Practical Example of Tcl Command Design in a Qt/C++ Graphical Application

Tony Johnson
October 18th, 2017

tony_johnson@mentor.com

Abstract

Tcl provides excellent support for creating complicated user commands in applications that have correct-by-construction built-in help, support for hidden commands, support for position-independent switches, and more. This paper discusses the technical details for how we created one such command in our Qt-based application.

Summary

In this paper I will discuss the technical details around the design for the Tcl "wave" command in our Qt-based [1] application. Visualizer is a GUI for displaying and analyzing simulation waveforms. It is driven by the user either via standard GUI operations such as menu selections, button clicks, drag and drop, key accelerators, etc or via commands passed in through a command prompt which is a Tcl shell. Because GUI operations are inherently difficult to automate among other reasons, there was a strong need to have a command for our waveform window that could do everything that could be done via the GUI. Since we already had a Tcl shell in our application, the obvious answer was to create a Tcl command to provide this capability. Luckily Tcl is well suited for implementing such a complex command.

Background and Motivation

Command-based control of graphical user interface interactions is important for many different reasons. The four key reasons that motivated this work were: Testing; User Control; 3rd Party Access; Save/Restore; and Expandability.

Testing

Unless you want to retest the GUI operations in your application by hand with every major release, minor release, feature addition and bug fix (trust me, you don't), then you will want to have some way to create automated tests for GUI interactions. There are solutions available to help in this area such as TkTest [2] based on the TkReplay [3] framework that was mentioned at last year's TclTk conference. Qt also has a similar solution called Squish [4] that verbosely logs every user interaction. Since our application is Qt-based we do use Squish for test automation. However, these Squish tests heavily use the Tcl commands that we created in this work because they make the resulting Squish test much shorter and more focused on the specific feature being tested. We have Squish tests that are less than 100 lines in length that would be thousands of lines long if created by logging purely GUI interactions.

User control

Users rely on a well-designed GUI to get their job done. However, once they master the GUI they inevitably want a faster way to do their work that doesn't involve bouncing their mouse around the GUI repeatedly doing the same operations over and over. Having robust scripting support with easy to use control commands is key to keeping the user happy.

3rd Party Access

Often applications end up being reused by others within the same company for their products. They want to leverage the functionality that your application provides and present it with whatever value their technology provides. Our wave window is one window that is highly leveraged within our company. Since we don't have time to provide direct support to each of these 3rd party groups, it was critical that we provided a way for them do this work on their own. Our Tcl "wave" command provides most of the functionality that is needed in this area.

