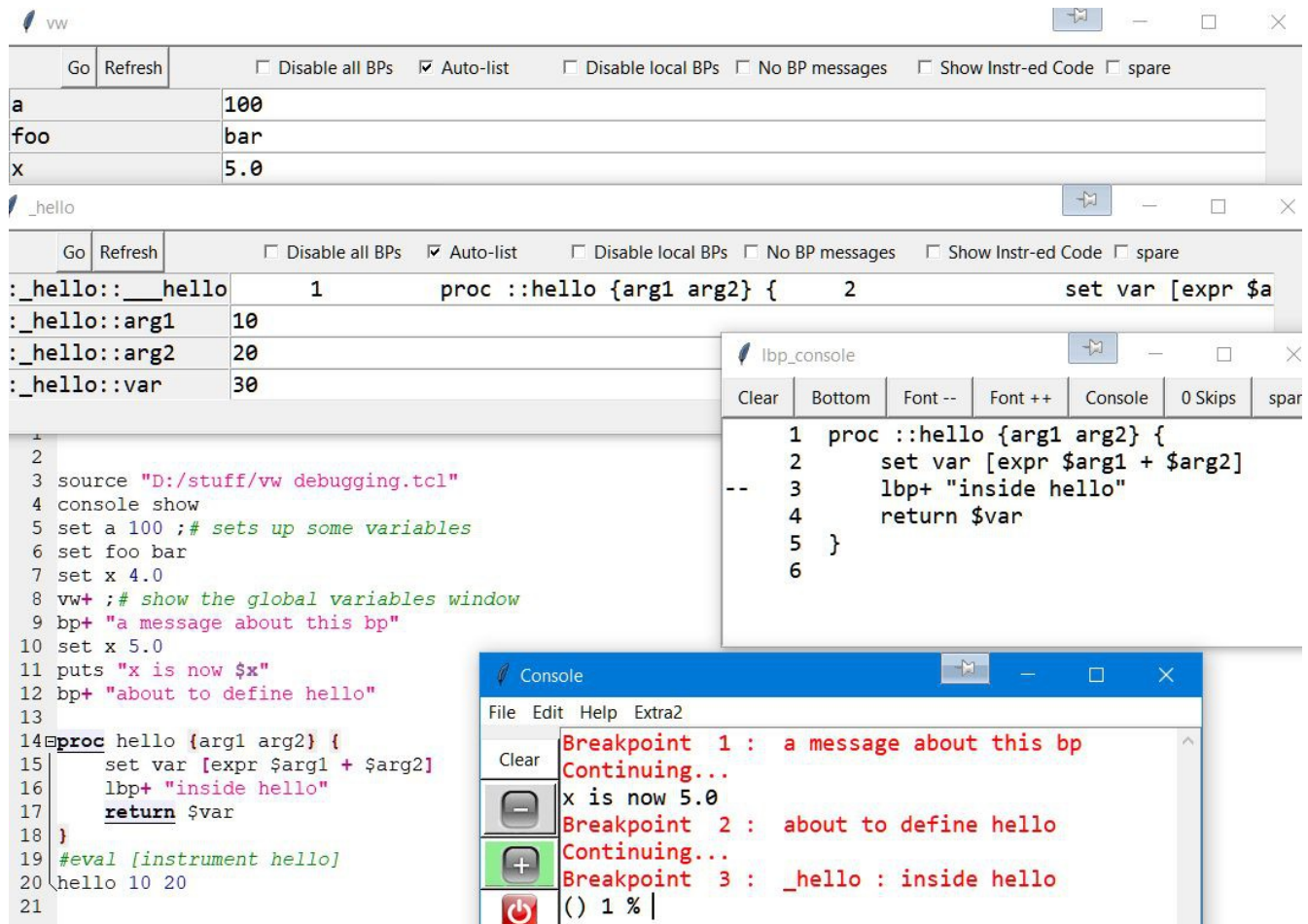
```

source "D:/stuff/vw debugging.tcl"
console show
set a 100 ;# sets up some variables
set foo bar
set x 4.0
vw+ ;# show the global variables window
bp+ "a message about this bp"
set x 5.0
puts "x is now $x"
bp+ "about to define hello"

proc hello {arg1 arg2} {
    set var [expr $arg1 + $arg2]
    lbp+ "inside hello"
    return $var
}
hello 10 20

