cādence®

Innovus Stylus Common UI Quick Start Guide

Product Version 21.10
May 2021

Contents

About This Manual

This document applies to and provides information about the Innovus Stylus user interface.

How This Document Is Organized

This manual is organized into chapters that cover broad areas of Innovus Stylus Common UI (Common UI) functionality. The manual is aimed at users who are new to Common UI. Each chapter contains topics that may address one or more of the following areas:

- Overview of the Stylus Common UI functionality
- Key areas of Stylus Common UI, including design initialization, database access, flow, and timing reports.

If you are migrating from legacy to the Stylus Common UI version of the software, refer to the *Innovus Stylus Common UI Migration Guide* instead.

Related Documents

For more information about Stylus Common UI, see the following documents. You can access these and other Cadence documents using the Cadence Help documentation system.

Innovus Stylus Common UI Documentation

- Innovus Stylus Common UI Migration Guide
  Provides information on migrating from legacy to the Stylus Common UI version of the Innovus software.
- What's New in Innovus Stylus Common UI
  Provides information about new and changed features in this release of the Innovus family of products.
- Innovus Stylus Common UI User Guide
  Describes how to install and configure the Innovus Stylus Common UI software, and provides strategies for implementing digital integrated circuits.
- Innovus Stylus Common UI Text Reference Manual
  Describes the Innovus Stylus Common UI text commands, including syntax and examples.
- Innovus Stylus Common UI Menu Reference
  Provides information specific to the forms and commands available from the Innovus Stylus Common UI graphical user interface.
- Stylus Common UI Database Object Information
  Provides information about Stylus Common UI database objects.
- Innovus Stylus Common UI Mixed Signal (MS) Interoperability Guide
  Describes the digital mixed-signal flow using Innovus Stylus Common UI.

Related Documents

For more information about Stylus Common UI, see the following documents. You can access these and other Cadence documents using the Cadence Help documentation system.

Voltus Stylus Common UI Documentation

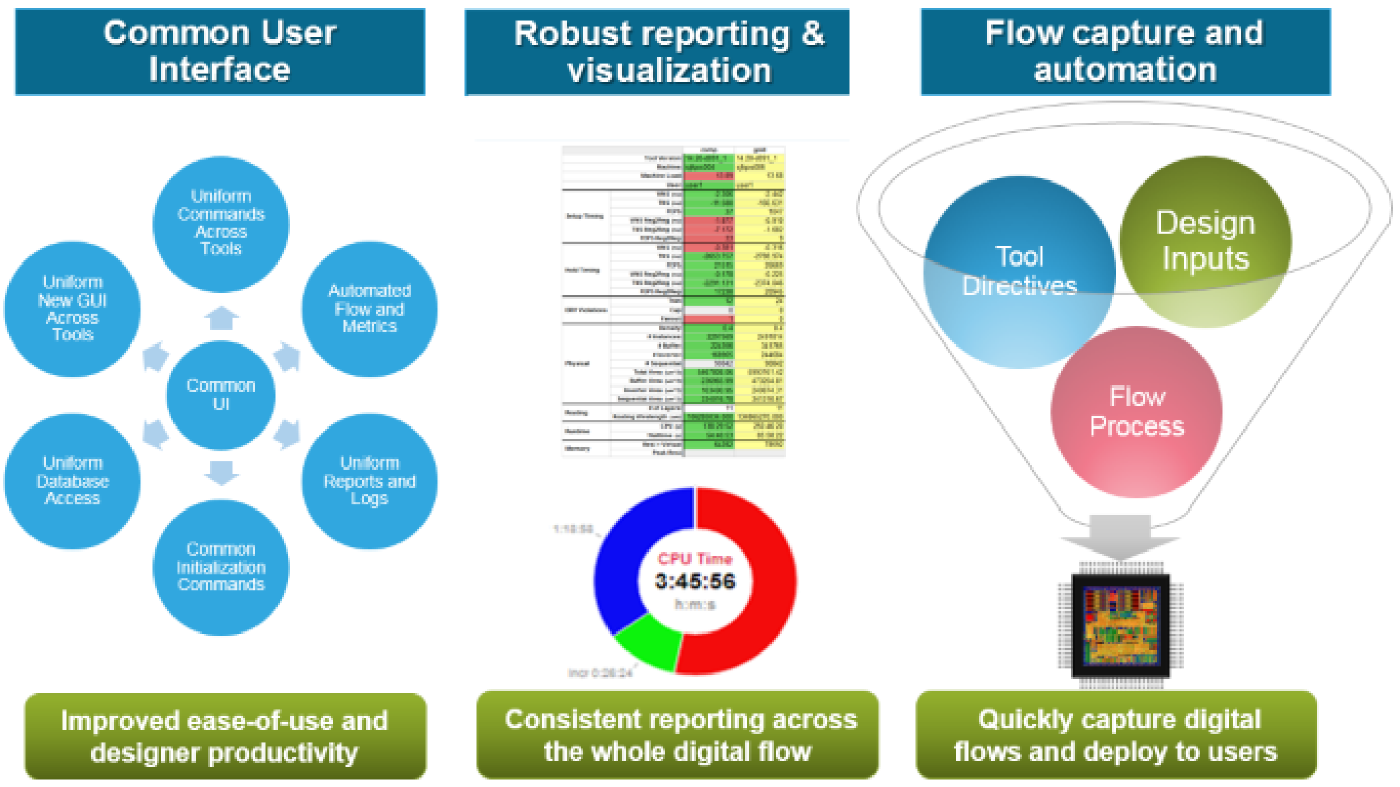
- Voltus Stylus Common UI Text Reference Manual (Limited Access)
  Describes the Voltus Stylus Common UI text commands, including syntax and examples.
- Stylus Common UI Database Object Information (Limited Access)
  Provides information about Stylus Common UI database objects.

1

Introduction to Stylus

Stylus is an infrastructure that offers three significant features:

- Stylus Common User Interface offering consistent commands across the whole digital flow
- Stylus Unified Metrics for capturing and reporting
- Stylus Flow Kit for design flow capture and deployment


Common User Interface
Robust reporting & visualization
Flow capture and automation
Uniform Commands Across Tools
Uniform New GUI Across Tools
Automated Flow and Metrics
Common UI
Uniform Database Access
Uniform Reports and Logs
Common Initialization Commands
CPU Time
3:45:56
h:m:s
Tool Directives
Design Inputs
Flow Process
Improved ease-of-use and designer productivity
Consistent reporting across the whole digital flow
Quickly capture digital flows and deploy to users

The Stylus Common User Interface (Common UI from this point onwards) has been designed to be used across Genus, Joules, Modus, Innovus, Tempus, and Voltus tools. By providing a common interface from RTL to signoff, Common UI enhances user experience by making it easier to work with multiple Cadence products.

Common UI uses consistent command naming and aligns common implementation methods across Cadence digital and signoff tools. For example, the processes of design initialization, database access, and metric collection are consistent across tools. In addition, Common UI uses shared methods to run, define, and deploy reference flows across core implementation and signoff products. You can take advantage of consistently robust RTL-to-signoff reporting and management, as well as a customizable environment in Common UI.

The key features of Common UI are:

- **Common database access methods in all tools**

  You can use a single command, `get_db`, to query and filter all database attributes. The `set_db` and `reset_db` commands are the companion commands to set values.

  For more information, see Database Access and Handling with get_db and set_db.

