DashRTL User Guide

Version 2025.06

DashThru

Copyright Notice

Trademarks

Disclaimer

Contents

1

1 DashRTL Introduction

DashRTL is a high-speed digital design platform developed by DashThru Technology, characterized by high performance and large capacity. To ensure the project schedule, RTL coding typically requires frequent compilation to check for issues during digital design development. As chip design scales up, using lint, synthesis or simulation tools for compilation checks can become overwhelmingly time-consuming.

DashRTL uses a parallel HDL compiler optimized for speed. By reading RTL code through DashRTL, fast design iterations can be performed. For large-scale SoC projects, DashRTL's ultra-large capacity enables full-chip RTL compilation and checks, eliminating the need for hierarchical RTL checks that were often required in the past.

DashRTL offers the following checks for RTL code:

- Checks for syntax errors and potential issues during the RTL compilation process.
- Verifies the synthesizability of RTL code, identifying non-synthesizable code early in the project.
- After RTL compilation is complete, performs design exploration such as inspecting hierarchies/parameters/cells/nets as well as their properties.

1-1 DashRTL Feature Overview

For detailed information on DashRTL's RTL checking capabilities, please refer to Section 4: DashRTL Features. DashRTL offers the following key features:

- Ultra-fast RTL code compilation to accelerate design iterations.
- Supports ultra-large capacity of full-chip design, enabling the reading of chip level RTL code.
- Provides both Tcl and Python user interfaces, with flexible switching between Tcl and Python shell modes.
- Supports both simulation-style and synthesis-style launch modes.
- Supports Verilog and SystemVerilog standards, with the ability to specify different standards on a per-file basis.
- Provides a powerful built-in GUI for inspecting RTL syntax and design issues.

1-2 DashRTL Installation and Deployment

- **Recommended Operating System Versions: RHEL/CentOS 6.5-7.9**
  Experimental support for RHEL/Rocky/Alma 8.x

To check the system version on RHEL and CentOS systems, view the `system-release` file

```
% cat /etc/system-release
CentOS Linux release 7.9.2009 (Core)
```

- **Installing DashRTL**

Simply extract the compressed package to the installation directory.

```
% tar xJvf DashRTL_v2024.09.tar.xz
```

- **Setting System Variables**

System variables can be added to the shell initialization file (e.g., `.cshrc`). In the following example, replace `<installation_path>` with the actual installation path. After setting this, you can directly use the `dashrtl` executable from the command line.

`DASHTHRU_LICENSE_SERVER`: Set this for the license server. Please refer to the **DashLM User Guide** for details on how to obtain a license and start the license service. `<port>` is the license service port, with a default of 28000. `<hostname>` refers to the machine name or IP address where the DashLM license service is running.

```
% setenv DASHTHRU_LICENSE_SERVER
<port>@<hostname>:<port>@<hostname>:…
% set path = ($path <installation_path>/DashRTL_v2024.09/bin)
```

The example for csh is as follows:

```
% setenv DASHTHRU_LICENSE_SERVER 28000@lic_server1:28000@lic_server2
% set path = ($path /edatool/DashRTL_v2024.09/bin)
```

- Launching DashRTL

```
% which dashrtl
/edatool/DashRTL_v2024.09/bin/dashrtl
% dashrtl
DashRTL(TM) Digital Design Platform
Copyright(c) 2024, DashThru Technology, Ltd. All rights reserved.

Version:    v2024.09-Alpha, build 2024/09/20
Date:       2024/09/20 20:11:45
Host:       EPYC / 128 Threads
Launch:     dashrtl
Feedback:   support@dashthru.com

Info: checked out license 'FusionShell-Tcl-Py'. (LIC-CO)
dashrtl-tcl>
```

2

2 DashRTL Launch Modes

To accommodate the user preferences of various design engineers in the chip industry, DashRTL offers two different launch modes: **simulation-style** and **synthesis-style**.

The synthesis-style launch mode can be accessed via either the Tcl Shell or Python Shell interface. The Tcl Shell facilitates the traditional Tcl script usage, making it easier for users to migrate scripts from other tools. The Python Shell provides a highly flexible Python platform as an interface, suitable for users who require high flexibility in script development.

Whether using Tcl Shell or Python Shell, users can choose between **interactive mode** or **batch mode** for launching.

The following table describes the specific launch modes and their features:

Launch Mode	Shell	Shell Mode	Features
Simulation Style	Automatically enters Tcl Shell after launch.		Suitable for engineers familiar with simulation tools (e.g., chip verification engineers). Allows direct launching of the tool with RTL files specified via Linux command line.
Synthesis Style	Tcl	Interactive	Suitable for engineers familiar with synthesis tools (e.g., chip synthesis engineers). Requires using Tcl commands or scripts to read RTL files.
		Batch	
	Python	Interactive	Suitable for engineers who need to develop Python scripts. Requires using Python commands or scripts to read RTL files.
		Batch	

DashRTL is executed via the executable file `dashrtl` along with its associated options. Options can be abbreviated; for example, `-files` and `-f` are the same option, and `-pymode` and `-py` are equivalent. However, options starting with `+`, such as `+incdir+`, cannot be abbreviated.

In addition, all launch modes and shell modes support the `-execute` / `-x` option to execute the initialization script, as well as the `dashrtl_rc.tcl` initialization script file. For detailed information, please refer to Section 2-4: Launch with Initialization Scripts.

Below are examples for both synthesis-style and simulation-style launch modes:

```
# Synthesis Style
% dashrtl -files run.tcl -log run.log

# Simulation Style: full option name
% dashrtl -log run.log -run -top mod_top -verilog -filelist rtl.lst

# Simulation Style: compact option name
% dashrtl -l run.log -r -t mod_top -ver -f rtl.lst
```

This chapter provides a detailed introduction to the two launch modes of DashRTL and their related features. Please refer to the following sections:

- Simulation-Style Launch Mode
- Synthesis-Style Launch Mode
- Handling File Lists
- Launch with Initialization Scripts

2-1 Simulation-Style Launch Mode

The DashRTL simulation-style launch mode allows the use of various options with the `dashrtl` executable directly from the Linux shell to specify input files such as RTL codes and the RTL file list. The usage is similar to mainstream simulation tools.

Preparing Input Files

- **RTL Code Files**: Supports Verilog or SystemVerilog standards.
- **RTL File List**: A file containing a list of RTL code files. For the file list format, please refer to Section 2-3: Handing File Lists.
- **LIB File List (Optional)**: Supports cell library files, such as memory and pad cells used in chip design. If this file is not provided during launch, the corresponding macro cells will be treated as black boxes.

Simulation-Style Launch Options

dashrtl <launch_options> -run <hdl_read_options>

launch_options: These are the launch options specific to DashRTL, such as `-log` to specify the log file name, `-gui`/`-no_gui` to control whether the GUI is launched, and other related options.

hdl_read_options: These are the options used to configure the reading of RTL files. For example: `dashrtl -run -f rtl.list`. Since all arguments following `-run` are treated as RTL reading options rather than DashRTL launch options, the `-run` switch must follow all other startup options and be the last option used.

Correct Usage:

```
% dashrtl -log top -run -f rtl.list
```

Incorrect Usage:

```
# Error: '-log' is an incorrect option here
% dashrtl -run -f rtl.list -log top
```

`-run` cannot be used multiple times. If repeated, an error will occur:

```
% dashrtl -run option1 -run option2
Error: cannot use '-run' switch after '-run' switch.
```

`-run` and `-files` cannot be used together as they correspond to different launch modes: simulation-style and synthesis-style, respectively. If multiple files are specified for `hdl_read_options`, they must be enclosed in double quotes, such as:

```
% dashrtl -run -f "rtl1.list rtl2.list"
```

There are two types of options for `hdl_read_options`:
Unordered options: These begin with a plus sign (+), such as `+define+` and `+incdir+`, and take effect as if they were specified at the beginning of the list, regardless of their position within the list.
Ordered options: These begin with a minus sign (-), such as `-verilog`, `-undef`, and `-define`, and take effect only in the order they appear in the list.

- **Launch Options: dashrtl <launch_options>**

[-pymode]
Launch DashRTL in Python Shell mode. If this option is not used, the default is to enter the Tcl Shell. When this option is used, DashRTL will stay in the Python Shell (`dashrtl-py`) after RTL compilation is complete.

[-log "log_file_name cmd_file_name" | -no_log]
These two options cannot be used simultaneously. `-log` specifies the log file and command file names. If this option is not used, `dashrtl.log` and `dashrtl.cmd` files are generated by default. For example:
`-log a` will generate `a.log` and `a.cmd`
`-log "a b"` will generate `a.log` and `b.cmd`
`-no_log` prevents the generation of log and command files.
For more flexible usage, please refer to Simulation-Style Launch Example

[-no_overwrite]
By default, the tool overwrites the original log and command files upon each launch. When this option is used, new log and command files will be generated with a suffix. The file naming format will be `xxx.logN` and `xxx.cmdN`. For example,

if the tool is launched twice, the following files will be created:

```
dashrtl.log1, dashrtl.cmd1
dashrtl.log2, dashrtl.cmd2
```

[-gui | -no_gui]

These two options cannot be used simultaneously. When `-gui` is used, the GUI will open during launch. When `-no_gui` is used, the GUI will be forced to be disabled during launch, and subsequent commands will not be able to enable the GUI.
In the simulation-style launch mode, if neither option is used, the GUI will be opened by default during launch.

- **RTL Reading Options：dashrtl -run <hdl_read_options>**

-run [-filelist <hdl_filelists> | -f <hdl_filelists>]

Specifies the RTL filelist. Filelists can be nested within each other. If there are two or more filelists, they can be enclosed in double quotes or specified multiple times using `-filelist`. For example:

```
dashrtl -run -f "rtl1.list rtl2.list"
dashrtl -run -f rtl1.list -f rtl2.list
```

-run [-v01|-v2001|-v2k]

Specifies that the compiler should use the Verilog 2001 standard. This can also be nested within the filelist.

-run [-v05|-v2005]

Specifies that the compiler should use the Verilog 2005 standard. This can also be nested within the filelist.

-run [-sv09|-sv2009]

Specifies that the compiler should use the SystemVerilog 2009 standard. This can also be nested within the filelist.

-run [-sv12|-sv2012]

Specifies that the compiler should use the SystemVerilog 2012 standard. This can also be nested within the filelist.

-run [-sv17|-sv2017]

Specifies that the compiler should use the SystemVerilog 2017 standard. This can also be nested within the filelist.

-run [-sv23|-sv2023]
Specifies that the compiler should use the SystemVerilog 2023 standard. This can also be nested within the filelist.

-run [-verilog]
Specifies that the compiler should use the standard defined by the tool variable `hdl_default_std`, which defaults to the Verilog 2005 standard. This can also be nested within the filelist.

-run [-sverilog]
Specifies that the compiler should use the standard defined by the tool variable `hdl_default_std`, which defaults to the SystemVerilog 2009 standard. This can also be nested within the filelist.

