Tcl Scripting in Vivado

**1 Introduction**

The Vivado®IDE uses Xilinx Design Constraints (XDC) to specify the design constraints. XDC is

based on a subset of all the Tcl commands available in Vivado and is interpreted exactly like Tcl.

The Vivado tools write a journal file called vivado.jou into the directory from which Vivado

was launched. The journal is a record of the Tcl commands run during the session that can be used as a starting point to create new Tcl scripts.

Tk是Tcl的图形界面工具集。对应于wish（windowing shell）运用程序。

**2 A Brief Overview of Tcl**

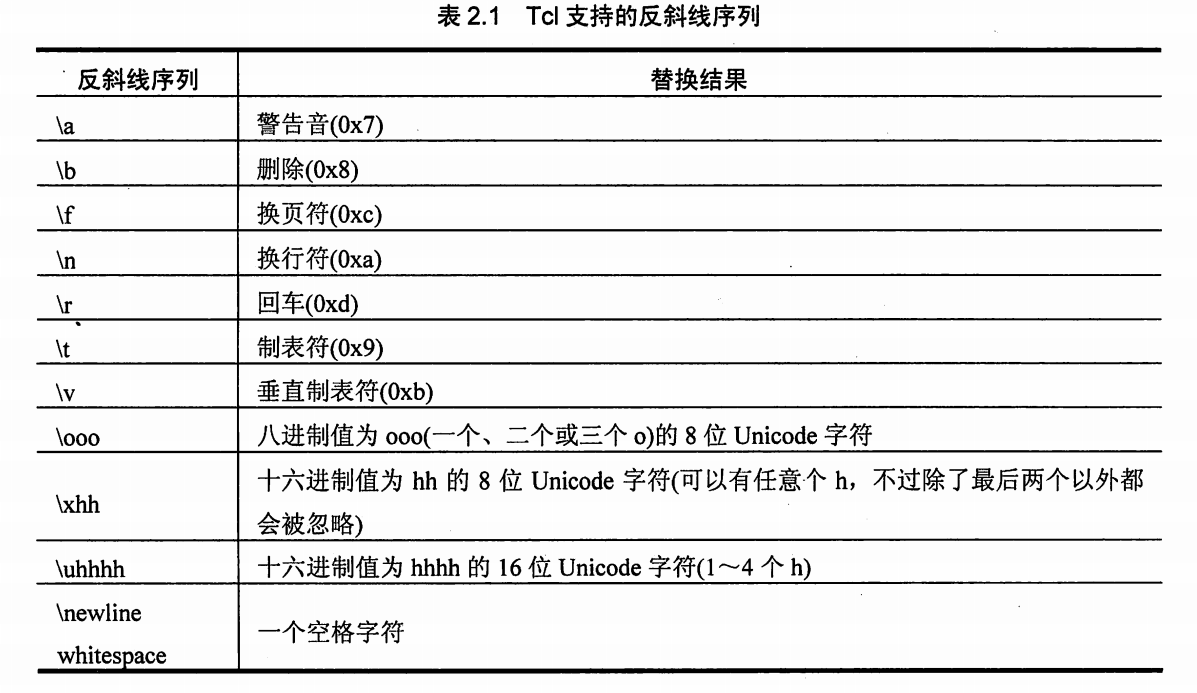
A Tcl script is a series of Tcl commands, separated by new-lines or semicolons. A Tcl command is a string of words, separated by blanks or tabs.

A word is a string that can be a single word, or multiple words within braces, {}（其中字符完全按照字面意义）, or multiple words within quotation marks, ””（其中字符或单词按照tcl语言关键字意义）.the backslash, \, is treated as a special character even within braces and quotation marks.

Commands can also be nested inside other commands within brackets, [], which are evaluated from left to right in a bottom-up manner.

[] Square brackets are treated as standard characters in Verilog and VHDL names (net, instances, etc.), and usually identify one or more elements of vectors, such as buses or arrays of instances.

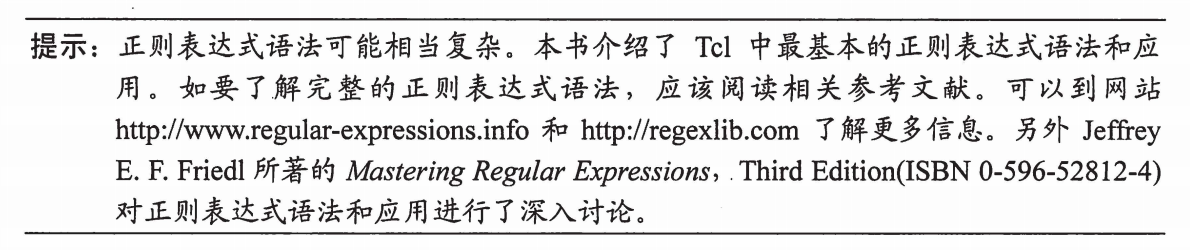
\替换。



\n 表示换行符。在双引号”字符串”中出现’\n’字符时，若输出该字符串将被分为两行。

\空格 表示空格字符。

\换行 表示空格字符，一个’\’紧接一个’换行’，中间没有空格。’\换行’字符通常用于命令行的分行书写，避免一行太长。在双引号”字符串”中出现’\换行’字符时，若输出该字符串，’\换行’字符将被替换成一个空格，且输出为一行，而不是两行。