- **Common initialization flow with common MMMC file**

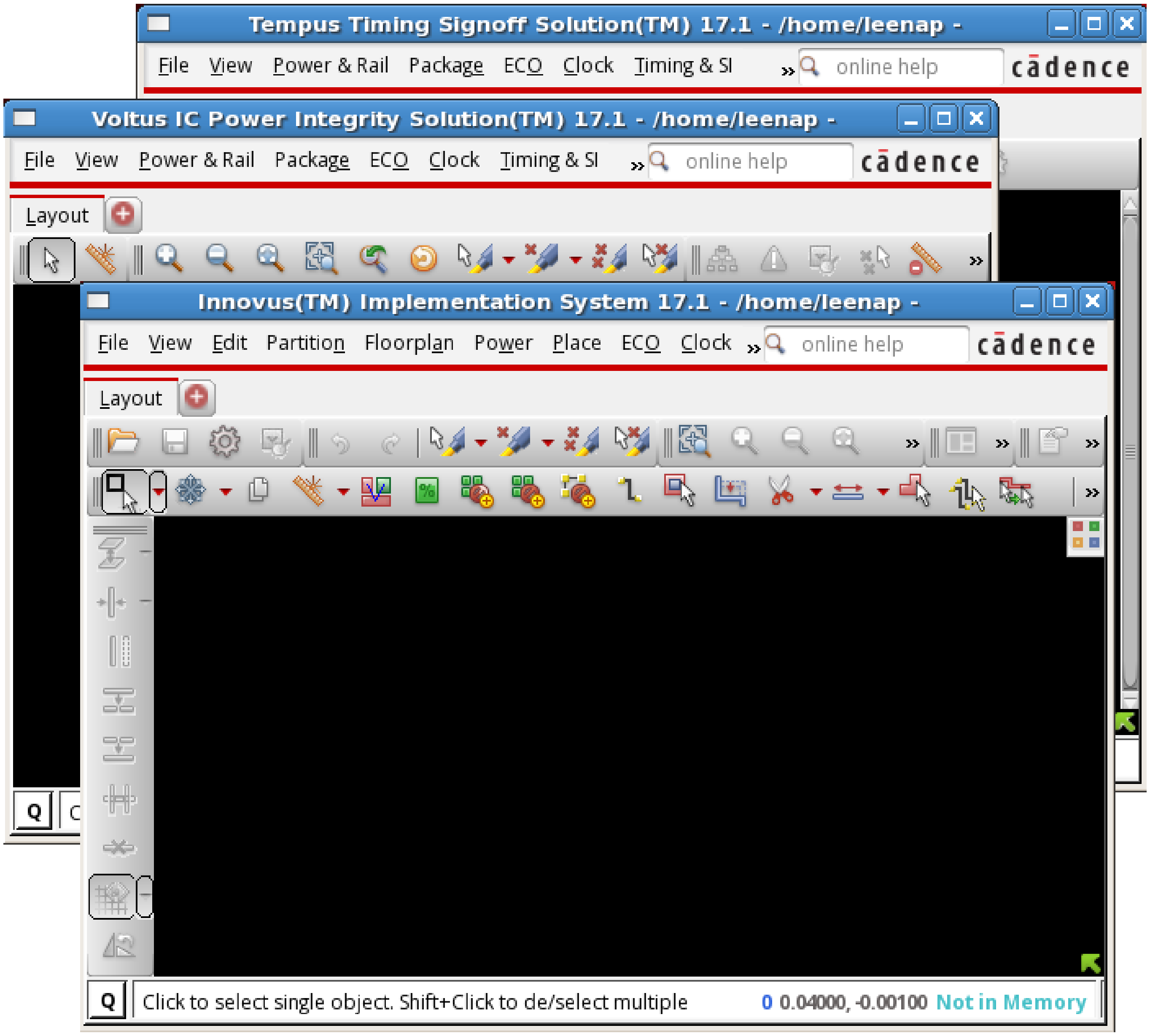
  In Common UI, the initialization flow is the same for all the tools and uses the same MMMC file.

  The `timing_condition` object in the MMMC syntax in the MMMC file makes it possible to bind power_domains to the MMMC timing data easily. With this object, no timing data is required from power_intent files.

  For more information, refer to Stylus Common UI Initialization and the following initialization-related commands in the *Text Command Reference*:

  - `read_mmmc`
  - `read_physical`
  - `read_netlist`
  - `read_power_intent`
  - `init_design`

- **Common GUI**
  Common UI provides a uniform graphical user interface across tools with consistent menus, widgets, and forms. The menus are organized in the same order, although each tool only shows the menus or sub-menus for commands supported by that tool. The widgets and forms for common commands and windows are the same.


Tempus Timing Signoff Solution(TM) 17.1 - /home/leenap -
File View Power & Rail Package ECO Clock Timing & SI
online help
cādence
Voltus IC Power Integrity Solution(TM) 17.1 - /home/leenap -
File View Power & Rail Package ECO Clock Timing & SI
online help
cādence
Layout
Innovus(TM) Implementation System 17.1 - /home/leenap -
File View Edit Partition Floorplan Power Place ECO Clock
online help
cādence
Layout
Q
Click to select single object. Shift+Click to de/select multiple
0 0.04000, -0.00100 Not in Memory

- **Common timing reports**

  In Common UI, the `report_timing` command:

  - Allows fast identification of issues related to clock definition, optimization, and constraints.
  - Facilitates analysis and debugging by making issue identification simple and by providing similar reports for different tools. The timing reports also support the cut and paste option.
  - Produces an efficient and consistent report format. This includes aligning similar data

from launch and capture paths and aggregating useful information that may not be visible in detailed paths.

For more information, see the Timing Reports section.

- **Flow kit**

  The set of flow commands, including `create_flow` and `create_flow_step`, support user flows easily. The `flow` and `flow_step` objects store the state of the flow with the database.

  For more information, refer to the Foundation Flows section.

- **Unified metrics**
  Common UI enables you to generate unified and holistic metrics, which make it easy to summarize and compare results and lead to effective debugging. For information on metric commands and attributes, refer to the Unified Metrics section.

- **Consistent command naming across tools**
  Common UI uses consistent command naming conventions across tools.

  - All commands and options are in lower case with an underscore as a separator. For example:

    ```
    innovus 1> delete_area_io_fillers -help
    ```

    ```
    Usage: delete_area_io_fillers [-help] -cell <fillerCellName> [-inst
    <fillerInstanceName>]
    [-prefix <prefix>] {-area_io_row_cluster <aioRowClusterName> | -
    all_area_io_row_cluster }
    ```

- **Consistent command usage across tools**

  - All root attributes are accessed and modified the same way across tools with the `get_db`, `set_db`, and `reset_db` commands.

  - A single `help` command can now be used to obtain information about attributes. commands, messages, and objects. The `help` command facilitates discovery and self-help. It supports pattern matching across command, object and attribute names.

    For more information on using help, see `help`.

  - Common UI provides a simple selection, deselection, and deletion mechanism for objects. A single command, `select_obj`, can be used to select different types of objects. Similarly, you can use the `deselect_obj` command to deselect and the `delete_obj` command to delete different object types, such as bump, bus_guide, blockages, and text.

- **Uniform logging**

  Command logging plays a vital role in the debug process. Stylus Common UI provides uniform logging across products by logging all commands in the log file, irrespective of whether they are issued interactively or through startup files and scripts.

  For more information, refer to the Uniform Logging section.