```

and when we run this, we click go 2 times to proceed from the 2 break-points. We would then see these windows:



\* Note this is just a modified console, but otherwise its just the same. Ignore for now the commented `eval` command, it will be discussed below (since this screenshot it's been renamed `instrument+`).

What we see now, is that on reaching the `lbp+` command with the label “inside hello” 2 more windows have appeared. One is called `_hello` (the `_` is added by the debugger, explained later) and the other is called `lbp_console`. This method doesn't actually display the local variables, but rather it copies them into a namespace and then uses the namespace viewer command from `vw+`. The `_` is added in case the user might want to have a namespace which is the same as the procedure.

One can modify the namespace copies and their new values are copied back to the local variables in the procedure `hello` – but only after a restart. Arrays cannot currently be modified in a local procedure.

The second window that shows up is used to list the current source code for the `hello` procedure. The “`__`” points to where the program has stopped on the breakpoint.

Note that the `lbp+` procedure will need a unique comment, since that is how the debugger finds the current line to display the “`--`”. There can be a 3<sup>rd</sup> parameter to the `lbp+` which would be used instead

and should be a unique text string. This is also used when we get to the next section on instrumentation. **NOTE:** break-points are NOT recursive, and any procedure that includes a break-point call will run while paused at a break-point but these break-points will be ignored.

