Introduction to Lattice Tool Scripting

Introduction to Lattice Tool Scripting

- Why create scripts?
  - Implement new approaches to designing
  - Improve productivity
  - Not as difficult as some might think
    - Reusability possible once you get going initially
  - Work entirely from the command line

```
set project "/home/lattice/Desktop/xo2_proj/xo2_proj.ldf"

prj_project open $project
prj_run Export -impl impl1
```

- What types of scripts are typically used within the Lattice tool flow?
  - **TCL (Tool Command Language)**
    - TCL scripts are used to automate the FPGA design flow for Lattice's main design tools
      - Diamond, Radiant, Propel Builder
    - Most actions in each tool's GUI have corresponding TCL commands which can be used in a scripted flow
  - **Makefile**
    - Makefile scripts are used to generate compiled code for embedded software projects (C/C++)
    - Lattice Propel™ SDK automatically generates the necessary build scripts to compile a C/C++ project
      - In the Lattice tool flow it is not required for users to create these own scripts
      - If users desire to customize their build scripts, they should create a new file called makefile.targets

Creating Lattice Tool Scripts

Radiant & Diamond Tool Scripting Overview

Radiant Scriptable Features

- Synthesis, MAP, PAR
- Bitstream generation
- General project management (TCL)
- Message promotion & demotion (TCL)
- Reveal Analyzer & Controller (TCL)
- Power Calculator (TCL)
- ECO Editor (TCL)
- Device Programming
- Deployment Tool
- IP Generation

Diamond Scriptable Features

- Synthesis, MAP, PAR
- Bitstream generation
- General project management (TCL)
- Reveal Analyzer (TCL)
- Power Calculator (TCL)
- ECO Editor (TCL)
- Device Programming
- Deployment Tool

Creating Lattice Tool Scripts

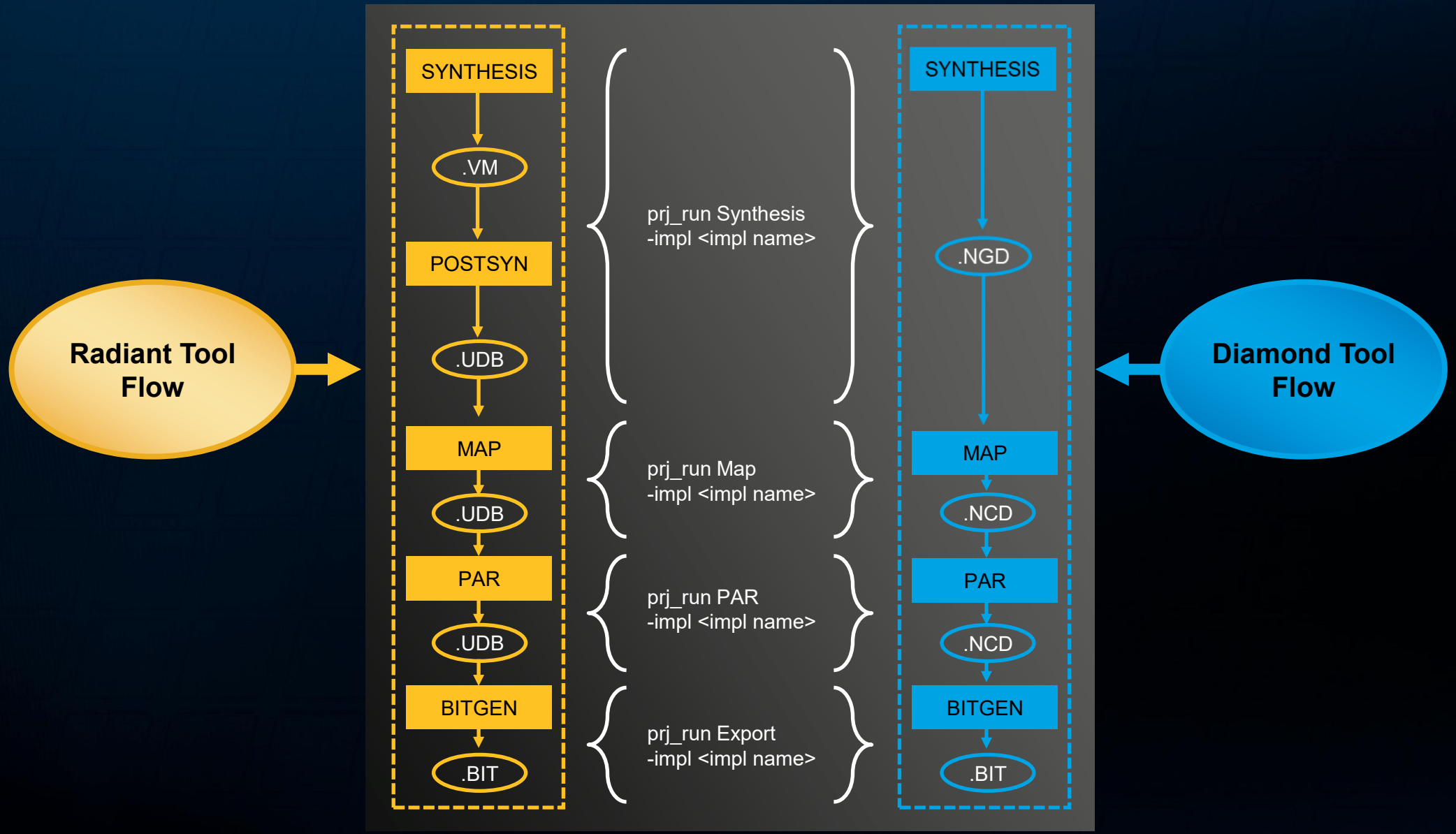
- The two main methods for scripting the Lattice FPGA build flow are TCL and batch
  - Depending on the type of script (TCL or batch), there are different commands that are used
  - How each script is invoked in the Lattice tool design flow also depends on the type of script
    - This is the first thing that you should consider when creating your script

