FusionShell User Guide

Version 2025.12

DashThru

Copyright Notice

Trademarks

Disclaimer

DashThru Technology makes no representations or warranties about the accuracy, reliability, completeness, or timeliness of the content provided by FusionShell. All content is provided "as is" without any express or implied warranties.

Contents

1

1 FusionShell Overview

FusionShell is a command-line user interface (CLI) developed by DashThru Technology. It serves as the standard interface for users operating DashThru's EDA products. FusionShell supports both Tcl and Python modes, accommodating traditional Tcl scripting preferences while simultaneously providing users with a highly open Python platform.

Users can switch between FusionShell's Tcl and Python modes freely and in real-time using specific commands. Consequently, users can choose a workflow that best suits their needs: utilizing exclusively Tcl mode, exclusively Python mode, or a hybrid combination of both.

In Tcl mode, FusionShell extends and enhances the native Tcl language with multiple advanced features to meet the development requirements of different user groups. In Python mode, users can work in the same way as in standard Python environments, while also leveraging a wide range of third-party Python packages to assist script development.

For more information about FusionShell features, please refer to the following sections:

- FusionShell Tcl Mode
- FusionShell Python Mode
- FusionShell Hybrid Mode

1-1 Starting and Exiting FusionShell

When launching FusionShell, users can specify whether to enter interactive mode or batch mode, and whether to start in Tcl mode or Python mode. In interactive mode, commands are entered and executed manually by the user. In batch mode, FusionShell automatically executes the specified script file.

Starting FusionShell in Interactive Mode

- **Starting Interactive Tcl Mode**

To start interactive Tcl mode, invoke the tool executable without any options. In the following examples, `<tool_exec_file>` represents the executable name of the tool, such as `dashrtl`.

```
% <tool_exec_file>
```

- **Starting Interactive Python Mode**

To start interactive Python mode, invoke the tool executable with the `-pymode` option.

```
% <tool_exec_file> -pymode
```

Starting FusionShell in Batch Mode

- **Starting Batch Tcl Mode**

To start batch Tcl mode, invoke the tool executable with the `-files` option to specify the Tcl script file path.

```
% <tool_exec_file> -files user.tcl
```

- **Starting Batch Python Mode**

To start batch Python mode, invoke the tool executable with both the `-files` option to specify the Python script file and the `-pymode` option.

```
% <tool_exec_file> -files user.py -pymode
```

Exiting FusionShell

- **Exiting FusionShell in Tcl Mode**

In Tcl mode, use the standard Tcl commands `exit` or `quit` to terminate

```
tool-tcl> exit
```

- **Exiting FusionShell in Tcl Mode**

In Python mode, use the Python built-in function `exit()` to terminate FusionShell.

```
tool-py> exit()
```

1-2 Switching Between Tcl Mode and Python Mode

FusionShell allows users to switch seamlessly between Tcl mode and Python mode at runtime without exiting the shell. After a mode switch command is issued, the command prompt changes accordingly, making it easy to identify the current operating mode.

Identifying the Current Command-Line Mode

FusionShell distinguishes Tcl mode and Python mode by the command-line prompt. A prompt ending with `-tcl` indicates Tcl mode. A prompt ending with `-py` indicates Python mode.

In the following prompt examples, `tool` represents the name of the DashThru's EDA tool, such as `dashrtl`.

- **Tcl Mode Prompt**

```
tool-tcl>
```

- **Python Mode Prompt**

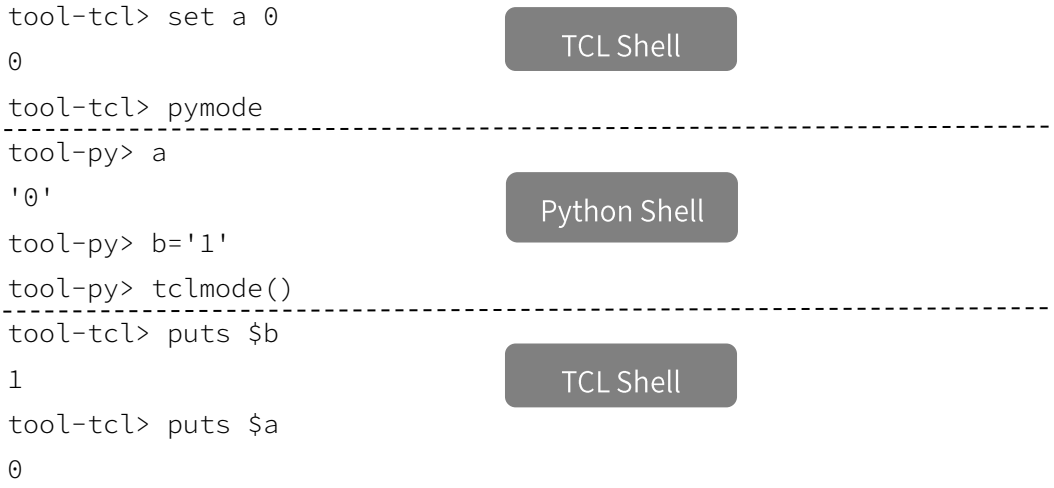
```
tool-py>
```

Switching Between Command-Line Modes

FusionShell supports dynamic switching between Tcl mode and Python mode without restarting the tool. From Tcl mode, use the `pymode` command to switch to Python mode. From Python mode, call the `tclmode()` function to switch back to Tcl mode.

The following example demonstrates switching from Tcl mode to Python mode and then back to Tcl mode.

After executing `pymode`, the prompt changes to `tool-py`, indicating Python mode. Similarly, after executing `tclmode()`, the prompt changes to `tool-tcl`, indicating Tcl mode.



Sharing Variables and Commands Between Modes

Variables and commands are shared between Tcl mode and Python mode in FusionShell. Variables and commands defined in Tcl mode can be accessed and used in Python mode. Variables and commands defined in Python mode can likewise be accessed and used in Tcl mode.

This shared execution environment allows users to switch freely between Tcl and Python while continuing to use previously defined variables and commands.

For more details on using shared variables and commands across modes, refer to 4 FusionShell Hybrid Mode.

2

2 FusionShell Tcl Mode

FusionShell Tcl mode is developed based on Tcl version 8.5. In addition to supporting the standard Tcl language, FusionShell provides multiple functional extensions and enhancements tailored for EDA workflows.

For more information about features available in FusionShell Tcl mode, refer to the following sections:

2-1 Tcl Language Overview

Tcl, short for Tool Command Language, is an interpreted scripting language created by John Ousterhout in 1988. It was originally designed as a glue language to connect different software components. Tcl can be used as an interactive shell, a standalone script interpreter, or embedded into applications as an extension language.

Tcl supports variables, procedures, and control structures, and provides a powerful built-in core command set. These commands can be used to perform a wide range of operations, including file manipulation, string processing, and mathematical computation.

Tcl uses several special characters to represent specific syntax and language constructs. Commonly used special characters include:

$ Used to reference variables in Tcl scripts.

() Used to group expressions, typically in mathematical expressions or command evaluation.

[] Used to perform command substitution. Commands enclosed in square brackets are evaluated first, and their results are substituted into the surrounding command.

\ Used for escaping characters and performing character substitution within strings.