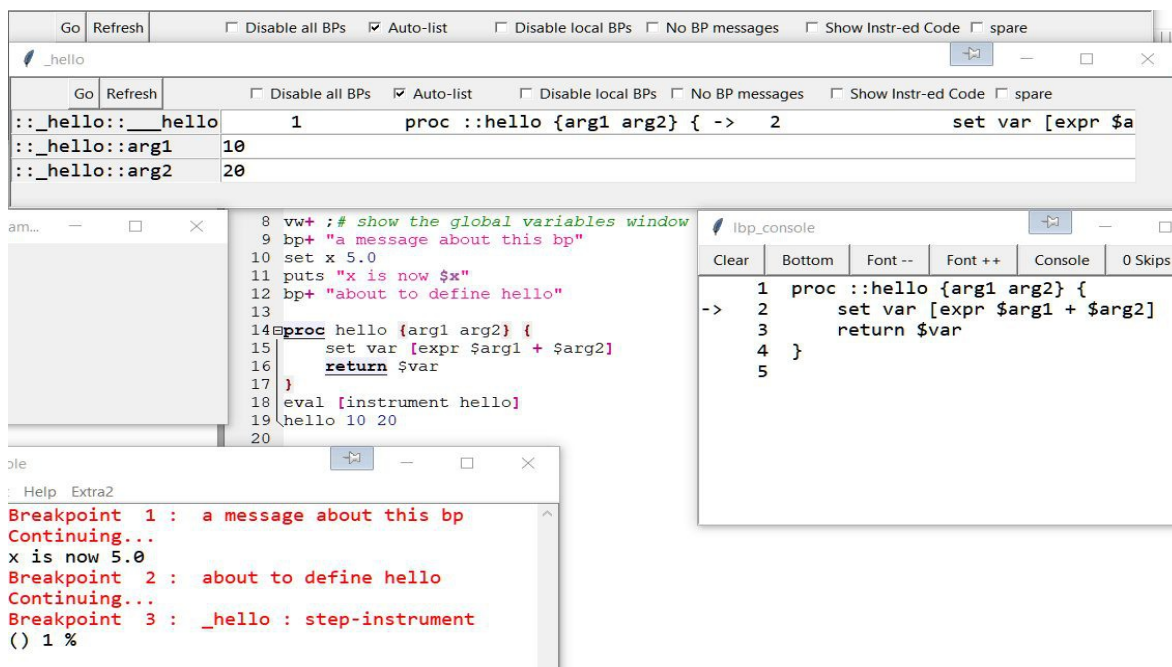
## Instrumentation

Since it is sometimes useful to also single step a program, there is an procedure that can redefine a procedure by adding calls to lbp+ with a fixed comment but a unique id. The command `[instrument+ <proc> ?options?]` takes one argument, the name of a procedure, and using tcl introspection will add the breakpoints and produce a new procedure text that can be `[eval]`'d to replace the current definition. The commented line in the above is how one would go about that.

Note there is one additional variable in the window, the first line `__hello` which holds the text of the procedure, with line numbers and the current position indicator. This can be clicked on and it will be written to stdout (console). **Update** there is a new menu button in the code display window that includes a command to display the call hierarchy in the stack frames.

When the program is being debugged, the normal case is to hide the instrumentation, but there's a checkbox that can enable it's display, in case the instrumentation was faulty. When you click on that checkbox (show instr-ed code) you will have to step at least once for the extra code to display.

With instrumentation, one does not need to manually add breakpoints to the code, although they are still permitted. And when stepping, one will see a different indicator for the instrumented lines `"->"`. The lbp+ command will call the debugger's `vw+` command on each step. It will update variables but otherwise reuse the window. If the window is closed, the next step will create another one.



Here we see the `lbp_console` with a `->` instead of `-` and it stopped on the first statement. We also removed the manual break-point from previously, since we no longer needed it. In the text editor window, we see that the `[eval]` statement has been enabled, and `hello` is called with 2 parameters. The window reflects those values, of 10 and 20. The reason the variables show up as `::_hello::arg1` and



similarly for arg2, is that they have been copied to those namespace variables. It is those variables that are actually being displayed. At each step, the user can type values into those entry boxes, and they are reflected back to the locals they came from in the hello proc after the next step. As mentioned previously, that's only for scalars at present.

Since single stepping can be tedious, the go+ command (abbreviated g) takes an argument, a number of steps to go. So,

```
g 100
```

will run the program until 100 breakpoint steps have been reached. This is the total count and not just specific to the proc or method which was paused when you entered the command. There's no gui way to do this at present, you need a console to type into or (**update**) you can use the 1-line console command entry now included in the code window.

While it is automatically stepping, it delays updating the windows for performance reasons. The globals windows should remain updated, however.

If the value for g is negative, then a command such as,

```
g -30
```

will automatically single step until line 30 is reached. While stepping this way, the data will be updated and so giving an animated look at the program's progress. The code window will also be updated, and with smaller proc's there will be no scrolling either, so the pointer will just move rapidly. If you don't want a large loop to dominate, you can use the on/off (#enable+ #disable+ special comments) ability of the instrumenter. In the code window, one can also **double-click** a line number to issue a goto line comand. The lock check-button can be used to keep the code window from scrolling. When the pointer is offscreen, there will be an up or down arrow at the top or bottom to indicate where the program is currently running.

There's now a new button (not seen in the screenshots above) called Stop. During the goto line or go a number of steps, this button will cause a break by zero-ing the number of steps remaining, or canceling the go to line number. This should stop the program quickly.

When skipping with the go number of steps, it will only report modulo 100 now. There is a command to change the modulo. See the util+ ensemble commands. This further reduces to overhead of stepping.

The checkbuttons are,

**disable all bps** – this will cause all breakpoints to be ignored. The one checkbutton is linked to all the others in any other data window.

**Auto-list** is on by default and causes the proc code window to be updated. If this is off, it will not, which might make the program run a bit faster. It's specific to that procedure only.

**Disable all local bp's** is a window and procedure specific option. It's a way to step out of a procedure. Windows that are not proc windows have no effect if that checkbutton is clicked.

