Customizable Keyboard Shortcuts

Ron Wold
Mentor Graphics Corporation
8005 SW Boeckman Road
Wilsonville, OR 97070
503-685-0878

Abstract

Anyone that spends a lot of time using the same software tool becomes very familiar with it. They know how the tool works, what the tool's commands are and when to execute these commands. Coined a "power user", these type of users operates fast. Power users want quick access to commands, they do not want to navigate through a menu to access commonly used operations. User interfaces often address this issue by adding toolbar buttons, but, the fastest method of access is through a keyboard shortcut.

A keyboard shortcut refers to the association of a key sequence with an operation. User interfaces typically have a predefined set of keyboard shortcuts. However, a tool that runs on multiple platforms and that has dozens of windows and hundreds of operations will be unable to define a single set of shortcuts that is sufficient for all users.

Modelsim is a software program written in Tcl/Tk that has been recently enhanced to support customizable keyboard shortcuts. Users can associate a key sequence with a menu pick, a toolbar button, a CLI command or a custom tcl script. In addition, users can specify that the key sequence is applicable for the entire tool or for just a specific window. Implementing this functionality presented several technical challenges. Tcl/Tk has a unique methodology for processing keyboard events and a successful solution requires an architecture that functions within the bounds of this methodology. This paper will discuss the basic architecture as well as the technical challenges that were faced and how they were addressed.

Keywords

Tcl/Tk, keyboard shortcuts, bind, bindtags, customizable

1. Introduction

Modelsim is an integrated development environment (IDE) used by electronic designers to develop, debug, simulate and test electronic designs. It supports several different hardware description languages (HDLs) - such as VHDL [1]

and Verilog [2]. Modelsim's user interface is comprised of many unique windows, toolbars, menus, popups and a command line interface. The user interface is written entirely in Tcl/Tk.

Developing a functional, well tested electronic design can take several weeks to several months. Users that spend this much time working with the same tool become 'power users' [3]. A key behavior of a power user is their desire to perform operations quickly. Within a graphical user interface, there are many ways to perform an operation such as selecting a menu item from a popup, clicking a tool bar or by typing the command into a command line interface. While each of these methods has their advantages, none of methods can be executed as quickly as a keyboard shortcut.

A keyboard shortcut[4] is a key combination that performs a certain command. The efficiency of a shortcut key comes from that fact that it can be invoked entirely from the keyboard. A user isn't required to position the mouse cursor or click mouse buttons.

Keyboard shortcuts are not a new concept. On some platforms, such as Microsoft Windows ™, there is a standard set of keyboard shortcuts. Given an operating system, a tool developer can identify the common shortcuts and implement them within their tool. However, a conflict can arise if the tool supports multiple platforms, and the standard shortcuts are different between the two platforms. For many years Modelsim detected the platform that was in use and defined the shortcuts based on the platform. Although this is more flexible than a single shortcut definition, the solution is still incomplete. Users want a shortcut definition that matches their own expectations, and they want shortcuts for the commands that they most often use. Since there is not a single set of keyboard shortcuts that will appease all 'power users', the best solution is to allow users to define and customize their own shortcuts.

2. Shortcut Fundamentals

Implementing a shortcut in Tcl/Tk requires the bind[5] command. In its simplest form, the bind command associates an event, like a key stroke or mouse event, with an action, like a procedure call. For example, consider this bind command:

```
bind .a.b.c <control-key-x> "Control_X_pressed"
```

This bind will result in the procedure "`Control_X_pressed`" being called when the widget .a.b.c receives the control-x key. In this example, the binding is placed on the widget .a.b.c, but bindings can also be placed on the name of a widget class. When a binding is placed on the name of a widget class, all instances of that widget inherit the binding. A binding can also be placed on "all" which causes all widgets to inherit the binding. Widgets also have the notion of bindtags. A widget's bind tags is a list of tag names, which may include the widget's name, that class name and all.

```
bindtags .a.b.c {.a.b.c My_Class . all }
```

When an event occurs on a widget, it is applied to each of the window's bind tags, in the order in which they are defined. If the bind tag has a binding definition for the event, then the bindtag's script is executed.