"" Text enclosed in double quotes is treated as a string, with variable and command substitution applied.

{} Text enclosed in braces is treated as a literal string, and no variable or command substitution is performed.

* Wildcard character that matches any string, including the empty string.

? Wildcard character that matches any single character.

; Command separator. Used to separate multiple commands on the same line.

Comment character. Any line beginning with this character is treated as a comment and is not executed.

2-2 Extended Tcl Commands for Object Handling

In standard Tcl, data is typically managed using lists. However, Tcl lists are essentially string-based lists, which can be inefficient when handling the large volumes of data commonly encountered in chip design workflows.

To address these limitations, FusionShell extends Tcl by introducing a set of built-in object commands. These commands operate on an internal object-based data model specifically designed to represent design data efficiently. Common object types include modules, cells, pins, ports, and parameters.

Compared with native Tcl lists, the FusionShell object-based data model provides the following advantages:

- **Type Awareness**

Tcl lists can only store string data, whereas FusionShell objects represent strongly typed design entities, including both primitive types and reference types.

- **Richer Functionality**

Tcl lists provide only basic operations such as insertion, deletion, and element access. In contrast, built-in object commands support advanced operations such as filtering, searching, sorting, and comparison.

- **Improved Performance**

Operations on Tcl lists often require full list traversal, which can be inefficient for large datasets. FusionShell objects are backed by optimized internal data structures and algorithms, enabling more efficient processing of large-scale design data.

For complex design analysis and manipulation tasks, the object-based commands provide significantly better performance and flexibility than native Tcl lists.

Creating and Storing Object Handles

FusionShell operates on objects through object handles, similar to how the `open` command returns a file descriptor. The `get_objects` command returns a handle to the specified objects. Object handles are represented as strings prefixed with `_obj`.

In the following example, the `get_objects -type cell` command creates an object handle for all cells matching `*reg*`. The command prints the names of the matched objects and returns an object handle `_obj1`.

The `get_objects -type port` command creates a handle for the port named `clk`, returning a new object handle `_obj2`.

```
tool-tcl> get_objects -type cell *reg*
out_reg[0] out_reg[1] out_reg[2]
_obj1
tool-tcl> get_objects -type port clk
clk
_obj2
```

In the following example, an object handle can be stored in a variable by assigning the return value of `get_objects`. Printing the variable directly displays the handle identifier rather than the object names. To retrieve the object name list, use the `get_object_name` command.

```
tool-tcl> set reg_cell [get_objects -type cell *reg*]
_obj1
tool-tcl> puts $reg_cell
_obj1
tool-tcl> get_object_name $reg_cell
out_reg[0] out_reg[1] out_reg[2]
```

Operating on Object Handles

Operations such as adding, removing, and comparing objects are performed using FusionShell built-in object commands.

These commands typically follow the naming convention `*_objects`, such as `add_to_objects` and `remove_from_objects`.

- **add_to_objects**

Adds existing object elements to a new or existing object handle. In the following example, `add_to_objects` creates a new handle containing all elements from `reg_cell` plus `data_reg`. The resulting handle is stored in the variable `all_reg`.

```
tool-tcl> set all_reg [add_to_objects $reg_cell [get_objects -
type cell data_reg]]
_obj3
```

- **compare_objects**

Compares two object handles. In the following example, if both handles reference exactly the same set of objects, the command returns `0`; otherwise, it returns `1`.

By default, object order is ignored. To perform an order-sensitive comparison, use the `-order_dependent` option.

```
tool-tcl> compare_objects [get_objects -type port *] $clk_port
1
tool-tcl> compare_objects [get_objects -type port *] [get_objects
-type port *]
0
```

- **filter_objects**

Filters objects from an existing handle based on a specified filter expression and returns a new object handle. In the following example, cells whose names start with `u` are selected.

```
tool-tcl> filter_objects -print [get_objects -type cell *] "name
=~ u*"
u_9 u_10 u_11
_obj2
```

- **foreach_in_objects**

Iterates over all objects referenced by an object handle. Its usage is similar to the Tcl `foreach` command. In the following example, all port objects are iterated over, and the name of each port is printed.

```
tool-tcl> foreach_in_objects port [get_objects -type port *] {
puts [get_object_name $port]}
in[0]
in[1]
.....
```

- **index_objects**

Retrieves one or more elements from an existing object handle by index, similar to the Tcl commands `lindex` and `lrange`.

In the following example, the first command retrieves the third element from the port objects. The second command retrieves the elements at indices 4, 5, and 6.

```
tool-tcl> index_objects -print [get_objects -type port *] 3
in[3]
_obj2.2
tool-tcl> index_objects -print [get_objects -type port *] 4 6
in[4] in[5] in[6]
_obj4
```

- **remove_from_objects**

Removes specified elements from an existing object handle and returns a new object handle containing the remaining elements. In the following example, the `clk` port is removed from the port objects, and the resulting object handle is stored in the variable `port`.

```
tool-tcl> set port [remove_from_objects [get_objects -type port
*] {clk}]
_obj2
```

In addition to specifying object names directly, `remove_from_objects` also supports removing elements by providing another object handle. In the following example, a handle for the `clk` port is first created and then used to remove that port from the full port collection.

```
tool-tcl> set clk_port [get_objects -type port clk]
_obj1
tool-tcl> set port [remove_from_object [get_objects -type port *]
$clk_port]
_obj3
```

- **sizeof_objects**

Returns the number of elements referenced by the specified object handle. In the following example, the total number of port objects is printed.

```
tool-tcl> sizeof_objects [get_objects -type port *]
36
```

- **sort_objects**

Sorts elements in an object handle based on a specified property. In the following example, objects are sorted by the `name` property using ascending order,

descending order, and dictionary order, respectively.

```
tool-tcl> sort_objects -print [get_objects -type cell] name
u_10 u_11 u_9
_obj2
tool-tcl> sort_objects -print -descending [get_objects -type
cell] name
u_9 u_11 u_10
_obj4
tool-tcl> sort_objects -print -dictionary [get_objects -type
cell] name
u_9 u_10 u_11
_obj6
```

2-3 Extended Tcl Commands for Procedure Handling

In Tcl, the `proc` command is used to create user-defined commands. In the following example, `proc` creates a new command named `add`, which can be invoked in the same way as any other Tcl command.

When the `add` command is invoked, the arguments `1` and `2` are passed to the local variables `x` and `y`, respectively, and the addition operation is performed.

```
tool-tcl> proc add {x y} {puts [expr $x+$y]}
tool-tcl> add 1 2
3
```

When invoking a procedure, the number of arguments provided must match the number of parameters defined in the `proc` declaration. However, if the parameter name is specified as `args`, the procedure can accept a variable number of arguments.

In the following example, the `add` procedure is defined with `args` and is invoked with three arguments.

```
tool-tcl> proc add {args} {puts $args}
tool-tcl> add 1 2 3
1 2 3
```

Option-Based Procedure Definition

Most built-in Tcl commands and tool-specific commands support option switches, which are typically specified using the following syntax:

```
tool-tcl> command_name -option_name option_value
```

When defining a user procedure that supports option switches, implementing such functionality using native Tcl constructs is complex and limited. The primary limitations include:

- Option switches may be specified in arbitrary order, whereas traditional `proc` definitions require fixed positional arguments.

