The vw+ Debugging system

Introduction

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

- Simple to install – it's just a single pure tcl file
- Inserted into a program at any time using a *source* statement
- Can dynamically instrument code for visual 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).
- Cruise mode / 3 spin-box dials / 0..999 ms / step - watch variables and code progress
- New – Supports usage with the Tasks module, see appendix at end for details

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

The single file comprises 6 procedures and 2 namespaces:

- vw+   A viewer for global/namespace variables/arrays / entry -textvariable
- 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

One global (array) is variable used, and the file includes a section at the top where this array is initialized, as discussed later. Several options, such as max sizes of entry data is used for safety.

There are also 4 optional aliases that can be uncommented out to use. They provide 4 single letter abbreviations (v, g, i, and u, for vw+, go+, instrument+ and util+) for the 4 commands often typed in by the user. As shipped these are not enabled, to use them requires an edit of the file, or just include a copy of them enabled after doing the source of the file.

When the file is sourced, it only sets values in its global array variable and defines the 6 procedures and the 4 aliases (if enabled) + 2 namespaces. It takes no further actions until some of the above commands are used. The first set of it's global variables can be edited, or overridden by including different values for the items after doing the source of the file.

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.

We first discuss the data viewer and then separately the use of the single code viewer window with the start and stop buttons etc.

Viewing Variable Data

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 works well with the console 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 of the vw+ toplevel windows.

The code viewer window has buttons to step/run/stop plus spin-boxes to control the speed of the program. Three are for 100s 10s and ones. For simplicity the 3 adjust the same var, but have seperate precisions. Mouse wheel support as well. This window is used only when in proc's and methods when lbp+ is used.

The windows have changed some from these screenshots and some updates in the appendix.

The first example is a small program with code that is simply down the page outside of all procedures that makes use of global variables but could also have some in namespaces.

To get started, suppose we have just a few lines of code at the beginning of a tcl file. We first source the debugger. It's name is usually vw_debugger.tcl, but it can be anything/where you want it.

```
source {pathto/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 tclkit with the tk gui) we will get the following window plus a console and one empty tk main window (.).

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, for 3 windows, this will use the default for the first, but provide names for the rest.
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" (if the alias is enabled).

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. update: The list can also take patterns, but does NOT append a * to each item.

v {foo bar baz a_global_array(x) err* myns::*} 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 *):
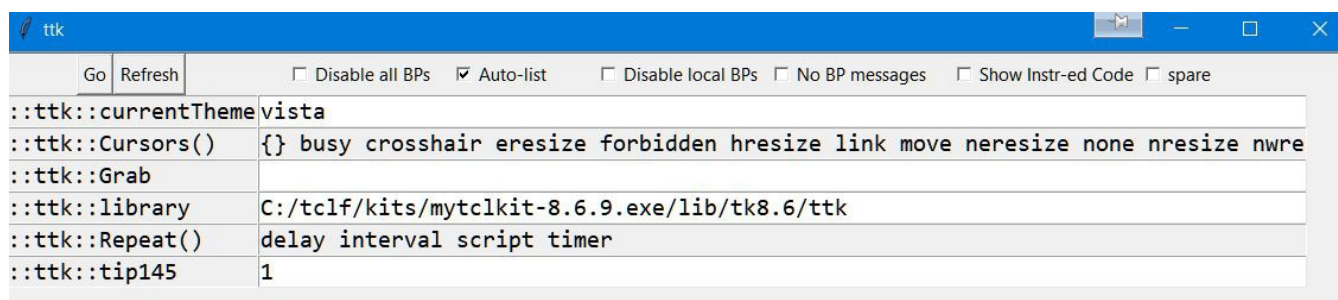
With variable names on the left and values on the right we can see the variable *currentTheme* has the value *vista*. For an array, the value (s) shown are not the data, but rather the indices.

```
---------
The array ::ttk::Cursors()
---------
::::ttk::Cursors()          =
::::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(seresize)  = 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
```

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 (use the list option to view individual array elements).

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

```
ple                                              ✒ vw
 Help  Extra2                            Go  Refresh      □ Disable all BPs   ☑ Auto-list   □ Disable local BP
------------
The variable websters               websters              Abacination {(n.) The act of a
------------
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
--------------
  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    llength: 18
--------
  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.

Note: The in the above dictionary example, only a few words were actually used. Had there been many more, some safety checks would have kicked in. Text entries are protected against too large data also.

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 in the appendix).

Globals and Namespace variables can be modified 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 in vw+.