3. Capturing keyboard events

Modelsim's user interface is comprised of many windows. These windows are actually just widgets that contain other widgets, but from a user's standpoint, a window is a self contained tool providing specific functionality.

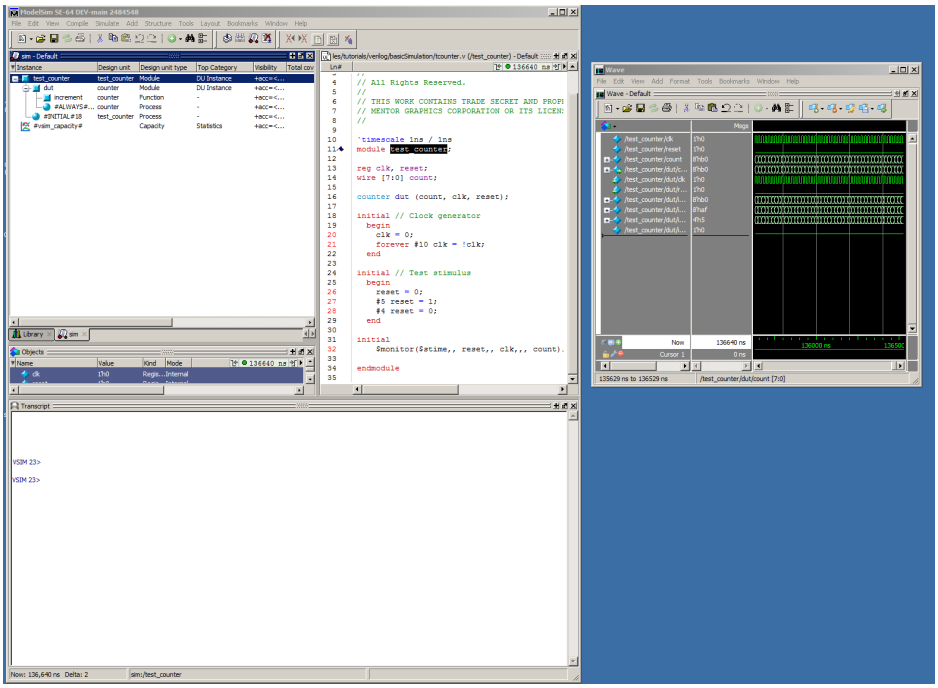


Figure 1 - Modelsim Windows

When a user activates a particular window and issues a keyboard event, such as key-delete, there is an expectation that whatever is selected in the window will be deleted. However, a window may be comprised of multiple subwidgets and each subwidget can take focus and thus be the target of the keyboard event. The user may have activated the window by clicking in any of the subwidgets. From the users standpoint, the shortcut key is defined by the window, not the individual subwidgets that make up the window. This creates a technical challenge in that every subwidget must have an identical binding. There are other possible solutions, such as forcing focus to a particular widget which contains the correct bindings, but these solutions require a detailed understanding of a window's construction. Modelsim has over 50 distinct windows, so defining a customizable binding architecture that requires significant window specific changes is not feasible given this project's time constraints.

Rather than managing the customizable bindings on a widget, an alternative approach was taken. Applying a binding to the 'all' tag provides a means of catching an event that is independent of the target widget. Given the following bind command:

```
bind all <key-delete> "Binding::ServiceBinding %W $key %k %x %y %X %Y %A"
```

a user can click anywhere in a window that is comprised of subwidgets, hit the delete key, and the procedure Binding::ServiceBinding will be executed. There are, however, two exceptions. The first is that all widgets must have the 'all' tag in their list of bindtags. This is not an issue with the Modelsim environment. Widgets receive the 'all' bind tag by default, and this bind tag is not removed from any widget. The second issue that can prevent the 'all' tag from receiving an event occurs if there is a binding for an event on one of the bind tags found earlier in the bind tag list and the binding script issues a break. For example, given these definitions:

```
bind .a.b.c <key-delete> "Delete_Something;break"
bind all <key-delete> "Binding::ServiceBinding %W <key-delete>"
bindtags .a.b.c { .a.b.c all}
```

the call to `Binding::ServiceBinding` will never occur. The binding tag .a.b.c is found earlier in the bind tag list than the bind tag all, so it is executed first. Since the binding script contains a break, all bindtag processes are halted. This behavior is fundamental to Tcl/Tk's bind processing algorthim, and the only way to assure that `Binding::ServiceBinding` is executed is by eliminating the break. The example below replaces the bind command on the widget with a procedure call, `Binding::DefineBinding:`

```
Binding::DefineBinding .a.b.c <key-delete> "Delete_Something"
bind all <key-delete> "Binding::ServiceBinding %W <key-delete>"
bindtags .a.b.c { .a.b.c all}
```