- Option switches may be specified multiple times, which cannot be properly handled by standard positional argument parsing.

- Option values cannot be validated, such as enforcing data types or value ranges.

To address these limitations, FusionShell provides two built-in commands — `parse_proc_args` and `define_proc_constraints` — to enable robust definition and parsing of option-based procedures.

Procedure Definition Using parse_proc_args and define_proc_constraints

For procedures that support option switches, the `parse_proc_args` command must be invoked within the procedure body to parse the `args` argument and assign option values to variables or arrays.

The `define_proc_constraints` command is used to define the syntax and constraints of option switches, including option names, value types, default values, and usage rules. Since `parse_proc_args` relies on these constraints to perform validation and argument assignment, both commands must be used together.

- **parse_proc_args -to_vars**

The following example illustrates the use of `parse_proc_args -to_vars`. In this example, `define_proc_constraints` defines two option switches, `-string` and `-int`, with value types `string` and `int`, respectively.

The `-string` option is mandatory, while the `-int` option has a default value of `1`. A positional (non-switch) argument named `bool` is also defined.

When the `-to_vars` option is used, local variables named `string`, `int`, and `bool` are automatically created within the procedure body and populated with the parsed values.

```
proc print {args} {
    parse_proc_args -to_vars
    puts $string
    puts $int
    puts $bool
}
```

```
define_proc_constraints print \
-info "puts all arguments" \
-default_arg {names=bool} \
-switch_arg "name=string type=string optional=false" \
-switch_arg "name=int type=int default=1"
```

After the procedure is defined, the command usage can be queried using the `-help` option:

```
tool-tcl> print -help
Usage: print  # puts all arguments
      -string <string>
      [-int <int>]
      [<bool>]
```

If a mandatory option switch is omitted, an error is reported:

```
tool-tcl> print -int 3
Error: wrong # args: '-string' must be specified for procedure
'print'
```

Option value type checking is enforced. Since the `-int` option is defined as an integer, specifying a non-integer value results in an error:

```
tool-tcl> print -string haha -int 3.0
Error: wrong # args: expect integer value for '-int' of procedure
'print' but get '3.0'
```

Option switches and positional arguments may be specified in any order. In the following examples, the positional argument value `true` is assigned to `bool`:

```
tool-tcl> print true -string haha -int 3
haha
3
true
tool-tcl> print -int 3 -string haha true
haha
3
true
```

- **parse_proc_args -to_array**

The `parse_proc_args -to_array` option provides functionality similar to `-to_vars`, except that parsed values are stored in an array instead of individual variables.

In the following example, all option and positional argument values are stored in the array `value`, indexed by option or argument name.

```
proc print {args} {
    parse_proc_args -to_array value
    puts $value(-string)
    puts $value(-int)
    puts $value(bool)
}

define_proc_constraints print \
-info "puts all arguments" \
-default_arg {names=bool} \
-switch_arg "name=string type=string optional=false" \
-switch_arg "name=int type=int default=1"
```

How to use define_proc_constraints

The `define_proc_constraints` command specifies the syntax rules and constraints for an option-based procedure. The available options are listed below. Note that the `-reset` option cannot be combined with any other options.

define_proc_constraints

[-reset]

Reset all constraints associated with the procedure.

[-info <string>]

Specify the `-help` descriptive information for the procedure.

[-switch_arg <subswitch_list>]+

Define option switches.

name=<string>

Option switch name.

[optional=<bool>]
Specify whether the option is optional.
[default=<string>]
Specify the default value.
[enum=<list_of_string>]
Specify a list of allowed values.
[type=<string|string|list|bool|null|int|float>]
Specify the value type.
[min=<int>/<float>]
Specify the minimum value for int or float types.
[max=<int>/<float>]
Specify the maximum value for int or float types.
[multi=<bool>]
Specify whether the option may be specified multiple times.
[exclude_all=<bool>]
Specify whether the option is mutually exclusive with other options.

[-default_arg <subswitch_list>]
Define positional (non-switch) arguments.
names=<list_of_string>
Specify argument names.
[optional=<bool>]
Specify whether the arguments are optional.
[enum=<list_of_string>]
Specify a list of allowed values.
[type=<string|string|list|int|float>]
Specify the argument type.
[min=<int>/<float>]
Specify the minimum value for int or float types.
[max=<int>/<float>]
Specify the maximum value for int or float types.

[-enable_redirect]
Enable output redirection using '>' and '>>'.

[-must_include_any <list_of_string>]+
Require that at least one option from the specified list be used.

[-exclusive <list_of_string>]+
Define mutual exclusion rules between option switches.

Enhanced Procedure Flow Control

When executing a long-running procedure, it is often necessary to guarantee that certain user-defined operations are performed when the procedure returns. Typical use cases include printing summary reports, releasing resources, or cleaning up the execution environment.

FusionShell provides the built-in `defer` command to support exit-time execution control within a procedure.

Using `defer`, users can register commands that are executed automatically when the procedure terminates. Two execution modes are supported:

1. Unconditional execution on procedure exit
2. Execution only when interrupted by Ctrl+C

- **Unconditional Execution on Procedure Exit**

In the following example, a procedure performs a long-running loop operation. The `defer` command is used to register a command that prints the value of a variable when the procedure exits.

This behavior is functionally similar to a Tcl8.6 `try … finally …` construct with additional support for Ctrl+C exit: the deferred command is always executed when the procedure terminates, independent of the exit path.

defer.tcl

```
proc incr_loop {} {
   defer {puts "i = $i when exiting proc"}

   set i 0
   while 1 {
      incr i
      # if {$i == 5000} {puts $ii}
      if {$i == 10000} {return $i}
   }
}

set var [incr_loop]
```

Normal Exit: When the procedure exits normally after reaching the termination condition, the deferred command is executed before returning the value.

```
tool-tcl> source defer.tcl
i = 10000 when exiting proc
10000
tool-tcl> puts $var
10000
```

Ctrl+C Interruption: If the procedure execution is interrupted by pressing Ctrl+C, the procedure exits immediately and the deferred command is still executed. However, since the procedure does not complete normally, subsequent `set var` command is not executed.

```
tool-tcl> source defer.tcl
^C i = 3072 when exiting proc
Info: Ctrl-C interrupt: 'source defer.tcl'. (SHELL-INTRCMD)
tool-tcl> puts $var
Error: can't read "var": no such variable
```

Runtime Error Exit: If a runtime error occurs during procedure execution triggered by `if {$i == 5000} {puts $ii}`, the deferred command is executed before the error is reported.

```
tool-tcl> source defer.tcl
i = 5000 when exiting proc
Error: can't read "ii": no such variable
tool-tcl> puts $var
Error: can't read "var": no such variable
```

- **Execution Only on Ctrl+C Interruption**

In some scenarios, users may want to execute cleanup or recovery logic only when a procedure is interrupted, but not on normal completion.