Save/Restore

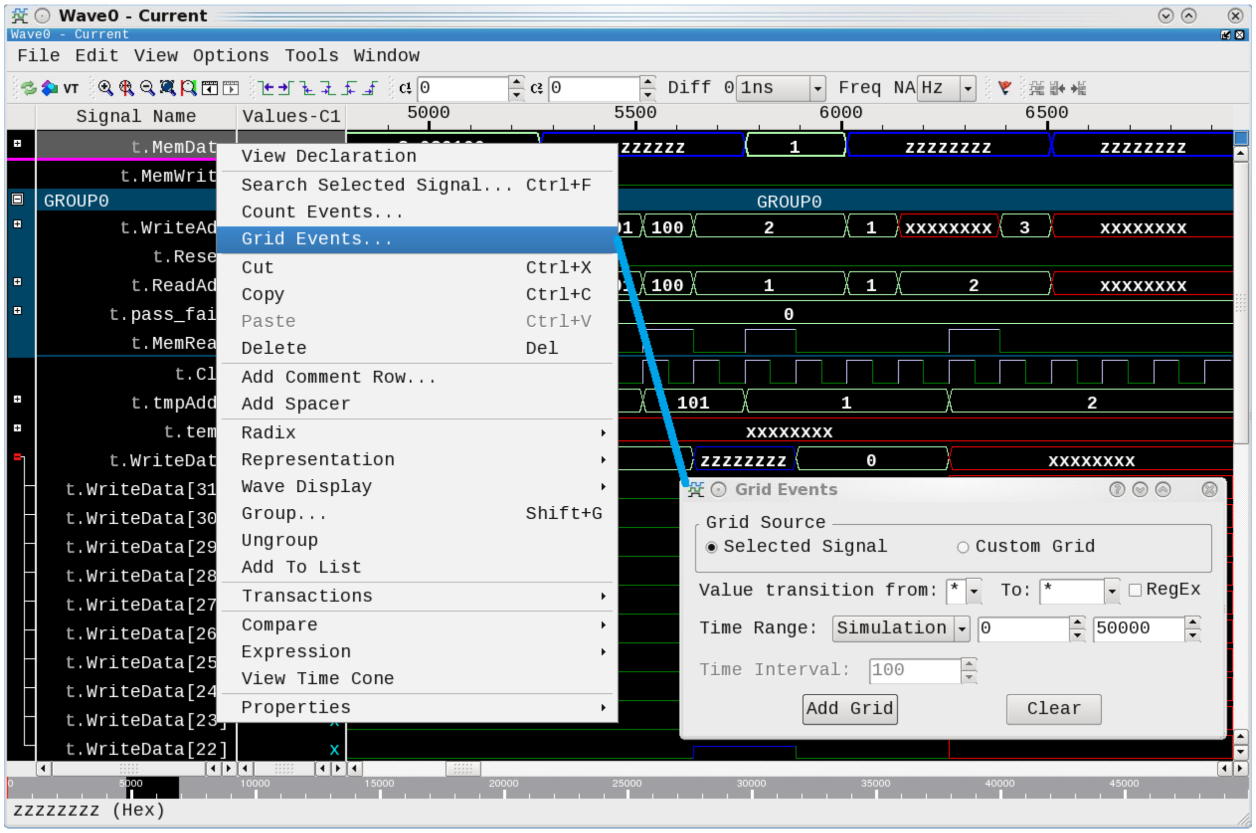
When closing a GUI application, it is usually a requirement that the current state of the windows in that application be saved so that they can be restored later. This could be done by saving the state to some internal representation. However, since we already need a "wave" command for the reasons mentioned above, it makes sense to leverage it in the Save/Restore operation.

Expandability

Whenever new functionality is added to the wave window, it is key that the "wave" command be updated to provide access to this functionality. Therefore it is critical that the method for adding new functionality be as streamlined and as easy to implement as possible. Even though this new functionality could be implemented by different developers, it still needs a common look and feel with the rest of the "wave" command functionality.

Brief Wave Window Description

Before diving into how we implemented the Tcl "wave" command, it makes sense to briefly describe the types of functionality that are used in our wave window. The following image shows the Visualizer Wave window with a RMB menu popup that selects one of the many dialog functions that will need to be supported by our Tcl command.



Grid Source: [✓ Selected Signal] Custom Grid

Value transition from: * To: * RegEx

Time Range: Simulation 0 50000

Time Interval: 100

Figure 1 – Example Wave window showing RMB submenu invocation.

What is a Wave window used for?

The Visualizer Wave window shows the names of digital and/or analog signals on the left along the Y-axis. The resulting waveforms for activity on those signals over time (i.e. binary "0"s and "1"s or composite bus values "100", "2", "3", etc) are shown along the X-axis starting at Time 0 on the left and the end of time on the far right. These waveforms could have been created via purely behavioral simulation of the system; emulation on hardware-based systems that emulate the design; or, theoretically, on actual implemented systems themselves should they ever feel the need to provide access to this data. This allows designers of complex digital/analog systems to analyze the internal activity to discover any problems with their design or to discover any deviation in implementation from the design intent.

What types of operations does a user do in a Wave window?

To give you some idea for the types of operations that our “wave” command will need to support, here’s a brief list of operations that our Wave window supports:

- Adding, deleting, moving, renaming and grouping signals.
- Creating new signals that are concatenations or expressions of other signals.
- Expanding and collapsing composites.
- Zooming in, zooming out, zooming full, zooming over a time range.
- Paging up, down, left, right or jumping to a particular time.
- Setting background and foreground colors.
- Placing and moving cursors in the wave window.
- Creating or modifying markers
- Etc, etc, etc

What does the wave command look like to the user?