*No bp messages* will suppress the message at a breakpoint as well as the continuing messages. Also window specific.

*Show instr-d code* will (after at least one step) show the code in the text window along with the instrumentation. Window specific.

## The Instrumentation process

The **instrument+** procedure uses several [info] calls to reconstruct the procedures. For the most part, the same text is returned; however, there are some differences. The most important one is that line continuation is removed. This affects the look, but more importantly the line numbers. This is good and bad. The good is that the **instrument+** proc doesn't need to worry about them, and the line numbers that are shown by the debugger will agree with line numbers in an error traceback report.

The bad thing is, well, it's not the true source code. But maybe that's not so important.

There is one other issue which is on-the-line comments. The **instrument** proc will generally append statements using a **;lbp+ step-instrument id#####** with a long number so it doesn't accidentally appear in the code somewhere else. When there's a **;** at the end of a statement, appending to that will have no affect when the program is run. To counter that, the **instrument** proc will attempt to position the breakpoint call in front of an on line comment. The **show instrument code** option can be used to see if the instrument process worked correctly. The user might add a manual **lbp+** break-point if one really needs to stop there.

Also, this **instrument-er** assumes a certain coding style. Control structures all expect to look like a style that is popular with tcl developers. For example, a **foreach** loop and a multi-way **if** would look like so,

```
proc {} {                                ← instrument here
  foreach item list {                    ← instrument here
    ...                                  ← instrument here for mostly all simple statements completely on 1 line
  }                                      ← instrument here

  if {condition} {                       ← instrument here
    ...
  } elseif {condition} {← instrument here
    ...
  } else {                               ← instrument here
    ...
  }                                      ← instrument here
                                          ← do not instrument here on a blank line
# comment                               ← do not instrument here on a comment (trims on left)

  puts "this is a test ;# of a string" ← do not instrument ;# followed by a "
  puts "this has a comment on line" ;# comment ← instrument before this → ;#
}
```

So, the open braces are at the end of the line, rather than on, say, the next line. However, to do that would cause a tcl error, so maybe it doesn't really matter, as the opening brace, if on the next line, would require that a backslash were placed on the line with the **foreach** or **if** statements. Then the

introspection would return the proc body as though the newline (and the `\`) were not really there. But it's worth mentioning, since it's not that easy to parse a tcl program. It does use `[info complete]` however.

There is a problem with switch statements and if used, one cannot instrument them between the cases in the switch, since they get interpreted as another case in the switch, and worse, one that has no body. So, the code can't attach a breakpoint at the end of a `}` which is normally the case. For that, there is one option, so one can call the instrument as follows:

`instrument+ procedure ?-norb"`

which is short for no right brace. If this must be used, then one can manually add breakpoints at the places they won't occur. For example,

In the below proc, the only possibility is to manually add breakpoints and set the `-norb` option. It's ok if a manual breakpoint is appended with another breakpoint, it will just take 2 go's to continue and you will be able to tell since the current line indicator will change from `→` to `—`. If a switch statement is found while processing, the `-norb` switch will be automatically turned on from that point on.

```
proc range {cmd {arg {}}} {
    global RANGE
    switch -- $cmd {
        "max" {
            set RANGE(max)      $arg
            set RANGE(current) -1
        } ← cannot have a statement here, it will be considered the next case
        "next" {
            upvar $arg var
            incr RANGE(current)
            set var $RANGE(current)
            if { $var < $RANGE(max) } {
                return 1
            } else {
                return 0
            }
        } ← nor have one here
    } ← this is the end of the switch, you can have one here, but we can't tell
}
```

There are some bugs in the instrumentation. If it finds code like this,

```
if {....
    || ....
    && ...
} { ...
}
```

Where there are no `\` continuation escapes, the debugger will crash. So, if this happens, the only workaround at present is to add the `\` to the lines. In the debugger, and in tracebacks, the escapes are removed and the the code appears different, but it is still functionally the same.