Related Information

- Database Access and Handling with get_db and set_db
- Stylus Common UI Initialization
- Foundation Flows
- Timing Reports
- Unified Metrics
- Uniform Logging
- Uniform Startup

2

Stylus Common UI Initialization

Initialization is the first step in Stylus Common UI. It is the process of reading libraries, physical data, HDL/netlist, power intent, and constraints. After initialization, the design is in a consistent state, and is ready for manipulation or timing.

In Common UI, there is a single initialization methodology that works across Genus, Innovus, and Tempus. Common UI provides two options in the initialization strategy, one for MMMC (Multi-Mode, Multi-Corner) and one for a “simple” methodology. The simple methodology does not require detailed MMMC.

The Common UI initialization strategy mostly has autonomous commands, with the db being in a consistent state after each initialization step is completed. The exception to this are power intent and full MMMC timing commands. These commands are deferred until an initialization command is issued.

Types of Initialization

The following table describes the various initialization types.

Types	Description
Setup	Sets the Common UI database attributes with the specified information. However, it does not populate the database. Some basic error checking may be done, but only on the database objects.
Load	Loads the database with the specified objects. Error checking on the data being loaded can be done. The database is consistent, but is not yet ready for execution.
Init	Populates the database with the full object set and does binding between timing modes and domains. The database is fully consistent and Common UI is ready for execution.

Basic Flow

The steps in the basic flow of Common UI Initialization are explained in the table below.

Step	Tasks Performed
1. Timing	During the Timing step, the timing libraries and constraints are specified. The Timing step is a combination of Setup and Load. The data that is loaded are the library sets pointed to by the `set_analysis_view` in the MMMC file.
2. Physical	In this step, the physical data is loaded into the database. Both LEF and OpenAccess (OA) data are supported. Captables and QRC files are defined in the MMMC file with the `create_rc_corner` command. Therefore, loading of these is deferred to init when the full MMMC data is available. This step is optional because Synthesis and STA do not always require physical data. It is also optional for implementation to support timing regressions. The Physical Step is a Load function. All data specified in this step is loaded into the database.
3. Design	In the Design step, the design data is in the form of a Verilog gate level netlist, or in the form of Verilog, VHDL, or System Verilog RTL. In the case of a netlist, a single command will read the netlist as well as elaborate. After this single command, the database is fully populated with the design data. In the case of RTL, there is a two-step process – reading the RTL and then elaborating. Reading the RTL is with the specific language and parameters. Elaborating is assembling the RTL into a fully-populated and consistent design view. Elaboration can also do additional RTL reads to resolve missing blocks, depending on the RTL search paths. Once elaboration is complete, the design is consistent. An optional floorplan can be loaded after the design is elaborated. Both the Innovus floorplan file format and the DEF file format are supported. If a floorplan file format is used, it must be read before the power intent is read. This ensures that the power intent power rails and definitions are used. The Design Step is always a Load function.

4. Power	During the Power Step, the power intent is set up in the database attributes. Both CPF and IEEE 1801 are supported. In Common UI, timing intent is removed from power intent. As a result, if the CPF contains timing intent (nominal conditions or analysis views), an error will be issued unless the user specifies to ignore the timing. The Power Step is optional and only required if the design needs a power_intent file. Only db attributes are set during this step. Binding of power domains, loading of power cells, etc is not done until the Init Step.	
5. Init	The Init step is when the full database is prepared for execution. The constraints are loaded into the correct mode, the power intent is loaded and applied to the appropriate instances, and the operating conditions are applied to the power domains. Extensive error checking is performed, including checking of objects, and checking for completeness of data. Once the init step is complete, the design is fully ready for execution.	

Related Information

- Flows in Stylus Common UI
- Initialization Considerations
- Initialization Flow in MMMC Mode
- Initialization Flow in Non-MMMC Mode

Flows in Stylus Common UI

The following table shows the different flows in Common UI and the tools that they support.

Name	**Description**	**Supported Tools**		
		Genus	**Innovus**	**Tempus**
Full MMMC	MMMC timing + physical	√	√	√
Phys only	No `read_mmmc` or `read_power_intent`		√	
MMMC only	No physical	√	√	√
Simple timing	Only read_libs and read_sdc	√		√

Simple physical	Only read_libs, read_physical -oa_ref_libs, read_qrc, and read_sdc	√		

The following table shows the various commands used at different steps in different tools.

Note: The highlighted commands are optional for the particular tool.

Command	Genus	Innovus	Tempus
Timing	`read_mmmc` or `read_libs`	`read_mmmc`	`read_mmmc`
Physical	`read_physical –oa_ref_libs` or `read_qrc`	`read_physical –oa_ref_libs`	`read_physical –oa_ref_libs`
Design	`read_hdl`/`elaborate` or `read_netlist`	`read_netlist`	`read_netlist`
Floorplan	`read_def`	`read_floorplan` or `read_def`	
Power	`read_power_intent`	`read_power_intent`	`read_power_intent`
Constraints (simple mode)	`read_sdc`		`Innovus_read_sdc`
Initialization	`init_design`	`init_design`	`init_design`

Related Information

- Initialization Considerations
- Initialization Flow in Non-MMMC Mode
- Initialization Flow in MMMC Mode
- Stylus Common UI Initialization

Initialization Considerations

You need to consider the following from the design initialization perspective:

- **Synthesis Considerations**
    - Library Data
      HDL elaboration requires the library data to prevent unresolved references. HDL can contain references to macros and standard cells. The current elaborator makes assumptions about port directions if a library cell is not present, and these assumptions can dramatically impact the resulting netlist. As a result, library data is needed before the design elaboration.
    - Physical Data
      Synthesis does not always require physical data, that is, LEF, OA or QRC files. Reading this data is optional.
    - Design Manipulation
      HDL elaboration can be quite complex due to various factors such as different languages and design parameters. For example, in some flows the full chip is elaborated to determine the parameters for the CPU. The design is then re-elaborated at the CPU level with the parameters. In addition, you may ungroup or rename the hierarchy before applying power intent or timing intent. Therefore, the initialization flow must support breaks for the commands after design import and before applying timing or power intent.
    - Simple Mode
      Both a simple setup, and an MMMC setup are supported.
- **Implementation Considerations**

  Back end is relatively simple from an initialization perspective. If timing is needed, only an MMMC setup is allowed, and design manipulation are not required. However, the design, timing, and power intent should be fully resolved before execution.
    - Immediate Implementation Load

      In Common UI, the implementation immediately loads some data. In Common UI, specific data is loaded into the database at command invocation. This immediate load will be common across Synthesis, Implementation, and STA.
    - Physical Only Flow
      Common UI supports a physical only flow – using OA/LEF without the logical or timing libraries. This is enabled by reading the physical data first without reading the timing data. When a physical-only flow is used, the timing setup and power intent setup is skipped.
- **STA Considerations**
    - Simple Mode

STA needs to support a simple non-MMMC mode. The mechanism will be identical between synthesis and STA.