-run [-netlist]
Specifies that the compiler should use netlist mode for compilation. This can also be nested within the filelist. This option can speed up netlist compilation but should not be used for RTL code compilation.

-run [-autoext]
Specifies that the compiler should automatically select the appropriate HDL standard for compilation based on the file extension of the RTL files. This can be nested within the filelist.
The default behavior is:

- Files with the `.v` extension are compiled using the Verilog 2005 standard.
- Files with the `.sv` extension are compiled using the SystemVerilog 2009 standard.
- Files with the `.vg` extension are compiled using netlist mode.

If no files in the list match these extensions, a matching error will occur. The extension matching can be modified using the `+hdlext+` option.
This is the default behavior of the compiler, which automatically determines the HDL standard based on file extensions if no other HDL standard is specified. The file extension of `include` files is not evaluated; they will be compiled using the same standard as the parent file.

-run [+hdlext+.<ext1>=<hdl_std1>+.<ext2>=<hdl_std2>+ ....."]
Used with `-autoext` to customize the suffix matching pattern. This can be nested within the filelist.

- `ext` specifies the file extension of the RTL files.
- `hdl_std` specifies the HDL standard and must be one of the following: v01, v05, sv09, sv12, sv17, sv23, verilog, sverilog, or netlist.

Example usage:

```
dashrtl -run -autoext +hdlext+.v=v05+.sv=sv23 -f rtl.list
```

-run [-v <ignore_files>]

Specifies files in the filelist that contain memory/pad macro cells or behavioral model Verilog files. Use `-v` to specify such files. These files are not RTL files, and the compiler will ignore their contents in relation to the `-v` option. If there are multiple files, they can be listed together or specified with `-v` multiple times.

Example usage:

```
dashrtl -run -f rtl.list -v "a.v b.v"
dashrtl -run -f rtl.list -v a.v -v b.v
```

-run [+incdir+<path1>+<path2>+...]

Specifies the include file paths. This can be nested within the filelist. If there are multiple paths, they can be separated by the `+` sign, or `+incdir+` can be specified multiple times. For example:

```
dashrtl -run +incdir+/rtl/top+/rtl/sub -f rtl.list
dashrtl -run +incdir+/rtl/top +incdir+/rtl/sub -f rtl.list
```

-run [+define+<macro_id1>(=<value1>)+<macro_id2>(=<value2>)+...]

Specifies additional custom macros to define during compilation. This can be nested within the filelist. `macro_id` is the name of the macro. `value` is optional and specifies the value of the macro definition.

For example, the following command:

```
dashrtl -run +define+A=0+B -f rtl.list
```

is equivalent to defining in RTL:

```
`define A 0
`define B
```

If there are multiple macros, they can be separated by the `+` sign, or `+define+` can be specified multiple times. For example:

```
dashrtl -run +define+A=0+B=0 -f rtl.list
dashrtl -run +define+A=0 +define+B=0 -f rtl.list
```

-run [-top <design_name>]

Specifies the name of the top-level module in the design. This cannot be nested within the filelist. If this option is not used, the tool will automatically infer the top-level module. In this case, ensure that the inferred top-level module is correct.

-run [-liblist <lib_filelists>]

Specifies the timing library list file for macro cells. This cannot be nested within the filelist. For example, if you have a file `lib.list` containing paths to all required

library files:
`/lib/cell1.lib`
`/lib/cell2.lib`
……
You can specify the `lib.list` file like this:
`dashrtl -run -liblist lib.list -f rtl.list`
Multiple `lib.list` files can be specified:
`dashrtl -run -liblist "lib1.list lib2.list" -f rtl.list`

-run [-search_path <paths>]
Specifies the RTL search path, which is used to support incomplete paths in the RTL file list. This cannot be nested within the filelist. If there are multiple paths, they can be enclosed in double quotes or specified multiple times using `-search_path`. For example:
```
dashrtl -run -search_path "/rtl/top /rtl/sub" -f rtl.list
dashrtl -run -search_path /rtl/top -search_path /rtl/sub -f
rtl.list
```

-run <hdl_files>
Specifies the HDL files directly without using a filelist. Any string that does not begin with `+` or `-` (which are reserved for options) is treated as an HDL file path. For example, the following command directly reads in `top.v` and `sub.v`:
`dashrtl -run -v05 /rtl/top/top.v -f rtl.list sub.v`

Simulation-Style Launch Example

- **DashRTL HDL Standard Usage**

DashRTL supports dynamically switching the compiler's HDL standard based on different files. The following options can be used to change the current compiler HDL standard. These options must be specified in the correct order, and it is recommended to place them before the filelist or RTL files.
-v01|-v2001|-v2k|-v05|-v2005|-sv09|-sv2009|-sv12|-sv2012|-sv17|-sv2017|-sv23|-sv2023|-verilog|-sverilog|-netlist|-autoext

In Example 1, both the RTL files in `rtl.list` and `top.sv` will be compiled using the default compiler behavior. The file extensions will determine the appropriate HDL standard:
`.v` files will use the Verilog 2005 standard by default.

`.sv` files will use the SystemVerilog 2009 standard by default.
Example 1:

```
dashrtl -run -top mod_top -f rtl.list top.sv
```

In Example 2, the RTL files in `rtl.list` will be compiled based on their file extensions. However, the `-v01` option explicitly switches the compiler to the Verilog 2001 standard. This ensures that `top.v` is compiled using Verilog 2001, overriding the default Verilog 2005 standard.
Example 2:

```
dashrtl -run -top mod_top -f rtl.list -v01 top.v
```

In example 3-1 (correct usage), the `-v01` option is specified before `top.v`, ensuring the compiler switches to the Verilog 2001 standard prior to compiling `top.v`. This is the correct order of operations and ensures that `top.v` is compiled using the Verilog 2001 standard.
Example 3-1：

```
dashrtl -run -top mod_top -v01 top.v
```

In example 3-2 (incorrect usage), `top.v` is compiled first with the default Verilog 2005 standard, and the compiler is switched to Verilog 2001 afterward. This results in `top.v` being compiled with the wrong standard, which is not the desired behavior.
Example 3-2：

```
dashrtl -run -top mod_top top.v -v01
```

In example 4, different files are compiled using different HDL standards:
`rtl1.list` is compiled using Verilog 2001 (`-v01`).
`sub.v` and `rtl2.list` are compiled using Verilog 2005 (`-v05`).
`top.sv` is compiled using SystemVerilog 2009 (`-sv09`).
This example demonstrates how to flexibly specify different HDL standards for various files in a single compilation process.
Example 4：

```
dashrtl -run -top mod_top -v01 -f rtl1.list -v05 sub.v -f
rtl2.list -sv09 top.sv
```

In example 5, `rtl.list` is compiled using SystemVerilog 2023 (`-sv23`), and the `-autoext` option is used to allow the compiler to automatically detect the HDL standard based on file extensions:
`sub.v` is compiled using the default Verilog 2009 standard (since the file extension `.v` is used).
`top.sv` is compiled using the default SystemVerilog 2009 standard (since the file

extension `.sv` is used).
`netlist.vg` is compiled using the default netlist mode (since the file extension `.vg` is used).
Example 5:

```
dashrtl -run -top mod_top -sv23 -f rtl.list -autoext sub.v top.sv
netlist.vg
```

In Example 6-1 (incorrect usage), an error will occur because the default `-autoext` settings do not include `.ve` as a recognized extension. Therefore, the compiler will not know how to handle `sub.ve`, leading to an error.
Example 6-1:

```
dashrtl -run -top mod_top -autoext sub.ve top.v -sv09 -f rtl.list
```

In Example 6-2 (correct usage), the `+hdlext+` option is used to customize the file extension matching for `-autoext`. Here, we define `.ve` to be compiled with the Verilog 2001 standard (`-v01`), and `.v` to be compiled with the Verilog 2005 standard (`-v05`), while the files in `rtl.list` will be compiled using the SystemVerilog 2009 standard (`-sv09`).
`sub.ve` will be compiled using Verilog 2001 (due to `+hdlext+v01=.ve`).
`top.v` will be compiled using Verilog 2005 (due to `+hdlext+v05=.v`).
`rtl.list` files will be compiled using SystemVerilog 2009 (due to `-sv09`).
Example 6-2:

```
dashrtl -run -top mod_top -autoext +hdlext+v01=.ve+v05=.v sub.ve
top.v -sv09 -f rtl.list
```

- **DashRTL Logging Usage**

DashRTL by default generates log and command files:
`dashrtl.log`: The log file that records screen output.
`dashrtl.cmd`: The file that records the command list used after the tool is launched.
The following examples demonstrate how to control the log and command file generation using the `-log` option.

Example 1-1: No `-log` option is used, so the default `dashrtl.log` and `dashrtl.cmd` files are generated.

```
dashrtl -run ...
```

Example 1-2: The `-no_overwrite` option is used, which prevents overwriting existing log and command files. New files are created with the suffix `N`.

```
dashrtl -no_overwrite -run ...
```

Example 2-1: The `-log` option specifies the prefix `a`, so the generated files are `a.log` and `a.cmd`.

```
dashrtl -log a -run ...
```

Example 2-2: The `-no_overwrite` option is used, generating `a.logN` and `a.cmdN` instead of overwriting `a.log` and `a.cmd`.

```
dashrtl -log a -no_overwrite -run ...
```

Example 3-1: The `-log` option specifies `a` for the log prefix and `b` for the command prefix, so `a.log` and `b.cmd` files are generated.

```
dashrtl -log "a b" -run ...
```

Example 3-2: The `-no_overwrite` option is used, so `a.logN` and `b.cmdN` are generated instead of overwriting the original files.

```
dashrtl -log "a b" -no_overwrite -run ...
```

Example 4: The `-no_log` option is used, so no log or command files are generated.

```
dashrtl -no_log -run ...
```

Example 5: The `-log` option is set with an empty string for the log name, resulting in no log file being generated, but the command file `b.cmd` is still created.

```
dashrtl -log '"" b' -run ...
```

Example 6: The `-log` option is set with an empty string for the command file name, resulting in the generation of only the log file `a.log` and no command file.

```
dashrtl -log 'a ""' -run ...
```

- **DashRTL GUI Usage**

In the simulation-style launch mode, DashRTL will by default launch the GUI, allowing users to review compilation results and perform RTL cross-probing. Therefore, **Example 1** and **Example 2** will automatically launch the GUI. **Example 3** disables the GUI, so it will not open the GUI after launch.