Before we dive into the code here’s the usage message for the “wave” command and for one of the subcommands it supports:

```
Visualizer> wave -help
# Usage: wave -help                  | This message.
#        wave activate               | Activate the specified window or report the active window.
#        wave add                    | Add signals to wave window.
#        wave biometricsearch        | Set or clear Biometric search values.
#        wave cget                   | Returns current configuration value for wave window.
#        wave clear                  | Clears the wave window.
#        wave clearselection         | Unselect all
#        wave collapse               | Collapse a particular waveform by index.
#        wave comment                | Add a comment row.
#        wave compare                | Compare two signals highlighting differences.
#        wave concatenate            | Create a concatenation of all selected signals.
#        wave configure              | Query or modify configuration options of the window.
#        wave deleteselected         | Delete the selected objects.
#        wave end                    | Return the index of the last object in the wave window.
#        wave expand                 | Expand a particular waveform by index.
#        wave expression             | Create an expression in the wave window.
#        wave find                   | Find the index of the next occurrence of a signal by name.
#        wave get                    | Get the name and/or value of signals in the wave window.
#        wave grid                   | Create a grid in the wave window.
#        wave group                  | Create a new group or subgroup.
#        wave grouprename            | Rename a particular group.
#        wave index                  | Get the index of selection, insertpoint or end of the wave window.
#        wave insertion              | Set the location of the insertpoint in the wave window.
#        wave list                   | Get the list of currently open wave windows.
#        wave marker                 | Create new or modify existing markers in the wave window.
#        wave name                   | Return the name of the active wave window.
#        wave positioncursor         | Control the location of the primary and secondary cursors.
#        wave remove                 | Remove (unload) a qwave.db file from Visualizer.
#        wave save                   | Save the wave window format to a file.
#        wave seetime                | Pan the wave window to see a particular simulation time.
#        wave selectall              | Select all objects in the wave window.
#        wave selection              | Set or clear selection of signals by index.
#        wave spacer                 | Create a spacer.
#        wave tags                   | Access the qwave.db Tag values (i.e. "F0", "F1", etc).
#        wave timeunit               | Get or set the current time unit of the wave window.
#        wave update                 | Enable or disable drawing in the wave window.
#        wave zoomfull               | Zoom full.
#        wave zoomin                 | Zoom in.
#        wave zoomout                | Zoom out.
#        wave zoomrange              | Zoom to a particular time range.
```

Figure 2 – Output from executing “wave –help” at the Tcl shell in Visualizer.

```
Visualizer> wave grid -help
# Usage: wave grid [-clear] [-help] [-from <value>] [-to <value>] [-regx]
#                  [-start <time>] [-end <time>] [-interval <time>] [<signalpathname>]
# Examples:
#        wave grid .top.dut.reset
#        wave grid -clear
#        wave grid -start 1000ns -end 2000ns -int 100ns
#        wave grid -start 1000ns -end 2000ns top.dut.rdy
#        wave grid -from 0 -to ffff* -regx top.dut.addr
#        wave grid -from xxxxxxxx -to * top.dut.data
```

Figure 3 – Output from executing “wave grid –help” at the Tcl shell in Visualizer.

Ok now that we know we need a command that can do a bunch of stuff, how did we actually implement it?

Wave Command Architecture

The major area where Tcl helped us in the design of the "wave" command was with the argument handling and common help generation via the structure used by **Tcl_GetIndexFromObjStruct**() [5]. Before discussing that area, some other key areas where Tcl made things easy were in the areas of Tcl Command Registration and String Conversion between Tcl and Qt/C++.

