“There’s a better way to do it!”
Simple DC/PT tricks that can change your life.

Paul Zimmer
Zimmer Design Services

Zimmer Design Services
1375 Sun Tree Drive
Roseville, CA 95661

paulzimmer@zimmerdesignservices.com

website: www.zimmerdesignservices.com

ABSTRACT

Spending as much time with DesignCompiler and PrimeTime as I do, every once in a while I stumble onto a tip or trick that makes me think, “Gee, why didn’t I think of that before?”. This paper is a collection of these “headslappers” – simple little tricks that can make your life a whole lot easier.

Table of contents

1 Introduction ... 3
2 Doing get_cells and get_pins in an arbitrary hierarchy ... 4
2.1 &get_cells ... 4
2.2 &get_pins ... 7
3 Parsing Command Output ... 9
3.1 Simple parsing of command output ... 9
3.2 Line-by-line parsing of command output ... 10
3.3 Extracting a subsection of the command output ... 11
3.4 Putting it all together - &master_of ... 16
4 CHIP_INFO, globals, and $:: ... 18
5 puts, echo, and &err ... 20
6 Netlistid ... 22
6.1 Using netlistid for ECOs ... 24
6.2 Using netlistid in a hierarchical layout ... 26
7 The “vim” trick and &uniq ... 28
8 Conclusion ... 32
9 Acknowledgements ... 33
10 References ... 34
11 Appendix ... 35
11.1 &get_cells ... 35
11.2 &get_pins ... 36
11.3 &get_netlistid ... 37
11.4 &netlistid ... 38

1 Introduction

Many of these techniques are the result of a special collaboration with another consultant by the name of Stu Hecht. I had the pleasure of working with Stu on a project a few years back – he was doing synthesis and I was doing static timing. It was an unusual experience for me to be working with another DC/PT expert. The synergy was incredible. I brought in some ideas, Stu brought in some ideas, and together we came up with a number of techniques that I have found invaluable, and that I hope others will as well.

2 Doing get_cells and get_pins in an arbitrary hierarchy

Here's a common problem. Constraints have been created on a fully hierarchical netlist. Then, some downstream tool does some flattening. In the process, it converts "/" hierarchy dividers into, say, "_" characters. So, this path:

I_ORCA_TOP/I_RISC_CORE/I_ALU/U483

Becomes this:

I_ORCA_TOP/I_RISC_CORE_I_ALU_U483

Now all your scripts are broken. You can:

1. Convert all your scripts to the new hierarchy and not have them work on hierarchical netlists ever again.
2. Sprinkle "if" statements or use "get_cells [list <original_path> <new_path>] " all over your code to handle either hierarchy. And pray no one ever changes the hierarchy again.
3. Go see if that job over in Marketing is still open…

It turns out that there is a simple solution to this. You can create a proc "&get_cells" that uses the original, full-hierarchy version of the path and always returns the targeted cell, even if some or all of the path has been flattened.

2.1 &get_cells

The way &get_cells works is simple - it does a "get_cells –hier *", then filters it using a pattern based on the original path using the built-in "-filter" option on get_cells.

Sounds slow, right? It isn't. I've used this on some very large designs, and the difference from normal "get_cells" is almost undetectable. Besides, it's so handy that I'd be willing to put up with some slowdown to make it work.

The key is the filter pattern. The filter pattern replaces all the "/" characters in the original path with a search pattern that will match "/" or "_".

Unfortunately, it isn't quite as simple as it sounds – primarily because the basic –filter pattern matching isn't up to much. It supports only "*" (any characters) and "?" (exactly one character, but any one character). What we want is "[]" like the string matching, so we could say "[/_]" to match exactly one occurance of either "/" or "_", but –filter doesn't have this.

The best we can do is to use "?". So, the command we will execute will look something like this:

```
get_cells * -hier -filter {@full_name =~ I_ORCA_TOP?I_RISC_CORE?I_ALU?U483}
```

This does come with a slight risk that we will match on something we don't want. If there were a component:

I_ORCA_TOP/I_RISC_CORE/I_ALUxU483

we would match (incorrectly) on this as well. In several years of using this technique, I've never seen it, but it is possible.

The only absolutely clean way to do it is to use the –regexp option of get_cells, which does support "[]". In that case, the command will look something like this:

```
get_cells -regexp {.*} -hier \
  -filter {@full_name =~ I_ORCA_TOP[/_]I_RISC_CORE[/_]I_ALU[/_]U483}
```

Unfortunately, my experience has been that the regular expression version really is very slow, so I've used the standard pattern version instead.

You could, I suppose, generate every possible combination of "/" and "_" as a separate pattern, and match on any of them, but this would get messy for realistic chip path lengths, and might not be any faster than regular expressions. I'd rather take the risk of a false match and deal with this one case if it ever occurs.

Assuming we want to use the fast, if not entirely accurate, basic search, we need to substitute all the "/" characters with "?". If the path is in the variable cellpath:

the command looks like this:

```
pt_shell> set cellpath "I_ORCA_TOP/I_RISC_CORE/I_ALU/U483"
I_ORCA_TOP/I_RISC_CORE/I_ALU/U483
```

the command looks like this:

```
pt_shell> set searchpattern [regsub -all {/} $cellpath {?}]
I_ORCA_TOP?I_RISC_CORE?I_ALU?U483
pt_shell> echo $searchpattern
I_ORCA_TOP?I_RISC_CORE?I_ALU?U483
```

The the get_cells call is:

```
pt_shell> get_cells * -quiet -hier -filter "@full_name =~ $searchpattern"
{"I_ORCA_TOP/I_RISC_CORE_I_ALU_U483"}
pt_shell>
```

Of course, we really want this rolled up in a proc.

First we declare the proc and use the built-in parser (for a more powerful parser, see Reference 1):

```
proc &get_cells { args } {

  parse_proc_arguments -args $args result_array
```

My return collection is called filtercells. Initialize it empty:

```
  set filtercells {}
```

The argument to the command is a *list* of paths (like get_cells), so we need to foreach through the list (note: arguments will be defined below using define_proc_attributes):

```
foreach cellpath $result_array(cellpaths) {
```

Now set up the searchpattern as show above:

```
    set searchpattern [regsub -all {/} $cellpath {?}]
```

If given the –hierarchical option, prepend "*" followed by the searchpattern:

```
    if {[info exists result_array(-hierarchical)]} {
      set searchpattern "*?${searchpattern}"
    }
```

Now do the search (for this pattern):

```
   set thispattern_filtercells [get_cells * -quiet -hier -filter "@full_name
=~ $searchpattern"]
```

If the pattern returned nothing (we used –quiet above to suppress the message from get_cells), and our own –quiet option was not set, squawk about it:

```
    if {![info exists result_array(-quiet)] &&
[sizeof_collection $thispattern_filtercells] == 0} {
      &err "Nothing matched for pattern $cellpath"
    }
```

Append it to the filtercells collection, and do the next pattern:

```
    append_to_collection -unique filtercells $thispattern_filtercells
  }
```

Finally, return the full collection:

```
  return $filtercells
}
```

Here’s the define_proc_attributes code:

```
define_proc_attributes &get_cells \
  -info "Does get_cells independent of hierarchy"  \
  -define_args {
      {cellpaths "(Hierarchical) path to cell" cellpaths list required} \
      {-quiet "Don't complain if no match found" "" boolean optional} \
      {-hierarchical "Do the search hier'ly" "" boolean optional} \
}
```

The complete code for &get_cells is in the appendix.