glob \* 命令相当于ls ./ 意思是将通配符‘\*’匹配的字符串展开。

encoding system -- 列出系统支持的字符编码，比如unicode

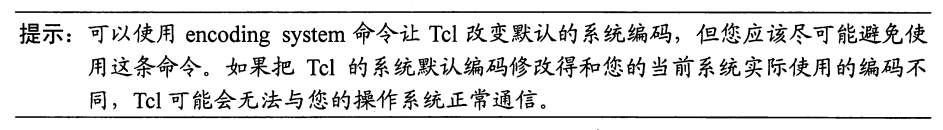
encoding convertfrom gb2312 “字符串” -- 将gb2312字符串转换为utf-8 unicode

encoding convertto gb2312 “str” -- 将utf-8字符串转换为gb2312

source -encoding gb2312 “script.tcl” 以gb2312编码方式执行script.tcl

set fd [open $file r]

Fconfigue $fd -enconding gb2312 调整文件的编码方式



**3 Getting Help**

You can get help directly from the Tcl console. Every Vivado command supports the -help

command line argument that can be used anywhere in the line.

In addition, there is a help command that provides additional information. Providing a command name to the help command (that is, help <command>) reports the same help information as <command> -help.

The help command can also just return a short description of the arguments using the -args

option: help <command> -arg

A short summary of the syntax of a command is also available with the -syntax option:

help <command> -syntax

A list of categories can be obtained by executing the “help” command without any argument or option. The categories is:

ChipScope

DRC

FileIO

Floorplan

GUIControl

IPFlow

Object

PinPlanning

Power

Project

PropertyAndParameter

Report

SDC

Simulation

TclBuiltIn

Timing

ToolLaunch

Tools

XDC

The list of commands available under each category can be also reported with the -category

option. For example, reporting all the commands under the Tools category:

help -category tools

数组是一组变量的集合。由，数组变量名（元素名） — 值，表达。$数组变量（$元素变量）可求出值。

set {capital(R china)} beijing #设置数组变量capital，元素R china 的值为beijing

>beijing

set capital(R\_china) beijing #设置数组变量capital，元素R\_china 的值为beijing

>beijing

上述capital(R china) 与 capital(R\_china)相当于两个变量。由于前一个包含空格，所以要用{}或””包含。

puts "$capital(R china)" #字符串中的变量需要替换

>beijing

Puts ${capital(R china)} #变量范围由{}界定

>beijing

puts {$capital(R china)} #字符串原样输出不替换

>$capital(R china)

注意{}或””的用法区别。

set nation "R china" #创建变量nation，其值是R china，是一个元素名

>R china

puts $capital($nation)

>beijing

set nati R\_china #创建变量nati，其值是R\_china，是一个元素名

>R\_china

puts $capital($nati)

>Beijing

puts $capital($R\_china)

can't read "R\_china": no such variable #元素变量与元素名不是一回事！

puts $capital(${R china})

can't read "R china": no such variable #同上，看来数组比后面的列表，字典麻烦

set matrix(1,1) 140 #多维（二维）数组

set matrix(1,2) 218

set matrix(1,3) 84

set i 1

set j 2

set cell $matrix($i,$j)

> 218

列表list {字符串0 字符串1，…… } 是字符串组。其索引从0开始的自然数。

字典dict {key0 value0 key1 value1 ……} 是key-value字符串对的组合，还是字符串集合。字符串value的索引是key，不是自然数。字典是具有偶数个元素的列表。

字典和数组有根本的不同，数组是变量的无序集合，其元素是变量；而字典是一组具有索引名称的值的有序集合。数组通常用于动态传递具备同类属性的数值；字典通常用于静态保存具备同类属性的数值。

set example {first joe sub schmoe title mr}

>first joe sub schmoe title mr

lindex $example 1

>joe

dict get $example sub

>schmoe

split命令将字符串按指定的分隔符如，空格符，转换为列表。

join命令将列表按照指定的分隔符，转换为字符串。

**4 Platform Specific Tcl Behaviors**

win32, win64, lnx32: sizeof(int) is 4bytes

lnx64: sizeof(int) is 8bytes

**5 Compilation and Reporting Example Scripts**

**5.1 Compilation with a Non-Project Flow**

**5.2 Compilation with a Project Flow**

**6 Loading and Running Tcl Scripts**

**6.1 Initializing Tcl Scripts**

The Vivado Design Suite can automatically load Tcl scripts defined in **Vivado\_init.tcl** file.

When you start the Vivado tools, it looks for a Tcl initialization script in several locations with the following precedence:

1. In the software installation: <installdir>/Vivado/<VivadoVersion>/scripts/Vivado\_init.tcl

Where <installdir> is the installation directory where the Vivado Design Suite is installed.