For this purpose, FusionShell provides the `defer -intr` option. In the following example, the deferred command is executed only when the procedure is interrupted by Ctrl+C.

defer.tcl

```
proc incr_loop {} {
   defer -intr {puts "i = $i when exiting proc"; return $i;}

   set i 0
   while 1 {
      incr i
      if {$i == 10000} {return $i}
   }
}

set var [incr_loop]
```

Normal Exit: When the procedure completes normally, the deferred command is not executed.

```
tool-tcl> source defer.tcl
10000
tool-tcl> puts $var
10000
```

Ctrl+C Interruption with Recovery: When the procedure is interrupted by Ctrl+C, the deferred command is executed, and control is returned to the script. This allows the script to continue execution instead of terminating immediately. As a result, `set var` is executed successfully after the interruption.

```
tool-tcl> source defer.tcl
^C Info: Ctrl-C interrupt is handled by 'defer -intr' command.
(SHELL-INTRRECOVER)
i = 4095 when exiting proc
4095
tool-tcl> puts $var
4095
```

In this mode, `defer -intr` effectively allows the procedure to intercept and recover from Ctrl+C interruptions, enabling customized interrupt handling and improving script robustness.

For detailed information about Ctrl+C interrupt behavior, refer to the section Ctrl+C Interrupt Handling.

2-4 Debugging Enhancements

To support advanced debugging requirements for user scripts, FusionShell extends native Tcl debugging functionality with the following enhancements:

Ctrl+C Interrupt Handling

FusionShell extends Ctrl+C interrupt handling to allow real-time termination of command execution. Interrupt handling is supported for:

- Native Tcl commands
- User-defined procedures
- Loop constructs such as `foreach`, `for`, and `while`

Pressing Ctrl+C during loop execution immediately terminates the loop:

```
tool-tcl> while 1 {}
^C
Info: Ctrl-C interrupt: 'while 1 {}'. (SHELL-INTRCMD)
```

Ctrl+C can also interrupt long-running Tcl commands:

```
tool-tcl> after 100000000
^C
Info: Ctrl-C interrupt: 'after 100000000'. (SHELL-INTRCMD)
```

Pressing Ctrl+C twice while in command input mode exits FusionShell:

```
tool-tcl>
Info: one more Ctrl-C to exit. (SHELL-INTR)
tool-tcl>
Info: thank you for using ……
```

When entering multi-line command mode using `{}`, Ctrl+C cancels the current command input:

```
tool-tcl> if {$a == 0} {
...
Info: Ctrl-C interrupt: 'if {$a == 0} {'. (SHELL-INTRCMD)
tool-tcl>
```

Enhanced Error Reporting

Script execution errors can be diagnosed using the `error_info` command, which reports detailed error location information.

The following example shows error reporting for an error occurring in the interactive shell. The `error_info` command reports the shell line number where the error occurred.

```
tool-tcl> set a
Error: can't read "a": no such variable
tool-tcl> error_info
.................
----------------------------------------------------------------
<shell> line 1:
  set a
```

When executing scripts using the `source` command, `error_info` reports both:

- The shell line that invoked `source`
- The line number within the script file where the error occurred

```
tool-tcl> source user.tcl
Error: can't read "a": no such variable
tool-tcl> error_info
.................
----------------------------------------------------------------
<shell> line 1:
  source user.tcl
"user.tcl", line 1:
  set a
```

Command Tracing

In native Tcl behavior, the `source` command prints only the command output results, not the commands themselves. This behavior makes script debugging more difficult.

FusionShell enhances the `source` command by adding the `-echo` option. When this option is specified, both the executed commands and their outputs are displayed. Commands are prefixed with `<tcl>` to indicate that they represent command statements.

Example

Assume the script file `user.tcl` contains the following commands:

```
set a 0
set b 1
```

Native Tcl behavior:

```
tool-tcl> source user.tcl
0
1
```

FusionShell enhanced behavior:

```
tool-tcl> source -echo user.tcl
<tcl> set a 0
0
<tcl> set b 1
1
```

OS Command Execution

FusionShell supports executing OS commands using two methods.

- **Executing OS Commands Using exec**

OS commands can be executed using the Tcl `exec` command:

```
tool-tcl> exec gvim user.tcl
tool-tcl> exec mkdir aa
```

The `exec` command supports different execution modes. When `-shellmode` is specified, shell wildcard expansion is supported:

```
tool-tcl> exec -shellmode ls *.log
```

When `-tty` is specified, the command runs in an interactive terminal mode. All user inputs are processed in real time and are not recorded in the log. This mode is typically used for text-based interactive applications such as editors:

```
tool-tcl> exec -tty vim
```

- **Executing OS Commands Directly**

FusionShell also supports direct execution of OS commands from the command line, provided the command is accessible in the system environment (i.e., discoverable via `which`):

```
tool-tcl> mkdir aa
tool-tcl> ls
aa
tool-tcl> cd aa
```

2-5 Filelist Enhancements: flist Files

In EDA tools, filelists are commonly used to describe HDL source files. Such filelists typically contain HDL file paths, compilation options, and comments.

FusionShell extends the conventional filelist concept and introduces an enhanced list file format named `flist`. While `flist` reuses the general syntax structure of traditional filelists, it can be applied to any command or variable configuration, rather than being limited to HDL compilation.

An `flist` file is a special file type in FusionShell. In practical usage, users frequently encounter scenarios where lists become excessively long, making scripts difficult to read and maintain. Typical examples include:

- HDL file lists passed to `analyze`
- Library file lists passed to `read_libs`
- Path lists specified by `set search_path`

All such lists can be placed into an `flist` file and converted into a standard Tcl list using the `read_flist` command, which can then be directly passed to commands or variables.

- **flist Syntax Rules**

Elements in an `flist` file may be separated by spaces or newlines. For example:

```
-verilog top.v sub.v
```

is equivalent to:

```
-verilog
top.v
sub.v
```

To use an `flist` file, the `read_flist` command must be invoked. The command expands the *flist* content into a regular list, which can be directly embedded into commands or variable assignments.

The `flist` mechanism supports nested `flist` inclusion, standard comments and special comments.

- **Standard Comment in flist file**

The following standard comment styles are supported:
Single-line comments: #, //
Multi-line comments: /* ... */

Example:

rtl1.list

```
# This is a comment
-f rtl2.list
// This is a comment
/design/rtl/top.v //This is a comment
/design/rtl/sub.v /* This is a comment*/
```

rtl2.list

```
/* This is a comment line1
   This is a comment line2
*/
/design/rtl/define.v
```

Using the command:

```
analyze -hdl_type verilog [read_flist rtl1.list]
```

is equivalent to executing:

```
analyze -hdl_type verilog "/design/rtl/define.v /design/rtl/top.v
/design/rtl/sub.v"
```

- **Special Comment in flist file**

FusionShell introduces a special comment syntax that allows commented content to be conditionally included during `flist` expansion. A special comment must follow the format:

```
// dashthru <flist_items>
```

Although `<flist_items>` appear within a comment, `read_flist` recognizes the keyword `dashthru` and includes the commented elements in the expanded list. This mechanism enables flexible modification of filelist behavior without altering the original list structure, ensuring compatibility with third-party tools.