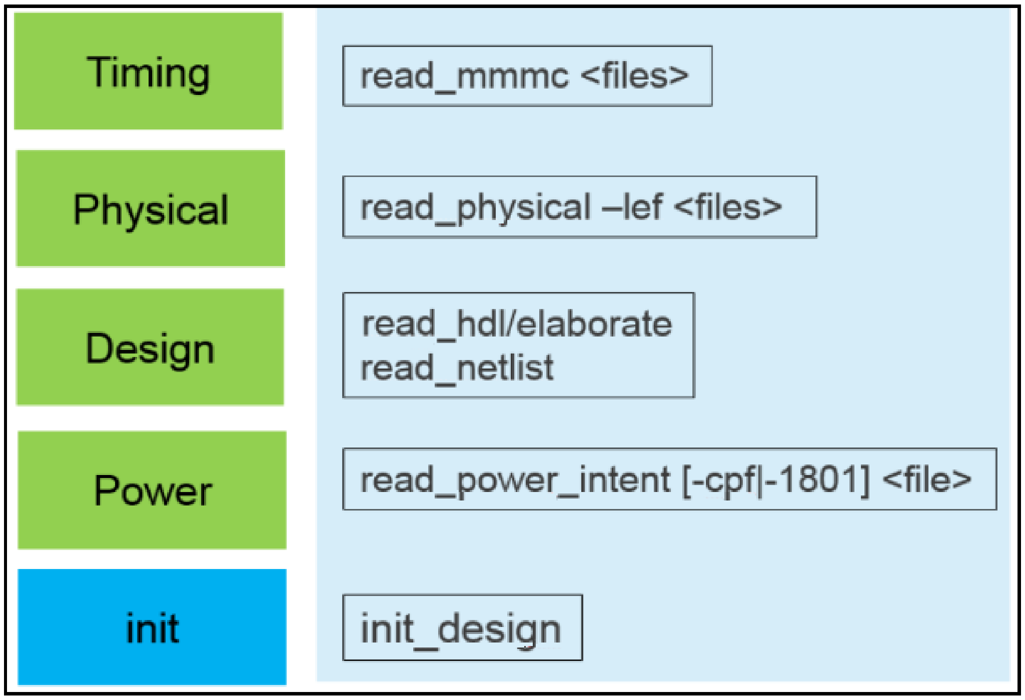
- Physical Data
  STA does not always require physical data, such as OA and QRC. Reading this data is optional.

Related Information

- Flows in Stylus Common UI
- Initialization Flow in Non-MMMC Mode
- Initialization Flow in MMMC Mode
- Stylus Common UI Initialization

Initialization Flow in MMMC Mode

The following image shows the initialization flow in the MMMC Mode.



- **Timing Step**

In the Timing Step, the full MMMC objects are set up, that is, populating the basic MMMC information attributes. Synthesis needs the library information before the Design Step, therefore, the Timing Step also loads some library information. The library data to be loaded is pointed by the `set_analysis_view` command in the MMMC file.

- **Physical Step**
  During the Physical Step, the physical libraries, OA or LEF, are loaded into the tool's database using the `read_physical -oa_ref_libs` command. The database will be loaded with the specified objects, and will be available for query.
- **Design Step**
  During the Design Step, the design (netlist or RTL) is loaded into the database. There are two command sets:
  - read_netlist
    - This is strictly for Verilog gate-level netlists
    - After issuing this command, the design is ready for querying in the database
    - This command can be used in Synthesis, Backend, and STA
  - read_hdl and elaborate
    - This command pair is used for Synthesis.
    - The read_hdl command supports Verilog, VHDL, and System Verilog. It parses the HDL, does basic syntax check, and builds the basic structure.
    - The elaborate command assembles the HDL objects, loads in missing HLD objects based on the search paths, and links the design. After elaboration, the design is ready for querying in the database

The design step is a pure load step because in this step the database objects are populated, and can be queried and manipulated (i.e ungrouped, renamed) after this step is completed. Full error checking on the design details will be performed. This includes, but is not limited to, unresolved references, port mismatches, and empty modules.

- **Power Step**
  During the Power Step, the power intent attributes are set up and populated. This reads the specified CPF or UPF, does some basic error checking (primarily syntax checking), and sets the appropriate attributes. The design database is NOT modified – this will be done during the Init Step. In Common UI, timing intent is separated from power intent. This is not a problem with UPF as UPF does not allow a mechanism to specify timing.
  This step is a setup only step. In this step, the tool database is not loaded with the power

intent objects, and no binding takes place. The reason for deferring to the Init Step is to allow further design manipulation. In this step the error checking is limited to CPF or UPF syntax checking.

After this command is issued and the user is satisfied with the results, the next step is initialization.

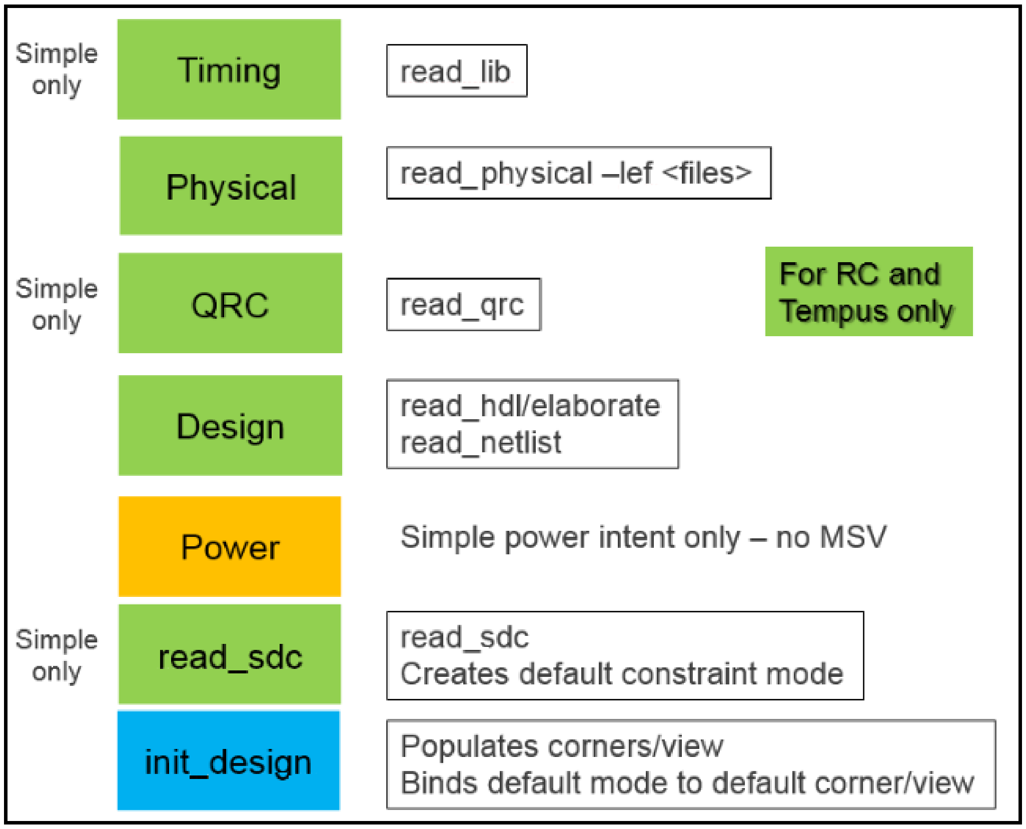
- **Init Step**
  The `init_design` command is an explicit command that causes the full system to be initialized. With this step, the remaining MMMC timing information is loaded, the power intent is initialized, the modes are bound to the power domains, and the design is prepared for timing. Once init is complete, the database is completely consistent and is ready for execution.

Related Information

- Initialization Considerations
- Flows in Stylus Common UI
- Initialization Flow in Non-MMMC Mode
- Stylus Common UI Initialization

Initialization Flow in Non-MMMC Mode

To support a simple non-MMMC initialization, there will be a simple initialization sequence consisting of simple commands. Effectively these simple commands are important for the MMMC. The data set will be stored in the same MMMC attributes, and will be queried in the same way.