Replacing the bind call with a call to `Binding::DefineBinding` serves two purposes. First it eliminates the binding conflict between the `.a.b.c` tag and the `all` tag. More importantly, it captures and stores the *intent* of the original bind command. In this example, the intent can be described as *"if widget .a.b.c is the target widget and the delete key is hit, the procedure Delete_Something should be called"*. When `Binding::ServiceBinding` recieves an event, it compares the target widget and the key event with the binding defintions that have been defined via `Binding::DefineBinding` command. If a match is found, the script associated with the binding definition is executed.

4. The Binding database

Replacing a bind command with a command that saves the bind's intentions results in a database of binding definitions. Not all bind commands use the bind database. Many bind commands are not candidates for shortcut keys, such as binds that are based on mouse buttons or motion events. For example, the right mouse button raises a window's popup. If a user changed this binding they could lose access to the popup menu. Only bindings that are intended as a shortcut key are stored in the binding database.

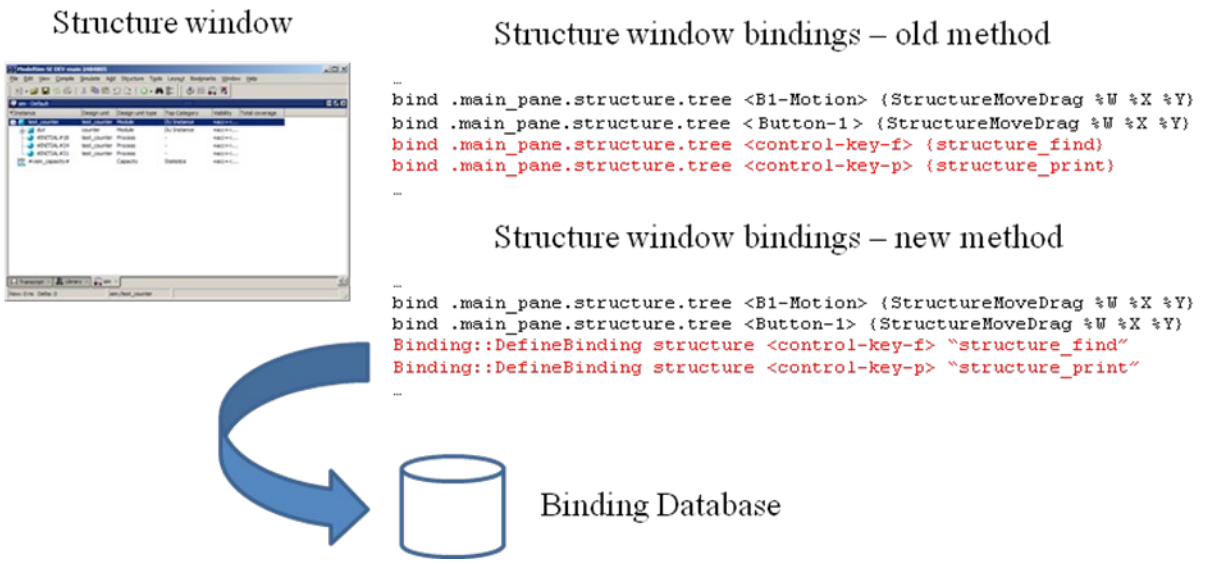


Figure 2

Storing binding definitions in a database has several advantages. First, changing a binding involves nothing more than changing the database. Since the bindings are no longer placed directly on widgets, knowledge of a window's widget hierarchy is not required when modify or adding a binding definition. The database also allows for persistent storage and binding definitions can easily be imported and exported. Lastly, determining what shortcuts are available in any given window becomes a trivial task.