2.2 &get_pins

You might suppose that the &get_pins proc looks similar. It doesn’t. It turns out that searching for pins this way *is* rather slow, and it is actually faster to split the pattern into a cell name and pin name, use &get_cells to find the cell name(s), then find the pins on that cell (or those cells) that match the pin name. It’s harder to describe than it is to do.

The first step is to parse the arguments (using parse_proc_arguments), save away the pinpaths list that was passed, then bundle the rest of the arguments together so we can pass them to &get_cells.

```
proc &get_pins { args } {

  parse_proc_arguments -args $args result_array

  # Surprisingly, it's actually FASTER to use &get_cells to do the
  # search than to do it directly.

  # Capture the pin paths, then delete them from result_array, since
  # result_array will be passed to &get_cells
  set pinpaths $result_array(pinpaths)
  unset result_array(pinpaths)

  # Shut off duplicate options warning so we don't have to remove
  # duplicate options.
  if {[info exists result_array(-quiet)]} {
    suppress_message CMD-018
  }

  # Pass other args to &get_cells command
  set passargs ""
  foreach arg [array names result_array] {
    if {$result_array($arg) == 1} {
      set passargs "$passargs $arg"
    } else {
```

```
      set passargs "$passargs $result_array($arg)"
    }
  }
```

Now the main part of the code loops through the pinpaths as described, splitting them into a cellpath and a pinname, and using &get_cells to get the matching cells. If any were found, it finds the pins and adds them to the collection.

```
  set mypins {}
  # Foreach pinpath, separate into cellpath and pinname.  Then call &get_cells
  # on the cellpath.  Loop through the return from &get_cells and look for
pins
  # on that cell that match pinname.
  foreach pinpath $pinpaths {
    set cellpath [regsub {(.*)/.*$} $pinpath {\1}]
    set pinname [regsub {.*/(.*)$} $pinpath {\1}]
    foreach_in_collection mycell [eval &get_cells -quiet $passargs $cellpath]
{
      if {[sizeof_collection $mycell] > 0} {
        append_to_collection -unique mypins [get_pins -
quiet [get_object_name $mycell]/$pinname]
      }
    }
  }
```

Note the use of “eval” when calling &get_cells. This allows the “$passargs” (the arguments given to &get_pins that will be passed to &get_cells) to be evaluated as arguments.

Check for problems, clean up, and exit:

```
  if {[info exists result_array(-quiet)]} {
    unsuppress_message CMD-018
  }

  if {[info exists result_array(-quiet)] || [sizeof_collection $mypins] > 0} {
    return $mypins
  } else {
    &err "Nothing matched for pins $pinpaths"
  }
}
```

Oh, and here’s the define_proc_attributes:

```
define_proc_attributes &get_pins \
  -info "Does get_pins independent of hierarchy"  \
  -define_args {
      {pinpaths "(Hierarchical) path to pin" pinpaths list required} \
      {-quiet "Don't complain if no match found" "" boolean optional} \
      {-hierarchical "Search hierarchically" "" boolean optional} \
}
```

3 Parsing Command Output

Ever wanted to know the master clock of a generated clock? I have. But there's no attribute for it. One time, as I was lamenting the lack of an attribute, it occurred to me –why not just parse the report_clock output? A few years back, Synopsys added a new option to the command "redirect" called "-variable". This option allows you to redirect command output to a variable, making it possible to easily parse report output without having to resort to temporary files. Let's explore this a little, then get back to our &master_of problem.

3.1 Simple parsing of command output

Let's start with a really simple case. Suppose you'd like to know whether you have the PrimeTime-SI license checked out, then do something in your script based on the result.

The command to see what licenses you have is "list_licenses":

```
pt_shell> list_licenses

Licenses in use:
        PrimeTime (1)
        PrimeTime-SI (1)

pt_shell>
```

We use "redirect –var" to put this into the variable rpt_out:

```
pt_shell> redirect -variable rpt_out {list_licenses}
```

And there it is:

```
pt_shell> echo $rpt_out

Licenses in use:
        PrimeTime (1)
        PrimeTime-SI (1)


pt_shell>
```

Note that the entire report output, newlines and all, is in the variable rpt_out. So, we can just scan the entire variable for the string "PrimeTime-SI":

```
pt_shell> string first PrimeTime-SI $rpt_out
34
pt_shell>
```

I used "string first" because it is a particularly handy way to search for a string in another string. If it returns any value greater than or equal to 0, you have a match. If I try it on a string that isn't in the var, I get -1:

```
pt_shell> string first foobar $rpt_out
-1
pt_shell>
```

The “-1” return means no match.

So, I can code up the check like this:

```
pt_shell> if {[string first PrimeTime-SI $rpt_out] >= 0} {echo "Have PT-SI"}
Have PT-SI
```

If I remove the license and do it over, it comes back blank:

```
pt_shell> remove_license PrimeTime-SI
Checked in license 'PrimeTime-SI'
1
pt_shell> redirect -variable rpt_out {list_licenses}
pt_shell> if {[string first PrimeTime-SI $rpt_out] >= 0} {echo "Have PT-SI"}
pt_shell>
```

All of this can be wrapped in a proc, of course. But the focus for now is on the parsing.

3.2 Line-by-line parsing of command output

In the previous example, we just looked for a string in the entire command output. But what if we want to extract information from a particular line? Look at the output of the report_analysis_coverage command, for example:

```
pt_shell> report_analysis_coverage -nosplit
****************************************
Report : analysis_coverage
Design : dashvar
Version: B-2008.12-SP1
Date   : Sun Jul 19 17:43:16 2009
****************************************

Type of Check         Total            Met         Violated         Untested
------------------------------------------------------------------------------
--
setup                     5       5 (100%)         0 (   0%)        0 (   0%)
hold                      5       4 ( 80%)         1 (  20%)        0 (   0%)
min_pulse_width          10      10 (100%)         0 (   0%)        0 (   0%)
------------------------------------------------------------------------------
--
All Checks               20      19 ( 95%)         1 (   5%)        0 (   0%)

1
```

Want to extract this line

Notice the use of –nosplit! Don’t forget this when parsing reports, or your code may do funny things (wouldn’t it be nice if Synopsys would add a global variable to set nosplit on all commands?)