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 and so used the default window name. The puts statement will also have run, outputting its data to the console:
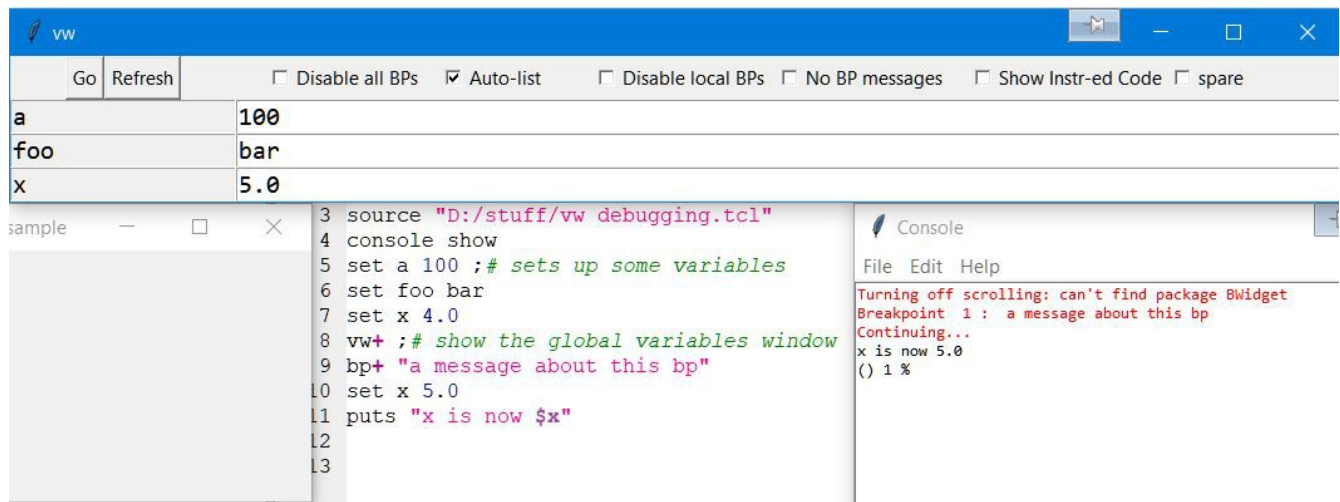
At this point, the program will have reached the end of the file, but will not exit, since it's a tk event program and it will pause in the event loop. You can check the variable windows to see the final values of the variables.

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,

```tcl
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:

vw window:

| Go | Refresh | | □ Disable all BPs | ☑ Auto-list | □ Disable local BPs | □ No BP messages | □ Show Instr-ed Code | □ spare |

a	100
foo	bar
x	5.0

_hello window:

| Go | Refresh | | □ Disable all BPs | ☑ Auto-list | □ Disable local BPs | □ No BP messages | □ Show Instr-ed Code | □ spare |

:_hello::___hello	1	proc ::hello {arg1 arg2} {	2	set var [expr $a
:_hello::arg1	10
:_hello::arg2	20
:_hello::var	30

lbp_console window:

| Clear | Bottom | Font -- | Font ++ | Console | 0 Skips | spar |