Example 1: `dashrtl -run ...`
Example 2: `dashrtl -gui -run ...`
Example 3: `dashrtl -no_gui -run ...`

Simulation-Style Launch Reference Examples

- **Absolute Path File List**

Prepare the RTL file list (rtl.list) and LIB library file (lib.list).

rtl.list

```
+define+MACRO+...
+incdir+/design/rtl+...
/design/rtl/top.v
/design/rtl/sub.v
......
```

lib.list

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

Example 1 (Recommended): All macro cells are correctly identified according to the lib file.

```
% dashrtl -log top -run -top mod_top -liblist lib.list -verilog
-f rtl.list
```

Example 2: All macro cells are treated as black boxes, with their pins considered as 1-bit wide inout.

```
% dashrtl -log top -run -top mod_top -verilog -f rtl.list
```

- **Relative Path File List**

Prepare the RTL file list (rtl.list) and LIB library file (lib.list).

rtl.list

```
+define+MACRO+...
+incdir+/design/rtl+...
top.v
sub.v
......
```

lib.list

```
pad.lib
mem32x32.lib
mem32x64.lib
......
```

Example 1 (Recommended): All macro cells are correctly identified according to the lib file.

```
% dashrtl -log top -run -top mod_top -search_path /design/rtl
-search_path /design/lib -liblist lib.list -verilog -f rtl.list
```

Example 2: All macro cells are treated as black boxes, with their pins considered as 1-bit wide inout.

```
% dashrtl -log top -run -top mod_top -search_path /design/rtl
-search_path /design/lib -verilog -f rtl.list
```

2-2 Synthesis-Style Launch Mode

DashRTL's synthesis-style launch mode enters the Tcl Shell by default, where RTL file reading and LIB file reading are controlled via Tcl Shell commands and variables, similar to mainstream synthesis tools.

Preparing Input Files

- **RTL Code Files**: Supports Verilog or SystemVerilog standards.
- **RTL Filelist**: A file containing a list of RTL code files. For the filelist format, please refer to Section 2-3: Handling File Lists.
- **LIB List File (Optional)**: Supports cell library files, such as memory and pad cells used in chip design. If this file is not provided during launch, the corresponding macro cells will be treated as black boxes.
- **Tcl Script File (Optional for Tcl Shell Mode)**: Can be executed directly upon launch. If this file is not provided, commands must be manually entered in the Tcl Shell.
- **Python Script File (Optional for Python Shell Mode)**: Can be executed directly upon launch. If this file is not provided, commands must be manually entered in the Python Shell.

Synthesis-Style Launch Options

[-files <startup_exec_tcl_or_python_script_files>]
`startup_exec_tcl_or_python_script_files` refers to the script files executed at launch. In Tcl mode, a Tcl script is required, and in Python mode, a Python script is required. Examples:

```
dashrtl -f run.tcl
dashrtl -f run.py -pymode
```

This option allows specifying two or more script files, which can be enclosed in double quotes `""` or specified multiple times using `-f`. Examples:

```
dashrtl -f "run1.tcl run2.tcl"
dashrtl -f run1.tcl -f run2.tcl
```

[-execute | -x <startup_exec_tcl_or_python_scripts>]

Executes user-defined scripts at launch. The script type must match the mode (Tcl or Python). Examples:

```
dashrtl -x "set a 0" -f run.tcl
dashrtl -x "a=0" -pymode -f run.py
```

This option can be used multiple times. In the example below, the contents inside -x will be executed in order during launch:

```
dashrtl -x "set a 0" -x "set b 1" -f run.tcl
```

[-pymode]

Enters the Python Shell upon launch. If this option is not used, the default is to enter the Tcl Shell.

[-log "log_file_name cmd_file_name" | -no_log]

These two options cannot be used simultaneously. `-log` specifies the log file and command file names. If this option is not used, `dashrtl.log` and `dashrtl.cmd` files are generated by default. For example:

`-log a` will generate `a.log` and `a.cmd`
`-log "a b"` will generate `a.log` and `b.cmd`
`-no_log` prevents the generation of log and command files.
For more flexible usage, please refer to Simulation-Style Launch Example

[-no_overwrite]

By default, the tool overwrites the original log and command files upon each launch. When this option is used, new log and command files will be generated with a suffix. The file naming format will be `xxx.logN` and `xxx.cmdN`. For example, if the tool is launched twice, the following files will be created:

```
dashrtl.log1, dashrtl.cmd1
dashrtl.log2, dashrtl.cmd2
```

[-gui | -no_gui]

These two options cannot be used simultaneously. When `-gui` is used, the GUI will open during launch. When `-no_gui` is used, the GUI will be forced to be disabled during launch, and subsequent commands will not be able to enable the GUI.

In synthesis-style launch mode, if neither option is used, the GUI will not be opened by default, but can later be launched using the `start_gui` command.

Interactive and Batch Shell Mode

- **Interactive Shell Mode**

If no script file is specified using the `-files` option during launch, DashRTL enters the interactive mode. Upon launch, DashRTL enters either the Tcl Shell or Python Shell, where you can manually execute Tcl or Python commands.

The following command will enter the Tcl Shell:

```
% dashrtl
DashRTL(TM) Digital Design Platform
Copyright(c) 2024, DashThru, Ltd. All rights reserved.

Version:    v2024.09-Alpha, build 2024/09/20
Date:      2024/09/20 20:11:45
Host:      EPYC / 128 Threads
Launch:    dashrtl
Feedback:  support@dashthru.com

Info: checked out license 'FusionShell-Tcl-Py'. (LIC-CO)
dashrtl-tcl>
```

The following command will enter the Python Shell:

```
% dashrtl -pymode
DashRTL(TM) Digital Design Platform
Copyright(c) 2024, DashThru, Ltd. All rights reserved.

Version:    v2024.09-Alpha, build 2024/09/20
Date:      2024/09/20 20:11:45
Host:      EPYC / 128 Threads
Launch:    dashrtl
Feedback:  support@dashthru.com

Info: checked out license 'FusionShell-Tcl-Py'. (LIC-CO)
dashrtl-py>
```

- **Batch Shell Mode**

If the `-files` option is used during launch to specify a script file, DashRTL enters the batch mode. The script file type executed in batch mode must match the Shell type, meaning Tcl Shell can only execute Tcl scripts, and Python Shell can only execute Python scripts.

The following command enters Tcl Shell and executes the `run.tcl` script:

```
% dashrtl -f run.tcl
DashRTL(TM) Digital Design Platform
Copyright(c) 2024, DashThru, Ltd. All rights reserved.

Version:    v2024.09-Alpha, build 2024/09/20
Date:      2024/09/20 20:11:45
Host:      EPYC / 128 Threads
Launch:     dashrtl
Feedback:   support@dashthru.com

Info: checked out license 'FusionShell-Tcl-Py'. (LIC-CO)
set_tool_var search_path "$search_path /design/rtl"
read_libs "pad.lib mem32x32.lib"
analyze -hdl_type verilog [read_flist rtl.list]
......
```

The following command enters Python Shell and executes the `run.py` script:

```
% dashrtl -f run.py -pymode
DashRTL(TM) Digital Design Platform
Copyright(c) 2024, DashThru, Ltd. All rights reserved.

Version:    v2024.09-Alpha, build 2024/09/20
Date:      2024/09/20 20:11:45
Host:      EPYC / 128 Threads
Launch:     dashrtl
Feedback:   support@dashthru.com

Info: checked out license 'FusionShell-Tcl-Py'. (LIC-CO)
tcl.proc.set_tool_var('search_path',search_path ... ')
tcl.proc.read_libs('pad.lib mem32x32.lib ... ')
tcl.proc.analyze('-hdl_type', 'verilog',
tcl.proc.read_flist('rtl.list'))
......
```

Dynamically Switching Between Tcl Shell and Python Shell

DashRTL supports dynamically switching between Tcl Shell and Python Shell without exiting, and the variables set before the switch are retained. To switch from Tcl Shell to Python Shell, use the `pymode` command, and to switch from Python Shell back to Tcl Shell, use the `tclmode()` command.

For the list of commands available in both Tcl Shell and Python Shell, please refer to Appendix C: DashRTL Tool Commands. Most of the commands and variables in Tcl Shell and Python Shell have corresponding counterparts, such as:

	Tool Command	Tool Variable
Tcl Shell	analyze	hdl_default_std
Python Shell	tcl.proc.analyze	tcl.var.hdl_default_std

The following example demonstrates the process of switching from Tcl Shell to Python Shell, and then back to Tcl Shell. The variable `a` set in Tcl Shell is visible in Python Shell, and similarly, the variable `b` set in Python Shell is visible in Tcl Shell.

```
dashrtl-tcl> set a 0
0
dashrtl-tcl> pymode
-----------------------------------------------------------------------
dashrtl-py> a
'0'
dashrtl-py> b="0"
dashrtl-py> tclmode()
-----------------------------------------------------------------------
dashrtl-tcl> return $b
0
dashrtl-tcl> return $a
0
```


Setting Tool Variables

DashRTL has predefined tool variables, and by setting these tool variables, you can control the corresponding tool behaviors. Tool variables can be set using either `set`

or `set_tool_var` in both Tcl Shell and Python Shell. It is recommended to use `set_tool_var`, as this command can check the validity of the variable name and value, as well as flexibly modify secondary options.

Complete setting of tool variables in Tcl Shell:

```
dashrtl-tcl> set_tool_var crossprobe_exec "code --goto
{filepath}:{line}"
code --goto {filepath}:{line}
dashrtl-tcl> set crossprobe_exec "code --goto {filepath}:{line}"
code --goto {filepath}:{line}
```

Partial modification of tool variables in Tcl Shell (only for variables with secondary options):

```
dashrtl-tcl> set_tool_var hdl_default_ext -verilog .v.vh
verilog=.v.vh sverilog=.sv netlist=.vg
```

Setting tool variables in Python Shell:

```
dashrtl-py> tcl.proc.set_tool_var('crossprobe_exec', 'code --goto
{filepath}:{line}')
dashrtl-py> tcl.proc.set('crossprobe_exec', 'code --goto
{filepath}:{line}')
```

Reading Cell Libraries

Reading the Cell Library is an optional step, typically used to identify macro cells like memory or pad cells in the RTL design. If this step is not performed, the tool will treat these cells as black boxes, and all the pins of the black boxes will be treated as 1-bit inout. Therefore, to ensure the completeness of the design, it is recommended to execute this step.

The libraries can be read using the `read_libs` command, and you can specify the paths of the lib files in a list, like this:

```
read_libs "/design/lib/mem.lib /design/lib/pad.lib"
```

Alternatively, you can put all the lib paths in a lib filelist and read them using the following command:

```
read_libs [read_flist lib.lst]
```

Reading RTL Files

RTL compilation involves two steps: `analyze` and `elaborate`, and these must be executed in the order of analyze -> elaborate. The analyze step specifies the HDL file paths and the compilation standard, while the elaborate step specifies the top-level design name.