- For TCL based scripts, Radiant and Diamond both have an integrated TCL console
  - TCL console is integrated at the GUI level, but there is also a console-only version as well
  - It is also possible to automatically invoke TCL scripts upon launching Radiant or Diamond's TCL consoles

- For batch mode scripts, there is additional setup required before Lattice tool commands can be used at the command line level
  - Depending on the operating system this setup varies slightly

General Scripting Flow

Radiant & Diamond General Scripting Flow



Synthesis & Post-Synthesis Commands

Synthesis & Post-Synthesis Commands (Radiant)

- Method #1: LSE batch mode
  - synthesis
  - postsyn
- Method #2: Synplify Pro batch mode
  - synpwrap
  - postsyn
- Method #3: Project TCL mode
  - prj_run Synthesis -impl <implementation name>

Synthesis & Post-Synthesis Commands (Diamond)

- Method #1: LSE batch mode
  - Synthesis

- Method #2: Synplify Pro batch mode
  - synpwrap
  - edif2ngd
  - Ngdbuild

- Method #3: Project TCL mode
  - prj_run Synthesis -impl <implementation name>
  - prj_run Translate -impl <implementation name>
    - Synplify Pro only

Map, Place & Route, and Bitstream Generation Commands

- Method #1: batch mode
  - map
  - par
  - bitgen
- Method #2: Project TCL mode
  - prj_run Map -impl <implementation name>
  - prj_run PAR -impl <implementation name>
  - prj_run Export -impl <implementation name>
    - -forceAll to rerun every stage before the selected one
    - -forceOne to force rerun this stage only

```
#Running MAP
map -a "MachXO2" -p LCMXO2-7000HE -t TQFP144 \
-s 4 -oc Commercial   "xo2_proj_impl1.ngd" \
-o "xo2_proj_impl1_map.ncd" -pr "xo2_proj_impl1.prf" \
-mp "xo2_proj_impl1.mrp" \
-lpf "$prj_path/xo2_proj.lpf"  -c 0

#Running PAR
par -w -l 5 -i 6 -t 1 -c 0 -e 0 -gui \
-exp parUseNBR=1:parCDP=0:parCDR=0:parPathBased=OFF:parASE=1 \
xo2_proj_impl1_map.ncd xo2_proj_impl1.dir/5_1.ncd xo2_proj_impl1.prf

#Generating bitstream and JEDEC
bitgen -f "xo2_proj_impl1.t2b" \
-w "xo2_proj_impl1.ncd"  "xo2_proj_impl1.prf"
bitgen -f "xo2_proj_impl1.t2b" \
-w "xo2_proj_impl1.ncd"  -jedec "xo2_proj_impl1.prf"
```