```
1   proc ::hello {arg1 arg2} {
2       set var [expr $arg1 + $arg2]
-- 3       lbp+ "inside hello"
4       return $var
5   }
6
```
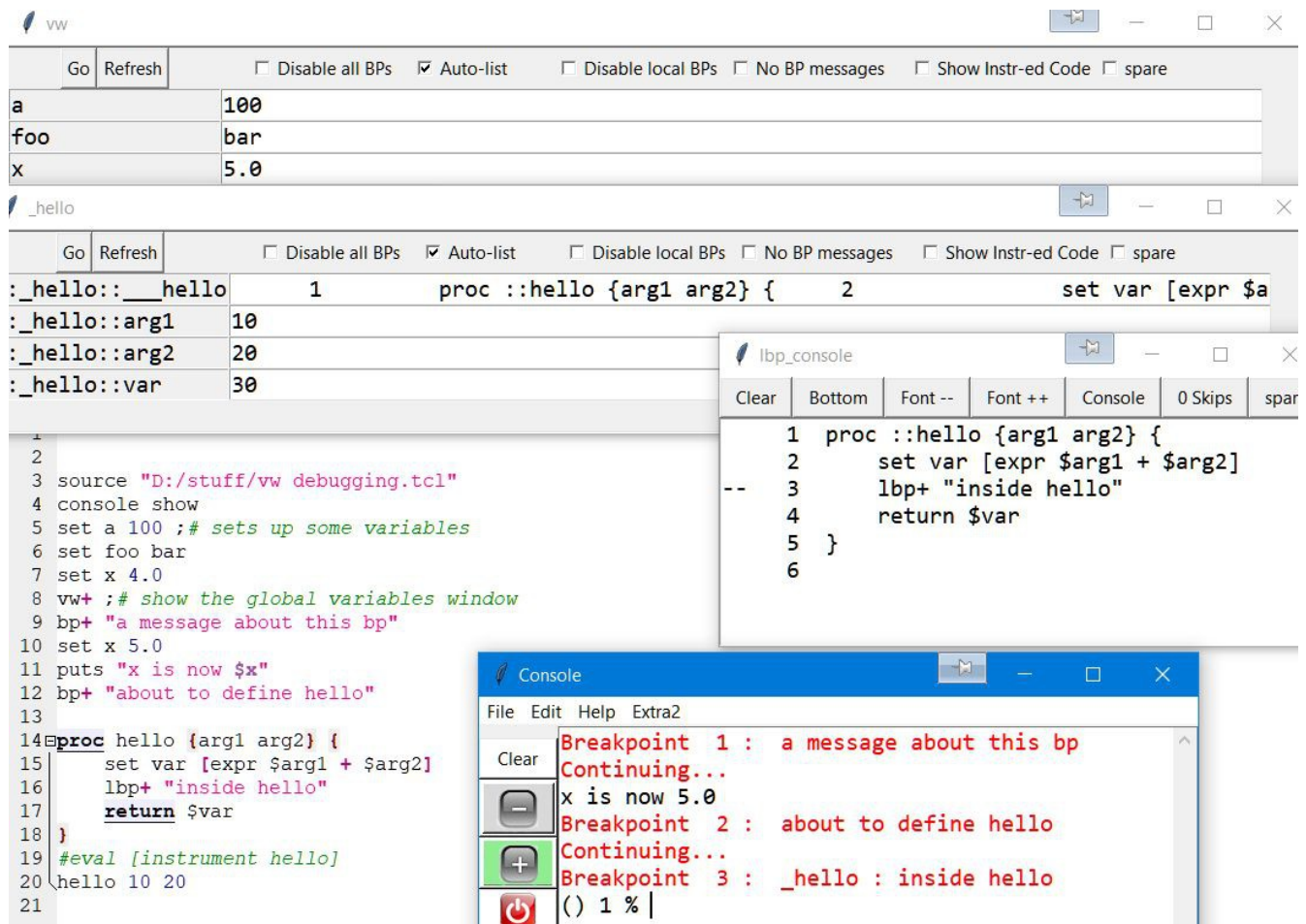
```
1
2
3  source "D:/stuff/vw debugging.tcl"
4  console show
5  set a 100 ;# sets up some variables
6  set foo bar
7  set x 4.0
8  vw+ ;# show the global variables window
9  bp+ "a message about this bp"
10 set x 5.0
11 puts "x is now $x"
12 bp+ "about to define hello"
13
14 proc hello {arg1 arg2} {
15     set var [expr $arg1 + $arg2]
16     lbp+ "inside hello"
17     return $var
18 }
19 #eval [instrument hello]
20 hello 10 20
21
```

Console window:

File Edit Help Extra2