The `analyze` step allows flexible control of RTL input through `hdl_read_options`. Users can specify RTL input using a file list, a list of HDL files, or a mixed list.

```
# filelist
analyze -hdl_type verilog [read_flist rtl.list]
# HDL files
analyze -hdl_type verilog "top.v sub.v"
# mixed list
analyze -hdl_type verilog "+define+TESTMODE -v05 top.v sub.v \
-sv09 [read_flist rtl.list]"
```

The elaborate step specifies the name of the top-level module, and the tool will compile based on this top-level. If the top-level name is not specified, elaborate will automatically infer the top-level. Users need to verify that the inferred top-level is correct. If more than one top-level is inferred, an error will occur.

```
# Automatic inference
elaborate
# Specify top-level module
elaborate top
```

analyze

-hdl_type <hdl_type_name>

The `hdl_type_name` option specifies the initial compilation mode for the compiler. Only one of the following methods can be selected:

[v01 | v2001 | v2k]:
Specifies that the compiler should use the Verilog 2001 standard.
[v05 | v2005]:
Specifies that the compiler should use the Verilog 2005 standard.
[sv09 | sv2009]:
Specifies that the compiler should use the SystemVerilog 2009 standard.
[sv12 | sv2012]:

Specifies that the compiler should use the SystemVerilog 2012 standard.
[sv17 | sv2017]:
Specifies that the compiler should use the SystemVerilog 2017 standard.
[sv23 | sv2023]:
Specifies that the compiler should use the SystemVerilog 2023 standard.
[verilog]:
Specifies that the compiler should use the standard defined by the tool variable `hdl_default_std`, which defaults to the Verilog 2005 standard.
[sverilog]:
Specifies that the compiler should use the standard defined by the tool variable `hdl_default_std`, which defaults to the SystemVerilog 2009 standard.
[netlist]:
Specifies that the compiler should use netlist mode for compilation. This option can speed up netlist compilation but should not be used for RTL code compilation.
[autoext] (default):
Specifies that the compiler should automatically select the appropriate HDL standard for compilation based on the file extension of the RTL files.
The default behavior is:

- Files with the `.v` extension are compiled using the Verilog 2005 standard.
- Files with the `.sv` extension are compiled using the SystemVerilog 2009 standard.
- Files with the `.vg` extension are compiled using netlist mode.

If no files in the list match these extensions, a matching error will occur. The extension matching can be modified using the `hdl_default_ext` variable.
This behavior is the default compilation method for analyze. If the `-hdl_type` option is not used, the `autoext` method will be applied by default.

-define "<macro_id1>(=<value1>) <macro_id2>(=<value2>) ..."
This option is used to define additional macros at the start of the compilation. The `macro_id` is the name of the macro, and the `value` is optional, representing the value of the macro definition. For example:

```
analyze -define "A=0 B"
```

is equivalent to the following definitions in RTL:

```
`define A 0
`define B
```

"<hdl_read_options> | [read_flist <hdl_filelists>]"
`hdl_read_options` can be used with various flexible options to sequentially read RTL files. These options can either be specified directly after the analyze

command or placed in an HDL filelist (which must be converted to a regular list using the `read_flist` command before being passed to `analyze`).

There are two types of options for `hdl_read_options`:
Unordered options: These begin with a plus sign (+), such as `+define+` and `+incdir+`, and take effect as if they were specified at the beginning of the list, regardless of their position within the list.
Ordered options: These begin with a minus sign (-), such as `-verilog`, `-undef`, and `-define`, and take effect only in the order they appear in the list.

[-v01|-v2001|-v2k]
Specifies that the compiler should use the Verilog 2001 standard. This can also be nested within the filelist.

[-v05|-v2005]
Specifies that the compiler should use the Verilog 2005 standard. This can also be nested within the filelist.

[-sv09|-sv2009]
Specifies that the compiler should use the SystemVerilog 2009 standard. This can also be nested within the filelist.

[-sv12|-sv2012]
Specifies that the compiler should use the SystemVerilog 2012 standard. This can also be nested within the filelist.

[-sv17|-sv2017]
Specifies that the compiler should use the SystemVerilog 2017 standard. This can also be nested within the filelist.

[-sv23|-sv2023]
Specifies that the compiler should use the SystemVerilog 2023 standard. This can also be nested within the filelist.

[-verilog]
Specifies that the compiler should use the standard defined by the tool variable `hdl_default_std`, which defaults to the Verilog 2005 standard. This can also be nested within the filelist.

[-sverilog]
Specifies that the compiler should use the standard defined by the tool

variable `hdl_default_std`, which defaults to the SystemVerilog 2009 standard. This can also be nested within the filelist.

[-netlist]
Specifies that the compiler should use netlist mode for compilation. This can also be nested within the filelist. This option can speed up netlist compilation but should not be used for RTL code compilation.

[-autoext]
Specifies that the compiler should automatically select the appropriate HDL standard for compilation based on the file extension of the RTL files. This can be nested within the filelist.
The default behavior is:

- Files with the `.v` extension are compiled using the Verilog 2005 standard.
- Files with the `.sv` extension are compiled using the SystemVerilog 2009 standard.
- Files with the `.vg` extension are compiled using netlist mode.

If no files in the list match these extensions, a matching error will occur. The extension matching can be modified using the `-hdlext` option.
This is the default behavior of the compiler, which automatically determines the HDL standard based on file extensions if no other HDL standard is specified. The file extension of `include` files is not evaluated; they will be compiled using the same standard as the parent file.

[-hdlext "<hdl_std1>=<ext_list1> <hdl_std2>=<ext_list2> ....."]
Used in conjunction with `-autoext`, this option customizes the extension matching pattern. It can be nested within a filelist. `ext_list` specifies a list of extensions, such as `.v.vh`. `hdl_std` specifies the HDL standard, and must be one of v01, v05, sv09, sv12, sv17, sv23, verilog, sverilog, or netlist. For example:

```
analyze "-autoext -hdlext \"verilog=.v.vh sverilog=.sv\" \
         [read_flist rtl.list]"
```

[-v <ignore_files>]
Specifies files in the filelist that contain memory/pad macro cells or behavioral model Verilog files. Use `-v` to specify such files. These files are not RTL files, and the compiler will ignore their contents in relation to the `-v` option. If there are multiple files, they can be listed together or specified with `-v` multiple times. For example:

```
analyze "[read_flist rtl.list] -v \"a.v b.v\""
analyze "[read_flist rtl.list] -v a.v -v b.v"
```

[+incdir+<path1>+<path2>+...]
Specifies the include file paths. This can be nested within the filelist. If there are multiple paths, they can be separated by the `+` sign, or `+incdir+` can be specified multiple times. For example:

```
analyze "+incdir+/rtl/top+/rtl/sub [read_flist rtl.list]"
analyze "+incdir+/rtl/top +incdir+/rtl/sub [read_flist
rtl.list]"
```

[+define+<macro_id1>(=<value1>)+<macro_id2>(=<value2>)+...]
Specifies additional custom macros to define during compilation. This can be nested within the filelist. `macro_id` is the name of the macro. `value` is optional and specifies the value of the macro definition.
For example, the following command:

```
analyze "+define+A=0+B [read_flist rtl.list]"
```

is equivalent to defining in RTL:

```
`define A 0
`define B
```

If there are multiple macros, they can be separated by the `+` sign, or `+define+` can be specified multiple times. For example:

```
analyze "+define+A=0+B=0 [read_flist rtl.list]"
analyze "+define+A=0 +define+B=0 [read_flist rtl.list]"
```

[-define "<macro_id1>(=<value1>) <macro_id2>(=<value2>) ... "]
Specifies additional custom macros to define during compilation. This can be nested within the filelist. `macro_id` is the name of the macro. `value` is optional and specifies the value of the macro definition. This option differs from `+define+` in that `+define+` is unordered, whereas `-define` is ordered.
For example, the following command:

```
analyze "-define A=0 -define B [read_flist rtl.list]"
```

is equivalent to defining in RTL:

```
`define A 0
`define B
```

If there are more than two macros, they can be specified as a list or using `-define` multiple times. For example:

```
analyze "-define \"A=0 B=0\" [read_flist rtl.list]"
analyze "-define A=0 -define B=0 [read_flist rtl.list]"
```

[-undef "<macro_id1> <macro_id2> ... "]
This option deletes the defined macros during compilation, and can be nested within a filelist. The `macro_id` refers to the macro names. After using this option, the specified macros will not exist in the subsequent RTL files.

```
analyze "-f rtl1.list -undef A -undef B -f rtl2.list"
```

This is equivalent to inserting the following between the compilation of `rtl1.list` and `rtl2.list`:

```
`undef A
`undef B
```

If there are more than two macros, they can be specified as a list or using `-undef` multiple times. For example:

```
analyze "-undef \"A B\" [read_flist rtl.list]"
analyze "-undef A -undef B [read_flist rtl.list]"
```

[-undef_all]

This option deletes all defined macros during compilation, and can be nested within a filelist. After using this option, no macros will exist in the subsequent RTL files.

```
analyze "[read_flist rtl1.list] -undef_all \
         [read_flist rtl2.list]"
```

This is equivalent to deleting all defined macros between the compilation of `rtl1.list` and `rtl2.list`.

[-print_macros <print_macro_file>]

This option prints the values of the defined macros during compilation, and can be nested within a filelist. If the `print_macro_file` path is not specified, the macro information will be printed to the screen. The following example prints the macro status after compiling `rtl1.list` to the screen:

```
analyze "[read_flist rtl1.list] -print_macros \"\""
```

If the `print_macro_file` path is specified, the macro information will be printed to the specified file. The following example prints the macro status after compiling `rtl1.list` to `macro.log`:

```
analyze "[read_flist rtl1.list] \
         -print_macros /design/log/macro.log \
         [read_flist rtl2.list]"
```

elaborate

-parameters "<param_1>=<value1> <param_2>=<value2> ..."

This option modifies the parameter values of the top-level module. When using automatic top-level inference, you must ensure that the parameter names specified in -parameters match those inferred in the top-level module. The parameter values support Verilog numeric formats. For example:

```
elaborate top -parameters "BIT_WIDTH=16 DATA=3'b110"
```

<top_module_name>

The elaborate step specifies the name of the top-level module, and the tool will compile based on this top-level. If the top-level name is not specified, elaborate will automatically infer the top-level. Users need to verify that the inferred top-level is correct. If more than one top-level is inferred, an error will occur. For example:

```
# Automatic inference
elaborate
# Specify top-level module
elaborate top
```

Synthesis-Style Launch Reference Examples

- **Absolute Path File List**

Prepare the RTL file list (rtl.list) and LIB library file (lib.list).

rtl.list

```
+define+MACRO+...
+incdir+/design/rtl+...
/design/rtl/top.v
/design/rtl/sub.v
......
```

lib.list

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

Tcl Shell Mode:

run.tcl

```
# Set Tool Variables
set_tool_var search_path "$search_path ..."
set_tool_var hdl_default_std -verilog v05
set_tool_var hdl_default_std -sverilog sv09

# Optional: Read Lib
read_libs "/design/lib/pad.lib /design/lib/mem32x32.lib ... "
# OR
read_libs [read_flist /design/filelist/lib.list]

# RTL Analyze+Elaborate
analyze -hdl_type verilog [read_flist /design/filelist/rtl.list]
elaborate top

# Debug From GUI
start_gui
```

```
% dashrtl -f run.tcl -log run
```