Timing Step

In this step, libraries are loaded with the `read_libs` command. This command sets the appropriate MMMC attributes with default names.

Physical Step

In the Physical Step, LEF or OA databases are loaded via `read_physical -oa_ref_libs` as done in the MMMC Physical Step. In addition, QRC files can be loaded with `read_qrc`. This will effectively do a `create_rc_corner -name default_emulate_rc_corner`, and will update the delay corner to include this default RC corner.

Design Step

This step is identical to the MMMC Design Step. Both command sets (read_netlist or read_hdl/elaborate) are supported. The `read_def` command is also supported as an optional step.The `read_floorplan` command is not supported because Innovus does not support the simple mode.

Power Step

Power intent is not supported in the Simple mode. If `read_libs` was used to initialize the libraries, `read_power_intent` will return an error.

SDC Step

The simplified setup adds a step `read_sdc` to load in the constraints. This command accepts a list of constraints and populates the default constraint mode. No initialization is done with this step. Multiple `read_sdc` are supported. By default, each file will be appended to the default constraint mode. If you issue `-reset`, then the constraint mode will be reset, and only the new files applied.

Init Step

The Init step is the same as in the MMMC setup through `init_design`. While read_sdc and init_design could be combined into a single step, maintaining init_design as a separate function is good for initialization consistency. When init_design is used, the constraints are loaded and bound to the design elements as with the MMMC flow. Once the design has been initialized, the design is consistent and ready for timing, reporting, and implementation.

The `init_design` command should be issued only once. After the design is initialized, all subsequent updates will be done via attributes or MMMC commands.

Once the design has been initialized via init_design, the design is ready for execution. Execution can include incremental updates to timing and power.

Related Information

- Initialization Considerations
- Flows in Stylus Common UI
- Initialization Flow in MMMC Mode
- Stylus Common UI Initialization

3

Database Access and Handling with get_db and set_db

The Stylus Common UI uses the `get_db`/`set_db` commands to provide a single and common way to access/query all the database information. These commands also give access to all the data available from the SDC collection-based commands such as `get_cells`, `get_pins`, `get_property`, and `set_property`.

The `get_db` and `set_db` commands support various chaining, filtering, and matching options with an easy to use model. These commands can access most object types directly from the root.

Note that `get_db` also allows string pattern matching directly on any object that has a `.name` attribute. This enables fetching the objects directly by name. For example:

```
get_db insts i1/* #returns all inst objects with .name matching i1/*

get_db insts -if {.name == i1/*} #returns all inst objects with .name matching i1/*

get_db insts .name i1/* #returns all inst names (not objects) matching i1/*
```

Note: `get_db` requires a space before the `.<attribute>` name.

Help on Database Attributes

You can use help to find the attributes and their help description for any given `obj_type`, as below:

```
help -obj pin     #note, "help pin:" also works, the ":" means only -obj help is
desired, so then -obj is not needed
Attributes:
arrival_max_fall(pin)
Returns the latest falling arrival time ...
arrival_max_rise(pin)
...
```

Pattern matching also works for help:

```
help pin: *slack*
Attributes:
slack_max(pin) Returns the worst slack across all concurrent MMMC views ...
slack_max_edge(pin) Returns the data edge of the path responsible ...
```

```
...
```

To find all the attributes with "`slack`" in their name, run:

```
help -attribute * *slack*
Attributes:
cts_move_clock_nodes_for_slack(clock_tree)
When set, CCOpt will consider moving nodes in the clock tree with poor ...
slack_max(pin) Returns the worst slack across all concurrent MMMC views for Setup-style
...
...
```

You can also use the <tab> key to see the attribute names of the current object when entering a `get_db` command:

get_db $my_obj .<tab>

Browsing the Database

You can use pattern matching for an attribute name to see multiple attribute values for many objects at the same time. For example:

get_db $my_pins .*

Or use this to see just a few attributes of every pin:

```
get_db $my_pins .capacitance_max*
Object: pin:top/CG/BC1/Y
capacitance_max_fall: 1.0
capacitance_max_rise: 1.1
Object: pin:top/CG/BC2/A
capacitance_max_fall: 1.1
capacitance_max_rise: 1.2
...
```

If some of the values are not computed (like timing-graph data), and you want to force them to be computed anyway, run:

```
get_db $my_pins .slack_max* -computed
Object: pin:top/CG/BC1/Y
slack_max: 9227.4
slack_max_edge: rise
slack_max_fall: inf
slack_max_rise: 9227.4
...
```

Object Chaining

Object chaining enables you to link to the related objects allowed.

Note: `get_db` requires at least one space before the "."

For example:

- Pins or ports that drive a specific pin:

```
get_db pin:top/rst_reg/D .net.drivers
```

- Pins or ports that are loads of a specific pin:

```
get_db pin:top/rst_reg/D .net.loads
```

Examples

- The following command finds all the insts that start with i1/i2/ and end with _buf:

```
get_db insts i1/i2/*_buf
inst:top/i1/i2/test_buf inst:top/i1/i2/i3/test_buf ...
```

- The following command counts the number of repeater cells:

```
llength [get_db insts -if { .base_cell.is_buffer || .base_cell.is_inverter }]
```

- The following `get_db` command uses `-foreach` to count the number of `base_cell` used in the `netlist` inside the `cell_count` Tcl array.

```
get_db insts .base_cell -foreach {incr cell_count($obj(.name))}
array get cell_count     #write out the array with the counts
BUF1 20 AND2 30 ...
```

- The following command defines a user-defined attribute, and then adds it on a net based on its fanout:

```
define_attribute high_fanout \
-category test \
-data_type bool \
-obj_type net
set_db [get_db nets -if {.num_loads > 20}] .high_fanout true
get_db net:clk .high_fanout
true
```

- The following command counts all the pins below the hinst, or all pins at one level of the hierarchy

```
llength [get_db hinst:tdsp_core/ALU_32_INST .insts.pins]
llength [get_db hinst:tdsp_core/ALU_32_INST .local_insts.pins]
```

- The following command filters out the buffer cells, and counts them in `buf_count`:

```
get_db insts -if {.base_cell.is_buffer==true} \
-foreach {incr buf_count($obj(.base_cell.name))}
array get buf_count
BUF1X 20 BUF2X 30 ...
```

- The following command returns all the placed sequential cells:

  ```
  get_db insts -if {.base_cell.is_sequential==true && .place_status==placed)
  ```

Related Information

For more information, see:

- `get_db`
- `set_db`
- Stylus Common UI Database Object Information

4

Dual Ported Objects

The Stylus Common UI uses the Tcl Dual Ported Objects (DPO) concept. A DPO is a Tcl_Obj in the Tcl C++ programming interface. The object pointer is kept as a C++ pointer inside Tcl unless Tcl forces it to be converted to its dual string form. In normal usage, it is never converted to a string which is more efficient, but if you run `puts $var` or return the value to the shell and echo to the `xterm`, it is converted to its string form. Note that:

- The string form is normally the name of the object preceded by the `obj_type`, so a layer object string form might look like `layer:metal1`.
- A design object name has the current design name included, so an instance named `i1/i2` in design top, looks like `inst:top/i1/i2`.
- An object with no name like a wire, just shows the pointer hex-value like `wire:0x22222222`.

The `get_db` and `set_db` commands allow both a DPO list and a collection from SDC commands like `get_pins` as input, but only returns objects in a DPO Tcl list.