Suppose we want to extract the "setup" line. One way to do this is to put the command output into a variable, then split it into lines, then foreach through the lines, like this:

```
pt_shell> redirect -variable rpt_out {report_analysis_coverage -nosplit}
pt_shell> foreach line [split $rpt_out \n] {
?   if {[string first setup $line] >= 0} {echo $line}
? }

setup                    5         5 (100%)         0 (   0%)          0 (   0%)
pt_shell>
```

By the way, another way to do this is to use regexp on the entire rpt_out:

```
pt_shell> regexp -linestop {\n(setup.*)\n} $rpt_out all line
1
```

That might require some explanation. We're looking for newline followed by setup followed by any (non-newline) characters followed by newline. The "-linestop" makes ".*" stop at newlines. $rpt_out is the string to search, "all" is the variable to put the whole match into, and "line" is the variable to put the submatch into. The submatch is the part in the parentheses (setup.*). So, line looks like this:

```
pt_shell> echo $line
setup                    5         5 (100%)         0 (   0%)          0 (   0%)
pt_shell>
```

But using "foreach" over the lines comes in handy for more complicated cases, as we shall see.

3.3 Extracting a subsection of the command output

Here's another handy trick. Look at the output of report_clock:

```
pt_shell> report_clock -nosplit
****************************************
Report : clock
Design : dashvar
Version: B-2008.12-SP1
Date   : Sun Jul 19 18:02:47 2009
****************************************


Attributes:
    p - Propagated clock
    G - Generated  clock
    I - Inactive   clock

Clock          Period   Waveform            Attrs     Sources
--------------------------------------------------------------------------------
clkin           10.00   {0 5}               p         {clkin}
clkout          20.00   {0 10}              p, G      {clkout}
div2clk         20.00   {0 10}              p, G      {div2_reg/Q}
div4clk         40.00   {0 20}              p, G      {div4_reg/Q}


Generated     Master          Generated       Master          Waveform
Clock         Source          Source          Clock           Modification
--------------------------------------------------------------------------------
clkout        div2_reg/Q      clkout          div2clk         div(1), combinational
div2clk       clkin           div2_reg/Q      clkin           div(2)
div4clk       div2_reg/Q      div4_reg/Q      div2clk         div(2)
1
pt_shell>
```

Want to extract this section

Denote start and end of section

It's a long report with a couple of sections. What if we want to grab data from the second section (and then maybe go line-by-line through this)? One way would be to go line-by-line through the report and set flags when we enter and exit the section, but there's an easier way.

First we stick the report output into our rpt_out variable as usual:

```
pt_shell> redirect -variable rpt_out {report_clock -nosplit}
```

Now we use regexp to find the second section. How? By looking for the start string (Generated Master) and the end string (\n1\n) – the circled parts above.

```
pt_shell> regexp {(Generated *Master.*\n1\n)} $rpt_out matched
1
```

Which sets “matched” to:

```
pt_shell> echo $matched
Generated     Master          Generated       Master          Waveform
Clock         Source          Source          Clock           Modification
------------------------------------------------------------------------------
-
clkout        div2_reg/Q      clkout          div2clk         div(1),
combinational
div2clk       clkin           div2_reg/Q      clkin           div(2)
div4clk       div2_reg/Q      div4_reg/Q      div2clk         div(2)
1
```

If we’d wanted the first section, it would look like this:

regexp {(Clock *Period.*\n\n)} $rpt_out matched

Notice the difference:

regexp {(Generated *Master.*\n1\n)} $rpt_out matched
regexp {(Clock *Period.*\n\n)} $rpt_out matched

Normally, we’d look for “\n\n” (2 line feeds) as the end of the block. But since PT echos the “1” at the end, the *last* block has to look for “\n1\n”.

What would we do without regular expressions?

Since this is a frequent occurance, the best way to do this is to change the regular expression to match whether or not the “1” is there. This can be done by adding “?” after the “1”. This is regular expression parlance for “match 0 or 1 of the previous”:

```
pt_shell> regexp {(Generated *Master.*\n1?\n)} $rpt_out matched
1
pt_shell> echo $matched
Generated     Master          Generated       Master          Waveform
Clock         Source          Source          Clock           Modification
------------------------------------------------------------------------------
-
clkout        div2_reg/Q      clkout          div2clk         div(1),
combinational
div2clk       clkin           div2_reg/Q      clkin           div(2)
div4clk       div2_reg/Q      div4_reg/Q      div2clk         div(2)
1
```

One more subtlety – look at what regexp actually captures when we modify the search for the first section in this way:

```
pt_shell> regexp {(Clock *Period.*\n1?\n)} $rpt_out matched
1
pt_shell> echo $matched
Clock          Period   Waveform           Attrs     Sources
```

```
--------------------------------------------------------------------------------
-
clkin           10.00   {0 5}              p         {clkin}
clkout          20.00   {0 10}             p, G      {clkout}
div2clk         20.00   {0 10}             p, G      {div2_reg/Q}
div4clk         40.00   {0 20}             p, G      {div4_reg/Q}


Generated     Master          Generated       Master          Waveform
Clock         Source          Source          Clock           Modification
--------------------------------------------------------------------------------
-
clkout        div2_reg/Q      clkout          div2clk         div(1),
combinational
div2clk       clkin           div2_reg/Q      clkin           div(2)
div4clk       div2_reg/Q      div4_reg/Q      div2clk         div(2)
1

pt_shell>
```

The problem here is that regexp matching is, by default, "greedy". That means it will match as large a section of text as possible. We can make it not-greedy using the following regular expression magic incantation:

```
pt_shell> regexp {(Clock *Period.*\n1?\n){1,1}?} $rpt_out matched
1
pt_shell> echo $matched
Clock          Period   Waveform            Attrs     Sources
--------------------------------------------------------------------------------
-
clkin           10.00   {0 5}              p         {clkin}
clkout          20.00   {0 10}             p, G      {clkout}
div2clk         20.00   {0 10}             p, G      {div2_reg/Q}
div4clk         40.00   {0 20}             p, G      {div4_reg/Q}


pt_shell>
```

Back to the original problem. We want the Generated/Master (second) section:

```
pt_shell> regexp {(Generated *Master.*\n1?\n){1,1}?} $rpt_out matched
1
pt_shell> echo $matched
Generated     Master          Generated       Master          Waveform
Clock         Source          Source          Clock           Modification
--------------------------------------------------------------------------------
-
clkout        div2_reg/Q      clkout          div2clk         div(1),
combinational
div2clk       clkin           div2_reg/Q      clkin           div(2)
div4clk       div2_reg/Q      div4_reg/Q      div2clk         div(2)
1

pt_shell>
```

But we're going to loop over the actual entries, and we don't really want the "---" and the header line stuff. So, we apply the same trick again:

```
pt_shell> regexp -- {-\n(.*)\n1?\n} $matched all justmylines
1
pt_shell> echo $justmylines
clkout         div2_reg/Q      clkout          div2clk         div(1),
combinational
div2clk        clkin           div2_reg/Q      clkin           div(2)
div4clk        div2_reg/Q      div4_reg/Q      div2clk         div(2)
pt_shell>
```

More regular expression magic. I'm scanning the $matched string for "-" followed by anything followed by "\n1?\n", and keeping the "anything" in a new variable called justmylines.