5. Processing a keyboard event

The following tcl code creates a binding on the bindtag `all`. The binding will be in place for all widgets that currently exist as well as any widget created in the future.

```
foreach key [Supported_Shorcut_Keys] {
      bind all $key "Binding::ServiceBinding %W $key %k %x %y %X %Y %A"
}
```

When a shortcut key event occurs, regardless of the target widget, `Binding::ServiceBinding will` catch the event. `Binding::ServiceBinding` then examines the binding database and determines if there is a binding script associated with the event. If there is an associated script, the script is invoked.

6. Binding Priority

Modelsim is comprised of many windows. Each window has commands that are specific to only that window as well as keyboard shortcuts that reference these commands. Executing a window specific command requires that the window be activated. Likewise, a window's keyboard shortcut is only valid if the window that it is associated with it is active. Binding definitions that are associated with a particular window are called *window bindings*.

There are, however, commands that are not window specific. These commands are available without regard to the active window. For example, Modelsim's 'open' command will open a source file for editing and this command is always available. Binding definitions that are not associated with a specific window are called *global bindings*.

In addition to window and global bindings, there is also a distinction between intrinsic bindings and custom bindings. An *intrinsic* binding is one that is defined by a tool developer. An intrinsic binding is built into the tool and it is available to a user the first time they run Modelsim. A binding that a user adds is called a *custom* binding.

The distinction between window and global bindings and whether they are intrinsic or custom is important when an event matches more than one binding definition.

Consider the following scenario; a global intrinsic binding is added by a developer, say control-s, which raises a generic search dialog. In addition, a window intrinsic binding is added by a developer to the source window, it also uses control-s, but the bind script is different, it issues a command to save outstanding edits. Next, a user redefines the control-s binding definition for the source window to yet another command. When the user opens a source file and issues control-s, `Binding::ServiceBinding` will be called and it will examine the binding definitions database, looking for a match. This search will result in three matches, each with a different binding script. Only one of the binding scripts should execute, determining which binding definition to execute requires rules of priority. There are four categories of binding definitions, intrinsic global, custom global, intrinsic window and custom window. Given these four categories, we concluded that a window binding definition has precedence over a global binding definition and that a custom binding definition has precedence over an intrinsic binding definition. Implementing these rules of priority results in the following order of precedence. (Figure 3).

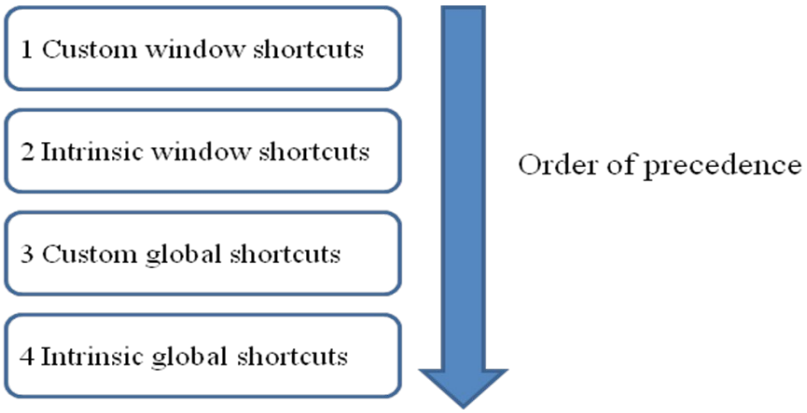


Figure 3

7. Editing the binding definition database

With an architecture that supports customized keyboard shortcuts, the final requirement is defining a user interface for adding, modifying and deleting keyboard shortcuts. The user interface must present the information found in the binding database in an understandable form, as well as provide operations for adding and modifying the database.