For example, if your design is `top`, then the output is like:

```
set i1_insts [get_db insts i1/*]
```

```
inst:top/i1/i2 inst:top/i1/i3 ...
```

You can also use the DPO name directly for input. So the last three queries mentioned above could be done this way:

Using Returned DPO Value	Using DPO Name Directly
`get_db [get_db layers metal1] .width`	`get_db layer:metal1 .width`
`get_db [get_db pins i1/p1] .arrival_max_fall`	`get_db pin:top/i1/p1 .arrival_max_fall`
`get_db [get_db clock_trees clk1] .cts_target_max_transition_time_early`	`get_db clock_tree:top/clk1 .cts_target_max_transition_time_early`

Related Information

For more information, see:

- `get_db`
- `set_db`
- Stylus Common UI Database Object Information

5

Significant get_db Objects

Most of the Stylus Common UI object names use all lower-case characters, with "_" for word separators. Following are few significant `get_db` objects:

Object Category	**Object Name**	**Description**
`Library`	`base_cell`	A library `base_cell` (like `AND2`) created from the LEF and Liberty `.lib` definitions. This is the library data that does not vary with different timing corners, so pin names, LEF attributes, and `.lib` attributes (like `is_flop`) are here. Most of the `base_cell` objects have multiple `lib_cell` objects (`.lib` data) attached for each timing corner, unless it is a physical-only `base_cell` like a filler cell that has no `lib_cell`. So every `inst` has a `base_cell`, but every `base_cell` does not have a `lib_cell`.
	`base_pin`	A logical pin on a library `base_cell`.
	`lib_cell`	A Liberty (`.lib`) defined library cell. There can be many `lib_cells` with different timing data for one `base_cell`.
	`lib_pin`	A Liberty (`.lib`) defined library cell pin. There can be many `lib_pins` with different timing data for one `base_pin`.
`Design`	`design`	A top-level design (the top-level `Verilog` module). Note that the `topCell`, `fplan`, and `hinst` attributes are directly available in the design object (similar to rows or `hinsts`).
	`hinst`	A hierarchical instance.

	`module`	A design module.
	`inst`	A leaf instance.
	`hpin`	A logical pin on the outside of an `hinst` (it connects to the `hnet` objects outside the `hinst`).
	`pin`	A logical pin on an `inst`.
	`port`	A logical port on the design. The `design` and `base_cell` usage is separated into two different objects, because they have many attributes that only apply to one of the associated two cases.

`Timing`	`analysis_view`
	`clock`
	`library`
	`library_set`
	`opcond`
	`rc_corner`
	`timing_condition`
	`arc`
	`lib_arc`
	`timing_point`
	`timing_path`
	`path_group`
		Timing objects accessed with the SDC commands, such as `get_pins` and `get_property`.

`Clock`	`clock_tree`
	`clock_tree_source_group`
	`clock_spine`
	`flexible_htree`
	`skew_group`
		CCOpt objects.

	`<direct access>`	A property value is directly accessed using the property name. So use: `get_db $inst .my_prop_name`
`Wires`	`via_def_rule`	A via definition rule (like a `LEF VIARULE GENERATE` statement).
	`via_def`	A via definition.
	`via`	An instance of a via.
	`special_via`	An instance of a special via.
	`obj_type` `attribute`	The database schema is available with these objects. Use these as below: `get_db obj_types #shows all obj_types get_db attributes *slack* #shows all attributes that have slack in their namehelp -attribute *slack* #shows formatted help output for all attributes with slack in their name`

Related Information

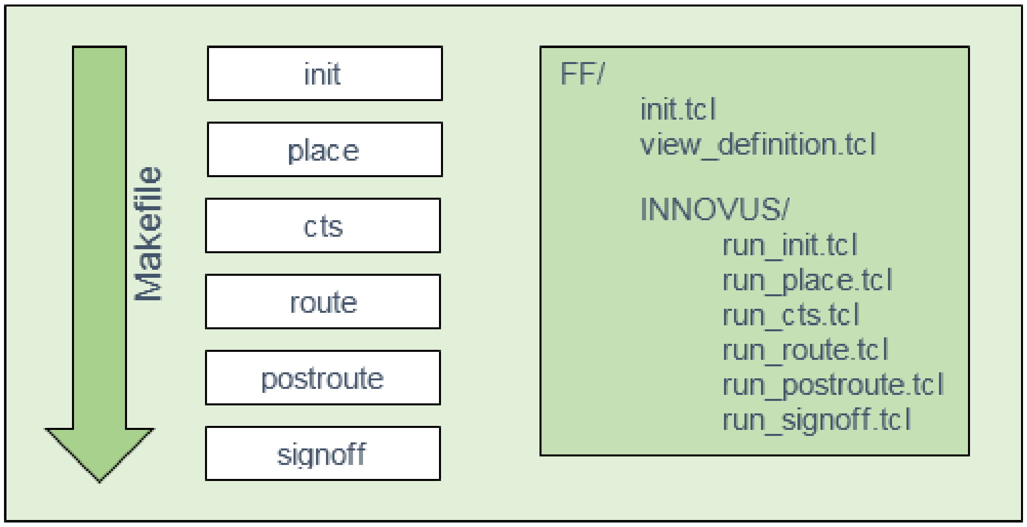
For more information, see:

- `get_db`
- `set_db`
- Stylus Common UI Database Object Information

6

Foundation Flows

The foundation flow is a code generation engine that takes a Tcl variable array (vars) that defines both the library/design data as well as tool options or flow control and generates a set of scripts to execute a baseline flow. Each flow step is a single Tcl script that gets executed via a Makefile. An example of a simple example flow (default baseline flow) and the scripts that execute the flow is shown below:



The flow can be broken down into two areas: design initialization and design execution. Design initialization takes place as part of `run_init.tcl` and is supported by the `FF/init.tcl` and `FF/view_definition.tcl` files, which define the design and library data required to construct the initial database.

In addition, the flow can be customized through the use of tags and plug-ins that allow you to augment or override the flow commands within the sequence of flow steps. It means that tag and plug-in files are included as part of the flow step script.

The Foundation Flow has been replaced with a more integrated approach to flow construction and execution that works across the entire flow seamless in the Stylus Common UI.

Related Information

- Flowkit
- flowtool

7

Flowkit

Flowkit enables the previous step scripts (standalone Tcl files) to become flow objects, which are stored in the database. In the example below, the flow block is defined as a flow of sub-flows, which map to the same basic flow sequence as the foundation flow:

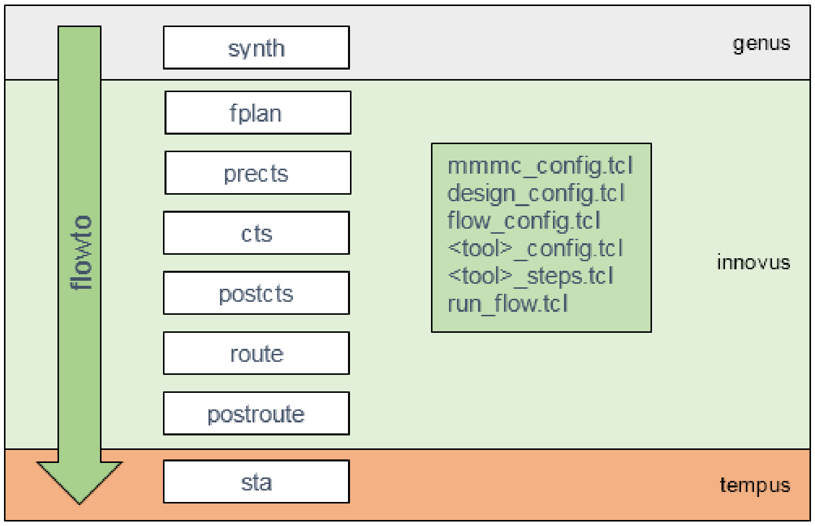
```
create_flow -name block {flow:fplan flow:prects flow:cts flow:route flow:postroute}
```

Also, there is `design_config.tcl`, which contains the design initialization commands and the `mmmc_config.tcl`, which is the flowkit equivalent of the `view_definition.tcl` files. The `*tool*_steps.tcl` file contains the `create_flow_step` commands for the steps associated with a particular tool (like Innovus). For a multi-tool flow (as depicted below), the block flow would be pre-pended with `flow:syn_generic, flow:syn_map, flow:syn_opt` and post-pended with `flow:sta`.