2. In the local user directory (Vivado tools version dependent):

• For Windows 7: %APPDATA%/Xilinx/Vivado/<VivadoVersion>/Vivado\_init.tcl

For example: %APPDATA%/Xilinx/Vivado/2017.1/Vivado\_init.tcl

• For Linux: $HOME/.Xilinx/Vivado/<VivadoVersion>/Vivado\_init.tcl

For example: $HOME/.Xilinx/Vivado/2017.1/Vivado\_init.tcl

3. In the local user directory (Vivado tools version independent):

• For Windows 7: %APPDATA%/Xilinx/Vivado/Vivado\_init.tcl

• For Linux: $HOME/.Xilinx/Vivado/Vivado\_init.tcl

If Vivado\_init.tcl exists in several locations, the Vivado tool sources the file following the

precedence explained above.The Vivado\_init.tcl file in the home directory allows each user to specify additional commands, or to override commands from the software installation to meet their specific design requirements.

**6.1 Sourcing Tcl Scripts**

source *<filename>*

Where <filename> specifies both the name of the file, as well as the relative or absolute path to the file.

Within the Vivado IDE you can also source a Tcl script from the Tools → Run Tcl Script menu command.

source <filename> -notrace

No echos.

**6.2 Executing a Tcl Script at Startup**

vivado -source myscript.tcl

source myscript.tcl

A checkpoint can also be specified on the command line along with a Tcl script:

vivado design.dcp -source myscript.tcl

open\_checkpoint design.dcp

source myscript.tcl

**6.3 Using Tcl Scripts in a Constraints Set**

Tcl scripts can be added to project constraint sets like any regular XDC file, except that the XDC files are managed by the tool, and not Tcl scripts.

**6.4 XDC Constraints: read\_xdc versus source**

When applying constraints to the design, the commands read\_xdc and source differ in

behavior. For more information, see the Vivado Design Suite User Guide: Using Constraints (UG903).

**6.5 Defining Tcl Hook Scripts**

In a Non-Project flow you have the ability to source a Tcl script at any point in the flow, such as before or after running the synth\_design command. You can also do this in a project-based flow, using the Vivado IDE, or by using the set\_property command to set a property on either a synthesis or implementation run. Tcl hook scripts allow you to run custom Tcl scripts prior to (tcl.pre) and after (tcl.post) synthesis and implementation design runs, or any of the implementation steps.

Right-click a run in the Design Runs window and select the Change Run Settings from the pop-up menu to open the Design Run Settings dialog box. The tcl.pre and tcl.post options can be used to specify a Tcl hook script.

**6.6 Sharing Hook Scripts Between Steps**

**6.7 Customizing the GUI**

**7 Writing a Tcl Script**

**7.1 Defining Tcl Procedures**

You can write Tcl scripts that can be loaded and run from the Vivado IDE, or you can write procedures, to act like new Tcl commands, taking arguments, checking for errors, and returning results.

A Tcl procedure is defined with the proc command which takes three arguments: the procedure name, the list of arguments, and the body of code to be executed. The following code provides a simple example of a procedure definition:

proc helloProc { arg1 } {

# This is a comment inside the body of the procedure

puts "Hello World! Arg1 is $arg1"

}

A procedure usually has predefined arguments. Each of them can optionally have a default value. When an argument has a default value, it does not need to be specified when calling the procedure if all the mandatory preceding arguments are specified. A procedure returns an empty string unless the return command is used to return a different value.

proc reportWorstViolations { nbrPaths corner delayType } {

report\_timing -max\_paths $nbrPaths -corner $corner -delay\_type $delayType

-nworst 1

}

proc reportWorstViolations { nbrPaths { corner Slow } { delayType Max } } {

report\_timing -max\_paths $nbrPaths -corner $corner -delay\_type $delayType

-nworst 1

}

The args keyword is a Tcl list that can have any number of elements, including none.

proc reportWorstViolations { nbrPaths args } {

eval report\_timing -max\_paths $nbrPaths $args （$args represent any arguments legally）

}

%> reportWorstViolations 2

%> reportWorstViolations 1 -to [get\_ports]

%> reportWorstViolations 10 -delay\_type min\_max -nworst 2

上述命令（过程）均能正常执行。

注意eval命令，eval是动态运行命令：

在shell中启动 tclsh

% puts abc

% eval puts abc

两条执行的结果是一样的，体会不到eval的好处，因为我们静态的编写了这条命令"puts abc"。如果想执行多条命令，命令是变化的，在 \*.tcl 脚本中可以定义字符串变量，变量对应不同的命令，每次最后调用：

eval $cmd

就可以运行不同的命令，动态命令就是变化的命令。

此外eval命令能扫描后续参数，并将其分解为独立单词，可解决一些语法问题，参见Tcl\_Tk入门经典(第2版)第46页，参数展开章节。

**7.2、Parsing Command Line Arguments**

A single procedure that can handle multiple contexts is easier to use and maintain that multiple procedures that cover the same range of contexts with duplicated code. Tcl provides an easy way to do this through the args variable. The keyword args used inside the list of arguments of a procedure can match any number of elements, including none. The args variable is a regular Tcl list（列表） that can be processed and analyzed like any Tcl list.

