

## Tcl/Tk and Skill

A brief introduction to GUI development in Cadence Design  
Systems PCB Design Environment

**Andreas Kulik**  
**PTC**  
**(781) 370-5893**  
**[akulik@ptc.com](mailto:akulik@ptc.com)**

INTERNATIONAL CADENCE USERSGROUP CONFERENCE  
September 15-17, 2003  
Manchester, NH

## **1. Introduction**

Tcl/Tk is a wide spread script language used to develop graphical user interfaces on a variety of operating systems. Inspired by the article “Tcl/Tk and Skill – Mix it up” from Christopher Nelson in Dr. Dobbs Journal from February 2002, the question of how to combine the power of Tcl/Tk with Cadence’s Skill was answered.

This paper provides a brief introduction to GUI development with Tcl/Tk, visualizes the basic methods of how Cadence IPC Skill functions can communicate with such Tcl/Tk applications and how to embed them into your PCB Design Environment.

You will find example applications such as:

- A simple UI showing zoom functionality in ConceptHDL
- Place components in ConceptHDL by Part Number

## 2. IPC – Interprocess Communication Skill Functions

Understanding interprocess communication or IPC means to understand what processes are. In general a process is an instance of a program. In UNIX and other operating systems, a process is started when a program is initiated. So every shell command a user enters or any program a user starts will result in a process.

A process can initiate a subprocess, also known as a *child* process. The initiating process is also called the *parent* process. A child process is a replica of the parent process and shares some of its resources, but cannot exist if the parent is terminated. Processes can exchange information or synchronize their operation through several methods of interprocess communication, which are described below.

Interprocess communication is a set of programming interfaces, meaning functions and/or procedures, that allow a programmer to create and manage individual program processes that can run concurrently in an operating system. This allows a program to handle many user requests at the same time. Since even a single user request may result in multiple processes running in the operating system on the user's behalf, the processes need to communicate with each other. The IPC interfaces make this possible. Therefore, IPC SKILL functions are easy to understand. The IPC Skill functions, which are part of the Cadence PCB Software release, enable a programmer or EDA Tool Developer to create and communicate with child processes, meaning applications launched/started from within a SKILL program. Using this functionality a Programmer benefits from the following:

- “Create encapsulation tools or utility programs.
- Communicate with encapsulated programs using standard IO channels.
- Control the encapsulated programs by sending signals like kill, interrupt, stop, and continue.
- Allow encapsulated programs to execute SKILL commands in the parent process.
- Run child processes on remote hosts. [1]”

Such an interface functionality is a very powerful utility if you think of its capabilities in terms of customizing Cadence Applications. An EDA Tool Developer has the ability to write Applications that execute commands in Cadence® ConceptHDL™ or Allegro™ using SKILL functions. This provides a variety of possibilities to incorporate and integrate custom applications written in Java, Perl, Visual Basic, C\C++ or Tcl/Tk in the PCB Design Environment.

Now that we understand what interprocess communication is, the following two questions still remain. How does the parent process communicate with the child process? How does the parent process launch the child process?

The answer to the first question is simple. The parent and child process communicate through **stdin**, **stdout** and **stderr** channels. The actual communication can be carried out either synchronous or asynchronous. You should be aware of the two possible modes while dealing with output from a child process. You either synchronize the flow of a program with the output of a child process by performing blocking read operations, meaning a read operation will wait until data arrives from the child process or you choose to deal with output from a child process by registering a callback function. This function is invoked asynchronously whenever data is received from a child process and the event manager in the parent program is ready to handle the data.

Writing to a child or parent process does not mean the data was received successfully. Appropriate checking or verification functions, procedures or mechanisms need to be in place.

For example, sending a Skill command to a parent Skill process doesn't mean it can be executed successfully.