Example:

old.lst

```
/design/rtl/define.v
/design/rtl/top.v
/design/rtl/sub.v
```

new.lst

```
/design/rtl/define.v
// dashthru +define+MACRO
/design/rtl/top.v
/design/rtl/sub.v
// dashthru /design/rtl/mod.v
```

The modified filelist `new.lst` remains fully compatible with third-party tools and behaves identically to the original list `old.lst`. However, in FusionShell, executing:

```
analyze -hdl_type verilog [read_flist new.list]
```

is equivalent to:

```
analyze -hdl_type verilog "/design/rtl/define.v +define+MACRO
/design/rtl/top.v /design/rtl/sub.v /design/rtl/mod.v"
```

- **Typical Usage Scenarios**

The `flist` mechanism allows users to externalize long command arguments or variable values into files, significantly improving script readability and maintainability.

The following sections demonstrate four typical usage scenarios:

Using flist to Specify Path Lists
Using flist to Specify HDL Filelists
Using flist to Specify Library Filelists

Using flist to Specify Path Lists

The `flist` mechanism can be used to define and extend the `search_path` variable in a structured and maintainable way. By externalizing path definitions into `flist` files, long and frequently modified path lists can be managed more cleanly.

// Search Path: search.lst
 	
$search_path
/design/rtl
/design/rtl/top
// dashthru -f libsearch.lst
	// Search Path: libsearch.lst
	
	/design/lib/mem
	/design/lib/pad

In the script, using the following command:

```
set search_path [read_flist search.lst]
```

is equivalent to executing:

```
set search_path "$search_path /design/rtl design/rtl/top
/design/lib/mem /design/lib/pad"
```

A regular Tcl list and an `flist` expansion can be freely mixed within the same variable assignment. For example:

```
set search_path "$search_path /design/rtl design/rtl/top
[read_flist libsearch.lst]"
```

This is equivalent to:

```
set search_path "$search_path /design/rtl design/rtl/top
/design/lib/mem /design/lib/pad"
```

Using flist to Specify HDL Filelists

The `flist` mechanism can be used to describe complex HDL filelists, including compiler directives, include paths, and mixed-language source files. This approach significantly improves the readability and maintainability of HDL compilation scripts, especially for large designs.

```
// RTL1 List: rtl1.lst

// dashthru +define+SIM
+incdir+/design/rtl
/design/rtl/top.v
/design/rtl/sub_top.v
-f rtl2.lst
```

```
// RTL2 List: rtl2.lst

-verilog /design/rtl/sub1.v
-sverilog /design/rtl/sub2.sv
```

In FusionShell, executing:

```
analyze -hdl_type verilog [read_flist rtl1.lst]
```

is equivalent to executing:

```
analyze -hdl_type verilog "+define+SIM +incdir+/design/rtl
/design/rtl/top.v /design/rtl/sub_top.v -verilog
/design/rtl/sub1.v -sverilog /design/rtl/sub2.sv"
```

Regular Tcl lists and `flist` expansions can be freely combined. For example:

```
analyze -hdl_type verilog "+incdir+/design [read_flist rtl1.lst]
/design/rtl/sub3.sv"
```

This is equivalent to:

```
analyze -hdl_type verilog "+incdir+/design +define+SIM
+incdir+/design/rtl /design/rtl/top.v /design/rtl/sub_top.v -
verilog /design/rtl/sub1.v -sverilog /design/rtl/sub2.sv
/design/rtl/sub3.sv"
```

When multiple `flist` files are required, they can be specified simultaneously in a single `read_flist` invocation. For example:

```
analyze -hdl_type verilog [read_flist rtl1.lst rtl2.lst]
```

Using flist to Specify Library Filelists

The `flist` mechanism can also be applied to library filelists used by the `read_libs` command. By externalizing library paths into `flist` files, users can manage large and frequently changing library sets in a modular and reusable manner.

```
// RTL1 List: lib1.lst

/design/lib/pad.lib
// dashthru /design/lib/pll.lib
-f lib2.lst
```

```
// RTL2 List: lib2.lst

/design/lib/mem32x32.lib
/design/lib/mem32x64.lib
```

In FusionShell, executing:

```
read_libs [read_flist rtl1.lst]
```

is equivalent to executing:

```
read_libs /design/lib/pad.lib /design/lib/pll.lib
/design/lib/mem32x32.lib /design/lib/mem32x64.lib
```

Regular Tcl lists and `flist` expansions can be freely combined. For example:

```
read_libs "[read_flist lib1.lst] /design/lib/mem64x64.lib"
```

This is equivalent to:

```
read_libs "/design/lib/pad.lib /design/lib/pll.lib
/design/lib/mem32x32.lib /design/lib/mem32x64.lib
/design/lib/mem64x64.lib"
```

When multiple `flist` files are required, they can be specified together in a single `read_flist` command. For example:

```
read_libs [read_flist lib1.lst lib2.lst]
```

2-6 Tab Key Enhancements

In conventional Tcl command-line environments, the Tab key is typically limited to command name completion and listing, and cannot be applied to subcommands or argument values.

FusionShell extends the Tab key functionality to provide comprehensive completion and listing support not only for commands, but also for subcommands, variables, command options, and valid option values. This enhancement enables more efficient and flexible command-line interaction.

Command Completion and Listing

When an incomplete command is entered at the command prompt and the Tab key is pressed, FusionShell attempts to automatically complete the command name. If multiple commands match the current input, FusionShell displays a list of all commands that begin with the specified prefix.

Command completion and listing apply to both native Tcl commands and user-defined procedures.

In the example below, entering `ge` and pressing Tab does not result in a unique match, since multiple commands start with `get`. FusionShell therefore completes the input to `get`. Pressing Tab again displays all matching commands.

```
tool-tcl> ge<tab>
tool-tcl> get<tab>
get_cells       get_functions    get_modules     get_nets ……
```

Command completion and listing are also supported within command substitutions `[]`. As shown below, Tab can be used in the same way inside brackets.

```
tool-tcl> puts [ge<tab>
tool-tcl> puts [get<tab>
get_cells       get_functions    get_modules     get_nets ……
```

Subcommand Completion and Listing

Commands with subcommands (such as `string` and `file`) also support Tab-based completion and listing for their subcommands. The behavior is identical to that of command completion and is supported both at the command prompt and within command substitutions.

Examples of subcommand completion at the command prompt and inside brackets are shown below.

```
tool-tcl> string tr<tab>
tool-tcl> string trim<tab>
trim  trimleft  trimright

tool-tcl> puts [string tr<tab>
tool-tcl> puts [string trim<tab>
trim  trimleft  trimright
```

Command Option Completion and Listing

Command options can also be completed and listed using the Tab key. This functionality is consistent with command completion behavior and is supported both at the command prompt and within command substitutions.

Examples of option completion and listing are shown below.

```
tool-tcl> get_objects -<tab>
-filter  -hier   -of    -quiet   -sorted  -type
tool-tcl> get_objects -ty<tab>
tool-tcl> get_objects -type

tool-tcl> puts [get_objects -<tab>
-filter  -hier   -of    -quiet   -sorted  -type
tool-tcl> puts [get_objects -ty<tab>
tool-tcl> puts [get_objects -type
```

In addition, when an option accepts an enumerated set of values, the Tab key can be used to complete and display all valid enumeration values. The following

example shows all supported values for the `-type` option.

```
tool-tcl> get_objects -type <tab>
cell      floorplan   interface   lef_pin     lef_tech    lib_cell
module      package     pin         task        class       function
lef_cell    lef_site    lib         lib_pin     net         parameter
port       text_macro
```

Variable Completion and Listing

Variable completion and listing follow the same principles as command completion. When an incomplete variable name is entered and the Tab key is pressed, FusionShell attempts to complete the variable name. If multiple variables match the prefix, all matching variable names are displayed.