在过程体中为了让$args列表展开成单词，通常需要使用eval命令。参见Tcl\_Tk入门经典(第2版)第46页，参数展开章节。

通常有三种参数列表的分析方案。

switch -exact

switch -glob

assigning the list of command arguments to a Tcl array

**Return语句**

return -code error {Oops, something is not correct}

return -code ok {Done}

**7.3、Local and Global Variables**

A local variable is a variable created inside a procedure. It is created at runtime inside the stack of the function. The variable is only accessible within the procedure and the variable name is not subject to name collision with variable names outside of the procedure.

A global variable is a variable created outside of a procedure and that belongs to the global

namespace. To refer to a global variable inside a procedure, the keyword global is used

followed by the variable name

**7.4、Namespaces for Procedures**

**7.5、Template Script**

**8 Accessing Design Objects**

The Vivado Design Suite Tcl interpreter provides access to many first class objects

（一级对象） such as project, device, nets, cells, and pins. The Vivado Design Suite updates these design objects dynamically, as the design progresses, and loads them into the in-memory database in both Project and Non-Project modes.

You can query design objects using the get\_\* Tcl commands which return list of design objects. The complete list of get\_\* commands can be returned with help get\_\*.

Each class of design object (net, pin, port, ) has a unique set of standard properties that can be read and sometimes written to modify their value in the database. In addition, the design

attributes specified in the RTL source files, the Verilog parameters and VHDL generics are stored with the associated netlist object as properties.

We can get the list of properties on an object by using the list\_property command. When a property type is enum, it is possible to get the list of all the valid values by using the list\_property\_value command.

There are two properties that are common to all objects: NAME and CLASS. When an object is assigned to a Tcl variable, a pointer to the object is stored in the variable. Objects can be passed by variable to Tcl commands or Tcl procs. When an object is passed as an argument to a Tcl proc or command which expects a string, the object's NAME property value is passed instead of the object itself.

set inst [get\_cells cpuEngine] # get object cpuEngine cell to variable inst

>cpuEngine

puts $inst # puts command outputs the NAME property of the cell

*>>cpuEngine # Name property of the cpuEngine cell*

report\_property $inst #show all properties of the *cpuEngine cell*

*>>Property Type Read-only Value*

*>>>CLASS string true cell*

*>>FILE\_NAME string true \*

*C:/2014.1/cpu/project\_1.srcs/sources\_1/imports/netlist/top.edf*

*>>IS\_BLACKBOX bool true 0*

*>>IS\_PRIMITIVE bool true 0*

*>>IS\_SEQUENTIAL bool true 0*

*>>LINE\_NUMBER int true 812044*

*>>NAME string true cpuEngine*

*>>PRIMITIVE\_COUNT int true 11720*

*>>REF\_NAME string true or1200\_top*

The following example creates a property, SELECTED, for a cell object. The value of the property is defined as an integer.

create\_property SELECTED cell -type int

The following example sets the SELECTED property to a value of 1 on all the cells that match the specified name pattern, \*aurora\_64b66b\*:

set\_property SELECTED 1 [get\_cells -hier \*aurora\_64b66b\*]

**8.1、Getting Objects By Name**

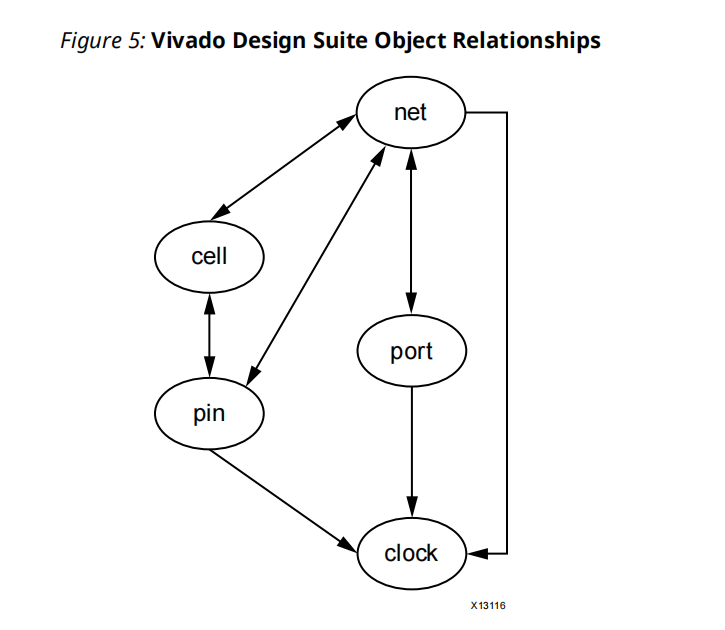
**Using the -hierarchical Option**

**Using the -filter and -regexp Options**

**Searching for Pins**

**Filtering Results**

**8.2、Getting Objects by Relationship**



The Help message for each of the get\_\* commands that supports the -of\_objects option