One other problem can occur. If one writes a new control command, say, a python like for loop with an else clause, then the instrumentation will become part of code block that would be run inside the new control command, not the proc it was intended for. One must use the #disable+ option here (see next section). Break-points can still be inserted, but not in code that will be interpreted (usually with an uplevel) in a different procedure.

If all else fails with instrumentation, there are now 2 special comments that can be used around some code to turn off instrumentation:

```
#disable+  
#enable+
```

These are not nest-able, they do not keep a level count, simply disable at a point and enable at another. If you use the enable, you will not see it in the code, but only a blank line unless you turn on show instrumentation. Due to the way they are implemented, if you use #enable, you should add a blank line following it, since it will not stop on that line. Show the instrumentation to see why. (fixed now)

In order to make the break-point counting as accurate as possible, along with the go to line command, blank lines and full line comments are now also instrumented unless turned off as above.

### **TclOO Method instrumentation**

In order to instrument a TclOO method, one uses the following format:

```
instrument+ -class <classname> <method>
```

Currently, there is no -revert option for methods. It returns code that can be eval'd immediately. If the method had been exported previously, this will be un-done and needs to be done again by the user.

## **Conclusion**

This has described a tool that is intended to be a small and simple debugger. It's roots began with a short snippet of about 12 lines of tcl code written by RS (Mr. Wiki). It's obviously grown some, but the core idea of using labels and entry widgets with an associated variable is still the core of the vw+ system.

Well, that's it for this tutorial introduction to the vw+ debugger. Hopefully you will find it useful.

# Appendix

## Command parameters

### **vw+ pattern/list window width**

The vp+ command has built in help using the “?” option, as shown below:

```
% v ?
vw+ pattern  window  width  - patterns are [string match] type
vw+ {a list} window  width  - alternate form, with list of >1 variable names
  pattern    * => all globals  ** => only user globals (the default)
             text => text* can also use glob with [abc] etc. * always added to end
             foo:: => foo::* vars, not globals but namespace variables only
  width      width of entry widget with variable data, defaults to 80
  window     default to .vw, can use several windows at same time
  {a list}   a list of specific variables, can be undefined

Any parameter can be a . = shorthand for default, none are required
Note: the patten parameter cannot look like a list of > 1 element
      it must be in {}'s to use the [ab] string pattern

On first call to vw+, BWidget's is loaded if possible to support scrolling
if it's not available, will fall back to a single window which could be too large
```

### **bp+ message nobreak nomessage**

This is not designed to be typed into a console. It is only to be placed in the user program where one wishes the program to pause. It is the lowest level break-point command. It does the waiting, but also checks to see if there are any windows to be refreshed when it is reached. The refresh is to update array variable indices that are displayed in a global or namespace display window.

It has 3 parameters. Only the first one is expected to be used when this statement is placed in the user program. The 2<sup>nd</sup> and 3<sup>rd</sup> parameters are for use when called by the lbp+ procedure.

- |           |  |
|-----------|--|
| message   | This is a text message that will appear when this break-point is reached<br>If there's no message, * will be displayed |
| nobreak   | A flag used to indicate don't pause, but it will do some other work. 0 by default                                      |
| nomessage | A flag to suppress messages, used when the checkbox to suppress BP messages is checked and is 0 by default             |



## lbp+ message ID

This is not designed to be typed into a console. It is only to be placed in the user program where one wishes the program to pause.

It has 2 parameters. Both are optional; however, the code window will very likely get confused as to what line this break-point was on, and could display many such indicators. Best to have something.

message	This is a text message that will appear when this break-point is reached If there's no message, * will be displayed
ID	An identifier to be used that must be a fixed string (not a variable) that is used by the single stepping to display an indicator of the current line. If not present, will use the message in the first argument. These are generated uniquely by the instrument+ procedure

## go+ skip window

This is used to continue from a breakpoint.