Lets take a look at an example to answer the second question. You start an application "myapp.exe" as a child process using the IPC Skill function *ipcBeginProcess*.

```
child_pid = ipcBeginProcess("myapp.exe" "" 'DataHandler 'ErrorHandler 'ExitHandler)
```

**DataHandler**, **ErrorHandler**, **ExitHandler** are callback functions and are called or executed whenever the parent receives data from the child process. In other words if "myapp.exe" writes something to its stdout, the function "DataHandler" will be executed with the process id of the child and the data as arguments. The function "ErrorHandler" will be invoked if the child produces errors on the stderr channel and "ExitHandler" will be called if the client process exits.

When writing IPC Skill routines remember the following:

### ***Input/Output handling***

- Determine early if synchronous or asynchronous
- Never mix synchronous or asynchronous
- errHandler always receives error messages

### ***Performance of dataHandlers***

- Change the frequency of incoming data

### ***Buffer limitations***

- IO channels only have a 4096 byte buffer

See Chapter 4 and Cadence's "IPC Skill Function Reference" manual for more details and examples.

### 3. TCL/TK – Tool Command Language and Graphical Tool Kit

Tcl (“tickle”) is a very simple programming language. No matter if a user has experience in writing scripts or programs, learning and understanding Tcl is not hard. A user should be able to write useful programs within hours. Tk (toolkit) is a standard add-on to Tcl that provides commands to quickly and easily create user interfaces.

For a scripting language, Tcl has a very simple syntax.

#### **cmd arg arg arg**

A Tcl command is formed by words separated by white space. The first word is the name of the command, and the remaining words are arguments to the command.

#### **\$myvariable**

The dollar sign substitutes the value of a variable. In this example, the variable name is myvariable.

#### **[expr 1 + 1]**

Square brackets execute a nested command. For example, if you want to pass the result of one command as the argument to another, you use this syntax.

#### **"this is a string"**

Double quotation marks group words as a single argument to a command. Dollar signs and square brackets are interpreted inside double quotation marks.

#### **{some stuff}**

Curly braces also group words into a single argument. In this case, however, elements within the braces are not interpreted.

**\**

The backslash (\) is used to quote special characters. For example, \n generates a new line. The backslash also is used to "turn off" the special meanings of the dollar sign, quotation marks, square brackets, and curly braces.

The following example shows the usage of three Tcl commands that print the current time. The commands are: set, clock, and puts.

The “set” command assigns the variable.

The “clock” command manipulates time values.

The “puts” command prints the values.

**set seconds [clock seconds]**

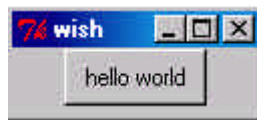
**puts "The current time is [clock format \$seconds]"**

Note that you do not use \$ when assigning to a variable. Only when you want the value do you use \$. The seconds variable isn't needed in the previous example. You could print the current time with one command:

```
puts "The current time is [clock format [clock seconds]]"
```

The following example shows the use of the TK toolkit. Note that the button command will return an error if a button with the name .b1 already exists.

```
set mytext "hello world"  
button .b1 -text $mytext -command exit  
pack .b1
```



**Figure 1**

## 4. Developing graphical interfaces with Skill and Tcl/Tk

The previous two chapters discussed IPC Skill functions and Tcl/Tk as an graphical extension language. This chapter describes how to combine IPC Skill and Tcl/Tk

### The Child Program

As mentioned in Chapter 2 the main communication between parent processes and child processes happens via *stdin*, *stdout*, *stderr*. We also know that it is a good practice to include some mechanism that allows to catch and analyze what we receive from our client process. For starters, lets suppose we want to create a simple graphical user interface (GUI) that provides us with simple zoom\_in/zoom\_out functionality. A program written in Tcl/Tk could look like this:

```
# create 3 button widgets
button .b1 -text "zoom in" -command [list TCLSkill_Send "zoom in"]
button .b2 -text "zoom out" -command [list TCLSkill_Send "zoom out"]
button .b3 -text "zoom fit" -command [list TCLSkill_Send "zoom fit"]

# place the button in the main widget using the window manager "grid"
grid .b1 -in . -row 0 -column 0 -sticky nswe -padx 10 -pady 5
grid .b2 -in . -row 1 -column 0 -sticky nswe -padx 10 -pady 5
grid .b3 -in . -row 2 -column 0 -sticky nswe -padx 10 -pady 5

# create a text widget and place it in the main widget
text .t -width 50 -height 10
grid .t -in . -row 3 -column 0 -sticky nswe
```

The resulting GUI is shown in Figure 2.