lists the related objects that can be traversed:

get\_cells -of\_objects {pins, timing paths, nets, bels or sites}

get\_clocks -of\_objects {nets, ports, or pins}

get\_nets -of\_objects {pins, ports, cells, timing paths or clocks}

get\_pins -of\_objects {cells, nets, bel pins, timing paths or clocks}

get\_ports -of\_objects {nets, instances, sites, clocks, timing paths, io standards, io

banks, package pins}

Note: If an empty list of objects is passed through -of\_objects, then the get\_\* command returns an empty Tcl list.

**8.3、get\_nets Command**

**8.4、Timing Path Object**

The get\_timing\_path command can be used to return a list of timing path objects.

report\_timing -of <timingPathObject>

**8.5、Persistency of Objects**

**9 Handling Lists of Objects**

When using the get\_\* commands, the returned object looks and acts like a standard Tcl list.

However, the Vivado Design Suite is returning a container of a single class of objects (for

example cells, nets, pins, or ports), which is not a generic Tcl list. The container of objects is

handled automatically by the Vivado Design Suite, and is totally transparent to the user. For

example, the standard Tcl llength command can be used on a container of objects (for example from get\_cells) and returns the number of elements in the container, like it would for any standard Tcl list.

The built-in Tcl commands that handle lists in the Vivado Design Suite have been overloaded and enhanced to fully support objects and containers of objects. For example, lsort, lappend, lindex, and llength, have been enhanced to manage the container based on the NAME property of the object.

You can add new objects to the container (using lappend, for instance), but you can only add

the same type of object that is currently in the container. Adding a different type of object, or string, to the list is not permitted and will result in a Tcl error.

**9.1、Collection versus String Representation**

The get\_\* commands do not create a regular Tcl list of elements. They create a Vivado

collection of objects, but the Vivado collections have been implemented in such way that they behave like any regular Tcl list.

For example, if a design has 20000 instances:

vivado% set cells [get\_cells -hier]

vivado% llength $cells

>>20000

vivado% set var [join $cells "\n"] #join命令，将列表转换为字符串，字符串的分隔符是\n

vivado% llength $var #将var字符串视为列表，求列表元素的数量

>>501 #只能取得20000个元素中的前500个，最后一个元素是...

vivado% lindex $var end

>>...

**10 Redirecting Output**

Many of the Vivado Design Suite Tcl commands allow you to redirect the information returned by the command to a file with the -file option, for printing or processing outside of the tool; or as a string that can be saved in a variable with the -return\_string option for further processing within the Vivado tools.

Some of the commands supporting file output include:

report\_datasheet

report\_drc

report\_power

report\_timing

report\_timing\_summary

report\_utilization

A relative or absolute path can be specified as part of the file name. A relative path is relative to the directory from which the Vivado tools have been started, or to the current working directory which can be retrieved with the pwd command.

To append the content from a command to an existing file, use the -append option in addition to -file. For example, the code below creates one file, all\_violations.rpt, that combines the output of two separate commands:

report\_timing -delay\_type max -file all\_violations.rpt

report\_timing -delay\_type min -file -append all\_violations.rpt

All report\_\* commands also support the -return\_string option. This option directs the

command to return its output as a string that can be assigned to a Tcl variable.

Some of the commands that support -return\_string are:

report\_clocks

report\_clock\_interaction

report\_disable\_timing

report\_environment

report\_high\_fanout\_nets

report\_operating\_conditions

report\_power

report\_property

report\_pulse\_width

report\_route\_status

report\_utilization

You can split the returned string from the report on the newline character, \n, to process the

string line by line as a list:

set timeLines [split [report\_timing -return\_string -max\_paths 10] \n ]

#split命令将字符串转换为列表

**10.1、 Accessing Files**

After a file has been written to the file system, Tcl provides many useful commands for working with the files. You can extract elements of a file, such as the file path, file name and file extension.

**10.2、Working with Strings**

The -return\_string argument directs the output of a report\_\* command to a Tcl string

rather than to stdout. The string can be assigned to a Tcl variable, and parsed or otherwise

processed.

**11 Controlling Loops**

Tcl has few built-in commands such as for, foreach and while that are used to loop or iterate

through a section of code.

Their syntax is:

for <{start}> <{testCondition>} <{next}> <{body}>

foreach <varname> <{list}> <{body}>

while <{testCondition}> <{body}>

With all the above commands, the entire Tcl script body is executed at each iteration. However, Tcl provides two commands to change the control flow of the loop: break and continue. The break statement is used to abort the looping command. The continue statement is used to jump to the next iteration of the loop.