Python Shell Mode:

run.py

```
# Set Tool Variables
tcl.proc.set_tool_var('search_path', f'{search_path} ... ')
tcl.proc.set_tool_var('hdl_default_std', 'verilog=v05 sverilog=sv09')

# Optional: Read Lib
tcl.proc.read_libs('/design/lib/pad.lib', '/design/lib/mem32x32.lib', …)
# OR
tcl.proc.read_libs(tcl.proc.read_flist('/design/filelist/lib.list'))

# RTL Analyze+Elaborate
tcl.proc.analyze('-hdl_type', 'verilog',
                 tcl.proc.read_flist('/design/filelist/rtl.list'))
tcl.proc.elaborate('top')

# Debug From GUI
tcl.proc.start_gui
```

```
% dashrtl -f run.py -log run -pymode
```

- **Relative Path File List**

Prepare the RTL file list (rtl.list) and LIB library file (lib.list).

rtl.list

```
+define+MACRO+...
+incdir+/design/rtl+...
top.v
sub.v
......
```

lib.list

```
pad.lib
mem32x32.lib
mem32x64.lib
......
```

Tcl Shell Mode:

run.tcl

```
# Set Tool Variables
set_tool_var search_path "$search_path \
                          /design/filelist /design/rtl /design/lib"
set_tool_var hdl_default_std -verilog v05
set_tool_var hdl_default_std -sverilog sv09

# Optional: Read Lib
read_libs "pad.lib mem32x32.lib ... "
# OR
read_libs [read_flist lib.list]

# RTL Analyze+Elaborate
analyze -hdl_type verilog [read_flist rtl.list]
elaborate top

# Debug From GUI
start_gui
```

```
% dashrtl -f run.tcl -log run
```

Python Shell Mode:

run.py

```
# Set Tool Variables
tcl.proc.set_tool_var('search_path',
                      f'{search_path} /design/filelist /design/rtl /design/lib')
tcl.proc.set_tool_var('hdl_default_std', 'verilog=v05 sverilog=sv09')

# Optional: Read Lib
tcl.proc.read_libs('pad.lib', 'mem32x32.lib', ... )
# OR
tcl.proc.read_libs(tcl.proc.read_flist('lib.list'))

# RTL Analyze+Elaborate
tcl.proc.analyze('-hdl_type', 'verilog', tcl.proc.read_flist('rtl.list'))
tcl.proc.elaborate('top')

# Debug From GUI
tcl.proc.start_gui
```

```
% dashrtl -f run.py -log run -pymode
```

2-3 Handling File Lists

The `flist` file is a special file format in DashRTL, commonly used when commands or variables involve long lists, which could make scripts unwieldy. For example, HDL file lists following the `analyze` command, library file lists following the `read_libs` command, and path lists following the set `search_path` command can all be placed in a `flist` file. You can then use the `read_flist` command to convert the contents into a regular list to be passed to commands or variables.

The elements in a `flist` file can be separated by spaces or newlines. For example:
`-verilog top.v sub.v`
is equivalent to:

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

When using a `flist`, you need to use the `read_flist` command to read the file. The converted list can then be directly used in commands or variable assignments. `flist` files can also be nested and support both regular and special comments. Regular comments include single-line comments (`#`, `//`) and multi-line comments (`/* */`).

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
```

For example:

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

is equivalent to:

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

Special comments must follow the format `// dashthru <flist_items>`. In this case, although the `flist_items` are within the comment, the `read_flist` command will still recognize the `dashthru` keyword and output the items from the comment into the list. This allows for more flexible modification of the contents of the file without changing the original file list.

```
// OLD LIST: old.lst

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

```
// NEW LIST: new.lst

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

In the above example, the left side represents the original HDL file list, and the right side is the new HDL file list with two lines of special comments added. This new HDL file list is still usable in third-party tools and is equivalent in effect to the original filel ist. However, in DashRTL, executing the command:

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

is equivalent to adding two more lines to the list, which is functionally equivalent to running:

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

Using `flist` makes it easy to store elements of a command or variable in a file and then convert them into a list using `read_flist`. The following sections will demonstrate how to use `flist` to specify `search_path`, `get_cells`, `hdl_filelist`, and `lib_filelist`.

Using flist to Specify search_path

```
// 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 a script, using the command `set search_path [read_flist search.lst]` is equivalent to:

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

You can also mix normal lists and `flist`. For example:

```
set search_path "$search_path /design/rtl design/rtl/top
```

```
[read_flist libsearch.list]"
```

This is functionally equivalent to:

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

Using flist to Specify get_cells

// Cell List: cell.lst	// Memory List: mem.lst
// dashthru u_chip/data_reg u_chip/u_sub1/data_reg* -f mem.lst	u_chip/u_mem32x32 u_chip/u_mem32x64

In a script, using the command `get_cells [read_flist cell.lst]` is equivalent to:

```
get_cells "u_chip/data_reg u_chip/u_sub1/data_reg*
u_chip/u_mem32x32 u_chip/u_mem32x64"
```

You can also mix normal lists and `flist`. For example:

```
get_cells "u_chip/u_sub1/data_reg* [read_flist mem.lst]"
```

This is functionally equivalent to:

```
get_cells "u_chip/u_sub1/data_reg* u_chip/u_mem32x32
u_chip/u_mem32x64"
```

Using flist to Specify HDL File List

// RTL1 List: rtl1.lst	// RTL2 List: rtl2.lst
// dashthru +define+TESTMODE +incdir+/design/rtl /design/rtl/top.v /design/rtl/sub_top.v -f rtl2.lst	-verilog /design/rtl/sub1.v -sverilog /design/rtl/sub2.sv

- **Simulation-Style Launch Mode**

Launch with `dashrtl -run -f rtl1.lst` is equivalent to executing:

```
dashrtl -run +define+TESTMODE +incdir+/design/rtl
/design/rtl/top.v /design/rtl/sub_top.v -verilog
/design/rtl/sub1.v -sverilog /design/rtl/sub2.sv
```

You can also mix normal lists and `flist`, for example:

```
dashrtl -run -top mod_top -f rtl1.lst
```

This is equivalent to:

```
dashrtl -run -top mod_top +define+TESTMODE +incdir+/design/rtl
/design/rtl/top.v /design/rtl/sub_top.v -verilog
/design/rtl/sub1.v -sverilog /design/rtl/sub2.sv
```

When multiple `flist` files are used, you can specify `-f` multiple times, such as:

```
dashrtl -run -top mod_top -f rtl1.lst -f rtl2.lst
```

- **Synthesis-Style Launch Mode**

In the script, using `analyze -hdl_type verilog [read_flist rtl1.lst]` is equivalent to executing:

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

You can also mix normal lists and `flist`, 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+TESTMODE
+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 there are multiple `flist` files, you can use `read_flist` to specify multiple `flist` paths, for example:

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

Using flist to Specify LIB File List

// 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

- **Simulation-Style Launch Mode**

Launch with `dashrtl -run -liblist lib1.lst -f rtl.lst` is equivalent to reading in 4 libraries:

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

If there are multiple `flist` files, you can specify `-liblist` multiple times, for example:

```
dashrtl -run -liblist lib1.lst -liblist lib2.lst -f rtl.lst
```

- **Synthesis-Style Launch Mode**

In the script, using `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"
```

You can also mix normal lists and `flist`, 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 there are multiple `flist` files, you can use `read_flist` to specify multiple `flist` paths, for example:

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

2-4 Launch with Initialization Scripts

Whether DashRTL is launched in simulation-style or synthesis-style mode, or in Tcl Shell or Python Shell mode, it supports the `-execute`/`-x` option for the execution of initialization scripts, as well as the `dashrtl_rc.tcl` initialization script file.

The `-execute`/`-x` option can be used once or multiple times. If used multiple times, the scripts will be executed in the order they are specified. For examples:

```
% dashrtl -x 'set DESIGN_TOP top'
% dashrtl -x 'set DESIGN_TOP top;puts $DESIGN_TOP;'
% dashrtl -x 'set DESIGN_TOP top' -x 'puts $DESIGN_TOP'
```

If the user wishes to execute an initialization script file before startup, the file must be placed in the following path. If the `.dashthru` hidden directory does not exist, the user needs to create it manually:

```
% ls ~/.dashthru/dashrtl_rc.tcl
```

Simulation-Style Launch Mode

In simulation-style launch mode, the initialization script execution sequence in DashRTL is as follows:

1. Execute the `~/.dashthru/dashrtl_rc.tcl` initialization script file (if present).
2. Execute any single or multiple `-execute`/`-x` options (if present).
3. Execute the compile options specified after the `-run` option.
4. Execute any single or multiple `-execute`/`-x` options again (if present).

Note: As mentioned in the introduction to simulation-style launch options, all options following `-run` are recognized as RTL compilation options. However, the `-execute`/`-x` options are exceptions. This allows users to execute customized scripts after RTL compilation.

- **Tcl Shell Initialization Script Example**

```
% dashrtl -x 'puts "Execute script1 before rtl read"' \
          -x 'puts "Execute script2 before rtl read"' \
          -run top.v \
          -x 'puts "Execute script3 after rtl read"' \
          -x 'puts "Execute script4 after rtl read"' \
          -x 'exit'
```

```
.................
Info: loading init script'/home/adam/.dashthru/dashrtl_rc.tcl'.
(SHELL-INIT)
Execute script1 before rtl read
Execute script2 before rtl read
.................
Info: elaborated design 'top' in 0s. (ELAB-DONE)
Execute script3 after rtl read
Execute script4 after rtl read
Info: thank you for using DashRTL! (err:0, crit:1, warn:1,
info:6)
```

- **Python Shell Initialization Script Example**

```
% dashrtl -py -x 'print("Execute script1 before rtl read")' \
              -x 'print("Execute script2 before rtl read")' \
              -run top.v \
              -x 'print("Execute script3 after rtl read")' \
              -x 'print("Execute script4 after rtl read")' \
              -x 'exit()'
.................
Info: loading init script'/home/adam/.dashthru/dashrtl_rc.tcl'.
(SHELL-INIT)
Execute script1 before rtl read
Execute script2 before rtl read
.................
Info: elaborated design 'top' in 0s. (ELAB-DONE)
Execute script3 after rtl read
Execute script4 after rtl read
Info: thank you for using DashRTL! (err:0, crit:1, warn:1,
info:6)
```

Synthesis-Style Launch Mode

In synthesis-style launch mode, the initialization script execution sequence in DashRTL is as follows:

1. Execute the `~/.dashthru/dashrtl_rc.tcl` initialization script file (if present).
2. Execute any single or multiple `-execute`/`-x` options (if present).
3. Execute the script files specified by the `-f` option (if present).

Note: Unlike in simulation-style launch mode, the order of `-execute`/`-x` options before or after the `-f` option does not affect the actual execution order. DashRTL always executes the `-execute`/`-x` scripts first, and then executes the script files specified by the `-f` option.

- Tcl Shell Initialization Script Example