Running TCL Scripts

Running TCL Scripts

- Methods for invoking TCL scripts vary slightly between each Lattice tool
  - **Lattice Tool Interactive TCL Console (Radiant, Diamond & Propel)**
    - source <TCL script location>
  - **Radiant**
    - Windows: <Radiant install path>/bin/nt64/pnmain.exe -t <TCL script location>
      - Console mode: <Radiant install path>/bin/nt64/pnmainc.exe <TCL script location>
    - Linux: <Radiant install path>/bin/lin64/radiant -t <TCL script location>
      - Console mode: <Radiant install path>/bin/nt64/radiantc <TCL script location>
  - **Diamond**
    - Windows: <Diamond install path>/bin/nt64/pnmain.exe -t <TCL script location>
      - Console mode: <Diamond install path>/bin/nt64/pnmainc <TCL script location>
    - Linux: <Diamond install path>/bin/lin64/diamond -t <TCL script location>
      - Console mode: <Diamond install path>/bin/lin64/diamondc <TCL script location>
  - **Propel Builder**
    - Windows: <Propel install path>/builder/rtf/bin/nt64/propelbld.exe <TCL script location> -gui
      - Console mode: <Propel install path>/builder/rtf/bin/nt64/propelbld.exe <TCL script location>
    - Linux: <Propel install path>/builder/rtf/bin/lin64/propelbldwrap <TCL script location> –gui
      - Console mode: <Propel install path>/builder/rtf/bin/lin64/propelbldwrap <TCL script location>

Running Batch Scripts

Running Batch Scripts (Windows)

- Setting up the command line for Radiant batch mode
  - Set the **PATH** & **FOUNDRY** environment variables
    - set PATH=<Radiant install path>/bin/nt64;<Radiant install path>/ispfpga/bin/nt64
    - set FOUNDRY=<Radiant install path>/ispfpga
  - Begin using Lattice tool commands

- Setting up the command line for Diamond batch mode
  - Set the **PATH** & **FOUNDRY** environment variables
    - set PATH=<Diamond install path>/bin/nt64;<Radiant install path>/ispfpga/bin/nt64
    - set FOUNDRY=<Diamond install path>/ispfpga
  - Begin using Lattice tool commands

```
set PATH=C:/lscc/radiant/2023.1/bin/nt64;C:/lscc/radiant/2023.1/ispfpga/bin/nt64
set FOUNDRY=C:/lscc/radiant/2023.1/ispfpga

set device=LIFCL
set part=LIFCL-40
set package=CABGA256
set speed=9_High-Performance_1.0V
set oc=Commercial

synpwrap -prj clnx_project_impl_1_synplify.tcl -log clnx_project_impl_1.srf
```

Running Batch Scripts (Linux)

- Setting up the command line for Radiant batch mode
  - Set the **bindir** environment variable
    - export bindir=<Radiant install path>/bin/lin64
  - Invoke Radiant environment setup script
    - source $bindir/radiant_env
  - Begin using Lattice tool commands

```
export bindir=/home/lattice/lscc/radiant/2023.1/bin/lin64
source $bindir/radiant_env

export proj_path=/home/lattice/Documents/prj_demo
cd $proj_path

postsyn -a LIFCL -p LIFCL-40 -t CABGA400 -sp 7_High-Performan

map -i prj_test_syn.udb -pdc impl1.pdc -o prj_test_map.udb -m
```