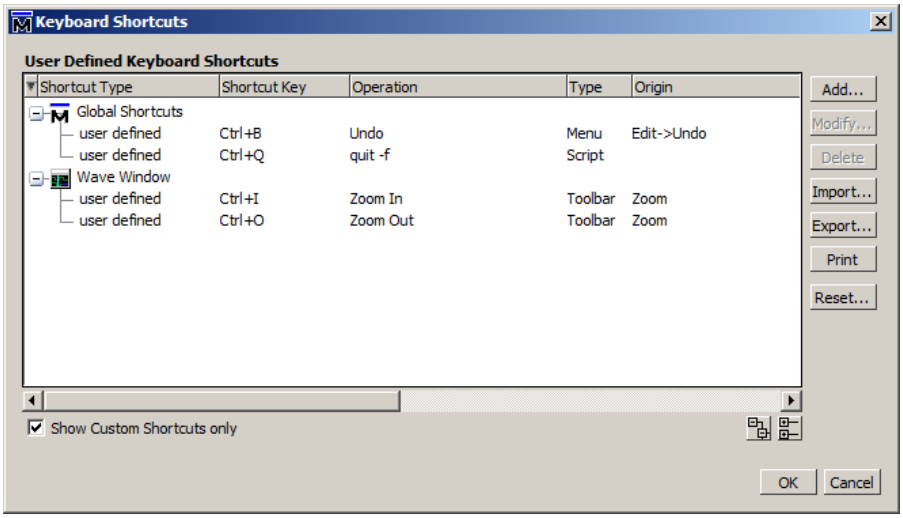
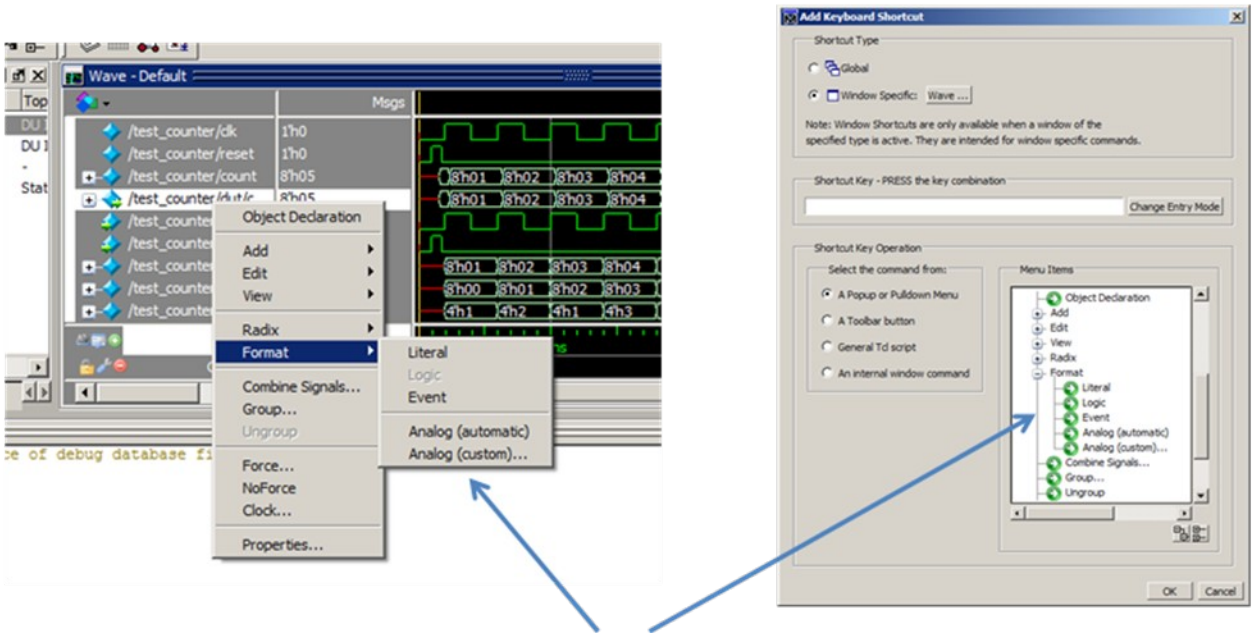


Figure 4

The keyboard shortcut dialog (Figure 4) lists the shortcuts found in the binding database. By default, only shortcuts added by the user are listed. Unchecking the check box 'Show Custom Shortcuts only' will cause intrinsic shortcuts to be displayed as well. Shortcuts are grouped by the window that they are associated with or as a global shortcut. A keyboard shortcut list item displays its shortcut type, the associated shortcut key, as well as information on the menu item or toolbar that the key is associated with.