```
% dashrtl -x 'puts "Execute script1 before rtl read"' \
          -x 'puts "Execute script2 before rtl read"' \
          -f run.tcl \
          -x 'puts "Execute script3 before rtl read"' \
          -x 'puts "Execute script4 before rtl read"'
..................
Info: loading init script'/home/adam/.dashthru/dashrtl_rc.tcl'.
(SHELL-INIT)
Execute script1 before rtl read
Execute script2 before rtl read
Execute script3 before rtl read
Execute script4 before rtl read
..................
Info: elaborated design 'top' in 0s. (ELAB-DONE)
Info: thank you for using DashRTL! (err:0, crit:1, warn:1,
info:6)
```

- Python Shell Initialization Script Example

```
% dashrtl -py -x 'print("Execute script1 before rtl read")' \
               -x 'print("Execute script2 before rtl read")' \
               -f run.tcl \
               -x 'print("Execute script3 before rtl read")' \
               -x 'print("Execute script4 before rtl read")'
..................
Info: loading init script'/home/adam/.dashthru/dashrtl_rc.tcl'.
(SHELL-INIT)
Execute script1 before rtl read
Execute script2 before rtl read
Execute script3 before rtl read
Execute script4 before rtl read
..................
Info: elaborated design 'top' in 0s. (ELAB-DONE)
Info: thank you for using DashRTL! (err:0, crit:1, warn:1,
info:6)
```

3

3 Graphical User Interface

DashRTL provides a powerful graphical user interface (GUI) that allows users to easily review various RTL design information and debug RTL code. DashRTL supports the GUI in both simulation-style or synthesis-style launch mode, and the functionality of the GUI remains the same regardless of the launch mode.

In simulation-style launch mode, the GUI will automatically start after RTL code compilation is complete by default, which is equivalent to explicitly using the `-gui` option. If the `-no_gui` option is used, the GUI will not be started automatically. The two examples below will both automatically launch the GUI:

```
% dashrtl -run ...
% dashrtl -gui -run ...
```

In synthesis-style launch mode, if you want the GUI to start automatically, you need to explicitly use the `-gui` option to launch DashRTL, as shown in the following examples:

```
% dashrtl -gui
% dashrtl -gui -files user.tcl
```

If you do not want the GUI to automatically start in synthesis-style launch mode (i.e., you don't use `-gui` to launch DashRTL), you can manually start the GUI using the `start_gui` command. The following examples show how to start the GUI in both Tcl mode and Python mode:

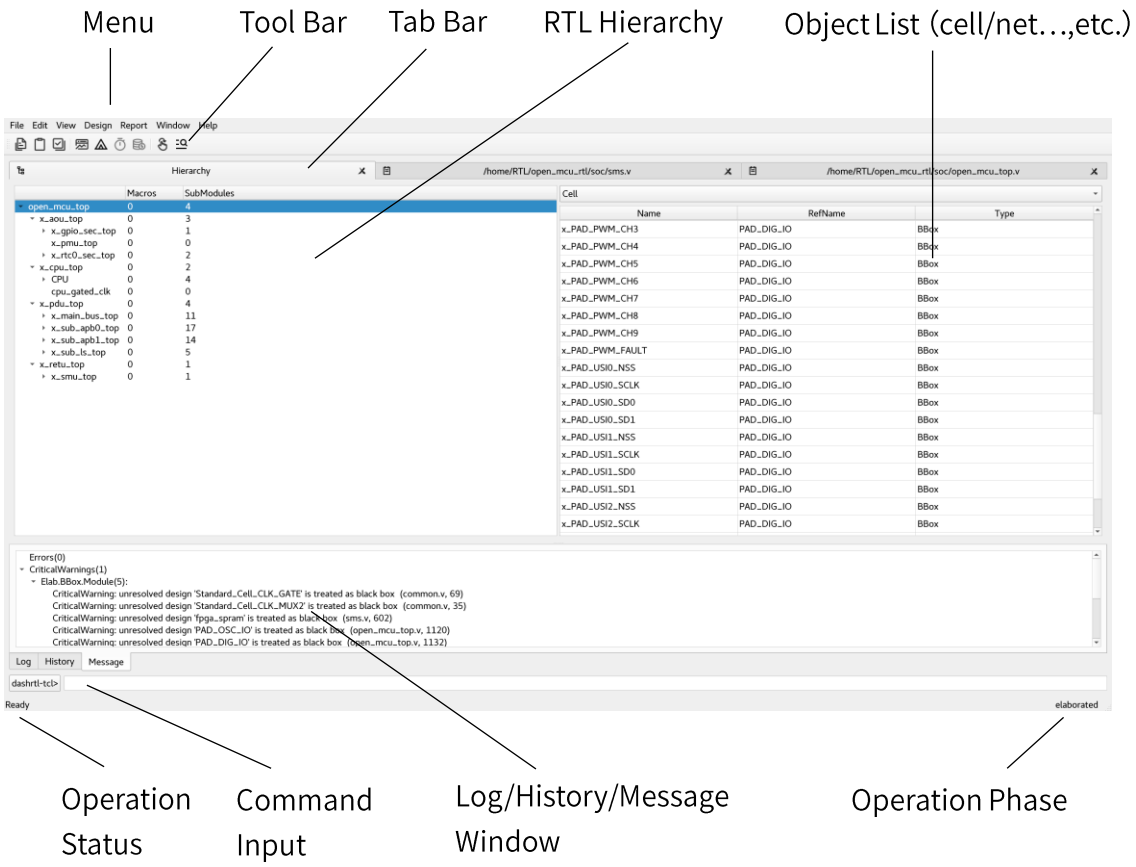
```
dashrtl-tcl> start_gui
dashrtl-py> tcl.proc.start_gui()
```

This chapter will introduce the various GUI features, including:

- GUI Window Layout
- Reviewing Classified Messages
- RTL Cross-probing within GUI
- RTL Cross-probing with External Editor
- Reloading RTL Design

3-1 GUI Window Layout

The following is a schematic of the GUI window layout. The specific content can be configured through the toolbar under `View`. It is recommended to check `View` → `Show Tab Bar` to enable the tab bar.

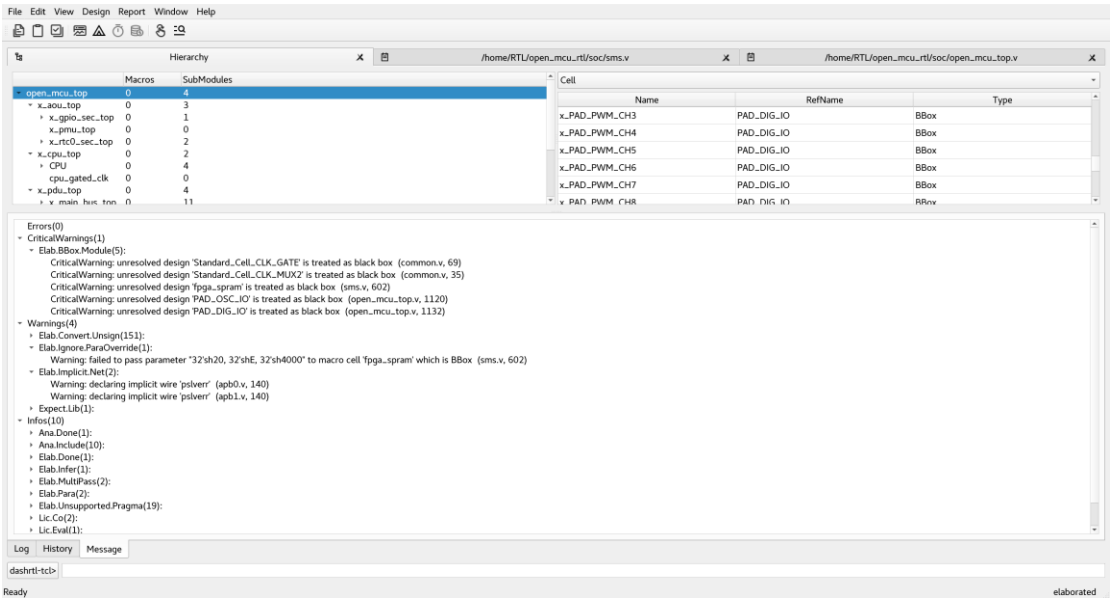


3-2 Reviewing Classified Messages

After executing the `analyze` and `elaborate` commands, the message window in the GUI will classify and display the RTL compilation messages. DashRTL divides messages into four severities: `Error`, `CriticalWarning`, `Warning`, and `Info`. For the specific meanings of these message levels, please refer to Section 4-2: RTL Syntax Checking.

The following is an example of the message window. It can be seen that under each message severity, the messages are further classified with their message IDs. DashRTL message IDs follow a functional naming style, such as `Elab.Convert.Unsign`, which indicates a message related to the conversion of unsigned numbers during the elaboration process. Under each message ID, all messages of this type can be expanded.

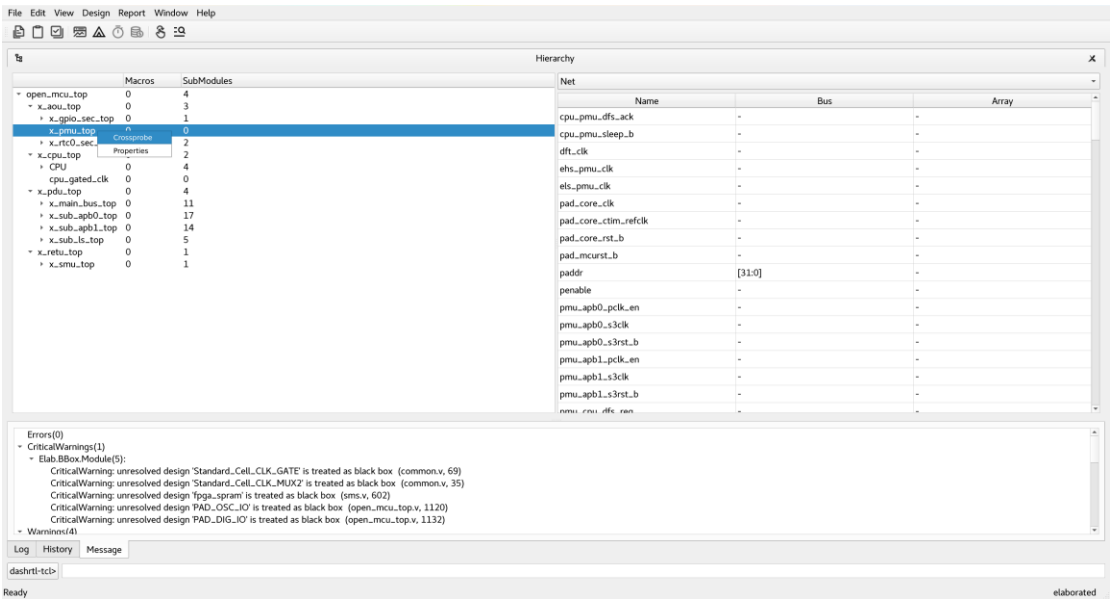
This format makes it clear for users to understand the actual issue related to any type of messages through the message ID, improving the readability of the messages.



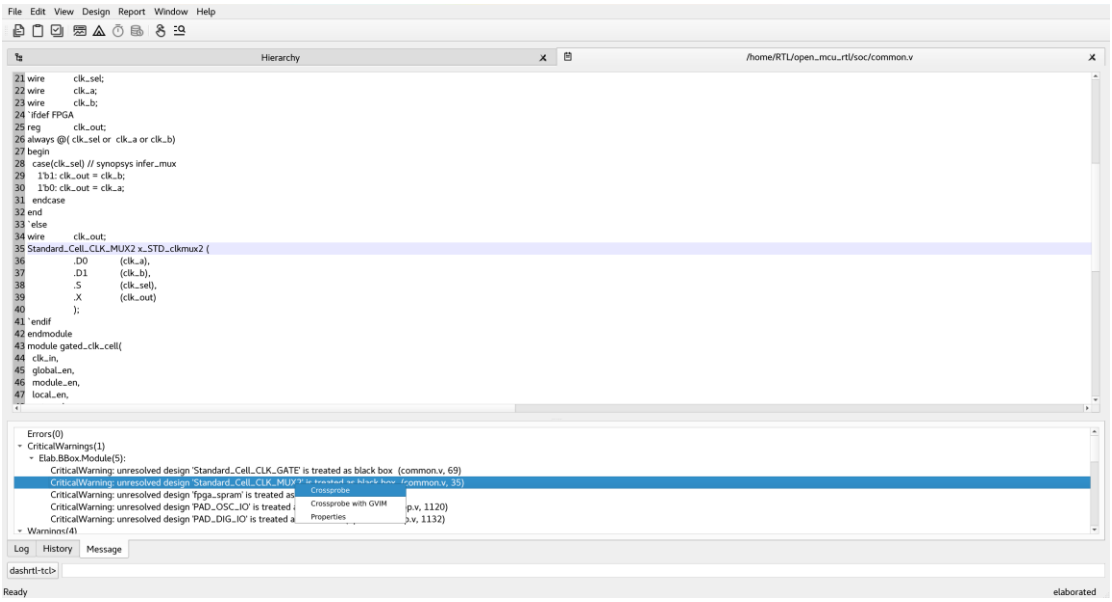
3-3 RTL Cross-probing within GUI

DashRTL GUI supports two ways to perform a one-click cross-probing to RTL code: one is from the hierarchy window and the other is from the message window.

To cross-probe from the hierarchy, simply right-click on the specified hierarchy level and select `Crossprobe`. This will open an external editor to review the code corresponding to the selected instance, as shown in the figure below.



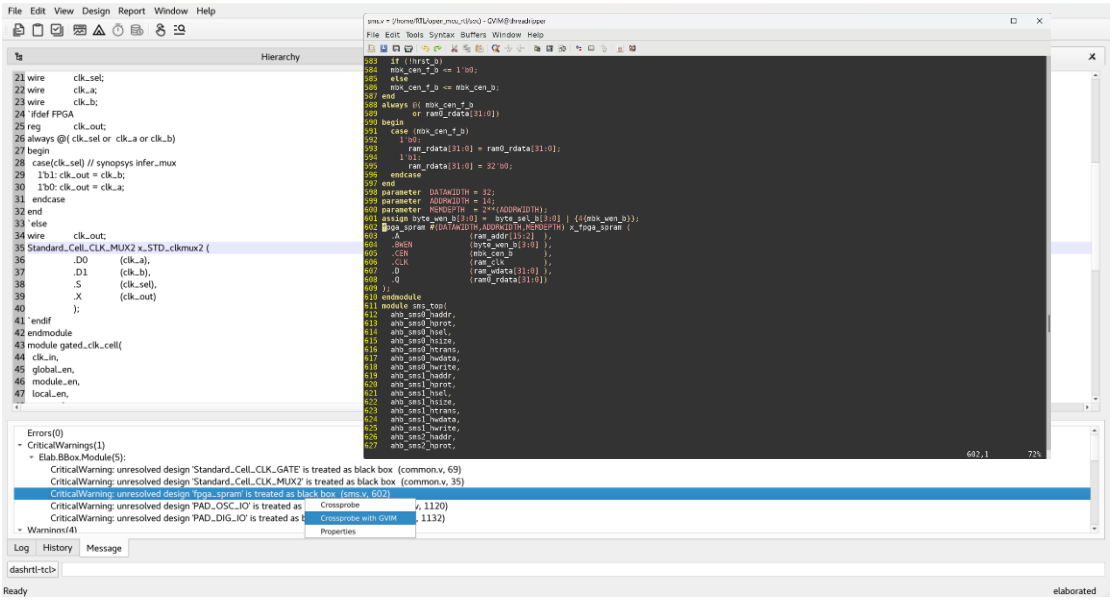
To cross-probe from the message, right-click on the selected message and select `Crossprobe`. This will automatically open a new tab displaying the corresponding line for the selected message.



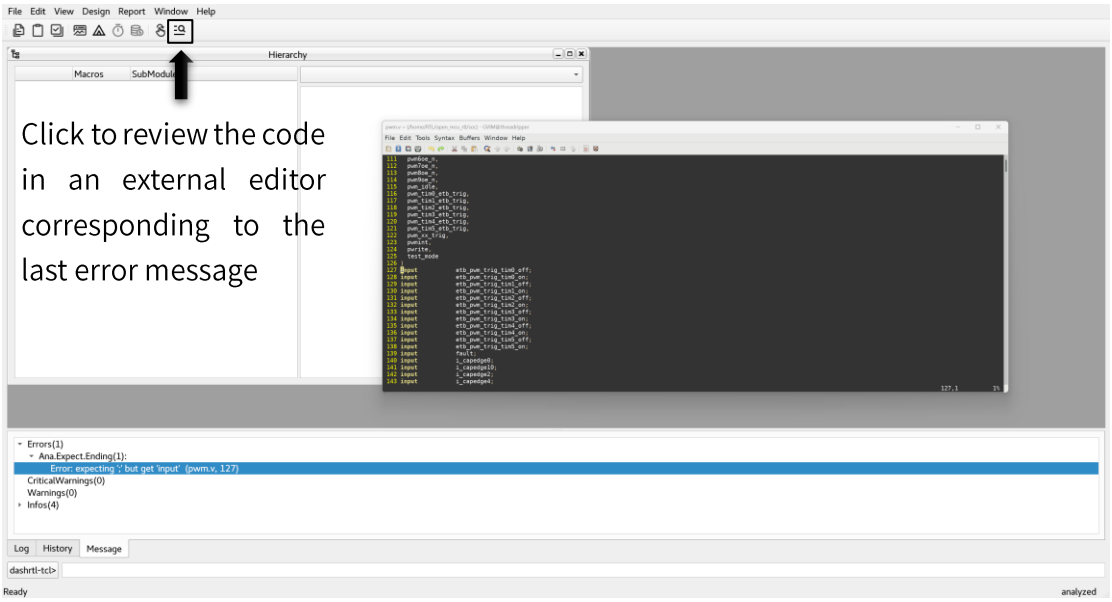
3-4 RTL Cross-probing with External Editor

DashRTL GUI also supports one-click cross-probing from a message to an external editor, allowing users to quickly modify problematic code and accelerate the debug process.

To cross-probe from the message in an external editor, right-click on the selected message and select `Crossprobe with XXXX`. This will automatically open the corresponding RTL file in an external editor specified in tool variable `crossprobe_exec`.

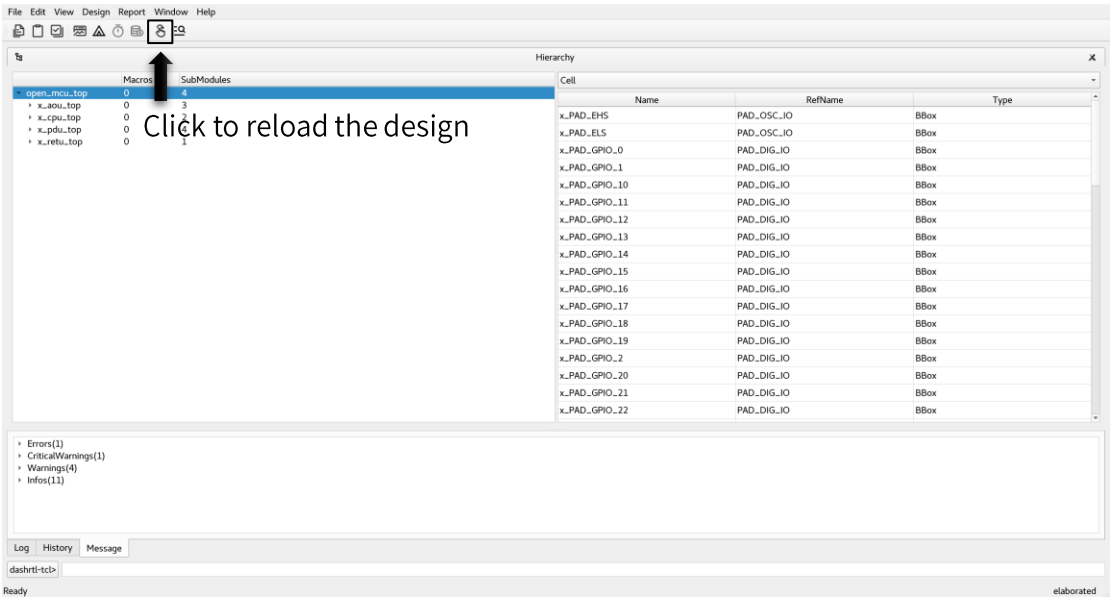


If the user only wants to review the code corresponding to the last error message, they can use the shortcut button in the toolbar, as shown in the below figure.



3-5 Reloading RTL Design

After the user modifies the problematic code, they can click the toolbar shortcut button to reload the design. DashRTL will execute the same `analyze` and `elaborate` commands to recompile the RTL.



4

4 DashRTL Features

The DashRTL function does not merely involve a simple check of RTL syntax. Instead, it breaks down the inspection of RTL design into multiple aspects.

First, before compilation, the `read_flist` command checks the file list. If the `flist` file is valid, the RTL compilation proceeds. The compilation process itself is divided into two aspects: syntax checking and synthesizability checking. After compilation is complete, users can explore the RTL design using Tcl commands, such as querying the hierarchies, parameters, cells, nets, and more.

This chapter will introduce the various features for checking RTL designs, including:

- RTL File List Checking
- RTL Syntax Checking
- RTL Synthesizability Checking
- RTL Design Exploration

4-1 RTL File List Checking

Before compiling RTL, DashRTL first checks the validity of the RTL file list beforehand to prevent users from discovering issues with the file list during the lengthy compilation process, which can save a significant amount of iteration time.

File path errors are common issues in the RTL file list. For example, in the file list below, the `rtl.list` file contains non-existent file and directory paths.

rtl.list

```
+incdir+/design/no_exist
/design/rtl/no_exist.v
```

In the following example, before executing the `analyze` command, the `read_flist` command first checks the legality of the file list. As shown, it reports the two non-existent file paths and errors out.