Skip	This is the number of breakpoints to skip over, it defaults to 0. If this is a <b>negative</b> value, it means run until this line number. It can be triggered with a double click of the line number in the code window as well.
window	for future use, internal only

## instrument+ procedure -norb -revert...

This is used to create an instrumented version of a proc.

procedure	This is the name of the proc to instrument or to revert to if previously instrumented
options:	
-norb	Do not instrument at right braces (has no effect if -revert is used)
-revert	Revert to a non-instrumented procedure

Methods are instrumented as,

```
instrument+ -class <class name> <method>
```

This command returns the source code of the specified procedure such that it can be **eval'd** to replace the procedure. Obviously, care must be taken when to do this. The statement would look like,

```
eval [instrument+ hello]
```

To revert is the similar, and you supply the same proc name:

```
eval [instrument+ hello -revert]
```

It does not keep more than one copy of the original code, so you can't run it twice (and will check for that now), since that would add many extra break-points to the code. While you can type this into the console, you would more likely include it in the program.

The most recent version now checks for manual lbp+ statements and does not instrument them. But this is only for those at the beginning of a line.

For a class, one might do this, for a method UpdateBalance in say, class Account:

```
eval [instrument+ -class Account UpdateBalance]
```

## **util+ sub-command args...**

This is a utility procedure that has several ensemble sub-commands. The args depend on the particular sub-command used.

subcommands:

lp <proc> This can be used to view the current source code, it takes on argument which is the name of a proc to show the source code for. It can be used to see the results of a call to instrument+ This now no longer outputs to stdout, but returns in a functional way to the caller. If entered in the console it will display the code, but can also be used to modify code. For example,

```
eval [regsub courier... [util+ lp someproc] {consolas 12 }]
```

This will retrieve the proc *someproc*, run a regsub on the code to change the font name and size (the 3 dots are regex dots, may need to place in braces), and then replace the original proc. Naturally, you need to be certain you know what you are doing here. Run in the console without the eval first to check the results.

smod N Set the modulo on how often to output skip remaining counts, also in config

clean closes all the data windows

grid x y re-position all data windows, 2 optional args y-incr x-incr (15/500 default)

? show little help message

## Configuration

Since this is all just one single .tcl file, and this is for programmers, editing text is no biggy, so all options are simply changed in this section at the top of the file.

```
# -----
# C O N N F I G U R A T I O N begin
# -----

set ::__zz__(proc_wid) 15      ;# the number of lines to show on either side of a breakpoint line
set ::__zz__(auto_list_default) 1 ;# this sets the auto list checkbox to this value at first creation of checkbox
set ::__zz__(bp_messages_default) 0 ;# this sets the no bp messages checkbox to this value at first creation of checkbox
set ::__zz__(console_hack) 0    ;# if 1, installs a console hack to allow an empty <cr> line on console, repeats last command
                                (handy for go+)
set ::__zz__(tooltips) 1000     ;# if > 0 tooltip enabled and value=delay, if the package require fails, it will report and
                                work w/o it, 0=don't use
set ::__zz__(use_ttk) 0         ;# if 1, the windows use the themed ttk
set ::__zz__(max_size) 3000     ;# the maximum size of a variable, for safety, also if the variable does not yet exist, we
                                can't monitor it
set ::__zz__(max_history) 50    ;# the maximum number of commands saved in the 2 command histories (command and uplevel)
set ::__zz__(skip_modulo) 100   ;# when using a large skip count on go+ this is the number of steps between reporting
                                remaining messages
set ::__zz__(arrow) "\u27F6"    ;# Unicode arrow, can be 2 char positions also, can cause a wobble of the line number, if you
                                like that

#set ::__zz__(black) black      ;# the code window colors, black is the foreground, white the background, yellow background
                                when proc done
#set ::__zz__(white) white      ;#
#set ::__zz__(yellow) {#ffffc0} ;# background when proc done
#set ::__zz__(yellowx) black    ;# foreground when proc done

set ::__zz__(black) {#ffffff}    ;# the code window colors, black is the foreground, white the background, yellow background
                                when proc done
set ::__zz__(white) {#33393b}    ;# the background color from awdark theme
set ::__zz__(yellow) {#ffffc0} ;# our shade of yellow
set ::__zz__(yellowx) black      ;# but need to make text dark to read it

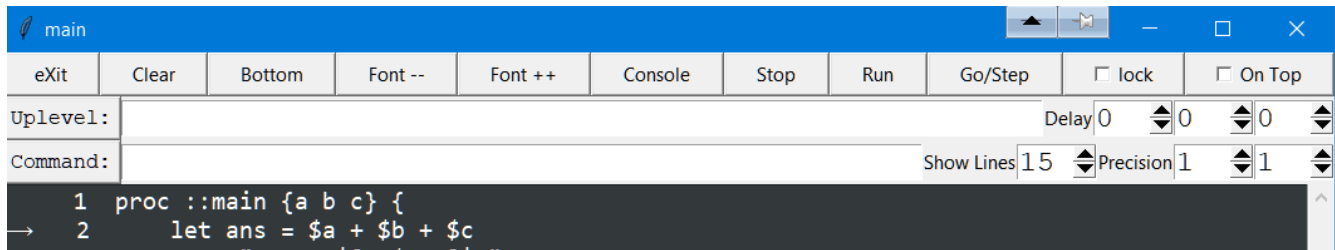
interp alias {} v {} vw+ ;# shorthands since we might be typing these, optional
interp alias {} g {} go+
interp alias {} u {} util+

# -----
# C O N N F I G U R A T I O N end
# -----
```