Adding a keyboard shortcut is nothing more than adding a new entry in the binding database. From a user's standpoint, they simply want to associate a key sequence with a menu item, toolbar button or a tcl script. This simplistic requirement contains a technical challenge; specifically how does one present a user with a complete list of menu items and toolbar buttons?

A fewl years ago Modelsim's user interface underwent a re-architecture to address issues due to an ever increasing number of windows[7]. This re-architecture included a well defined API for creating menus, menu items and toolbars. The API is essentially a group of wrapper functions that embed the actual Tk menu functions. Since these wrapper functions are used for all menu and toolbar creation, they provide a single point for capturing and saving menu creation and hierarchy.



The menu items and their commands are captured at the time of creation. This data is used later by the "Add Keyboard Shortcut" dialog, providing a list of menu items that can be selected.

It is important to display menu items using the same name and hierarchy that is found in the menu itself. This is also true for toolbars, their name and listing order must also match the actual toolbar. Matching hierarchy makes finding the menu item or toolbar much easier.

8. Teaching the shortcuts

If a user is not aware of a keyboard shortcut, the shortcut will not be used. Although a listing of intrinsic shortcuts can be found in Modelsim's documentation, quite often users do not read the documentation. Modelsim has two features that are intended to help users learn the available shortcuts.

Keyboard Shortcut Quick Help

Modelsim has an intrinsic keyboard shortcut that will raise a temporary dialog. This dialog lists the keyboard shortcuts that are currently available for the active window.

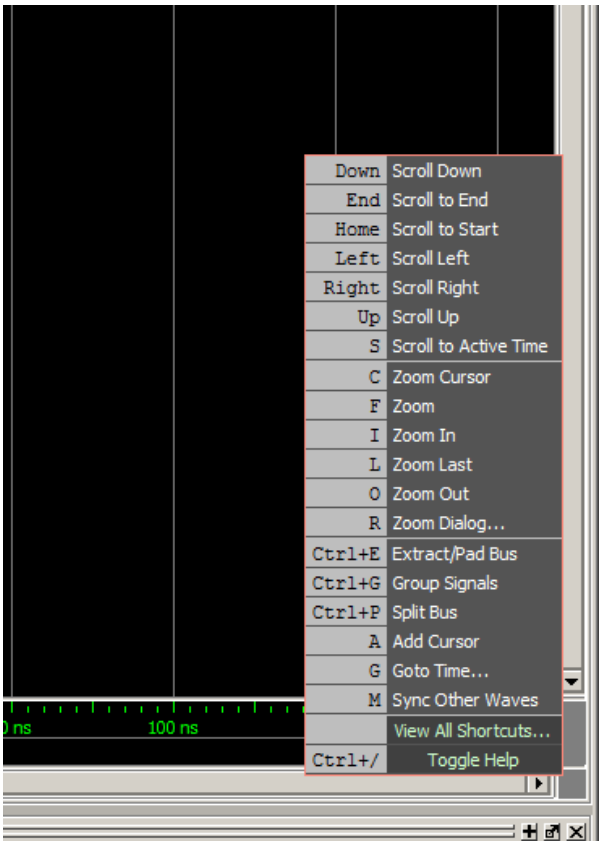
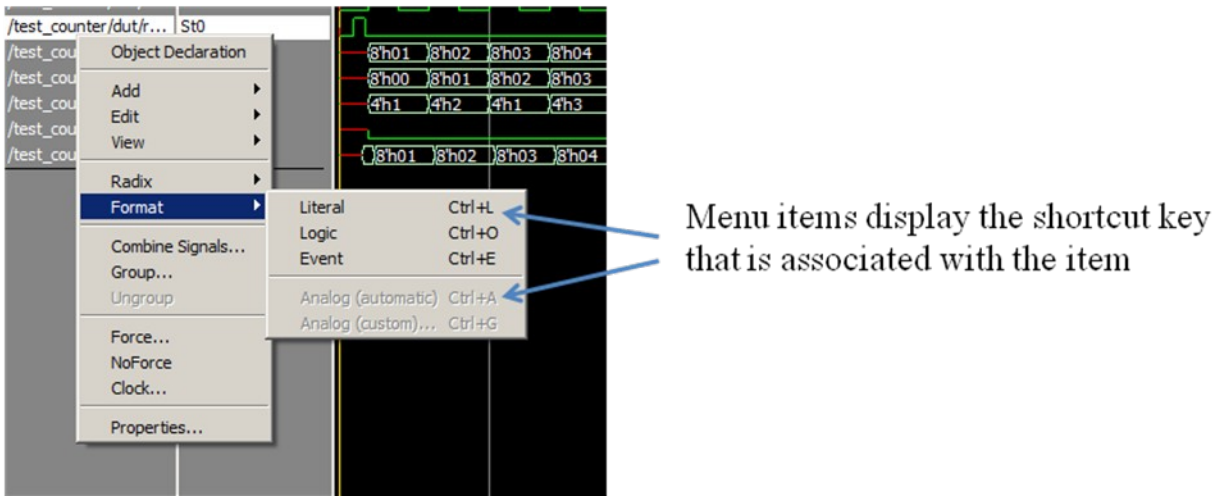