Since the flow definitions for the (sub)flows specify the tool, flowkit knows how to launch them. Example:

```
create_flow -name prects -tool innovus {flow_start run_place_opt flow_finish}
```

The flowkit stylus flow is presented below:



Each of the sub-flow consists of the flow steps. The flow customization is done by creating the specified flow steps and attaching them before or after one of these flow steps.

For example, the previous `pre_cts` plug-in in the foundation flow would be handled in the following way. Consider a plug-in file (`plug/pre_cts.tcl`) that contains the following:

```
set_interactive_constraint_modes [all_constraint_modes -active]

set_clock_uncertainty -setup 0.050 [get_clocks {CLK}]

set_clock_uncertainty -hold 0.003 [get_clocks {CLK}]

reset_propagated_clock [all_clocks]

reset_propagated_clock [get_ports [get_property [all_clocks] sources]]
```

The Stylus Common UI flowkit `cts` flow is defined as the following sequence of steps:

```
create_flow -name cts -tool innovus \

{flow_start add_clock_spec build_clock_tree add_tieoffs flow_finish}
```

You can create a flow step that contains the plug-in commands
(use `convert_legacy_to_common_ui`).

```
create_flow_step -name pre_cts {

set_interactive_constraint_modes [all_constraint_modes -active]

set_clock_uncertainty -setup 0.050 [get_clocks {CLK}]

set_clock_uncertainty -hold 0.003 [get_clocks {CLK}]

reset_propagated_clock [all_clocks]

reset_propagated_clock [get_ports [get_property [all_clocks] sources]]

}
```

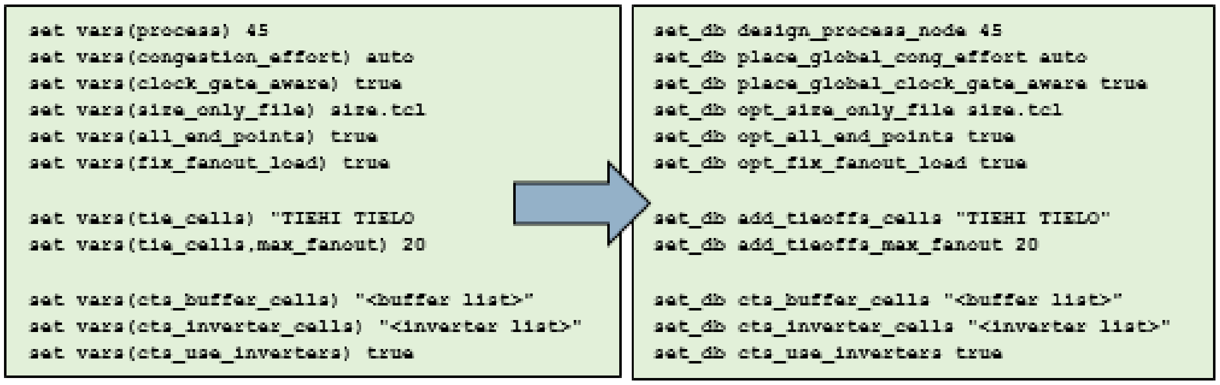
This flow step would then be added to the flow before the `add_clock_spec` flow step:

```
edit_flow -append flow_step:pre_cts -before flow_step:add_clock_spec
```

The flow execution is managed by `flowtool`, as show below.

The foundation flow `setup.tcl` contained variable definitions that the code generator would use to create the command sequence and determine the flow step sequence (for example, `place_opt_design` and `fix_hold`). With a flowkit, `write_flow_template` allows you to enable features that control the step content and the sequence of the flow templates that are generated.

With respect to the variable definition in the `setup.tcl` that is used to map commands or command options, the large majority of these now map directly to Stylus Common UI attributes, which makes the need for a layer of abstraction unnecessary:



```
set vars(process) 45
set vars(congestion_effort) auto
set vars(clock_gate_aware) true
set vars(size_only_file) size.tcl
set vars(all_end_points) true
set vars(fix_fanout_load) true

set vars(tie_cells) "TIEHI TIELO"
set vars(tie_cells,max_fanout) 20

set vars(cts_buffer_cells) "<buffer list>"
set vars(cts_inverter_cells) "<inverter list>"
set vars(cts_use_inverters) true
```

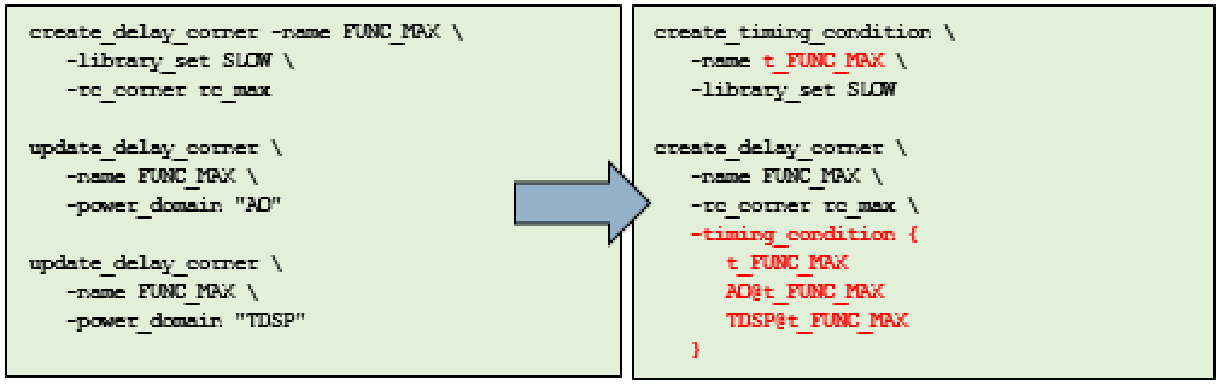
```
set_db design_process_node 45
set_db place_global_cong_effort auto
set_db place_global_clock_gate_aware true
set_db opt_size_only_file size.tcl
set_db opt_all_end_points true
set_db opt_fix_fanout_load true

set_db add_tieoffs_cells "TIEHI TIELO"
set_db add_tieoffs_max_fanout 20

set_db cts_buffer_cells "<buffer list>"
set_db cts_inverter_cells "<inverter list>"
set_db cts_use_inverters true
```

Additionally, the rest of the vars were used to define the timing information for the MMMC view definition file generation (for
example, `library_sets`, `rc_corners`, `delay_corners`, `constraint_modes`, and `analysis_views`). The Stylus Common UI (and hence the `flowkit`) requires additional migration of the view definition file. In particular, the MMMC syntax has an object called the `timing_condition`.