```
Breakpoint  1 :  a message about this bp
Continuing...
x is now 5.0
Breakpoint  2 :  about to define hello
Continuing...
Breakpoint  3 :  _hello : inside hello
() 1 %
```

* 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** be modified in a local procedure however a left click on them display their contents. (update) they can be modified using the uplevel: entry.

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 at a break-point.

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 3rd 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 nested break-points will be ignored.

Instrumentation

Since it is often desirable to single step a program, there is an procedure that can redefine a procedure or method by adding calls to lbp+ with 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 is [eval]'d to replace the current definition.

The commented line in the above is how one would go about that. For methods, there is a -class option as well, to provide the class name of a method. UPDATE: the eval is no longer needed, but doesn't hurt if it's used. In the console, you likely would use the alias, so for example: i hello   could be entered to setup stepping for that procedure. And later,   i hello -revert   would restore the original code.
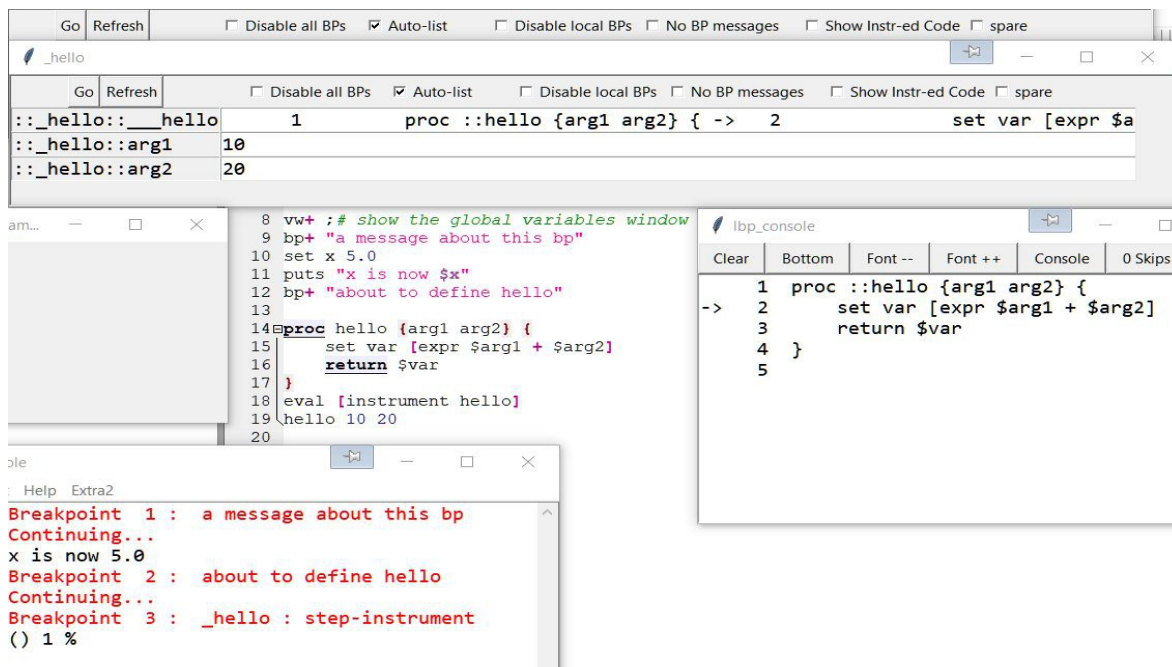
Note there is one additional variable in the window, the first line __hello which holds the text of the procedure, with line numbers and the current position indicator. This can be clicked on and it will be written to stdout (console). This can be used to see the upper call frames while in a lower one.

There is a menu button in the code display window that includes several commands to display the call hierarchy of the stack frames and toggle showing instrumentation.

When the program is being debugged, the normal case is to hide the instrumentation, but there's a menu command that can enable it's display, in case the instrumentation is faulty. When you click on that checkbutton (show instr-ed code) the instrumentation is shown or removed.

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. This is sometimes useful to refresh the window.

In the most recent version, there is now a menu command which will list global procedures (not method however) that can be instrumented. It corresponds to the instrument+ -pick command (but the manual command allows for wildcards, the menu does not). Items that have already been instrumented show at the top of the menu with an arrow and these can be selected to revert to uninstrumented.

_hello

::_hello::___hello	1	proc ::hello {arg1 arg2} { ->	2	set var [expr $a
::_hello::arg1	10			
::_hello::arg2	20			

```
 8 vw+ ;# show the global variables window
 9 bp+ "a message about this bp"
10 set x 5.0
11 puts "x is now $x"
12 bp+ "about to define hello"
13
14 proc hello {arg1 arg2} {
15     set var [expr $arg1 + $arg2]
16     return $var
17 }
18 eval [instrument hello]
19 hello 10 20
20
```

lbp_console

Clear | Bottom | Font -- | Font ++ | Console | 0 Skips

```
    1  proc ::hello {arg1 arg2} {
->  2      set var [expr $arg1 + $arg2]
    3      return $var
    4  }
    5
```

Help Extra2