Variable completion and listing apply to both tool-defined variables and user-defined variables, and support both variable definition and variable reference contexts.

In the example below, multiple variables begin with `hdl_`, so FusionShell completes the prefix and then displays all matching variables.

```
tool-tcl> set hd<tab>
tool-tcl> set hdl_<tab>
hdl_allow_static_task   hdl_default_ext   hdl_default_std
hdl_inst_array_pre_postfix   hdl_max_limit
hdl_soc_integration_mode   hdl_warn_threshold
```

Both `set` and `unset` commands support Tab-based completion for user-defined variables, as shown below.

```
tool-tcl> set aa 0
tool-tcl> set aaa 0
tool-tcl> set a<tab>
tool-tcl> set aa<tab>
aa aaa
tool-tcl> unset a<tab>
tool-tcl> unset aa<tab>
aa aaa
```

Variable references using the `$` prefix also support completion and listing.

```
tool-tcl> puts $hd<tab>
tool-tcl> puts $hdl_<tab>
hdl_allow_static_task   hdl_default_ext   hdl_default_std
hdl_inst_array_pre_postfix   hdl_max_limit
hdl_soc_integration_mode   hdl_warn_threshold
```

File Path Completion and Listing

The Tab key can also be used to complete and list file paths. When entering a file name or directory path, pressing Tab completes the path or displays all matching files and directories.

The example below shows file name completion.

```
tool-tcl> open dash<tab>
tool-tcl> open dashrtl.<tab>
dashrtl.cmd  dashrtl.log
```

The following example lists files and directories under the `../tcl/` path.

```
tool-tcl> open ../tcl/<tab>
tc  work  script
```

3

3 FusionShell Python Mode

FusionShell Python mode is fully compatible with the official CPython release, supporting the complete Python syntax and functionality. Users can freely leverage third-party Python libraries to perform secondary development. In addition, Python mode provides built-in functions corresponding to Tcl commands, allowing users to directly invoke EDA tool operations within the Python environment.

For more information about the capabilities of FusionShell Python mode, refer to the following sections:

- Python Language Overview
- Python Version
- Importing Third-Party Python Libraries
- Invoking Built-in Tool Functions and Variables

3-1 Python Language Overview

Python is a widely adopted scripting language known for its design philosophy of elegance, clarity, and simplicity. Its concise and readable syntax enables developers to accomplish more with fewer lines of code. Python supports multiple programming paradigms, including procedural, object-oriented, and functional programming.

Python is extensively used in a wide range of domains, such as web development, data analysis, artificial intelligence, scientific computing, and scripting.

Key characteristics of Python include:

- **Interpreted Language**

Python is an interpreted language, meaning that code is executed line by line at runtime. This allows rapid development, testing, and debugging.

- **Dynamic Typing**

Python uses dynamic typing, so variable types do not need to be explicitly declared. While this increases flexibility, users should be aware of potential type-related runtime errors.

- **Object-Oriented Support**

Python supports object-oriented programming, including classes, objects, inheritance, and polymorphism.

- **Extensive Standard and Third-Party Libraries**

Python provides a rich standard library and a vast ecosystem of third-party libraries, enabling efficient implementation of a wide variety of applications, such as data analysis, machine learning, and visualization.

- **Cross-Platform Support**

Python can run on multiple operating systems.

3-2 Python Version

FusionShell Python mode is based on **CPython version 3.8.20**. The Python version information can be queried as shown below:

```
tool-py> import sys
tool-py> print(sys.version)
3.8.20 (default, Oct 11 2024, 10:13:03)
[GCC 9.3.1 20200408 (Red Hat 9.3.1-2)]
```

3-3 Importing Third-Party Python Libraries

Integrating Python with EDA tools provides significant advantages in multiple dimensions, including data analysis, automation, and ecosystem support. FusionShell Python mode enables these benefits by allowing direct use of Python's extensive third-party library ecosystem. Key advantages include:

- **Powerful Data Analysis and Visualization**

Python offers robust data processing and visualization libraries such as NumPy, pandas, and matplotlib. These libraries can be combined with EDA tool outputs to enable in-depth analysis and intuitive visualization of design data.

- **Automation**

Python scripting allows users to automate many EDA workflows, including simulation, layout verification, and performance analysis. This improves efficiency and reduces the risk of human error.

- **Community Ecosystem**

Python has a large and active developer community, providing abundant resources such as tutorials, libraries, and example code. Users can easily find solutions and benefit from shared experience.

In FusionShell Python mode, importing third-party libraries follows the same syntax and behavior as standard Python. Examples are shown below.

Import an entire module:

```
tool-py> import math
```

Import a module with an alias:

```
tool-py> import numpy as np
```

Import specific objects from a module:

```
tool-py> from datetime import date
```

Import all objects from a module:

```
tool-py> from math import *
```

3-4 Invoking Built-in Tool Functions and Variables

Similar to Tcl mode, where Tcl commands and variables are used to control and configure the tool, Python mode provides Python functions and variables for tool interaction.

Most commands and variables in Tcl mode have corresponding representations in Python mode. An example of the naming correspondence is shown below.

Mode	Built-in Tool Commands (Functions)	Built-in Tool Variables
Tcl	analyze	$hdl_default_std
Python	tcl.proc.analyze	hdl_default_std

Python Built-in Functions

Python built-in functions correspond directly to Tcl built-in commands, with a one-to-one name mapping. These functions are provided in the `tcl.proc` namespace.

The Tcl commands `puts` and `string` correspond to the Python functions `tcl.proc.puts` and `tcl.proc.string`. In the below example, `tcl.proc.puts` prints the string `haha` and returns an empty string, while `tcl.proc.string` returns the result as a string `'4'`.

```
tool-py> tcl.proc.puts('haha')
haha
''
tool-py> tcl.proc.string('length','haha')
'4'
```

The following example demonstrates how to read RTL and execute EDA operations in Python mode, equivalent to invoking `analyze`, `read_flist`, and `elaborate` commands in Tcl mode.

```
tool-py> tcl.proc.analyze('-hdl_type', 'verilog',
tcl.proc.read_flist('rtl.list'))
tool-py> tcl.proc.elaborate('top')
```

Python Built-in Variables

Python built-in tool variables share the same names as their Tcl counterparts and can be accessed directly, as shown below:

```
tool-py> search_path
'.'
```

However, directly assigning values to these variables using standard Python assignment syntax does not apply the change to the tool environment, even though the variable value appears to be updated:

```
tool-py> search_path = '. /design/rtl'
tool-py> search_path
'. /design/rtl'
```

To correctly modify tool variables in Python mode, users must use `tcl.proc.set_tool_var` or `tcl.proc.set`, as shown below:

```
tool-py> tcl.proc.set_tool_var('search_path','. /design/rtl')
'. /design/rtl'
tool-py> search_path
'. /design/rtl'
tool-py> tcl.proc.set('search_path','. /design/lib')
'. /design/lib'
tool-py> search_path
'. /design/lib'
```

4

4 FusionShell Hybrid Mode

FusionShell Hybrid Mode is not a standalone command-line mode. Instead, it refers to the capability of using Tcl and Python statements together within the same script. As described previously, FusionShell allows seamless and real-time switching between Tcl mode and Python mode without exiting the shell. For more information, refer to 1-2 Switching Between Tcl Mode and Python Mode.