Tcl Command Registration

We chose to implement all of the "wave" command functionality in C and C++. To create a Tcl command that can be implemented in C and used in Tcl, we used the **Tcl_CreateObjCommand**() [6] function which is defined to be:

```
    #include <tcl.h>
    Tcl_Command
    Tcl_CreateObjCommand(interp, cmdName, proc, clientData,
                         deleteProc)
```

We didn't need to pass any client data (the arguments are our data) and didn't need a deleteProc. We simply used the following to create our "wave" command that would be accessible via our Tcl shell "xInterp" which is the command prompt in Visualizer:

```
    Tcl_CreateObjCommand(xInterp, "wave", gTclWave, 0, 0);
```

The "proc" argument defines the C function ("gTclWave" in our case) that will be invoked when the Tcl "wave" command is called. The arguments to this function are defined to be: the ClientData, the Tcl_Interp (which we will use to output results from the wave command), an integer representing the number of arguments passed to the wave command, and a Tcl_Obj array that contains those arguments.

```
int gTclWave(ClientData xClientData, Tcl_Interp *xInterp,
                  int xObjc, Tcl_Obj *const xObjv[])
```

String Conversion between Tcl and Qt/C++

Our application is implemented in Qt which handles strings via a QString [7] data type. The first thing we needed to do was to decide how we were going to convert strings back and forth between Tcl and our Qt code. The arguments passed to our "gTclWave" C function are passed as a Tcl_Obj [8] array. We used the following strategy to convert these back and forth:

Tcl to QString:

```
  QString lGroupArg;
  lGroupArg.sprintf("%s", Tcl_GetString(xObjv[iii]));
```

QString to Tcl:

```
  tclObjCls lResult;
  tclObjCls lBufObj(qPrintable(lWin->mGetName()));
  lResult.mLappend(lBufObj);
```

We defined "tclObjCls" as a class that contains the Tcl_Obj* (as "dObj") along with convenience methods for creating, printing, reference counting, etc Tcl_Obj objects. For example the mLappend() is defined to be:

```
 mLappend(tclObjCls &xObj) {Tcl_ListObjAppendElement(NULL, dObj,
                                                     xObj.dObj);
```

The function "qPrintable" is defined to be one of the many ways to get string data out of a QString.

```
  #define qPrintable(string) (string).toLocal8Bit().constData()
```

The Power of Tcl_GetIndexFromObjStruct()

This function [6] is defined as

```
     #include <tcl.h>
     int
     Tcl_GetIndexFromObjStruct(interp, objPtr, structTablePtr,
                               offset, msg, flags, indexPtr)
```

This procedure is used to lookup keywords which, in this case, are the subcommands we created in our "wave" command. It automatically handles the matching of unique abbreviations in the keywords. The key to leveraging this strategy of sub-command handling is to define a structure (structTablePtr) to pass in that allows you to define everything about the sub-commands in one place. The following is the structure that we defined and an enum definition we use in the code below:

```
typedef struct _optionWaveTableStruct {
    char* optionName;
    int   optionEnum;
    char* optionDesc;
    int   optionFlags;
} optionWaveTable;

enum optionEnumsWindowCmd {
    VIS_WAVE_CMD_ADD,
    VIS_WAVE_CMD_BLANK,
    VIS_WAVE_CMD_CLEAR, … }
```

Where "optionName" is the name of the subcommand; "optionEnum" is the enum that our command handler will switch on; "optionDesc" is the brief help text that will be presented if help is requested; and "optionFlags" is an integer that we can use to set attributes for each subcommand. The following is an example of some of the fields in this struct:

```
static optionWaveTable optionWaveCmds[] = {
//optionName optionEnum  optionDesc                   optionFlags
{"add",     CMD_ADD,     "Add signals to wave window.", 0},
{"blank",   CMD_BLANK,   "",                            DEP},
{"clear",   CMD_CLEAR,   "Clears the wave window.",     0},
{"comment", CMD_COMMENT, "Add a comment row.",          0},
{"cursor",  CMD_CURSOR,  "Control the cursors.",        TBD},
{"InvokeMenu", CMD_INVOKEMENU, "",                      HID},
...
```

The benefits to using a structure like this to define the subcommands are:

1. The act of adding a command to this table will remind the programmer to add a description that will be displayed when help is requested on the wave command (i.e. "wave –help", see Figure 2 above).
2. Subcommand name definitions that are close together like this will help to ensure a common look and feel to their naming and their descriptive text.
3. Defining the list of all the subcommands as "optionEnums" in one place provides an easy location for our developers to jump to when they are looking for the other supported subcommands. Simply jump to the definition of one command you remember (i.e. "CMD_CURSOR") and you have quick access to all the other commands.
4. The "optionFlags" field provides a handy way to set flags on particular commands. For example, if you want a subcommand that is hidden from the user, you can set the HID flag. If there is a command that has been deprecated, you can set the DEP flag.

Code Highlights Executing a Wave Command

User Enter's Command "wave"

```
    Visualizer> wave grid –start 1000ns –end 2000ns –int 100ns
```

This calls the C function gTclWave that we registered for the Tcl command "wave" above with **Tcl_CreateObjCommand().**

gTclWave is called to Handle invocation errors and provide subcommand help

```
int gTclWave(ClientData xClientData, Tcl_Interp *xInterp, int
            xObjc, Tcl_Obj *const xObjv[]) {

    if (Tcl_GetIndexFromObjStruct(xInterp, xObjv[1],
           optionWaveCmds, sizeof(optionWaveTable), "command",
           0, &index) != TCL_OK) {
        return TCL_ERROR;
    } //Else If “-help” passed in for a particular command
        switch((enum optionEnumsWindowCmd)index) {
            case VIS_WAVE_CMD_GRID:
                sWaveGridHelpMsg(xInterp);
                return TCL_OK; break;
            case …
    } //Else call “wave” window command handler passing args
        lActiveWaveWinPtr->mWindowCmd(xInterp, xObjc, xObjv);
```

The **sWaveGridHelpMsg**(xInterp) function takes simply our xInterp pointer so that it can output the help message for that command (i.e. the output from “wave grid –help”). Here is the code for that function:

```
int sWaveGridHelpMsg(Tcl_Interp* xTclInterp) {
    Tcl_AppendResult(xTclInterp,
      "Usage: wave grid [-clear] [-help] [-from <value>] [-to
      <value>] [-regx]\n", "                 [-start <time>]
      [-end <time>] [-interval <time>] [<signalpathname>]\n",
      "Examples:\n",
      "     wave grid .top.dut.reset\n",
      "     wave grid -clear\n",
      "     wave grid -start 1000ns -end 2000ns -int 100ns\n",
      "     wave grid -start 1000ns -end 2000ns top.dut.rdy\n",
      "     wave grid -from 0 -to ffff* -regx top.dut.addr\n",
      "     wave grid -from xxxxxxxx -to * top.dut.data",
      (char *) NULL);
return TCL_OK; }
```

Key Lesson Learned: It is highly recommended that you write the help output for your command and subcommands first before starting implementation. The act of explaining something in a clear and concise manner (a requirement in help output) often uncovers problems in your design that causes you to change your implementation plan, switch organization, etc. Implementing this help output first will save you a ton of time and result in a more elegant implementation.

Subcommand Called Passing Interp and Rest of Arguments

```
int waveFormWinCls::mWindowCmd(Tcl_Interp *xInterp, int xObjc,
                               Tcl_Obj *const xObjv[]){
    //Retrieve index as above
    switch((enum optionEnumsWindowCmd)index) {
      case VIS_WAVE_CMD_ADD: //Strip “wave add” args
          mWindowCmdAdd(xInterp, xObjc-2, &xObjv[2]);
          break;
      case VIS_WAVE_CMD_GRID: //Strip “wave grid” args
          mWindowCmdGrid(xInterp, xObjc-2, &xObjv[2]);
          break;
```