Figure 5 Keyboard Shortcut Quick Help

Dynamic Menu Item shortcut key association

If a menu item has an associated shortcut key, it has become common practice to display the key sequence to the right of the menu item text. We modified our default menu post command to query the default binding database before rendering each menu item. If a menu item has an associated shortcut key, the shortcut key is displayed to the right the menu text. The shortcut key display is dynamic in that it is not statically defined with the menu item. If a user changes or deletes a shortcut key, the associated menu item will reflect the change immediately.



9. Issues

Custom Window Bindings

When a user creates a custom window binding, they must specify the window type. If they are adding a custom binding to a menu or toolbar, the dialog provides a list of menu items or toolbars to select from. The list of menu items or toolbars is created when the menu item or toolbars is created. If a menu hasn't been created it will not be in the list that the user can select from. If a window has not been opened at least once, there will be no information to display in the add shortcut key dialog. We address this issue when the user selects the window type that they want to apply the binding to. The user selects from a list of window types and only windows that have been instanced at least once in the current session can be chosen. This solution is not ideal, but Modelsim's user interface must support 3rd party windows seamlessly and this approach achieves this.

Dialogs and in place edit boxes

Adding a binding to the "`all`" bindtag generates a lot of event traffic sent to `Binding::ServiceBinding`. The traffic does not generate a performance issue, but there are situations where the active window and the shortcut key match a binding definition, but the binding script should not be executed. A text entry box in a dialog is one example. Modelsim follows a model dialog model. When a dialog is raised, all key events are intended for the dialog, not the underlying window. The service binding routine must first detect whether a dialog is raised before processing a key event. When a dialog is raised and a key is detected, all key events received by the service routine are ignored..

An even more difficult issue occurs when in-place text entry boxes are used. Several of Modelsim's windows use in-place text boxes. For example, when the user double clicks on some text, a text box is placed directly on the window. Unlike a dialog, detecting a text entry box requires examining the widget class name. The service routine must exclude processing for certain class names.

Context specific menus

Context menus are created on the fly, the menu items are typically based upon a current state within the window, such as selection. Context specific menu items are cleared and recreated each time the menu is raised. For example, consider a debugger's breakpoint menu, when the user places the mouse over a visual break point and issues the popup, the menu items are created based upon the specific breakpoint. The menu items could have the name of the breakpoint in the menu text, as well as in the menu command. This type of menu item is not a candidate for a keyboard shortcut and special work is needed to prevent a user from binding to the menu item.

10.0 References

[1] Doulos, A Brief History of VHDL, http://www.doulos.com/fi/desguidevhdl/vb2_history.htm.
[2] Doulos, A Brief History of Verilog, http://www.doulos.com/fi/desguidevlg/vb2_history.htm
[3] http://en.wikipedia.org/wiki/Power_user
[4] http://www.techterms.com/definition/keyboardshortcut
[5] http://www.tcl.tk/man/tcl8.5/TkCmd/bind.htm
[6] http://www.tcl.tk/man/tcl8.5/TkCmd/bindtags.htm
[7] Too Many Windows, Ron Wold, 2010 Tcl/Tk Conference.