One more thing to fix before looping over the lines. That "combinational" is really part of the "div(1)" field, so let's get rid of the space:

```
pt_shell> regsub -all {, comb} $justmylines {,comb} justmylines
1
pt_shell>
pt_shell> echo $justmylines
clkout         div2_reg/Q      clkout          div2clk
div(1),combinational
div2clk        clkin           div2_reg/Q      clkin           div(2)
div4clk        div2_reg/Q      div4_reg/Q      div2clk         div(2)
pt_shell>
```

OK, now we can use our split and loop trick to look at each line in turn:

```
foreach line [split $justmylines \n] {
  ….
}
```

But what should we do to each line? We could parse it using a regexp, something like {\s+(\S+)…}, but it's handier to split the line into words. Unfortunately, unlike perl, tcl split doesn't allow you to split on multi-char sequences (like "any number of whitespace characters"), but we can still use split if we first turn all the whitespace characters into single spaces, like this:

```
pt_shell> set words [split [regsub -all {\s+} $line { }] { }]
clkout div2_reg/Q clkout div2clk div(1),combinational
```

Now, the words array contains our entries. For example, words[3] contains "div2clk":

```
pt_shell> lindex $words 3
div2clk
pt_shell>
```

So, if we put the generated clock name in the variable genclk_name, our loop looks like this:

```
pt_shell> foreach line [split $justmylines \n] {
?   set words [split [regsub -all {\s+} $line { }] { }]
```

```
?   if {[string match [lindex $words 0] $genclk_name]} {echo "$genclk_name :
[lindex $words 3]"}
? }
div2clk : clkin
```

3.4 Putting it all together - &master_of

Wrap it all up in a proc, and you have &master_of!

```
########################################################################
#
# &master_of -- return the master of a generated clock
#
# usage: &master_of genclk_name
#

proc &master_of { args } {

  parse_proc_arguments -args $args result_array

  # Get report_clock output
  redirect -variable rpt_out {report_clock -nosplit}
  # Grab just the Generated/Master section of the report
  regexp {(Master *Waveform.*\n1?\n){1,1}?} $rpt_out matched
  # Grab just the clock lines themselves
  regexp -- {-\n(.*)\n1?\n} $matched all justmylines
  # pushd ,comb into previous div
  regsub -all {, comb} $justmylines {,comb} justmylines
  # Loop through the lines
  foreach line [split $justmylines \n] {
    # split words into array
    set words [split [regsub -all {\s+} $line { }] { }]
    # If entry 0 matches, return master value (entry 3)
    if {[string match [lindex $words 0] $result_array(genclk_name)]} {
      return [lindex $words 3]
    }
  }
}

#
# Define the arguments, command information and usage
#
define_proc_attributes &master_of \
  -info "Returns master of a generated clock"   \
  -define_args {
      {genclk_name "Name of generated clock" genclk_name string required} \
}
```

Here's &master_of in action. You can, of course, add options to return period, waveform, etc.

```
pt_shell> &master_of clkout
div2clk
```

```
pt_shell> &master_of div2clk
clkin
pt_shell> &master_of div4clk
div2clk
pt_shell>
```

4 CHIP_INFO, globals, and $::

If you're like me, you tend to create a lot of variables and flags as your script grows. After a while, it becomes difficult to remember all of them – particularly if you're sharing the script with someone else. Using wildcards in printvar helps, but is there perhaps a better way?

Why not keep all your variables in a single array? I call this array $CHIP_INFO, but you might want to call is something shorter and simpler, like $c (or $m for you magma people!). That way, you can easily see how all your flags are set:

```
pt_shell> parray CHIP_INFO
CHIP_INFO(CORNER)      = wccom
CHIP_INFO(HAS_TEST)    = 0
CHIP_INFO(HOLDMARGIN)  = 0.01
CHIP_INFO(MODE)        = MISSION
CHIP_INFO(NETLISTID)   = 20090817_1119_00
CHIP_INFO(SETUPMARGIN) = 0.10
pt_shell>
```

Simple, but effective.

There's another reason you might want to keep your flags and variables in a single array – accessing them in a proc.

Variable names in tcl procs are local by default. To use a global variable, you have to use the "global" command:

```
proc &netlistid { args } {
  global CORNER
  global MODE
  ...
```

And then you come along later and try to access, say, "HAS_TEST", and you get an error…

But if you put all of them in a single array, you just globalize the array and get access to all the variables:

```
proc &netlistid { args } {
  global CHIP_INFO

  if {$CHIP_INFO(HAS_TEST)} {
  ...
```

Incidentally, there is another way to access global variables from within a proc – the "::" method. If you want to access a top-level variable inside a proc, you can do this:

```
proc &netlistid { args } {
```

```
  if {$::HAS_TEST} {
  ...
```

This also works for CHIP_INFO as well:

```
proc &netlistid { args } {

  if {$::CHIP_INFO(HAS_TEST)} {
  ...
```

5 puts, echo, and &err

Here's a quicky. I often see things like this in PT/DC scripts:

```
if {...} {
  echo "Error: <some error>"
```

The problem with this is that when someone does "source –echo" of this script, the "echo…" line will show up in the log file. Then, when I grep for "Error:", I get the darn "echo …" line. I could change my grep to look for "^Error:', but that's a hassle…

There's a better way to do it!

```
echo [format "Err%s" "or: $message"]
```

I generally bury this in a proc called &err:

```
proc &err args {
  set message [regsub -all {[\{\}]} $args {}] ;# get rid of random {} stuff
  echo [format "Err%s" "or: $message"]
}
```

But that's not the only reason to send error messages from a proc. There's also the issue of redirection.

If you were to use the above &err proc in a script, then redirect the script output, the error message would go to the redirected output. Fine, you say. Well, maybe. As a consultant, this is not always what I want. My procs might be used anywhere, and when they emit error messages, I want to be sure the messages are seen. But in many client flows, script output is redirected to log files all over the place. If I don't know where those log files are, and what they're called, the error message might get lost. And the chip might fail timing and I don't know it. This is bad.