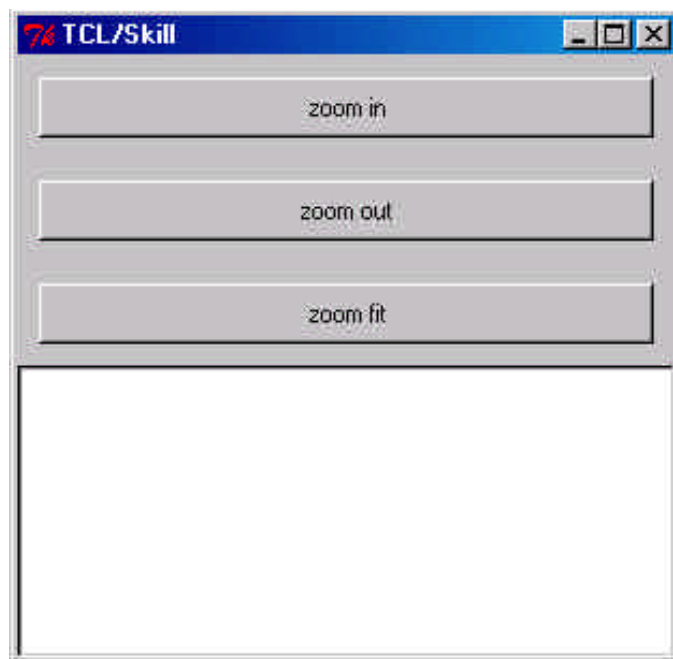


Figure 2

Each of the three button widgets has a command associated with it. This command is a procedure called “TCLSkill\_Send”, which takes one argument in form of a string and is executed whenever the user selects the button with a left mouse button. In our example above the commands we want to send to a Cadence Application are “zoom in”, “zoom out” and “zoom fit”. The purpose of the procedure TCLSkill\_Send is to write these commands to the *stdout* channel. The procedure itself could look like this:

```
proc TCLSkill_Send {command} {
    puts stdout $command
    flush stdout
}
```

In order to accomplish a full flow of communication you need a callback function in Tcl that detects and evaluates data it receives on its *stdin* channel. You accomplish this with the following commands:

```
#configure the receiving channel
fconfigure stdin -buffering none -blocking 0 -translation auto

#define the callback procedure
fileevent stdin readable "TCLSkill_Recv"
```

The referenced callback function “TCLSkill\_Recv” first reads the data from *stdin* and in case the data isn’t complete, keeps reading. Note that this approach could freeze your application if the parent process never sends the <crLf> to finish the command. In our example, the complete response will be displayed in the text widget “.t”. A more useful program would contain a parser, evaluating the response to test if it is a valid tcl command and execute it.

```
proc TCLSkill_Recv {} {
    set response [read stdin]
    while {![info complete $response]} {
        append response [read stdin]
    }
    # insert the response into the text widget
    .t insert end "R> $response"
    # adjust the current position in the text widget
    .t yview moveto 1
}
```

## The Parent program

You launch the child process using the skill function “ipcBeginProcess()”. Below are two examples, one for Concept Skill one for Allegro Skill, illustrating the use of this function. The command line “wish tcl\_skill.tcl” will start the child program described above as a child process (wish is Tcl/Tk’s window shell). The assumption is made that the Tcl program can be found in the skill path. Note that the skill function tclInterp assumes that the data received can be send straight to Allegro or ConceptHDL without any error checking.