This chapter describes how to use FusionShell Hybrid Mode. The following sections are covered:

4-1 Using Tcl Commands and Variables in Python Mode

Tcl commands and variables can be classified into two categories: built-in tool commands/variables and user-defined commands/variables.

The usage of built-in Tcl commands and variables in Python mode has been described in 3-4 Invoking Built-in Tool Functions and Variables. This section focuses on the usage of user-defined Tcl commands and variables in Python mode.

Using User-Defined Tcl Commands in Python Mode

Similar to built-in Tcl commands, user-defined Tcl procedures are exposed in the `tcl.proc` namespace when operating in Python mode.

The following example illustrates this behavior. A Tcl procedure named `add` is first defined in Tcl mode. After switching to Python mode using the `pymode` command, the procedure can be invoked directly through `tcl.proc.add`.

```
tool-tcl> proc add {prefix a b} {return "$prefix: [expr $a+$b]"}
tool-tcl> pymode
tool-py> tcl.proc.add('result', '1', '2')
result: 3
```

Using User-Defined Tcl Variables in Python Mode

In Python mode, user-defined Tcl variables can be accessed and modified directly. As shown in the following example, Tcl variable values — regardless of whether they represent integers, booleans, or strings — are converted to string type when accessed from Python.

```
tool-tcl> set v1 123
123
tool-tcl> set v2 true
true
tool-tcl> set v3 "hello world"
```

```
hello world
tool-tcl> pymode
tool-py> v1
'123'
tool-py> v2
'true'
tool-py> v3
'hello world'
```

Variables are shared across Python mode and Tcl mode. Therefore, modifying a variable in Python mode will also update its value in Tcl mode, as demonstrated below.

```
tool-py> v1='456'
tool-py> v2='false'
tool-py> v3='hello'
tool-py> tclmode()
tool-tcl> puts $v1
456
tool-tcl> puts $v2
false
tool-tcl> puts $v3
hello
```

4-2 Using Python Functions and Variables in Tcl Mode

Python mode and Tcl mode share the same execution environment. As a result, Python functions and variables defined in Python mode can be accessed directly from Tcl mode.

This section describes how to use user-defined Python functions and variables in Tcl mode.

Using User-Defined Python Functions in Tcl Mode

To invoke a user-defined Python function as a Tcl command, the function must be defined within the `tcl.proc` namespace. If a Python function is defined at the top level (outside of `tcl.proc`), it must be invoked using the Tcl `call` command.

The following example demonstrates invoking a Python function using `call`.

```
tool-py> def add(prefix,a,b): return f"{prefix}: {int(a)+int(b)}"
tool-py> tclmode()
tool-tcl> call add result 1 2
result: 3
```

Alternatively, a Python function can be explicitly registered as a Tcl command by adding it to the `tcl._procs` dictionary, as shown below.

```
tool-py> def add(prefix,a,b): return f"{prefix}: {int(a)+int(b)}"
tool-py> tcl._procs['add'] = add
tool-py> tclmode()
tool-tcl> add result 1 2
Warning: proc constraint not found for "add". (TCL-UNCONSTR)
result: 3
```

Using User-Defined Python Variables in Tcl Mode

User-defined Python variables can be accessed and modified directly in Tcl mode. However, to be accessible from Tcl mode, Python variables must be defined as

strings in Python mode.

If a Python variable is defined using a non-string data type (for example, an integer), referencing it in Tcl mode will result in an error, as shown below.

```
tool-py> v1=123
tool-py> tclmode()
tool-tcl> $v1
TypeError: startswith first arg must be str or a tuple of str,
not int
```

The correct usage is shown in the following example:

```
tool-py> v1='123'
tool-py> v2='true'
tool-py> v3='hello world'
tool-py> tclmode()
tool-tcl> puts $v1
123
tool-tcl> puts $v2
true
tool-tcl> puts $v3
hello world
```

Variables are shared across Python mode and Tcl mode. Therefore, modifying a variable in Tcl mode will also update its value in Python mode, as demonstrated below.

```
tool-tcl> set v1 456
456
tool-tcl> set v2 false
false
tool-tcl> set v3 hello
hello
tool-tcl> pymode
tool-py> v1
'456'
tool-py> v2
'false'
tool-py> v3
'hello'
```

4-3 Tcl+Python Hybrid Script Examples

Tcl and Python hybrid scripts can be executed in both interactive (command-line) mode and batch (script) mode.

Interactive Mode Example

When starting FusionShell in interactive mode, the language mode (Tcl or Python) must match the first language used in the hybrid script. If the script begins with Tcl statements, FusionShell must be launched in Tcl mode. If the script begins with Python statements, FusionShell must be launched in Python mode.

In the following example, the hybrid script begins with Python statements. Therefore, FusionShell is launched with the `-pymode` option to enter Python mode initially.

```
% <tool_exec_file> -pymode
tool-py> v1='123'
tool-py> v2='true'
tool-py> v3='hello world'
tool-py> tclmode()
tool-tcl> puts $v1
123
tool-tcl> puts $v2
true
tool-tcl> puts $v3
hello world
```

Batch Mode Example

In batch mode, the language mode (Tcl or Python) is also determined by the first language used in the hybrid script. If the script starts with Tcl statements, FusionShell must be launched in Tcl mode. If the script starts with Python statements, FusionShell must be launched in Python mode.

In the following example, the hybrid script file `mix_tcl_py.scr` begins with Tcl statements. Therefore, FusionShell is launched without the `-pymode` option to enter Tcl mode initially.

mix_tcl_py.scr

```
# Start With Tcl Script
set v1 123
set v2 true
set v3 "hello world"
pymode

# Switch To Python Script
v1='456'
v2='false'
v3='hello'
tclmode()

# Switch To Tcl Script
puts $v1
puts $v2
puts $v3
```

```
% <tool_exec_file> -f mix_tcl_py.scr
# Start With Tcl Script
<tcl> set v1 123
123
<tcl> set v2 true
true
<tcl> set v3 "hello world"
hello world
<tcl> pymode

# Switch To Python Script
<py> v1='456'
<py> v2='false'
<py> v3='hello'
<py> tclmode()

# Switch To Tcl Script
<tcl> puts $v1
456
```

```
<tcl> puts $v2
false
<tcl> puts $v3
Hello
```

A

Appendix A Common Tcl Commands

This appendix lists the common Tcl8.5 commands supported by FusionShell, including their available subcommands and support status.