It's also bad when you're working interactively and source a script with redirection. Do you really want to grep all the log files involved every time you source the script? Of course not, so you don't bother – then you waste hours debugging something because you didn't see the error message. Wouldn't it be nice if you could be sure the error would come to the console?

Another problem with this setup is that I have to check all of these log files every time I run the script interactively to make sure it ran cleanly.

But I need the error message to go to the redirect target for sure – that's where the context is.

What I want is for the message to go to both places – the redirect target *and* stdout (which means the console when I'm running interactively and the top-level log file when I'm running batch). That way it is available in its normal context, but I can't miss it because it's in the top-level log file, too.

In older versions of DC/PT tcl, the "puts" command would ignore redirect, always sending its output to stdout. So, my &err proc had both an echo and a puts.

Ideally, the proc could sense its redirection and only emit 2 messages when required. But I haven't found a way to do this. So, &err emits 2 messages all the time. Annoying, but it's the price I pay for being absolutely sure that no error messages get lost.

Then, starting in about 2008, Synopsys "fixed" the puts command to obey redirection. Uh-oh. But, fortunately, they gave us a switch to enable the old behavior - sh_enable_stdout_redirect. When set to "false" (it defaults to "true"), puts will ignore redirection and send its output to stdout.

So, here's what the consultant version of &err looks like:

```
proc &err args {
  if {[info exists ::sh_enable_stdout_redirect]} {
    set sh_enable_stdout_redirect_save $::sh_enable_stdout_redirect
    set $::sh_enable_stdout_redirect false
  }

  set message [regsub -all {[\{\}]} $args {}]
  echo [format "Err%s" "or: $message (echo)"]
  puts stdout [format "Err%s" "or: $message (puts)"]

  if {[info exists ::sh_enable_stdout_redirect]} {
    redirect /dev/null {set ::sh_enable_stdout_redirect $sh_enable_stdout_redirect_save}
  }
}
```

Does both puts and echo

6 Netlistid

Suppose you've got clocks created on a particular flop. At some point (maybe at your request!), the RTL changes such that the flop names change, or the clock moves.

If you're like me, you use a common script for constraints in all phases – synthesis, layout, and STA. Maybe you run the script in all tools, with appropriate checks of $synopsys_program_name, or maybe you always run in PT and just write out the constraints. It doesn't matter. The point is, you now need to synthesize with new RTL and use the new flop names.

Except you already sent the previous version to layout, and the results won't be back for a couple of weeks.

So, you need one create_clock for the old design, and a different one for the new one. What now?

You could handle this with revision control. Except that 90% of the stuff you're currently fixing in the script applies to the old netlist as well. So then you've got to do a bunch of merging and remember which bits are common and which not. *And* you've probably got more than one netlist oustanding in layout. Yech.

Another way around this is to do something like:

```
if {[sizeof_collection [&get_cells <...new_cell_path...>]] > 0} {
  ...new code...
} else {
  ...old code...
}
```

But it gets REALLY tedious writing and debugging these, and it is very difficult to read.

There's a better way – netlistid.

The idea is (painfully) simple. During synthesis, you create (or rename) a cell to indicate the revision id of the netlist. This is much better than, say, writing the creation date as a comment into the netlist, since it stays with the netlist throughout the flow – provided you do the equivalent of a dont_touch in all the tools.

You can then write a little proc to find this cell and return the netlistid:

```
proc &get_netlistid { args } {

  set nlid [get_object_name [get_cells -quiet u_DT_NETLISTID_*]]

  return [regsub {.*u_DT_NETLISTID_} $nlid {}]
```

```
}

#
# Define the arguments, command information and usage
#
define_proc_attributes &get_netlistid \
  -info "Gets the netlist id based on the dt cell.  Returns null string if not
found"   \
  -define_args {
}
```

and your code now looks like this:

```
if {[&get_netlistid] == <…old_netlistid...>} {
  ...old code...
} else {
  ...new code...
}
```

Actually, I use something a little more complicated, as we shall see, but that's the basic idea.

What you use for a netlistid is up to you, but I have a few suggestions:

1. Prefix the cell name with whatever your group uses to indicate "don't_touch" cells (say, "u_DT") and the string NETLISTID (like, "u_DT_NETLISTID").
2. Use the creation time of the netlist as a unique identifier. The "clock" command is a handy way to do this:
   pt_shell> clock format [clock seconds] -format {%Y%m%d_%H%M}
   returns:
   20090817_1119
3. Add a suffix to indicate eco level. More on this later. Adding "_00" will allow for 99 eco's on one netlist – which seems like enough. Ahem.

If you follow these suggestions, you end up with a cell in the design named "u_DT_NETLISTID_20090817_1119_00".

To apply this name, you can either instantiate a new cell, or rename an existing one. The choice is yours.

You'll want to do some sort of comparison on this netlistid, so best to remove the "u_DT_NETLISTID_" part while extracting it:

```
pt_shell> set nlid [get_object_name [get_cells -quiet u_DT_NETLISTID_*]]
u_DT_NETLISTID_20090817_1119_00
pt_shell> regsub {.*u_DT_NETLISTID_} $nlid {} nlid
1
pt_shell> echo $nlid
20090817_1119_00
```

Actually, you'll want to wrap this in a proc. This proc will be examined in more detail when we discuss hierarchical netlistid's. For now, just assume we have a proc called &get_netlistid that does the above and returns the result.

Now, we can do stuff like:

```
if {[&get_netlistid] == 20090817_1119_00} {
```

or this:

```
if {[&get_netlistid] > 20090817_1119_00} {
```

But I prefer to do the comparison *itself* in a proc, because it allows me to do stuff like this:

```
  if {[info exists results(-ge)]} {
    if {$nlid >= $results(-ge)} {
      return 1
    } else {
      &err "[&myname] returned ge false - probably using code for an old
netlist"
      return 0
    }
  }
```

So, if I do:

```
if {[&netlistid -ge 20090817_1119_00]} {
```

and it returns 0 (the netlist is NOT greater than or equal to 20090817_1119_00), I'll get an error message indicating that the code took a branch associated with an old netlist.

6.1 Using netlistid for ECOs

This netlistid trick is very handy for managing multiple netlists using the same constraint script. But it's even better for managing ECOs.

Here's the scenario:

You synthesize the "final" netlist (No, really. Stop laughing. I'm serious. Oh, get off the floor. It's not that funny. No, really, I'm serious, this is the final one. RTL is frozen, right?…).

Then the inevitable RTL change comes in. Some director somewhere has been told synthesis is complete, so you're not allowed to re-synthesize (no matter *how* much faster or easier that would be). So, you implement the change with connect_net, disconnect_net, etc and send the "write_changes" or "write_astro_changes" file to layout. This is ECO1.