**12 Handling Tcl Errors**

**12.1、Checking Validity of Variables**

When developing Tcl scripts, it is recommended to always check for corner cases and for

conditions where the code could fail.

proc get\_pin\_dir { pinName } {

if {$pinName == {}} {

puts " Error - no pin name provided"

return {}

}

set pin [get\_pins $pinName]

if {$pin == {}} {

puts " Error - pin $pinName does not exist"

return {}

}

set direction [get\_property DIRECTION $pin]

return $direction

}

**12.2、Handling Tcl Errors**

To do safe programming and catch the TCL\_ERROR condition, Tcl has a built-in catch command that returns 1 when an error is caught and otherwise returns 0. The catch command can be used on a single command or a set of commands.

The basic syntax of the catch command is:

catch <{script}> [varname]

if {[file exists $filename]} {

if {[catch { set FH [open $filename r] } errorstring]} {

puts " File $filename could not be opened : $errorstring"

} else {

# The file is opened, do something

#

close $FH

}

} else {

puts " File $filename does not exist"

}

**13 Accessing Environment Variables**

Tcl provides a convenient way to access the environment variables in a read-only mode through the Tcl global variable env. The variable env is a Tcl array that is automatically created and initialized at startup inside the Tcl interpreter.

The env array is a global variable and can therefore be referenced inside a proc after being

declared as global.

Vivado% puts "The PATH variable is $env(PATH) "

Vivado%: set all\_env\_var [array names env] #get list of all variables

**14 Calling External Programs**

It is possible to call external programs from within Tcl and capture the returned result. This is done using the exec command.

The following example (Linux) gets the list of all files and directories under the run directory by calling the ls command.

vivado% set result [exec ls]

vivado% foreach element [split $result \n] { ... }

**15 Vivado Integrated Design Environment**

**(IDE)/Tcl Modes versus Batch Mode**

The Tcl console that runs under the Vivado Integrated Design Environment (IDE) (vivado -mode gui) or Tcl mode (vivado -mode tcl) is slightly different from the one that runs

under the batch mode (vivado -mode batch).

But the consequence is that during an interactive session (Tcl or GUI modes), Vivado runs under tclsh mode. However, in batch mode, Vivado runs in Tcl script mode. That means in Tcl and GUI mode, the tcl command was usually run in interactive session on tclsh; while in batch mode, the tcl commands was organized by scripts that was executed in batches.

tcl模式与gui模式可视为命令行与图形交互模式；而batch模式是命令批处理模式。

**16 Creating Custom Design Rules Checks (DRCs)**

The Vivado Design Suite lets you define and use custom design rule checks (DRCs) written in Tcl.

When creating custom DRCs be aware of the following:

• The basis of creating a custom DRC is a Tcl checker procedure that gets design objects of

interest, or attributes of those design objects, and a checking function that defines the design rule. The Tcl checker procedure is defined in a separate Tcl script that must be loaded into the Vivado Design Suite prior to running report\_drc. Inside of the Tcl checker procedure, the create\_drc\_violation command is used to identify and flag violations when checking

the rule against a design. The create\_drc\_violation command creates a violation object

within the in-memory design, with properties that can be reported and further processed in

the Vivado Design Suite.

• The Tcl checker procedure is associated to a user-defined DRC that is created using the

create\_drc\_check command. Call this rule by name when you run the report\_drc

command.

• Optionally, you can elect to create a DRC rule deck by using the create\_drc\_ruledeck

command. A DRC rule deck is a collection of DRCs. A mix of both user-created and predefined DRCs can be added to the user-created DRC rule deck by using the add\_drc\_checks command.

• Checking the design against the design rules is performed by running the report\_drc

command. When running report\_drc, you can specify to run either a DRC rule deck, user

defined design rule checks, or predefined DRCs.

**16.1、Creating a Tcl Checker Procedure**

The Tcl checker procedure selects the design objects of interest to be checked. It then performs the necessary tests or evaluations of the design objects, and finally returns the results in the form of DRC violation objects that identify the objects associated with the specific error.

# This is a simplistic check -- report BRAM cells with WRITE\_WIDTH\_B wider

than 36.

proc dataWidthCheck {} {

# list to hold violations

set vios {}

# iterate through the objects to be checked

foreach bram [get\_cells -hier -filter {PRIMITIVE\_SUBGROUP == bram}] {

set bwidth [get\_property WRITE\_WIDTH\_B $bram]

if { $bwidth > 36} {

# define the message to report when violations are found

set msg "On cell %ELG, WRITE\_WIDTH\_B is $bwidth"

set vio [ create\_drc\_violation -name {RAMW-1} -msg $msg $bram ]

lappend vios $vio

}; # End IF

}; # End FOR

if {[llength $vios] > 0} {

return -code error $vios

} else {

return {}

}; # End IF

} ; # End PROC

This Tcl script file must be loaded into the Vivado tools prior to

running the report\_drc command. Refer to Loading and Running Tcl Scripts for more

information on loading the Tcl checker procedure.

**16.2、Creating a DRC Check**

Once the Tcl checker procedure is defined, you must now define the DRC as part of the DRC

reporting system within the Vivado Design Suite.