Subcommand handler mWindowCmdGrid() for example:

```
int waveFormWinCls::mWindowCmdGrid(Tcl_Interp *xInterp,
                   int xObjc, Tcl_Obj *const xObjv[]) {
static const char *options[] = {
          "-help", "-clear", "-from", "-to", "-regx",
          "-start", "-end",” -interval", (char*)NULL };

enum wavegridopt {
    WAVEGRID_HELP, WAVEGRID_CLEAR, WAVEGRID_FROM,
    WAVEGRID_TO, WAVEGRID_REGX, WAVEGRID_START,
    WAVEGRID_END, WAVEGRID_INTERVAL };
//Call Tcl_GetIndexFromObj to get “wave grid” subcommand
Tcl_GetIndexFromObj(xInterp, xObjv[iii], options, "option",
    0, &lOptIndex)
        switch (lOptIndex) {
        case WAVEGRID_HELP:
            sWaveGridHelpMsg(xInterp);
            return TCL_OK; break;
        case WAVEGRID_CLEAR:  // GUI Clear Method
            mGetWaveFormViewPtr()->mClearGrid();
            return TCL_OK; break;
        case …
```

Code Highlight Summary

In the above code, we’ve shown how the Tcl invocation of “wave” causes the C function gTclWave() to be called with all the arguments to that command. We’ve shown how the functions Tcl_GetIndexFromObjStruct() and Tcl_GetIndexFromObj() can be used to provide useful help output and to control the execution of GUI operations.

Future Work

1. Investigate recent developments in the area of supporting named arguments [9] when calling a Tcl function. This would entail handling the command arguments in Tcl rather than in C/C++. Additionally should Tcl_GetIndexFromObj() be extended to have more direct support available in C for named arguments?
2. Investigate providing raw help output for easier parsing and a “pretty print” Tcl proc to present the help output in whatever format is desired (indent, spacing, etc).

3. During review of this paper it was pointed out that the method of converting strings between Tcl and Qt that we utilize could run into problems with certain rare characters: "*The QString sprint method is expecting a char* containing ISO8859 characters, while Tcl_GetString returns a char* containting UTF8 characters. The correct way to translate Tcl string values to QStrings is by using the QString method 'fromUtf8()'. Similarly going from QString to Tcl_Obj should use the toUtf8() method.*" [10]

Conclusion

As we've shown Tcl provides excellent support for leveraging the power of Tcl for crafting user commands in your applications. It is up to you to choose how extensively you use this support. Before jumping into command implementation, plan ahead of time for: what the command user experience will be, what testing requirements are needed, what functionality 3rd party's will require and whether features like save and restore are going to be an issue. If done right, your customers will appreciate a clear and easy-to-use feature, testing will be easier, you'll have to do less special work for 3rd parties, and new features will be easier to implement.

Acknowledgements

I'd like to acknowledge Brian Griffin whose infinite knowledge in all things Tcl/Tk has made this work possible. I'd also like to thank my manager Ron Wold for supporting my desire to write this paper and in leading a team that inspires one to want to write it to somehow show a small inkling of the good work we are doing. And finally I'd like to acknowledge my fellow teammates Skyler Arabak, David Lowder, Joe Mueller, Chuck Pahlmeyer, Michelle Shao and Mike Webb for their stellar work that not only permeates this paper but who also make coming to work fun.

References

[1] "Qt", https://www.qt.io/

[2] "Editomat: Adventures in Commercializing a Tcl Application", Clif Flynt, http://www.tcl.tk/community/tcl2016/assets/talk34/editomat-paper.pdf

[3] "TkReplay", http://nikit.tcl.tk/page/TKReplay

[4] "Squish", https://www.froglogic.com/squish/editions/qt-gui-test-automation/

[5] "TclGetIndexFromObjStruct()", https://www.tcl.tk/man/tcl/TclLib/GetIndex.htm

[6] "TclCreateObjCommand()", https://www.tcl.tk/man/tcl/TclLib/CrtObjCmd.htm

[7] "QString", http://doc.qt.io/qt-5/qstring.html

[8] "Tcl_Obj", https://www.tcl.tk/man/tcl/TclLib/Object.htm

[9] "Tcl Support for Named Arguments", https://core.tcl.tk/tips/doc/trunk/tip/457.md

[10] Brian Griffin, superhero character name: "The Answer Key"