```
Breakpoint  1 :  a message about this bp
Continuing...
x is now 5.0
Breakpoint  2 :  about to define hello
Continuing...
Breakpoint  3 :  _hello : step-instrument
() 1 %
```

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 (optionally abreviated 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. At any break-point one can turn on the message reporting to see the current step count. If the program is repeatable, one can use this to effectively go back a few steps by restarting and running to just a few steps earlier.

You can use the 1-line console command entry now included in the code window. A second 1-liner is for issuing commands that run a level up in the context of the procedure or method where the program is currently paused. This can be used to set array variables.

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

g -30

will automatically single step until line 30 in the current proc/method 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. To specify another procedure other than the current one, add the name,

g -30 testproc

If you don't want a large loop to dominate, you can use the on/off (#enable+ #disable+ special comments) ability of the instrumenter.

Note that it is often easier to just use the run/stop/go buttons in the code window than typing the g command. (See the updated screenshots later in the Appendix). In the code window, one can also double-click a line number to issue a goto line comand.

During the goto line or go a number of steps (or run mode), the stop button will cause a break by zeroing the number of steps remaining, or canceling the go to line number. This should stop the program quickly if one's entered a very large number of steps.

Note if a program in run mode stops because it's finished, or is in the event loop waiting for some gui action, it will continue running after the next user event. The stop button can be used before the event so it will not be running when some new code is first entered.

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. This is the default and is rarely used

Conclusion

This has described a tool that is intended to be a small and simple debugger. It 's roots began with a short snippit 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 or [info global/vars], can be undefined (uses var for namespaces)
             A * is NOT added to the end of these list items, only if there's just the one pattern as above

   Any parameter can be a . = shorthand for default, none are required
   Note: when a single pattern, a * is added, but if a list, it's not
         a pattern 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 actual waiting.

It has 3 parameters. Only the first one is expected to be used when this statement is placed in the user program. The 2nd and 3rd 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
                  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. If set to 1, it becomes and AlertPoint, where it will only output the message. This can be setup with a program variable that 1 or more bp+ calls would share to turn a group of break-points into a group of AlertPoints.

    nomessage A flag to suppress messages. If used, one might not realize that a break-point is waiting to be continued. Use with appropriate caution.

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 varible) 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 ?procedure?

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 **negative** value, it means run until this line number. It can also be triggered with a double click of a line number in the code window as well.

procedure optional default is the current procedure

instrument+ procedure -norb -revert -nowarn...

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

    -nowarn   Do not warn when no return statement is found in a proc or method

    -list   list to stdout the procedures and methods that are currently instrumented
    -pick pat   pops up a menu to select a procedure to instrument or -revert (not methods)
                 pat is a string match (glob) type pattern, default is *

Methods are instrumented as,

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

All the methods can be instrumented as follows:

    instrument+ -class <class name> *

This command will lookup all the methods (public or private) and will include an export for each method so they can be instrumented. The instrumentation also generates an export statement since exports are canceled when the new instrumented version is eval'd.

This command now does the eval internally. In the past one needed to eval the results. If it is eval'd it will still work, since instrument+ now does the eval and returns a null string.

instrument+ hello

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

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. This is not currently implemented for methods, however.

The command now also verifies that you are not running the procedure to instrument. It also now includes a "just see it" with,

instrument+ hello -noeval

This will merely return the code that would be generated. However, it does not save the original, so even if this is eval'd it will not be able to restore with a -revert.

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:

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 one 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 retreive 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

tab N   Set the tab size

refresh win   Refreshes array values for a given window (in a vw+ * window). Normally arrays don't refresh since they're not values but indices (done for performance costs).

These can be refreshed also using **shift-refresh** button which will only refresh arrays or **refresh** button (alone) will refresh all variables and add any new ones, but costs more in performance, so this call has a lower cost than a full refresh and is suitable for use by the program. It returns "ok" or an message if the window has been closed or does not exist, but does not cause an error abort.

clean   closes all the data windows

debug   provides a data window for many of the debugger's internal variables. Mostly used to

debug the debugger, but also contains info such as the current break-point count. That could be used in the g # command as mentioned, to re-run to an earlier point.

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 easy, so all user 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) 5          ;# the min number of lines to show BELOW (just changed) a breakpoint line, adjust with spinbox
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) 1 ;# 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 package require fails, it will report and work w/o it, 0=don't use
set ::___zz___(tooltipsbuiltin) 0   ;# if > 0 use hobb's tooltip package, at bottom of this file if no tooltips package fnd, e.g. on linux w/o tcllib
set ::___zz___(use_ttk) 0           ;# if 1, some windows use the themed ttk, but not the label or entries since we use -bg
set ::___zz___(max_size) 1000       ;# the maximum size of a variable, for safety, also if the variable does not yet exist, we can't monitor it
set ::___zz___(max_array_size) 500  ;# the maximum size of a array in indices
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___(tabsize) 4           ;# code window tabsize
set ::___zz___(fontsize) 12         ;# data window font size
set ::___zz___(minupdate) 1         ;# this causes an update of only the arrow, not redraws code on each step, can't be on to show instrumentation
set ::___zz___(deadman) 100         ;# when all bp's are off, we can appear to freeze if there's a lot of work to do, so every so often we update
set ::___zz___(deadman2) -1         ;# decr this guy until he reaches 0, then set to deadman, first thing we check in bp and lbp

# choose one set or the other of the below, must be defined to something however
#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, whilte the background, yellow backgound when proc done
set ::___zz___(white) {#33393b}     ;# the bacground 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


#set ::___zz___(bwidget) 0 ;# uncomment this if BWidgets are not wanted, leave undefined and it will try to use it (do not set to 1 here)
catch {history keep 100}            ;# keep console history more than just 20, can comment this out, it's for debugging the debugger

#interp alias {} v {} vw+ ;# shorthands since we might be typing these, optional, now commented out to avoid collisions with unknown code
#interp alias {} g {} go+
#interp alias {} u {} util+
#interp alias {} i {} instrument+

# ------------------------------
# C O N N F I G U R A T I O N end
# ------------------------------
```
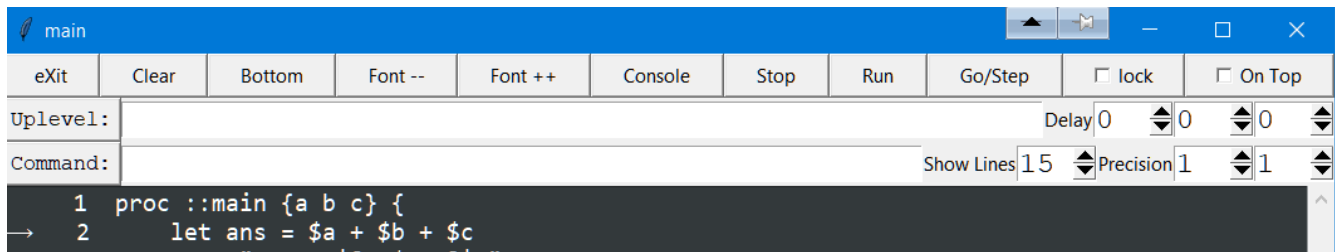
The vw+ command can modify the console (on it's first call) to make a null command be a repeat of the previous command. This lets one use the go+ command by holding down the enter key to repeat for rapid single stepping. This is one of the options that can be user configured by editing the file in the top section. **It is turned 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, though some windows may not get auto-sized correctly. Of course it also needs Tk. It now also attempts to include the tooltip package, but can work w/o that as well.