First, you must register the new design rule using the create\_drc\_check command. This

command requires you to provide a unique name for the user-defined rule check. This name that must match the name given to the violation created by the Tcl checker procedure. You will need to specify this unique name when adding the check to DRC rule decks or when running report\_drc.

create\_drc\_check -name {RAMW-1} -hiername {RAMB Checks} \

-desc {Block RAM Data Width Check} -rule\_body dataWidthCheck -severity

Advisory

You can optionally group the DRC into a special category, provide a description of the rule for reporting purposes, and assign a severity.

You can define a message to add to the DRC report when violations are encountered. By default, the message created by the create\_drc\_violation command in the Tcl checker procedure is passed upward to the DRC object. In this case, any message defined by

create\_drc\_violation is simply passed through to the DRC report.

The DRC object features the is\_enabled property that can be set to TRUE or FALSE using the

set\_property command. When a new rule check is created, the is\_enabled property is set

to TRUE as a default. Set the is\_enabled property to FALSE to disable the DRC from being

used when report\_drc is run.

**16.3、Creating a DRC Rule Deck**

You can optionally group multiple related DRC checks that can be run together into a DRC rule deck. To do this, you must first create the DRC rule deck by using the create\_drc\_ruledeck

command. Once the DRC rule deck is created, DRCs can be added and removed from the DRC rule deck by using the add\_drc\_checks and remove\_drc\_checks commands. Mixing userdefined checks and predefined checks into a single DRC rule deck is allowed in the Vivado Design Suite. Below is an example of creating a DRC rule deck called myrules along with the addition and removal of DRCs from the DRC rule deck.

create\_drc\_ruledeck myrules

add\_drc\_checks -ruledeck myrules {RAMW-1 RAMW-2 RAMW-3}

remove\_drc\_checks {RAMW-2} -ruledeck myrules

Note: If the is\_enabled property of the DRC is set to FALSE, then the DRC will not be run as part of the DRC rule deck when running report\_drc. In some cases, it might be more desirable to disable the DRC than to remove it from the DRC rule deck.

**16.4、Reporting Custom DRCs**

A user-defined DRC can be run individually, with other rules, or as part of a DRC rule deck using the report\_drc command. Below are examples of running the previously defined RAMW-1 rule individually, with other rules, and as part of the previously created DRC rule deck.

report\_drc -check {RAMW-1}

report\_drc -check {RAMW-1 RAMW-2}

report\_drc -ruledecks myrules

Remember that the is\_enabled property of the rule check must be set to TRUE in order for

report\_drc to run the check.

**16.5、DRC Explanation Script**

There are times when the designer has a DRC rule name or a pattern of DRC rules and wants to get an explanation about what these rules are doing. This can be done by reporting properties on the DRC objects.

proc explain\_drc { drcs } {

package require struct::matrix

set loop\_drcs [get\_drc\_checks $drcs -quiet]

if {$loop\_drcs == {}} {

puts " Error: $drcs does not match any existing DRC rule"

return

}

struct::matrix drcsm

drcsm add columns 3

drcsm add row {DRC\_ID SEVERITY DESCRIPTION}

foreach drc $loop\_drcs {

set description "\{[get\_property DESCRIPTION [get\_drc\_checks $drc]]\}"

set severity "\{[get\_property SEVERITY [get\_drc\_checks $drc]]\}"

set key "\{[get\_property NAME [get\_drc\_checks $drc]]\}"

drcsm add row "$key $severity $description"

}

puts "[drcsm format 2chan]";

drcsm destroy

}

Vivado% explain\_drc CFGBVS-1

DRC\_ID SEVERITY DESCRIPTION

CFGBVS-1 Warning Missing CFGBVS and CONFIG\_VOLTAGE Design Properties

Vivado% explain\_drc CFGBVS-\*

DRC\_ID SEVERITY DESCRIPTION

CFGBVS-1 Warning Missing CFGBVS and CONFIG\_VOLTAGE Design Properties

CFGBVS-2 Critical Warning CFGBVS Design Property

CFGBVS-3 Warning CONFIG\_VOLTAGE Design Property

CFGBVS-4 Critical Warning CFGBVS and CONFIG\_VOLTAGE Design Properties

CFGBVS-5 Critical Warning CONFIG\_VOLTAGE Design Property

CFGBVS-6 Critical Warning CONFIG\_VOLTAGE with HP Config Banks

CFGBVS-7 Warning CONFIG\_VOLTAGE with Config Bank VCCO

Vivado% explain\_drc foo

Error: foo does not match any existing DRC rule

**16.6、Manipulating DRCs**

DRCs are just like other objects in Tcl and their properties can be changed. Use the following

command to list the properties of a DRC Object.