While the design is in layout, 2 more “small” RTL changes come in. You implement these as ECO2, and release a new write_changes file to layout. Except the ECO1 layout is still going. You don’t want to hold up your timing info over a silly RTL fix, right? So you tell them to start a new layout by applying the ECO2 changes to the ECO1 netlist.

Now the ECO1 netlist comes back. You find some timing fixes you want to do using ECO commands (resize_cell, etc). This is ECO3. But you’ll need to do these on the ECO2 netlist when it comes back.

You send the ECO3 netlist mods to layout (using “write_changes”, or “write_astro_changes”), because you want to get those fixes in so you can run hold checks and add more timing ECOs.

Then the ECO2 layout data comes back.

Oh, and someone wants to run back-annotated gate sims. And they need the latest RTL. And the timing fixes (or at least an approximation of them).

How do you manage all this mess?

Netlistid can help. Every time you do a set of netlist edits, bump the netlistid value (remember the _00 at the end?). Use PT editing commands to do the edits. Put these commands in a block of code that only runs when the edited netlistid matches the current netlistid. And include the command to bump the netlistid. It looks like this:

```
if {[&netlistid -eq 20090817_1119_00]} {
  rename_cell 20090817_1119_00 20090817_1119_01
  # ECO1 edits...
  connect_net ...
  disconnect_net ...
  create_cell ...
  size_cell ...
}
if {[&netlistid -eq 20090817_1119_01]} {
  rename_cell 20090817_1119_01 20090817_1119_02
  # ECO2 edits...
  connect_net ...
  disconnect_net ...
  create_cell ...
  size_cell ...
}
if {[&netlistid -eq 20090817_1119_02]} {
  rename_cell 20090817_1119_02 20090817_1119_03
  # ECO3 edits...
  connect_net ...
  disconnect_net ...
  create_cell ...
  size_cell ...
}
```

Whatever netlist you load will automatically pick up at the right place and do the PT editing commands necessary to bring it up to the latest revision. Why? Because it's existing netlistid indicates what eco's have already been done. The ECO1 netlist loads up, skips the "-eq …_00" part (the first "if"), then proceeds from the second. The ECO2 netlist starts with the third "if", and so on.

Consider how this interacts with layout. Recall that the ECO1 netlist went to layout, then ECO2 edits were created. These commands (using write_changes or write_astro_changes) were sent to layout along with the original ECO1 netlist (since you can't write the netlist out of PT). When the ECO1 netlist comes back from layout, it has the netlistid _01. The code runs and automatically adds the ECO2 and ECO3 edits.

Now the ECO2 netlist comes back from layout. Since the edit commands included a "rename_cell" to make the netlistid _02, the _02 edits are skipped and only the new _03 edits are applied.

If you stick to monotonically increasing ECOs, this system will allow you to load any netlist and have PT edit it to bring it up-to-date, without attempting to apply edits that are already present in that netlist.

6.2 Using netlistid in a hierarchical layout

With a few minor enhancements, the netlistid trick works on hierarchical designs as well. Each block gets its own netlistid, and its own "eco block" (the sequence of "if" statements shown above).

The &get_netlistid proc gets a new option "-path" that tells it to look only down a particular hierarchial path. Specify a block's path and you get that block's netlistid. Similarly, the &netlistid proc (that does the ==, >=, etc checks) gets a "-use" option that tells it what netlistid value to use instead of doing &get_netlistid itself. This is usually the output of a [&get_netlistid –path …] command.

The full procs are in the appendix. I want to focus here on the usage.

You end up with commands that look like this:

```
if {[&netlistid -use [&get_netlistid -path core/block1] -eq 20090818_1320_00]}
{
```

It doesn't have to be that hardcoded, of course. The real command might look like this:

```
if {[&netlistid -use [&get_netlistid -path ${CPU_HIER}] -eq $nlid_cpu_eco0]} {
```

which is probably easier to read.

This netlistid technique has proven invaluable to me in managing multiple outstanding netlists.

7 The “vim” trick and &uniq

Here’s the scenario. You’re working interactively in DC/PT, and you run some report. Then you run some more commands, reports etc. Then you need to refer to the first report output again. So you use the scroll bar, or put “screen” into edit mode, or you move around in emacs, or whatver, and go back. Then you scroll back to use this info in a command, then you have to look at the report again, etc. After a while you get tired of this, dump the report output into a temp file, then open the temp file in your favorite editor.

There is a better way to do it! We can automate that process.

To give credit where credit is due, I got the idea for this trick from a Synopsys training class. But the implementation is entirely my own.

You create a proc (“vim”) that will take the report (or any command) output, and put it in a new window so you can see it and access it without messing around with your DC/PT window.

The basic idea is:

1. Dump the report output to a temporary file.
   I’ve tried to avoid the need for a temporary file, but haven’t found a way around it yet. If you figure it out, please send me an email!
2. Use “exec” to start vim (or some other editor/viewer) on this file, making sure that control returns to DC/PT while the editor continues to run.
3. When the editor is terminated, remove the temporary file.

Dumping the report output to a temporary file is easy enough – use redirect. But we need a unique temporary file name. The original Synopsys implementation used random numbers, but, “there is a better way!”.

We can use “pid” to generate a reasonably unique id for this DC/PT session. True, sessions on other machines sharing the same directory might not be unique, but this is a pretty unlikely scenario, especially if we tack on something that makes it unique within the session. We certainly need a unique identifier within the session, since we might run several of these at once. But what can we use to generate a unique number within this session?

Give up? The “history” index! It increments with every command, and every time we execute this proc it will be a new command. So, it is guaranteed to be unique!

What do I mean by the history index? It’s the number prefix output by history:

```
pt_shell> history
     1  source test_netlistid.tcl
     2  report_design
     3  source get_netlistid.tcl
     4  &get_netlistid
     5  history
pt_shell>
```

If we do “history –r 1”, we will get the current command, and its index:

```
pt_shell> history -r 1
     6  history -r 1
pt_shell>
```

Do a little regexp magic, and extract the index:

```
pt_shell> regexp {^\s+(\d+)} [history -r 1] junk uniqid
1
pt_shell> echo $uniqid
9
```

Add in a prefix (tmp4vim), the output of pid (process id), and a filetype suffix (.ptlog), and we get our unique file name:

```
pt_shell> set tmpfile tmp4vim[pid]${uniqid}.ptlog
tmp4vim292219.ptlog
pt_shell>
```

Put some sample output in tmpfile (this will be more generic in the proc),

```
pt_shell> redirect $tmpfile {report_design}
```

And run gvim:

```
pt_shell> exec gvim $tmpfile
```

Up pops our window!