### **Concept SKILL**

```
(defun tclInterp (childID data)

  cnSendCommand(cnhandle, data)
  ipcWriteProcess(childID data)

)

;; add the dfII/bin path to the path environment variable
;; otherwise the ipc server will not launch when ipcBeginProcess is invoked.
sprintf( dfii_bin_path "%s\\tools\\dfII\\bin" getShellEnvVar("CDSROOT"))
setShellEnvVar( sprintf(nil "path=%s;%s" getShellEnvVar("path"),dfii_bin_path))

;; we need to import a concept handle in order
;; to send the data we received
;; from the client to ConceptHDL
cnhandle = cnmpsImport()
;; Launch the child process
tclp = ipcBeginProcess("wish tcl_skill.tcl" "" 'tclInterp )
```

### **Allegro SKILL**

```
(defun tclInterp (childID data)

  axlShell(data)
  ipcWriteProcess(childID data)

)

;; add the dfII/bin path to the path environment variable
;; otherwise the ipc server will not launch when ipcBeginProcess is invoked.
sprintf( dfii_bin_path "%s\\tools\\dfII\\bin" getShellEnvVar("CDSROOT"))
setShellEnvVar( sprintf(nil "path=%s;%s" getShellEnvVar("path"),dfii_bin_path))

;; Launch the child process
tclp = ipcBeginProcess("wish tcl_skill.tcl" "" 'tclInterp )
```

Below is a more sophisticated approach to the function tclInterp. The skill function stringToList can be found in the Appendix of this paper.

```
(defun tclInterp (x_childID t_data)
  ;; this function assumes that the data received
  ;; is a skill command list such as (setq a 1)

  ;; convert the data received to a string
  cmd_list = stringToList(t_data)

  if( null(cmd_list) then
    printf("Could not process request from client\n\t%s\n" t_data)

    else

      ;; again even though cmd_list could contain multiple commands
      ;; we just evaluate and execute the first one for now...
      cmd = car(cmd_list)

      ;; print the cmd we execute, so we can see whats going on
      printf("\n*****\n")
      printf("eval( %L )\n" cmd)

      ;; evaluate the cmd, this will execute the command or
      ;; report an error
      ;; if the command had issues
      result = errset( eval( cmd ) )

      ;; print it so we can see
      printf("\neval_result %L\n")
      printf("*****\n" result)

      ;; send \n to tcl app indicating that we are done
      ipcWriteProcess(x_childID "\n")

    );if null(cmd_list)

  ) ;tclInterp
```

The above examples are rather simple. A more useful application could be a graphical user interface in ConceptHDL, for example, that allows the user to find and place logic symbols by properties such as a PART\_NUMBER. Although Cadence Design Systems does offer solutions such as Part Browser (free), it seems more user friendly to have such an application available within ConceptHDL. Since most Companies do have Part Numbers stored in either part table files or chips files and most Designers or Engineers only know the part by part number, a "place by part number" interface does make sense. The benefits of such interface in view of drawing schematics efficiently and quickly are obvious. To add additional value, one also could incorporate a footprint and datasheet viewer. Possible additions could be on-hand stock quantities or component lead times which could be directly retrieved from the MRP system.

Figure 3 shows this limited but efficient part/component browser, which allows the user to search for part numbers in the library. A part number could be the company specific part number in the MRP system or a manufacturers part number of the component itself. If available the user can view the corresponding Allegro footprint or datasheet of the part.