For an individual program, one can override these values and define the aliases by setting them after sourcing the debugger file. For example, the max_size at the default of 1000 can be overridden since it's a pretty conservative value. It's there to limit how much a text entry can handle when it uses the -textvariable attribute. If set too large, however, it can very much slow down the running of a program when there's some very large data or a very large procedure, since that too is stored in an entry variable with an updated instruction pointer.

Recently Updated Screenshot and Tips

main											
eXit	Clear	Bottom	Font --	Font ++	Console	Stop	Run	Go/Step	☐ lock	☐ On Top

Uplevel: [   ] Delay 0 ▲▼ 0 ▲▼ 0 ▲▼

Command: [   ] Show Lines 15 ▲▼ Precision 1 ▲▼ 1 ▲▼

```
    1  proc ::main {a b c} {
→   2      let ans = $a + $b + $c
```

Since the earlier 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 spin-boxes, 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 at the bottom below 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 it's return value is sent to the console. The command entry is for executing commands globally, similar to the console. (update) command results are also now sent to stderr/stdout.

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 a command causes an error, the error text is output, and the current line is not erased. However, it is not in the history, and is only added on a successful command. If not edited and resubmitted, it will be gone when the history is accessed.

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 should not be used to set a variable value (except a new variable) since on the next go/step from the break-point it will restore the values from the data window. So, to change a variables value, change it in the data window for the current procedure and step the program.

The refresh button now does a full refresh, as though the command to create it was re-executed 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 current value.

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.

The clear and bottom buttons have been replaced by a **menu** command with several commands plus a tearoff capability. The commands include some to re-allign windows in multiple columns, close all or just code windows and several commands for displaying the current calling hierarchy. Most of these commands just call the util+ or instrument+ command.