Command	Subcommand	Support Status
after	--	Supported
	cancel	Planned
	idle	Planned
	info	Planned
append	--	Supported
apply	--	Supported
array	anymore	Supported
	donesearch	Supported
	exists	Supported
	get	Supported
	names	Supported
	nextelement	Supported
	set	Supported
	size	Supported
	startsearch	Supported
	statistics	Supported
	unset	Supported
bgerror	--	Planned
binary	format	Planned
	scan	Planned
break	--	Supported
catch	--	Supported
cd	--	Supported
chan	blocked	Planned
	close	Supported
	configure	Planned

	copy	Planned
	create	Planned
	eof	Supported
	event	Planned
	flush	Supported
	gets	Supported
	names	Supported
	pending	Planned
	postevent	Planned
	puts	Supported
	read	Supported
	seek	Supported
	tell	Supported
	truncate	Planned
clock	add	Planned
	clicks	Planned
	format	Planned
	microseconds	Planned
	milliseconds	Planned
	scan	Planned
	seconds	Planned
close	--	Supported
concat	--	Supported
continue	--	Supported
dde	servername	Planned
	execute	Planned
	poke	Planned
	request	Planned
	services	Planned
	eval	Planned
dict	append	Supported
	create	Supported
	exists	Supported
	filter	Supported
	for	Supported
	get	Supported
	incr	Supported
	info	Supported
	keys	Supported

	lappend	Supported
	merge	Supported
	remove	Supported
	replace	Supported
	set	Supported
	size	Supported
	unset	Supported
	update	Supported
	values	Supported
	with	Supported
encoding	convertfrom	Unsupported
	convertto	Unsupported
	dirs	Unsupported
	names	Unsupported
	system	Unsupported
eof	--	Supported
error	--	Supported
eval	--	Supported
exec	--	Supported
exit	--	Supported
expr	--	Supported
fblocked	--	Planned
fconfigure	--	Planned
fcopy	--	Planned
file	atime	Planned
	attributes	Planned
	channels	Supported
	copy	Planned
	delete	Planned
	dirname	Supported
	executable	Supported
	exists	Supported
	extension	Supported
	isdirectory	Supported
	isfile	Supported
	join	Supported
	link	Planned
	lstat	Planned
	mkdir	Planned

	mtime	Planned
	nativename	Planned
	normalize	Planned
	owned	Supported
	pathtype	Supported
	readable	Supported
	readlink	Supported
	rename	Planned
	rootname	Supported
	separator	Supported
	size	Supported
	split	Planned
	stat	Planned
	system	Planned
	tail	Supported
	type	Supported
	volumes	Supported
	writable	Supported
fileevent	--	Planned
flush	--	Supported
for	--	Supported
foreach	--	Supported
format	--	Supported
gets	--	Supported
glob	--	Supported
global	--	Supported
history	--	Supported
	add	Planned
	change	Planned
	clear	Supported
	event	Planned
	info	Planned
	keep	Supported
	nextid	Planned
	redo	Planned
if	--	Supported
incr	--	Supported
info	args	Planned
	body	Planned

	cmdcount	Planned
	commands	Supported
	complete	Planned
	default	Planned
	exists	Supported
	frame	Planned
	functions	Planned
	globals	Planned
	hostname	Supported
	level	Planned
	library	Planned
	loaded	Planned
	locals	Planned
	nameofexecutable	Supported
	patchlevel	Planned
	procs	Planned
	script	Planned
	sharedlibextension	Planned
	tclversion	Supported
	vars	Planned
interp	alias	Planned
	aliases	Planned
	bgerror	Planned
	create	Planned
	debug	Planned
	delete	Planned
	eval	Planned
	exists	Planned
	expose	Planned
	hide	Planned
	hidden	Planned
	invokehidden	Planned
	limit	Planned
	issafe	Planned
	marktrusted	Planned
	recursionlimit	Planned
	share	Planned
	slaves	Planned
	target	Planned

	transfer	Planned
join	--	Supported
lappend	--	Supported
lassign	--	Supported
lindex	--	Supported
linsert	--	Supported
list	--	Supported
llength	--	Supported
lmap	--	Supported
load	--	Supported
lrange	--	Supported
lrepeat	--	Supported
lreplace	--	Supported
lreverse	--	Supported
lsearch	--	Supported
lset	--	Supported
lsort	--	Supported
memory	active	Unsupported
	break_on_malloc	Unsupported
	info	Unsupported
	init	Unsupported
	objs	Unsupported
	onexit	Unsupported
	tag	Unsupported
	trace	Unsupported
	trace_on_at_malloc	Unsupported
	validate	Unsupported
namespace	children	Supported
	code	Planned
	current	Supported
	delete	Planned
	ensemble	Planned
	eval	Supported
	exists	Supported
	export	Planned
	forget	Planned
	import	Planned
	inscope	Planned
	origin	Planned

	parent	Supported
	path	Planned
	qualifiers	Supported
	tail	Supported
	upvar	Planned
	unknown	Planned
	which	Planned
open	--	Supported
package	forget	Unsupported
	ifneeded	Unsupported
	names	Unsupported
	prefer	Unsupported
	present	Unsupported
	provide	Unsupported
	require	Unsupported
	unknown	Unsupported
	vcompare	Unsupported
	versions	Unsupported
	vsatisfiers	Unsupported
parray	--	Planned
pid	--	Supported
pkg_mkIndex	--	Planned
proc	--	Supported
puts	--	Supported
pwd	--	Supported
re_syntax	--	Planned
read	--	Supported
regexp	--	Supported
regsub	--	Supported
rename	--	Supported
return	--	Supported
scan	--	Supported
seek	--	Supported
set	--	Supported
socket	--	Planned
source	--	Supported
split	--	Supported
string	compare	Supported
	equal	Supported

	first	Supported
	index	Supported
	is	Supported
	last	Supported
	length	Supported
	map	Supported
	match	Supported
	range	Supported
	repeat	Supported
	replace	Supported
	reverse	Supported
	tolower	Supported
	totitle	Supported
	toupper	Supported
	trim	Supported
	trimleft	Supported
	trimright	Supported
	bytelength	Supported
	wordend	Unsupported
	wordstart	Unsupported
subst	--	Supported
switch	--	Supported
tell	--	Supported
time	--	Supported
trace	add	Planned
	remove	Planned
	info	Planned
	variable	Planned
	vdelete	Planned
	vinfo	Planned
unknown	--	Supported
unload	--	Planned
unset	--	Supported
update	--	Planned
uplevel	--	Supported
upvar	--	Supported
variable	--	Supported
vwait	--	Planned
while	--	Supported

B

Appendix B Additional Tool Commands

This appendix lists additional built-in tool commands provided by FusionShell. These commands extend standard Tcl functionality and are categorized by command type.

Command Category	Tool Command	Support Status
Procedure Handling	define_proc_constraints	Supported
	parse_proc_args	Supported
	defer	Supported
Object Handling	add_to_objects	Supported
	compare_objects	Supported
	filter_objects	Supported
	foreach_in_objects	Supported
	get_objects	Supported
	get_object_name	Supported
	get_object_type	Supported
	index_objects	Supported
	remove_from_objects	Supported
	sizeof_objects	Supported
	sort_objects	Supported
Others	alias	Supported
	apropos	Supported
	assert	Supported
	call	Supported
	help	Supported
	man	Supported
	plugin	Supported
	pymode	Supported
	unalias	Supported