```
dashrtl-tcl> analyze -hdl_type verilog [read_flist rtl.list]
Warning: directory '/design/no_exist' not found. (PATH-NOTFOUND)
Error: file '/design/rtl/no_exist.v' not found. (PATH-NOTFOUND)
Line: 2
File: /home/dashthru/rtl.list
```

4-2 RTL Syntax Checking

DashRTL checks the syntax of RTL code during compilation to ensure it complies with Verilog/SystemVerilog HDL syntax. The syntax checks are divided into four severity levels:

- **Error**

The highest severity of compilation syntax error, making compilation to abort. The user must correct this immediately.

- **CriticalWarning**

Complies with HDL syntax but is likely an RTL design error.

- **Warning**

Complies with HDL syntax but may indicate a potential issue in the RTL design.

- **Info**

General reports during the compilation process, which the user may review as needed.

DashRTL's RTL compilation consists of two stages: `analyze` and `elaborate`, each responsible for different types of syntax checks. The two stages are equally important for RTL syntax checking, in which users should review both messages.

Messages from the RTL compilation can be reviewed through log files or the GUI. While reviewing the messages in a log file allows for easy searching with an editor, it is recommended to use the GUI to review the messages. The GUI classifies messages by their severity levels, providing a clearer overview of RTL syntax issues.

For more information on how to use the GUI to view messages and perform RTL cross-probing, please refer to Section 3: Graphical User Interface.

4-3 RTL Synthesizability Checking

RTL synthesizability checking is also a crucial part of RTL checking. Not all Verilog HDL syntax is synthesizable; many keywords are only supported for simulation purposes, such as `real`, `specify`, `primitive`, and others.

In projects, simulation model files are often mistakenly included in the RTL file list. DashRTL helps identify these errors early, preventing the issue from being discovered later during the synthesis phase.

4-4 RTL Design Exploration

- **Query Design Objects with the get_objects Command**

After the RTL compilation is complete, users can explore the RTL design based on their needs, such as querying cells, nets, pins, and other design objects. DashRTL uses the `get_objects` command to query RTL design objects including: cell, net, pin, port, module, parameter, task, function, lib, lib_cell, lib_pin.

Using `get_objects`, you can retrieve different types of objects as follows:

```
dashrtl-tcl> get_objects -type cell *reg*
out_reg[0] out_reg[1] out_reg[2]
_obj0
dashrtl-tcl> get_objects -type port clk
clk
_obj1
```

To meet user preferences, DashRTL also provides equivalent commands for different object types, as shown in the table below:

Original Command	Equivalent Command
get_objects -type cell	get_cells
get_objects -type net	get_nets
get_objects -type pin	get_pins
get_objects -type port	get_ports
get_objects -type module	get_modules
get_objects -type parameter	get_parameters
get_objects -type task	get_tasks
get_objects -type function	get_functions
get_objects -type lib	get_libs
get_objects -type lib_cell	get_lib_cells
get_objects -type lib_pin	get_lib_pins
get_objects -type lef_tech	get_lef_tech
get_objects -type lef_site	get_lef_sites
get_objects -type lef_cell	get_lef_cells
get_objects -type lef_pin	get_lef_pins
get_objects -type floorplan	get_floorplan

The usage of equivalent commands is as follows:

```
dashrtl-tcl> get_cells *reg*
out_reg[0] out_reg[1] out_reg[2]
_obj0
dashrtl-tcl> get_ports clk
clk
_obj1
```

- **Query Design Properties with the get_property Command**

In addition to design objects, users can query the properties of objects to access more related design information, such as `full_name` and `ref_name`.

The `list_property` command can be used to query the property list of a specified object type. The usage is as follows:

```
dashrtl-tcl> list_property -type cell
 full_name    (string)
 is_multibit  (bool)
 ref_name     (string)
 type         (string)
```

Then, the `get_property` command can be used to query the properties of a specified object. The usage is as follows:

```
dashrtl-tcl> get_property [get_cells x_PAD_GPIO_0] ref_name
PAD_DIG_IO
```

A

Appendix A DashRTL Launch Options

% **dashrtl** [-files | -f | -init] <startup_exec_tcl_or_python_script_files>
Execute Tcl or Python script files. Multiple files can be specified in quotes. The option can be used multiple times.

[-execute | -x] <startup_exec_tcl_or_python_scripts>
Execute Tcl or Python scripts. The option can be used multiple times.

[-pymode]
Launch in Python mode. If not used, DashRTL will launch in Tcl mode.

[-gui | -no_gui]
Choose whether to launch the GUI. If `-no_gui` is used, the `start_gui` command will be disabled.

[-lic_timeout <integer>]
Specify the time (in seconds) to wait for license checkout. If this time is exceeded, DashRTL will exit.

[-log <log_file_name <cmd_file_name>> | -no_log]
Use `-log` to specify log and cmd file names. If not used, the default names are `dashrtl.log` and `dashrtl.cmd`. Use `-no_log` to prevent the creation of these files.

[-no_overwrite]
By default, DashRTL overwrites the previous log files. If this option is used, DashRTL will generate log and cmd file names with sequential numbers (e.g., `dashrtl.log2`, `dashrtl.log3`, etc.).

[-version]
Displays the DashRTL tool version.

[-help | -h | --help]
Displays the DashRTL help information.

[-run <design_read_options>]
All options after this are considered `<design_read_options>`, not DashRTL's launch options. Therefore, it must be the last option used to launch DashRTL.

-run [-filelist | -f] <hdl_filelists>
Specifies the filelist files.

-run [-verilog|-v01|-v2001|-v2k|-v05|-v2005|-sverilog|-sv09|
-sv2009|-sv12|-sv2012|-sv17|-sv2017|-sv23|-sv2023|
-netlist|-autoext]
Specifies the HDL standard for compilation.

-run [-v <ignore_files>]
Specifies simulation model files which is ignored during compilation.

-run [+incdir+<path>]
Specifies the search path for include files.

-run [+define+<macro_id>(=<value>)]
Defines macros before compilation.

-run [+hdlext+<ext>=<hdlstd>]
Specifies customized automatic extension matching for HDL standards.

-run [-top <design_name>]
Specifies the top module name for elaboration.

-run [-liblist <timing_library_filelists>]
Specifies the library list files.

-run [-search_path <paths>]
Specifies the search paths.

-run <hdl_files>
Specifies the HDL files for compilation.

B

Appendix B DashRTL Tool Variables

Tool Variable Name	Default Value
crossprobe_exec	gvim +{line} {filepath}
Set the custom third-party editor options for cross-probing.	
elab_allow_bbox	true
Set whether to allow RTL unresolved instances. If `true`, black boxes will not trigger an error.	
get_objects_fast_match	true
Set whether to use a fast match algorithm for the `get_objects` command.	
get_objects_of_sequence	object_pointer cell net pin port module
Set the priority sequence for matching object types in the `get_objects -of_object` option. Matching will be done in this specified order.	
hdl_allow_static_task	true
Allow local latch behavior of variables inside static tasks for Verilog HDL and SystemVerilog HDL. Setting to 'false' will disable static task processing and take all tasks as automatic.	
hdl_default_ext	verilog=.v sverilog=.sv netlist=.vg
Set the default file extensions for different HDL standards in `autoext` setting.	
hdl_default_std	verilog=v05 sverilog=sv09
Set the default standards for Verilog and SystemVerilog in the compiler.	
hdl_inst_array_pre_postfix	prefix=[postfix=]
Set the prefix and postfix for block name arrays.	

hdl_max_limit	loop=65536 recursion=512
`Loop` : Set the maximum iterations for `for`, `while`, and `repeat` loops. `Recursion` : Set the maximum number of recursive calls for `function` and `task`.	
hdl_soc_integration_mode	**false**
Control whether to issue system connection related warnings as critical warnings in SoC integration mode.	
hier_separator	**inst=/ inst_alt=| genblk=. blk=.**
Set the hierarchy separator.	
host_max_threads	**0**
Set the maximum number of CPU threads. If set to `0`, the number of threads will be based on the number of CPUs in the current machine.	
link_hdl_lib_priority	**error**
Control which to reference when a cell exists both in HDL description and timing library. `error` : Not allowed `lib` : Use timing library `hdl` : Use HDL description	
search_path	**.**
Set the search path for source files.	
tcl_shell_print_limit	**1000**
Set the character limit for printing in the Tcl shell.	

C

Appendix C DashRTL Tool Commands

The following command list includes DashRTL tool commands, available in both Tcl and Python modes.

This list contains only the DashRTL tool-specific commands and does not include native Tcl commands (e.g., set, string, lappend, etc.) or Tcl extension commands (e.g., define_proc_constraints, add_to_objects, etc.).

For information on native Tcl commands and Tcl extension commands, please refer to the **FusionShell User Guide** (FusionShell_UserGuide_EN.pdf).

Tcl Command	Python Command	Function
alias	tcl.proc.alias()	Create an alias name
analyze	tcl.proc.analyze()	Read RTL files
apropos	tcl.proc.apropos()	Search for command names related to a specified string
crossprobe	tcl.proc.crossprobe()	Use external editor for RTL cross-probing
current_design	tcl.proc.current_design()	Set the working design
elaborate	tcl.proc.elaborate()	Perform design elaboration
error_info	tcl.proc.error_info()	Print error information
get_cells	tcl.proc.get_cells()	Return cell objects
get_functions	tcl.proc.get_functions()	Return function objects
get_lib_cells	tcl.proc.get_lib_cells()	Return lib_cell objects
get_lib_pins	tcl.proc.get_lib_pins()	Return lib_pin objects
get_libs	tcl.proc.get_libs()	Return lib objects
get_modules	tcl.proc.get_modules()	Return module objects
get_nets	tcl.proc.get_nets()	Return net objects
get_object_name	tcl.proc.get_object_name()	Return object names
get_objects	tcl.proc.get_objects()	Return objects of the specified type

get_parameters	tcl.proc.get_parameters()	Return parameter objects
get_pins	tcl.proc.get_pins()	Return pin objects
get_ports	tcl.proc.get_ports()	Return port objects
get_property	tcl.proc.get_property()	Return object properties
get_tasks	tcl.proc.get_tasks()	Return task objects
get_tool_var	tcl.proc.get_tool_var()	Return the value of a tool variable
list_property	tcl.proc.list_property()	Return the property name list of a specific object type
list_tool_var	tcl.proc.list_tool_var()	Return the list of all tool variable names
man	tcl.proc.man()	Enter manual page
pymode	tclmode()	Switch between Tcl/Python modes
read_flist	tcl.proc.read_flist()	Read and parse the flist file
read_libs	tcl.proc.read_libss()	Read LIB files
read_lefs	tcl.proc.read_lefs()	Read LEF files
read_def	tcl.proc.read_def()	Read DEF files
reload_design	tcl.proc.reload_design()	Reload the RTL design
reset_design	tcl.proc.reset_design()	Clear all compiled RTL design
reset_tool_var	tcl.proc.reset_tool_var()	Reset a tool variable to its default value
set_tool_var	tcl.proc.set_tool_var()	Set a tool variable
start_gui	tcl.proc.start_gui()	Open GUI
stop_gui	tcl.proc.stop_gui()	Close GUI