The vw+ command can modify the console (on it's first call) to make a null command be a repeat of the previous command. This lets one use the go+ command by holding down the enter key to repeat for rapid single stepping. This is one of the options that can be user configured by editing the file in the top section. It is off by default now since the one line command entry supports repeat of last command in a similar fashion.

These initialization statements provide a way to configure certain options. Just edit the one file at the top to customize a handful of parameters. It will try to use the BWidgets package to add scrolling to frames, but can work without it, But of course it also needs Tk. It now also attempts to include the tooltip package, but can work w/o that as well.

## Recently Updated Screenshot and Tips



Since the tutorial section was created, there have been some changes to the program source code window. Notice the dark mode for the code itself, which can be adjusted or changed in the configuration section in the source code.

There is now a Stop button, to stop when a go+ command is running (positive values mean N breakpoints to skip, negative values indicate a line number to stop at). The 3 delay spinboxes, can be used to set a number of milliseconds to delay at each break-point that is being skipped when using the go+ command, in 100's 10's and 1's. It understands the mouse wheel (where supported) and one can enter a specific (valid) integer. Precision is how many instructions per break-point, in 10's and 1's. Show lines is the configuration parameter for the number of lines to show around the current line.

The 2 buttons (Uplevel and Command) are both a label and an action: which clears the entry widget on their right. The uplevel allows executing commands in the context of the proc or method where the break-point was issued, and its return value is sent to the console. The command entry is for executing commands globally, similar to the console, except it does not automatically output any values. The command must do that. Or one must use a puts [some-command].

Both entry boxes include a 50 command history available using the up/down arrows. An empty line (i.e. just the enter key or the keypad equivalent) will repeat the last command. This replaces the need for the console hack.

If the go+ value is positive, i.e. skipping break-points, it is engineered to run as fast as it can, while reporting every so often that N break-points remain (unless reporting is turned off, the default now). If the value is negative, and it's a line number to skip to (which can be set by double-clicking a line number), it will run much slower, on purpose.

The new *run* button is actually just a goto line number with a huge line number, so it will simply run until the user stops the program with *stop*.

The uplevel entry now works, however, do NOT try to set a variable that way (except a new variable) since on the next go from the break-point it will restore the values from the data window. So, to change a variables value, do it in the data window for the current procedure.