But there’s a fly in the ointment. Out DC/PT prompt is now hanging – waiting for gvim to exit. We can fix this using “&” to run gvim in the background.

```
pt_shell> exec gvim $tmpfile &
29292
pt_shell>
```

This works, but leaves us with the messy pid (29292) of the gvim process as our return value. So, use redirect to get rid of it. Throw in a “catch” to ignore errors.

```
pt_shell> redirect /dev/null {catch {exec gvim $tmpfile &}}
pt_shell>
```

This works, but how do we get rid of the temporary file when gvim exits? We need the command to be a sequence of commands: “gvim $tmpfile ; rm $tmpfile”. Except that gvim, by default, will fork. So, we can use –nofork: “gvim –nofork $tempfile; rm $tmpfile”. Except that sometimes screws up, with rm complaining the file isn’t there. I think gvim does something on exit that does

not always get completed before rm runs. I have found that adding a "sleep 1" in the sequence generally does the trick: "gvim –nofork $tempfile; sleep 1; rm $tmpfile".

But we need this whole sequence to run as one exec command, and in the background. After much messing around, I found that the easiest way to do this is to use "/bin/csh –c". So, the command looks like this:

```
pt_shell> redirect /dev/null {catch {exec /bin/csh -c "gvim --nofork $tmpfile
;sleep 1; rm $tmpfile" &}}
pt_shell>
```

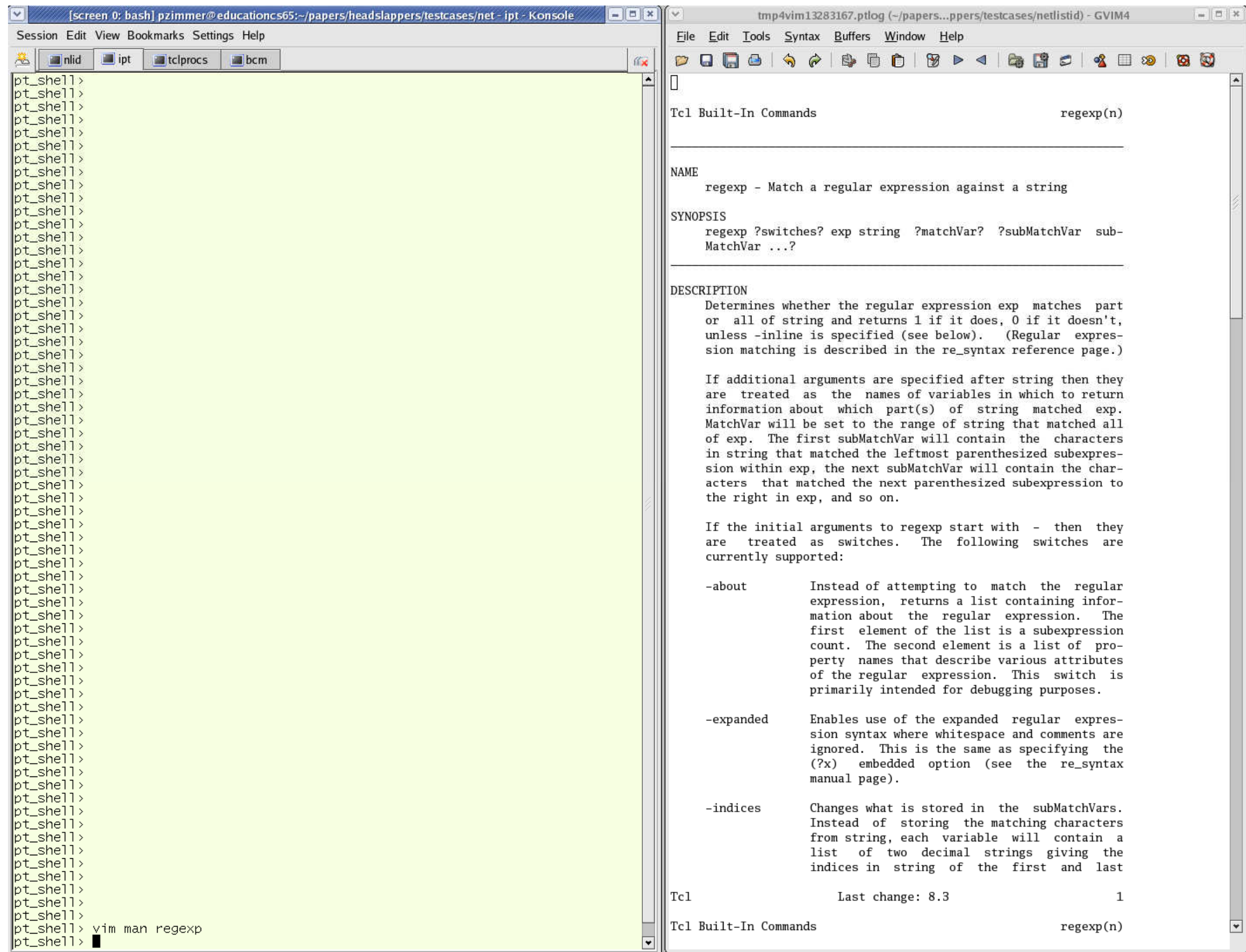
Now, we really want this in a proc whose argument is the entire command line to be run. To do this, we use "proc vim {args} {…. redirect $tmpfile {uplevel $args} …}.

Here's the complete vim proc:

```
proc vim {args} {
  # Make a unique id by using pid and current history index
  regexp {^\s+(\d+)} [history -r 1] junk uniqid
  set tmpfile tmp4vim[pid]${uniqid}.ptlog

  redirect $tmpfile {uplevel $args}
  # Without redirect, exec echos the PID of the new process to the screen
  redirect /dev/null {catch {exec /bin/csh -c "gvim --nofork $tmpfile ; sleep
1; rm $tmpfile" &}}
}
```

vim is pretty handy – especially for “man” commands. Here’s what it looks like running “man regexp”:



While we’re at it, that uniq number trick is pretty neat, too. So, we’ll make an &uniq proc:

```
proc &uniq { } {
  # Make a unique id by using pid and current history index
  regexp {^\s+(\d+)} [history -r 1] junk uniqid
  return [pid]${uniqid}
}
```

Simple, but very handy.

8 Conclusion

We have seen a number of handy tricks for DC and PT. I hope you find them as useful as I have!

9 Acknowledgements

The authors would like to acknowledge the following individuals for their assistance:

Stuart Hecht, SJH Clear *Consulting* LLC for helping develop many of the ideas shown.