Vivado% report\_property [get\_drc\_checks RAMW-1]

Property Type Read-only Visible Value

ARCHITECTURES string\* true true

CLASS string true true drc\_check

DESCRIPTION string true true Block RAM Data Width Check

GROUP string true true RAMW

HIERNAME string true true RAMB Checks

IS\_ENABLED bool false true 1

IS\_USER\_DEFINED bool true true 1

MESSAGE string true true

MSG\_ID int true true 1

NAME string true true RAMW-1

SEVERITY enum false true Advisory

Note: There are two properties of the RAMW-1 DRC check that you can modify. Those are the

IS\_ENABLED property and the SEVERITY property. Change the values of these properties on the DRC

Check object by using the set\_property command, just like any other object.

To disable the RAMW-1 DRC check, issue the Tcl command:

Vivado% set\_property IS\_ENABLED false [get\_drc\_checks RAMW-1]

To increase the severity of the RAMW-1 DRC Check, issue the Tcl command:

Vivado% set\_property SEVERITY {Critical Warning} [get\_drc\_checks RAMW-1]

These properties can also be changed on built-in DRC rules. In order to reset the built-in DRC

rules to factor defaults, use the following Tcl command:

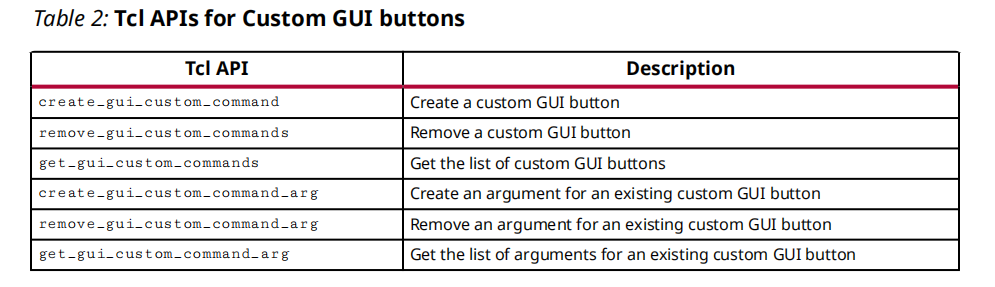
Vivado% reset\_drc\_check [get\_drc\_checks]

**17 Custom GUI Buttons**

There are two ways to define a custom button:

• Through the GUI: Tools → Custom Commands → Customize Commands.

• Programmatically through a Tcl API.



A custom button can be defined with or without one or more arguments. After the custom

button has been created using create\_gui\_custom\_command, each argument must be added with create\_gui\_custom\_command\_arg.

When a custom button is removed, Vivado Design Suite automatically removes all its arguments. There is no need to manually remove the arguments.

**18 Xilinx Tcl Store**

**18.1、Accessing the Xilinx Tcl Store**

**18.2、Coding Guidelines**

**18.3、Contributing to the Xilinx Tcl Store**

**19 Tcl Scripting Tips**

**19.1、Performance via Nesting**

**19.2、Caching Objects**

**19.3、Object Names and the NAME Property**

**19.4、Connecting and Disconnecting Nets**

**19.5、get\_property and Sorted Lists**

**19.6、Formatting Lists of Objects**

**19.7、Finding Vivado Tcl Commands by Options**

**19.8、Writing Efficient Code**

**19.9、Getting User Input**

20 使用tcl库和包

**20.1、tcl库**

Tcl库是一个tcl脚本目录，该目录包含一个或多个实现相关过程的tcl文件，以及过程与文件相对应的索引文件（tclIndex）。

例如:

info library #显示tcl标准库的路径

>/usr/share/tcltk/tcl8.6

puts $tcl\_library #全局变量tcl\_library同样保存tcl标准库的路径

>/usr/share/tcltk/tcl8.6

库的使用

A、比如建立自己tcl库：/home/my\_tcllib, 该目录下有foo.tcl、 bar.tcl两个过程相关文件。这两个文件均在其中按格式定义了若干proc，也就是若干自定义命令。

1. auto\_mkindex /home/my\_tcllib \*.tcl #auto\_mkindex不是内置tcl命令，是tcl标准库的过程
2. set auto\_path [linsert $auto\_path 0 /home/my\_tcllib]

#将/home/my\_tcllib加入全局变量$auto\_path列表的首位

1. tclsh利用unknown过程机制即可自动识别自定义库中的过程并调用。但是tcl库不方便移植，要使用tcl包。

**20.2、tcl包**

通过创建tcl包，可以对tcl库进行版本管理，方便tcl库代码的组织。

使用包：

package require pack\_name version #package require命令加载版本至少为version以上的pack\_name包。没有version参数将加载最新版本；使用-exact version将仅用version版本。

创建包：

package provide pack\_name version

#package provide命令说明本文件的源码提供了名为pack\_name的包。目录中可以有多于一个tcl脚本使用package provide pack\_name version命令。

使用库过程pkg\_mkindex，在目录中创建pkg\_Index.tcl文件：

pkg\_mkIndex \*.tcl

包安装：将包文件夹放置在：：tcl\_pkgPath变量指向的路径中。通常第一个路径是平台相关的包，第二个路径是平台无关的包。

package name 列出所有已安装的包

package versions pack\_name 列出pack\_name的version