The refresh button now does a full refresh, as though the command to create it was re-executed (the command is in fact saved for just that purpose) and so update and possibly add any new variables

specified in the pattern match that now match a variable. If a variable was added using the {list} form of the vw+ command, and that variable was not yet defined when the command was issued, a message is displayed for the variable's value instead and it cannot be changed. If, however, since the time the window was created the variable has been defined, a refresh will enable it and show the value.

Still need some tooltips however, for the data windows. That's a todo item.

The variables windows now have a stay on top check-box, as does the code window.

The vw+ command will accept array(element) when using the {list1 list2...} form of the command, and will preserve the order in the list, instead of sorting alphabetically, as it does in other cases with a pattern being entered.

The lock checkbox will keep the current set of lines in display rather than scroll to the line.



## **Performance Considerations**

As with any debugger, especially one where extra statements are added to the source code, there is the potential for a performance hit. Here are some tips.

### **Don't display too many variable windows**

The biggest performance cost is likely to be from the variable display windows. They monitor changes to variables by using the Tk `-variable` option to cause the entry widgets to be updated when the variable in the left column (either a global or a *variable* from a namespace) is modified. Careful use of the pattern matching argument to `vw+` can be used to reduce the variables being watched. However, in a procedure, this is not an option as all *local* variables will be displayed (hence the name `lbp+`). Too many variable windows will very much slow down the `g -N` command when it's used to animate the code.

### **Only instrument procedures you want to single step**

The `instrument+` procedure is completely dynamic. It can be used at any time, so if you don't want or need to single step a procedure, don't use it until it's needed. And it can be removed also.

### **Use the disable all break-points toggle**

There is much less overhead when this is set. It will check for this at the beginning of a call to both `bp+` and `lbp+` and do an immediate return. While it's not documented explicitly, the variable that is being checked for is,

```
set ::__zz__(cb1) 1 ;# 1 to turn off all break-point code, 0 to re-enable
```

So, one can turn this on and off from the user program if desired. It's the same variable that is used to monitor the first checkbox in each of the data windows. It's a 0/1 where 1 means don't break.

### **Use only `bp+` and not `lbp+`**

`lbp+` is for breakpoints in a proc where you want to see the code in a window, and have it update when you reach another `lbp+` breakpoint. The `instrument+` procedure places `lbp+` break-points into the program and is intended for when one is interactively single stepping, where performance should not be an issue. `bp+` break-points are more efficient, but only pause the program and don't show any source text in a window.

### **Close variable display windows when not needed**

When you close a variable window, since it's a toplevel, all widgets and `-variable` associations are removed by the Tk system. You can always add them again if you are using a console.

## **Add a console show button to your program**

Make use of the console. On Linux, you can get the console.tcl file from the Wiki and use it to give you the console command. I've TIpEd this for linux so it would become part of the core, but that has not yet been approved. If you have a gui program, add a button or a menu item to do a “console show” so you can make adjustments. The debugger is designed to take advantage of the console. After all, it really just another text widget. And it allows for interactive execution of commands and procedures.

## **Use go+ with a large skip count**

When this command is used, it will only update every N steps. The “u smod” command can be used to set the modulo on when to show a message which will also affect how often the data displays are updated. There's a button break, that can clear the remaing steps to stop the program at the next step. That button also causes a go to line to be zero'd so the program should stop after the next step.

## **vw+ system requirements**

- ◆ Needs only core tcl/tk 8.6+ functions and optionally Bwidgets and tooltip, works with 8.7a4
- ◆ Small footprint: pure tcl/tk, 6 proceduures and 1 global variable in 1 file

(the end – ignore these extra paragraphs – just there to aid adding more)

-----

## **vw+ pattern/list window width**

As with any debugger, especially one where extra statements are added to the source code, there is the potential for a performance hit. Here are some tips.

## **vw+ pattern/list window width**

As with any debugger, especially one where extra statements are added to the source code, there is the potential for a performance hit. Here are some tips.