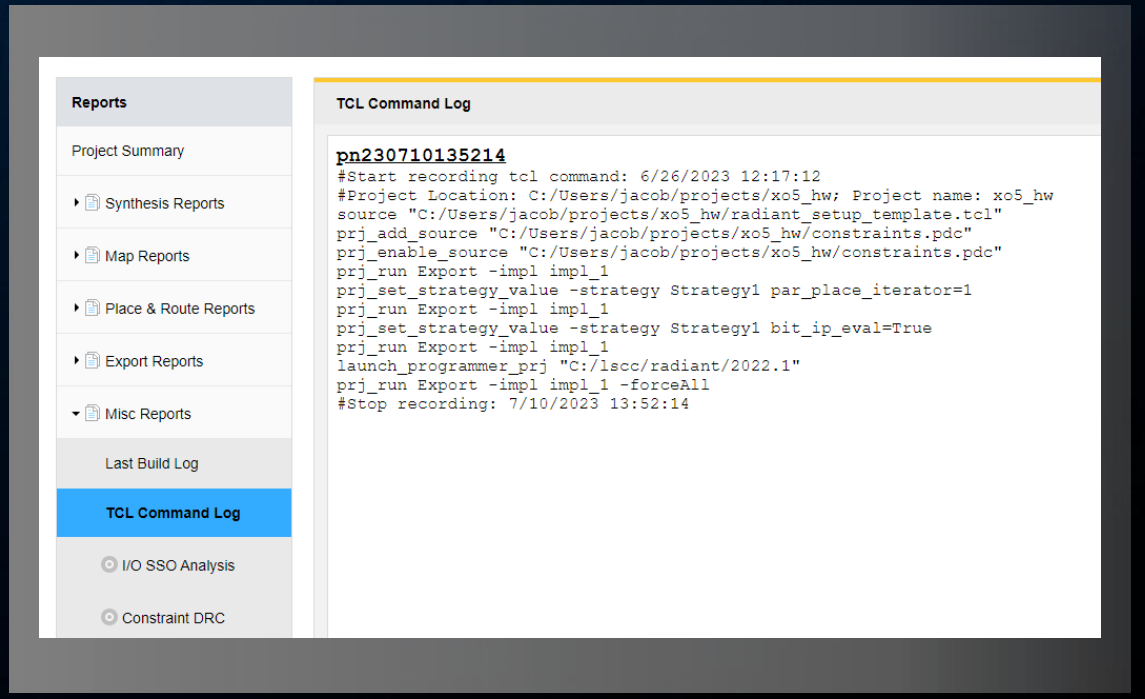
- Setting up the command line for Diamond batch mode
  - Set the **bindir** environment variable
    - export bindir=<Diamond install path>/bin/lin64
  - Invoke Diamond environment setup script
    - source $bindir/diamond_env
  - Begin using Lattice tool commands

```
export bindir=/usr/local/diamond/3.12/bin/lin64
source $bindir/diamond_env

export prj_path=/home/lattice/Desktop/xo2_proj
cd /home/lattice/Desktop/xo2_proj/impl1

synthesis -a "MachXO2" -d LCMXO2-7000HE  \
-optimization_goal timing -s 4 -top HelloWorld_Top \
-ver "$prj_path/xo2_proj/xo2_proj_Top.v"
```

Useful Reports for Creating Scripts

Useful Reports for Creating Scripts

- Radiant
  - Last build log (batch mode commands)
    - Contains console output from last time the project was ran
    - Useful to find how Radiant invokes each specific process (synthesis, map, par, etc.)
      - Use CTRL + F to easily parse through the file
- Radiant & Diamond
  - TCL command log
    - Tracks all TCL commands executed from each build
    - Useful to create a simple script to reproduce GUI functionality



Techniques & Tips - Setting and Using Variables

Setting and Using Variables

- A useful feature of TCL is the ability to set variable names and values
  - Very important for script readability and reusability
  - Enables better customizability of scripts

- To set a variable in TCL, use the set command followed by the value of the variable
  - E.g., “set project_directory /home/usr/jmerc/projects/hw_soc_lifcl”

- To reuse an existing variable after one has been set, simply call the variable by its name with a $ preceding it
  - If multiple variables are called in the same line, use ${} syntax to avoid syntax errors
  - E.g., “prj_project open ${project_directory}/${project_name}.rdf
    - Without ${}, the TCL interpreter would look for a variable called project_directory$project_name.rdf and error out
    - Using ${} tells the interpreter to elaborate both variables and treat the result as a single combined string

Techniques & Tips - Creating and Managing a List of Elements

Creating a List of Elements

- What is a list?
  - A list is an array of elements which can be used for various different purposes
    - Iterate through the list of elements
    - Manage and use contents of list to determine script functionality
  - Contents of a list can be any value type (e.g., integer, float, string, etc.)

- Creating a List of Elements
  - set <list name> {item1 item2 …}
  - set <list name> [list item1 item2 …]

Managing a List of Elements (1/2)

- Once a list of elements has been created, there are a variety of ways which it can be interacted with or used in a TCL scripted flow

- Methods for Managing and Using Lists:
  - lsort <list name>
    - Alphabetically sort list contents
  - lappend <list name> <value> | append <list name> <value>
    - Appends a value to the end of the specified list
  - lindex <list name> <list index>
    - Returns the value of a list at the specified index
    - $<list name>(list index)
      - E.g., $my_list(3)
  - llength <list name>
    - Returns the number of elements in the specified list

Managing a List of Elements (2/2)

- Methods for Managing and Using Lists:
  - linsert <list name> <list index> <value>
    - Insert a new element at the specified index of the list
  - lset <list name> <list index> <value>
    - Set the value of an element at the specified index of a list
  - lreplace <list name> <first index> <last index> <value 1> <value 2> …
    - Replace multiple list items from the specified index range