The *new* menu button in the code window includes a check-box to group move the data windows. This check-box is not set automatically when the data windows are arranged. This only affects the first window in the group. One can manually re-arrange the other data windows. Then moving the first window will move all the windows as a group, unless the check-box is off. The menu can be torn off:
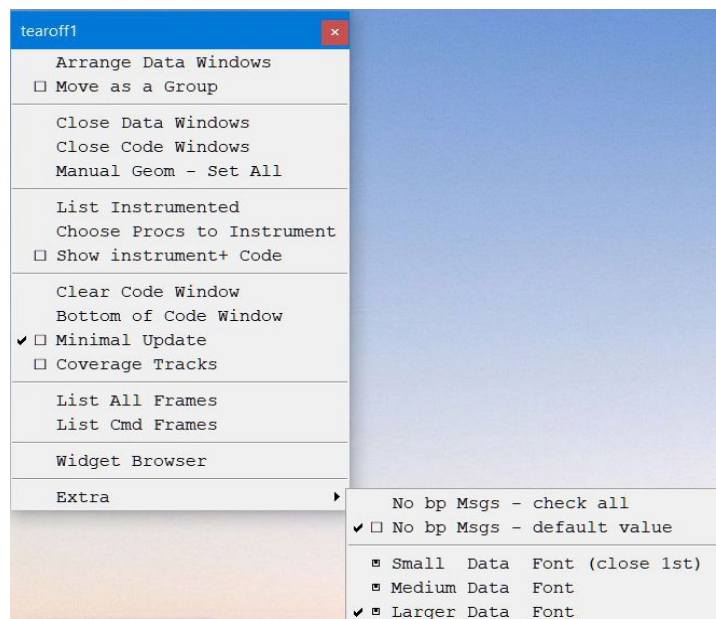
Minimal Update (required with coverage tracks) only redraws the arrow in the code window. Coverage tracks show which statements have been executed at least once. The data windows can have 3 font sizes. To take affect, one needs to close the window. At the next step, it will be re-opened with the new font. The extra menu also can check the No bp messages in all the data windows. The check-box is for the setting when a data window is first opened.

Using the Tasks Module

Some commands are available now to support using this debugger/viewer with the Tasks module. The Tasks module is hosted at the wiki (https://wiki.tcl-lang.org/page/Tasks) and a github site, along with this manual and the vw debugger.tcl file. As this is an experimental use of the debugger, it still has some glitches.

Note: A Task also imports the aliases (see above) v, g, i, and u. These can be suppressed by commenting them out, at about line 36 in the source.

Tip: it's best NOT to run more than 1 or 2 tasks in a group that has the debugger since the debugger in one task will not know about debuggers in other tasks. This can end up creating a lot of windows. Even in 1 task many windows are often created, especially if single stepping is used. It will work, but w/o lots of large monitors, it could get unwieldy.

Tasks can be created in 3 ways in the Tasks module: using the **Task** command directly and indirectly via the **tgroup** and **Tproc** commands. In each case, one of the parameters to these commands is a list of imports and command initialization. In this list, there can be the names (with wildcards) of proc's to import, or commands if they begin with a dash (-). Debugger support now includes commands that begin with a plus sign (+). There are 2 types as shown in this example.

```
Tproc sample {arg1 arg2} {
    set   a 100 ;# sets up some variables
    set foo bar
    set   x 4.0
} -tasks 1 [list -import_tasks {+debug=D:/stuff/vw debugging.tcl} +sample \
                 -#+comment_out {-vw+ * globs/$::t_name}              \
             ]
```

In this example, Tproc creates the proc sample in each Task and also the Main thread – but only loads the debugger in tasks. This is specified with the initialization where it loads the debugging tool using the +debug=path/to/debugger command. This needs to be in {} if the name has a space in it.

The +sample indicates that the procedure sample should be instrumented, by inserting [instrument+ sample] in the initialization section (after the proc sample). This sets up single stepping in the sample procedure. Other procedures can also be instrumented, but they also need to first be imported. So, for example, if we have a procedure foo, then 2 elements: foo and +foo should be added, with the +foo after the first one. The Tproc name, sample, is automatically imported, so doesn't need to be specified. If it is, it would still work by simply importing it twice as with any proc that was defined twice.

Note, a Tproc creates an actual proc, while the Task or tgroup commands do not; instead these run at global level. This means there's no proc to instrument. However, any proc's imported and used by them can be instrumented. The script in a Task or tgroup runs in a while loop (typically) and can be debugged using the bp+ breakpoints. They can also use puts or putz to output to a stdout or stderr.

In order to be able to use the debugger, the Tasks module has been updated to include a puts wrapper. This will cause any puts (or parray) output to be redirected to putz. This change was necessary to be able to use this debugger w/o changes. Since you're reading this, you will likely have gotten it from github, and the latest source to the tasks module and the debugger are there also (not kept up to date on the wiki recently).

An initialization command `-vw+ * globs/$::t_name` creates a vw window to monitor all the global variables. The task name will be appended to globs/ in case more than one task is running with a debugger.

To temporarily disable a command is shown by using the -# to comment out an initialization command. Additionally, an initializer or import can be commented out directly with a # as the first char. If this would cause the line to look to tcl like an actual comment, then enclose in braces.