Brian Kane, Northrop Grumman Corp for excellent and detailed review of this paper and the related presentation

10 References

(1) **My Favorite DC/PT Tcl Tricks**
Paul Zimmer
Synopsys Users Group 2003 San Jose
(available at www.zimmerdesignservices.com)

11 Appendix

11.1 &get_cells

```
#########################################################################
#
# &get_cells -- get_cells, independent of hierarchy
#
# usage: &get_cells hierarchical_path_to_cell
#

proc &get_cells { args } {

  parse_proc_arguments -args $args result_array

  set filtercells {}
  foreach cellpath $result_array(cellpaths) {
    set searchpattern [regsub -all {/} $cellpath {?}]
    if {[info exists result_array(-hierarchical)]} {
      set searchpattern "*?${searchpattern}"
    }
    # Tell the tool -quiet, and check the result so we can report
    # which pattern failed.
    set thispattern_filtercells [get_cells * -quiet -hier -filter "@full_name
=~ $searchpattern"]

    if {![info exists result_array(-quiet)] &&
[sizeof_collection $thispattern_filtercells] == 0} {
      &err "Nothing matched for pattern $cellpath"
    }
    append_to_collection -unique filtercells $thispattern_filtercells
  }

  return $filtercells

}

#
# Define the arguments, command information and usage
#
define_proc_attributes &get_cells \
  -info "Does get_cells independent of hierarchy"  \
  -define_args {
      {cellpaths "(Hierarchical) path to cell" cellpaths list required} \
      {-quiet "Don't complain if no match found" "" boolean optional} \
      {-hierarchical "Do the search hier'ly" "" boolean optional} \
}
```

11.2 &get_pins

```
###########################################################################
#
# &get_pins -- get_pins, independent of hierarchy
#
# usage: &get_pins hierarchical_path_to_pin
#

proc &get_pins { args } {

  parse_proc_arguments -args $args result_array

  # Surprisingly, it's actually FASTER to use &get_cells to do the
  # search than to do it directly.

  # Capture the pin paths, then delete them from result_array, since
  # result_array will be passed to &get_cells.
  set pinpaths $result_array(pinpaths)
  unset result_array(pinpaths)

  # Shut off duplicate options warning so we don't have to remove
  # duplicate options.
  if {[info exists result_array(-quiet)]} {
    suppress_message CMD-018
  }

  # Pass other args to &get_cells command
  set passargs ""
  foreach arg [array names result_array] {
    if {$result_array($arg) == 1} {
      set passargs "$passargs $arg"
    } else {
      set passargs "$passargs $result_array($arg)"
    }
  }

  set mypins {}
  # Foreach pinpath, separate into cellpath and pinname.  Then call &get_cells
  # on the cellpath.  Loop through the return from &get_cells and look for
pins
  # on that cell that match pinname.
  foreach pinpath $pinpaths {
    set cellpath [regsub {(.*)/.*$} $pinpath {\1}]
    set pinname [regsub {.*/(.*)$} $pinpath {\1}]
    foreach_in_collection mycell [eval &get_cells -quiet $passargs $cellpath]
{
      if {[sizeof_collection $mycell] > 0} {
        append_to_collection -unique mypins [get_pins -
quiet [get_object_name $mycell]/$pinname]
      }
    }
  }

  if {[info exists result_array(-quiet)]} {
    unsuppress_message CMD-018
```

```
  }

  if {[info exists result_array(-quiet)] || [sizeof_collection $mypins] > 0} {
    return $mypins
  } else {
    &err "Nothing matched for pins $pinpaths"
  }
}

#
# Define the arguments, command information and usage
#
define_proc_attributes &get_pins \
  -info "Does get_pins independent of hierarchy"  \
  -define_args {
      {pinpaths "(Hierarchical) path to pin" pinpaths list required} \
      {-quiet "Don't complain if no match found" "" boolean optional} \
      {-hierarchical "Search hierarchically" "" boolean optional} \
}
```

11.3 &get_netlistid

```
########################################################################
#
# &get_netlistid -- Get netlist id
#
# usage: &get_netlistid
#

proc &get_netlistid { args } {

  parse_proc_arguments -args $args results

  if {[info exists results(-path)]} {
    set path $results(-path)
    set nlid [get_object_name [&get_cells -quiet ${path}*u_DT_NETLISTID_*]]
  } else {
    set nlid [get_object_name [get_cells -quiet u_DT_NETLISTID_*]]
  }

  return [regsub {.*u_DT_NETLISTID_} $nlid {}]


}

#
# Define the arguments, command information and usage
#
define_proc_attributes &get_netlistid \
  -info "Gets the netlist id based on the dt cell.  Returns null string if not
found"  \
```

```
  -define_args {
      {-path "Hier path to search if not top" path  string optional} \
}
```

11.4 &netlistid

```
##########################################################################
#
# &netlistid -- Check netlist id
#
# usage: &netlistid -ge <num>
#        &netlistid -le <num>
#        &netlistid -gt <num>
#        &netlistid -lt <num>
#        &netlistid -eq <num>
#

proc &netlistid { args } {

  global CHIP_INFO

  # Default options
  set results(-use) [&get_netlistid]

  parse_proc_arguments -args $args results

  # Shorthand
  set nlid $results(-use)
  unset results(-use)

  if {[array size results] != 1} {
    &err "Must specify exactly one of:
    -gt, -ge, -lt, -le, -eq, -notyet
    "
    return
  }

  if {[info exists results(-ge)]} {
    if {$nlid >= $results(-ge)} {
      return 1
    } else {
      &err "[&myname] returned ge false - probably using code for an old
netlist"
      return 0
    }
  } elseif {[info exists results(-gt)]} {
    if {$nlid > $results(-gt)} {
      return 1
    } else {
      &err "[&myname] returned gt false - probably using code for an old
netlist"
      return 0
    }
```

```
  } elseif {[info exists results(-le)]} {
    if {$nlid <= $results(-le)} {
      return 1
      &err "[&myname] returned le true - probably using code for an old
netlist"
    } else {
      return 0
    }
  } elseif {[info exists results(-lt)]} {
    if {$nlid < $results(-lt)} {
      return 1
      &err "[&myname] returned lt true - probably using code for an old
netlist"
    } else {
      return 0
    }
  } elseif {[info exists results(-eq)]} {
    if {$nlid == $results(-eq)} {
      return 1
    } else {
      return 0
    }
  } elseif {[info exists results(-notyet)]} {
    &err "Hit netlistid -notyet - message \"$results(-notyet)\""
    return 0
  }

}

#
# Define the arguments, command information and usage
#
define_proc_attributes &netlistid \
  -info "Compares the current value of CHIP_INFO(NETLISTID) to some other
value"  \
  -define_args {
      {-ge "greater than or equal to" ge  string optional} \
      {-gt "greater than" gt  string optional} \
      {-le "less than or equal to" le  string optional} \
      {-lt "less than" lt  string optional} \
      {-eq "equal to" eq  string optional} \
      {-notyet "Never match, but issue &err.  Value is message to issue with
&err" notyet  string optional} \
      {-use "Value to use instead of CHIP_INFO(NETLISTID)" use  string
optional} \
}
```