Techniques & Tips - Controlling Loops

Controlling Loops (1/2)

- Similar to most other programming languages, TCL has a few types of loops which can be used to automatically iterate through certain lists or integer ranges

- For Loop
  - Executes a statement multiple times, updating the loop variable each time
  - Syntax: for {<initialization>} {<condition>} {<increment>} { <statement> };
    - E.g., “for {set i 0} {$i < 5} {incr i} { …”

- Foreach Loop
  - Iterates through all the elements in one or more lists
  - Syntax: foreach <variable> <list name or list contents> { <statement> };
    - E.g., “foreach x $my_list { ...”

```
foreach i $VFILE_LIST {
    if { [catch {prj_add_source $i)} fid] } {
        puts "file already exists in project."
    }
}
```

Controlling Loops (2/2)

- While
  - Executes a statement indefinitely as long as its logical expression is true
  - Syntax: while {<logical expression>} { <statement> };
    - E.g., “while {$x<10} {…”
    - E.g., “while {1} {…”
- Break
  - Causes a break exception to occur when the command is encountered in order to exit a loop
  - Syntax: break
- Continue
  - Causes a continue exception which causes the current iteration of the loop to exit, and continue on to the next iteration
  - Syntax: continue

Techniques & Tips - Accessing Files

Accessing Files (1/2)

- Check the Existence of a File
  - Boolean check to determine if a file or directory exists (returns 1 if True)
  - Syntax: file exists <file name>

- Open an Existing File
  - Opens the specified file with set access permissions and returns the name of the file
  - Syntax: open <file name> [r | w | a]
    - R = read only
    - W = write only
    - A = append only

- Close an Opened File
  - Syntax: close <file name>

Accessing Files (2/2)

- Parse through an Opened File
  - Syntax: read [-nonewline] <file name>
    - Reads all remaining contents from the specified file when used after the **open** command
    - (Optional) -nonewline removes all new line characters (\n) from the file when reading through it
  - Syntax: gets <file name> [<variable name>]
    - Reads the next line from the specified file
    - Stores the value of the next line to a variable if one is set
      - E.g., gets C:/projects/data.txt lines
- Check for the End of an Opened File
  - Boolean check to determine whether the specified file has reached its end of file
  - Syntax: eof <file name>
- Write to an Opened File
  - Writes a string to the end of the specified file
  - Syntax: puts [-nonewline] [file name] <string>
    - -nonewline will omit the new line character (\n) when writing to the file

Techniques & Tips - Calling External Program

Calling External Programs

- Aside from global and tool specific TCL commands, external programs can also directly be called from a TCL script
  - Useful to incorporate non-TCL commands into a TCL scripted flow
  - Expands possibilities for what can be done in a TCL script
    - Can also directly invoke other types of scripts

- To call an external program, use the exec command
  - Exec functions the same between Windows & Linux and is reusable
    - Main difference to keep in mind would be when OS specific commands are used
      - E.g., “exec ls” (Linux) vs “exec cmd /c dir /B” (Windows) to return a list of files in the current directory
  - Use the set command in combination with exec to store console output results from invoking a command
    - E.g., “set results [exec C:/users/usr/programs/python/python.exe C:/projects/scripts/my_script.py]”

Techniques & Tips – Error Handling

Error Handling (1/2)

- Although Lattice tools do not have any built-in error handling TCL commands, TCL has a few native commands which can be used to implement preventative error handling in your own scripts

- Method #1: Catch Errors during Runtime
  - Use the **catch** TCL command to handle errors which occur during a script's runtime
  - Useful in avoiding errors that may occur and cause command abortion
  - If an error is encountered the command will return a non-zero value corresponding to the error code
  - Syntax: catch <statements or script> <variable name>
    - E.g., “catch {gets $my_file} read_error”
      - Error code output from the TCL interpreter is stored in the “read_error” variable

```
foreach i $VFILE_LIST {
    if { [catch {prj_add_source $i)} fid] } {
        puts "file already exists in project."
    }
}
```

Error Handling (2/2)

- Method #2: Preemptively Check the Validity of a Script's Inputs
  - If users know the configurable parts of a script that are key to its operation, then preemptive error handling conditions can be set to prevent it from erroring out
  - Revolves around the use of **if else** statements to provide conditions which ensure that the script is used within its intended parameters
  - Syntax: if {boolean expression #1} {

```
    statements
} elseif {boolean expression #2} {
    statements
} else {
    statements
}
```

The Low Power Programmable Leader