If the default . Window is not desired, it should be withdrawn using an initializer, such as with the command: {-wm withdraw .}

If there has been no putz statements in any of the initializers, then trying to withdraw the . Window will fail. To remedy, do it as {-package require Tk; wm withdraw .}, since the package require Tk always opens the . Window, and so it can then be withdrawn immediately.

Currently there is no + command to instrument methods and these need to be done using the eval technique shown. Also, OO code needs to be imported as a text script. For example,

```
set oo_code {
    oo::class create Bitfield {
        variable data
        variable maxposition
        variable lastfree

        constructor { { numbits -1 } } {
            # ... code for constructor ...
        }
        method alloc {  } {
            # .... alloc code ...
            return $pos
        }

        method free { pos } {
            my set $pos 0
            set lastfree $pos
        }

        method set { position bit } {
            # ...... set code ....
        }

        method get { position } {
            # ...... get code ....
            return $value
        }
        destructor {}
    }
} ;# end oo_code script

Tproc sample {arg1 arg2} {
    set   a 100 ;# sets up some variables
    set foo bar
    set   x 4.0
} -tasks 1 [list -import_tasks {+debug=D:/stuff/vw debugging.tcl}         \
                +sample                                                    \
                -$oo_code                                                  \
                {-instrument+ -class Bitfield set get}          \
            ]
```

Here we first load the debugger, with the +debug= initializer, and then instrument our Tproc, with +sample. Next, we use -$oo_code which will insert the OO script from the *oo_code* variable into the task so we can then instrument it using the [instrument+ -class ...]. The first element after -class, *Bitfield* is the class name and *set* and *get* are two methods we want to instrument. If we wanted all of them we could specify * instead of set and get. This can cause a lot of windows to open however.

Note that any script code we wish to include can use the -$text technique which will see the initial "-", followed by the text in the variable, and insert it inline in the task initialization section, after removing the "-". This is the technique of choice when importing a large section of code.

Tip: be careful not to have any (invisible) spaces following the \ at the end of each line. This will negate line continuation and produces some very strange error messages.

The Instrumentation process

The instrument+ procedure uses several [info] calls to reconstruct procedures or methods. 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+ procedure 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 unique number. 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 break-point 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 also add a manual lbp+ break-point if one really needs to stop there and the instrumenting doesn't work as desired.

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 → ;#
}
```

Very small (1 command) procedures or methods will not be instrumented unless they are on separate lines from their definition. So for example, the following will not be instrumented:

proc mysimpleproc {args} {puts $args}

and should be changed to,

proc mysimpleproc {args} {
    puts $args
}

Alternatively, one can add a manual break-point instead, either before or after the puts command.

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. If a switch statement is encountered, it is skipped and only instrumented once after the statement is complete. To instrument the several cases in a switch statement, one would need to manually insert break-points.

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

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

Where there are no \ continuation escapes, the instrumentaton will run but not instrument the entire if statement. So, if this happens, the only workaround at present is to add the \ to the lines or add manual break-points. In the debugger, and in tracebacks, the escapes are removed and the the code appears different, but it is still functionally the same.

This is also the case with a while statement where there are newlines in the conditional part.

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.

Another problem can occur with procs or methods that do not use a return statement to return a value, but rely on the Tcl rule which states that the last command return value will be returned by a proc or method if there is no explicit return.

It is highly recommended that this feature not be used as it can cause the return value to be lost if any statements follow the one which is to return the value. This can also leave a lurking bug trap to the unsuspecting programmer who doesn't realize that by adding an additional statement near the end of a procedure can change the return value completely or eliminate one causing an empty string to be returned instead.

To counter this possibility, the instrument+ proc will note if it does not have a return statement and issue a warning to stderr. The option "-nowarn" can be used to suppress the warning.

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+). To many variable windows will very much slow down the g -N command when it's used to antimate 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.

One situation that can slow things down are instrumenting large procedures. If the procedure is too large, then the first data item, the text of the procedure, will be suppressed. If one changes the configuration for max_size, it can work, but will be quite slow. Any data item that is big, but still not exceeding max_size can be suppressed manually by right-clicking the variable name. The text will change color to indicate that it is not being monitored. It's the -textvariable attribute of the entry which can get bogged down when the text is constantly changing. It changes to indicate the location of the pointer to the current line. What is too large? Anything over a few hundred lines.

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 checkbutton 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 /method 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 stop, that can clear the remaining 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 tooltips, works with 8.7a4

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

(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.