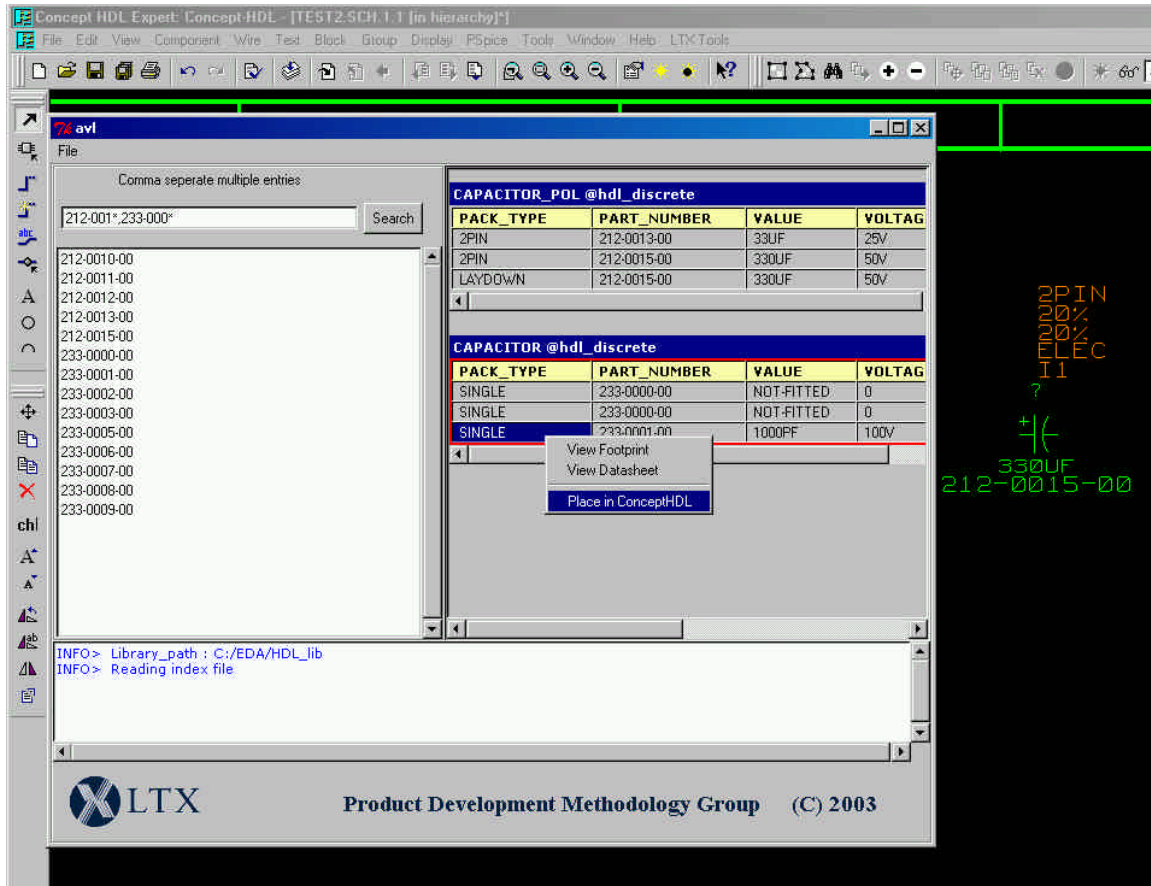


Figure 3

Please note that launching external applications such as the “Allegro Free Viewer” or Adobe Acrobat Reader™ may not work as expected. It seems that both application do not launch successfully if started as a sub process of a subprocess. One workaround for this problem, especially Acrobat Reader, would be to identify the application associated with the pdf file extension and add it's path, which most likely will contain spaces on Microsoft operating systems, to the path environment variable using the procedure below. Please note that this procedure uses and therefore requires the tcl package “registry”, which is loaded using commands such as:

```
package require registry 1.1
load_unsupported reg1.1.dll
```

Doing so will allow you to execute the following command in skill successfully.

```
(setq retmsg system(sprintf(nil "start /separate /realtime $PDFVIEWER %s" $pdf_file)) )
```

```
proc SetPdfViewer {} {
```

```

global PDFVIEWER PDFVIEWERPATH

set err [catch {registry get HKEY_CLASSES_ROOT\\.pdf {}} ret_val]
if {$err > 0} {
    # error
    set msg "Couldn't find pdf filetype registry entry! "
    tk_messageBox -type ok -icon error -message $msg -type ok
} else {
    set ftype_assoc [lindex $ret_val 0]
    set err [catch {registry get HKEY_CLASSES_ROOT\\$ftype_assoc\\shell\\open\\command {}}
ret_val]
    if {$err > 0} {
        # error
        set msg "Couldn't find PDFViewer application registry entry"
        tk_messageBox -type ok -icon error -message $msg -type ok
    } else {
        set strlen [string length $ret_val]
        set start 0
        set outstr ""

        #we need to remove any leading or trailing quotation marks
        #(there are other ways of doing this)
        for {set i 0} {$i < $strlen} {incr i} {
            set outchar [string index $ret_val $i]

            if {[string match {\"} [string index $ret_val $i] ] } {
                if {$start == 0} {
                    set outchar \{
                    set start 1
                } else {
                    set outchar \}
                    set start 0
                }
            }
            append outstr $outchar
        }
        regsub -all {[\\]} $outstr {/} outstr

        set pdfapp "[lindex $outstr 0]"

        set PDFVIEWER [file tail $pdfapp]
        set PDFVIEWERPATH [file dirname $pdfapp]
        # We will send a command to Skill, which will add the path to the pdf viewer
        # to the path environment variable.
        set pdfpath "simplifyFilename\\(\"$PDFVIEWERPATH\\\""
        set command "\"( setq retmsg setShellEnvVar( sprintf\\(nil \\\"path=%s;%s\\\"
getShellEnvVar\\(\"path\\\" , $pdfpath \\) \\) \\)"
        TCL_SendSkillCommand $command
    }
}

update
}

```

## **5. Conclusion**

The current functionality in front-end and back-end tools such as Allegro and ConceptHDL is sufficient enough to 90%. The missing 10% is the key to success, because the custom tools ensure that the new key product is on the market prior to any competition.

Screening the PCB Design Process for repetitive tasks is crucial to success in today's market place. But all the screening doesn't help if you are not able to quickly automate the process. Combining Tcl/Tk with Cadence's Skill, is just one of many examples, allowing you to quickly address the need of custom interfaces in Cadence Design Systems PCB Design Environment. As a matter of fact any other script language or system level programming language can be used in combination with Cadence's IPC Skill functions. Java, Perl, Visual Basic, C, C++ to name a few. The only aspect a programmer has to keep in mind is the fact that the processes communicate via the standard channels stdin, stdout and stderr. This does require a change in the thinking process and some time to get used to while creating the algorithm or program flow. You are not programming in the theme of if-then-else anymore, your are implementing code that deals with events and processes

## **6. References**

- [1] Cadence Online Documentation - PSD 14.2 – 2003
- [2] Interprocess Communication Skill Function Reference, November 2001
- [3] <http://www.scriptics.com>
- [4] <http://www.ddj.com/articles/2002/0202/>

Listing and examples can be found on the ICU2003 Conference CD-ROM

## **7. Acknowledgements**

Many thanks to the Skill Experts and Guru's at the ICU-PCB-FORUM for answering my questions regarding skill programming. I also would like to thank R&D at Cadence Design Systems for pointing out the Tcl/Tk article from Christopher Nelson.

## 8. Appendix

The function “stringToList” is taken from the code listings provided at “Dr. Dobbs Journal” website [4]. Those listings were part of the article “TCL/TK AND SKILL MIX IT UP” from Christopher Nelson which was published in Dr.Dobbs Journal in February 2002.

```
;; -----  
;; Convert a string to a list.  
;;  
;; The input string might be "(setq a 1) (setq b 2)"  
;; which would make (quote) fail because it expects a single argument.  
;; To avoid this, we wrap the input in another level of ().  
;; The caller should use (foreach) or similar to loop through  
;; each list from the input string.  
;;  
;; For example:  
;;   (setq a '((a b c) (d e f)))  
;;   (sprintf b "%L" a)  
;;   (setq c (car (stringToList b)))  
;; makes a and c equivalent lists (that is, (equal a c) returns t).  
(defun stringToList (s)  
  (let (result)  
    ;; Make sure no warnings are hanging about to interfere with our test  
    ;; after the errsetstring.  
    (getWarn)  
    ;; Try to convert the string to a list  
    (setq result (car (errsetstring (sprintf nil "(quote (%s))" s) nil)))  
    ;; If we have a warning in the conversion above, return nil.  
    ;; Otherwise, return the result of the conversion.  
    (if (getWarn) nil result)  
    );; let  
  );; stringToList
```