The `timing_condition` object defines the operating condition and library set:

Parameter	**Description**
`-help`	Displays the command usage.
`-library_sets` *`string`*	Specifies the list of libsets to associate to this timing condition.
`-name` *`tc_name`*	Specifies unique name for the timing condition.
`-opcond` *`string`*	Specifies single name of an operating condition.
`-opcond_library` *`string`*	Specifies name of library to search for the named opcond.

In addition, the `timing_condition` argument to `create_delay_corner` supports the syntax for power domain binding:



```
create_delay_corner -name FUNC_MAX \
   -library_set SLOW \
   -rc_corner rc_max

update_delay_corner \
   -name FUNC_MAX \
   -power_domain "AO"

update_delay_corner \
   -name FUNC_MAX \
   -power_domain "TDSP"
```

```
create_timing_condition \
   -name t_FUNC_MAX \
   -library_set SLOW

create_delay_corner \
   -name FUNC_MAX \
   -rc_corner rc_max \
   -timing_condition {
      t_FUNC_MAX
      AO@t_FUNC_MAX
      TDSP@t_FUNC_MAX
   }
```

The command `update_delay_corner` is not necessary because the power domains are bound directly to a timing condition using the `@` syntax.

Related Information

For more information on the Flowkit, see the following sections in the *Stylus Common UI Text Command Reference* document:

- Flowkit Commands and Attributes
- flowtool

8

flowtool

`flowtool` is an external command in Unix for running flows to work interactively within a Tcl session and is used for:

- Hierarchical flows in which multiple independent block flows run in parallel.
- Each step of a flow in its own session to improve productivity (in the absence of session reset).
- Flows that cover required tools (in particular, RTL-GDSII flows).

In addition to providing another interface to the existing `flowkit` functionality, the `flowtool` enables multi-session, multi-tool, and branching flows.

Make versus flowtool

The command behavior of `make` varies significantly across platforms (BSD and GNU), and the capabilities of GNU in particular varies significantly across versions. This can be mitigated by writing to a lowest common denominator, but still creates a testing issue.

`make` targets the files that are regenerated based on other files. This model works well for compiling C, but not for the complex flows. Currently, the foundation flow gets around this problem by creating dummy files corresponding to steps run. This works, but can lead to unexpected behavior if you do not know the location. Another strategy is to use the databases saved at the end of steps instead of the dummy files. This is less clear in the cases where steps do not create a database (notably, partition).

A significant problem with the `make` approach is that details of the flow, most importantly the flow order, are embedded in the Makefile. When these details change, the Makefile must be regenerated which is contradicts the desire to perform `codegen` only once. The complexity of using `make` in a dynamic flow environment can be avoided by developing a tool which is more tailored to running flows. Flowtool is specialized for running flows across multiple tools and multiple sessions and overcomes the limitations described previously.

Flowtool has the following advantages over `make`:

- Flowtool is entirely self-contained and independent of the flow. The `make` approach requires a flow-specific Makefile to be generated for each project.
- You can modify the command line syntax based on the specific task, unlike being restricted to

the make syntax.

- This approach reduces or eliminates dependencies on the deployment environment. For example, you can implement it as a Tcl script, and allows shipping the Tcl interpreter, or bundling the two together to make a standalone executable.
- Dependency tracking and task scheduling do not need to be based on files (although state would probably still be stored in the file system).
- Interface to machine distribution (LSF, SSH, etc.)

Benefits of flowtool

The benefits of using flowtool include:

- No code is generated; flowtool is a static script or executable.
- Flow information is only stored in the database, so cannot become inconsistent.
- The next steps can be determined. This is important for the partitioning flow, as it allows the partitioning step itself to determine the number and inputs of its successors at the run-time, based on the partitioning it has performed.

Related Information

For more information on flowtool syntax and parameters, see the flowtool section in the *Stylus Common UI Text Command Reference* document.

9

Timing Reports

The Stylus Common UI interface provides consistent UI and output for timing reports that improve the ability to analyze and debug the timing information. The timing reports are designed to accomplish the following goals:

- Easier and faster identification of issues.
- Customized reports for analyzing specific problems.
- Additional content to identify certain challenges (such as, power domains, congestion, etc).
- Consistent report output as all the timing report functions are accessible in the same way.
- Better use of hardware resources.

Reporting Commands

The reporting commands, mainly report_timing provide enhanced syntax for better clarity.

The commands support text and CSV output formats that can be redirected to a file through the standard redirect characters (that is, >, >>). The default output is text to stdout.

For more information on the supported reporting commands used for generating timing reports, see the Timing Analysis Commands and Attributes chapter.

Supported Reporting Attributes

You can use the timing_report_fields attribute to control the default output of timing reports - that is, the report fields and options can be changed.

For more information on attributes, see Timing Category Attributes.

The `get_db` command can be used to obtain information associated with the default and added fields. For example, the following command shows the added fields:

```
> get_db timing_report_fields
```

Output:

```
timing_point flags arc edge cell fanout transition delay arrival
```

Related Information

- Timing Report Output Format

Timing Report Output Format

The timing reports follow a standard formatting structure. The report output contains the following sections:

- Path Summary Section - The path summary section provides a quick snapshot of each reported path that allows you to identify various problems easily and quickly.
- Path Description Section - The path description section shows the cell-by-cell delay, slew, arc, exception information, and informational flag legends.

By default, the timing reports show the 'path summary' and 'path description' section. However, the default can be changed by using the `report_timing -path_type` parameter.

A sample path description section is shown below.

```
Path 1: MET (3595 ps) Setup Check with Pin FF_O2_reg[0]/CK->D
Group:        ck1
Startpoint:  (R) FF_I1_reg[0]/CK
Clock:       (R) ck1
Endpoint:    (F) FF_O2_reg[0]/D
Clock:       (R) ck1

             Capture      Launch
Clock Edge:+    9000           0
Src Latency:+      0           0
Net Latency:+      0 (I)    0 (I)
Arrival:=       9000           0

Setup:-          318
Uncertainty:-    125
Required Time:= 8557
Launch Clock:-     0
Data Path:-     4962
Slack:=         3595

#---------------------------------------------------------------------------------
# Timing Point            Arc     Edge    Cell       Load   Trans   Delay  Arrival
#                                                    (fF)   (ps)    (ps)   (ps)
#---------------------------------------------------------------------------------
FF_I1_reg[0]/CK           <<< -   R       -          -      150     -        0
FF_I1_reg[0]/Q            CK->Q   R       DFFHQX4    45.4   201     334    334
mul_26_28/Fn0755D13176/Y  A->Y    F       INVX2      15.7   103     68     402
```

```
mul_26_28/Fn0755D13172/Y  A->Y    R       INVX4     53.0  223     130    532
mul_26_28/n0617DTtn0525/Y B->Y    F       NAND2X1   10.9  169     108    640
mul_26_28/n0634Dtn0548/Y  A1N->Y  F       AOI2BB2X4 19.3  164     232    872
mul_26_28/Fn0612D/Y       A->Y    R       INVX2     18.5  183     119    992
. . .
mul_26_43/n0014Dtn0331/Y  A->Y    F       NAND2X2   17.3  147     94     4168
mul_26_43/Fn0013D16832/Y  A->Y    R       INVX1     5.6   145     97     4265
mul_26_43/n0078Dtn0331/Y  A1N->Y  R       OAI2BB1X2 18.9  201     203    4468
mul_26_43/n0064Dtn0331/Y  A->Y    R       XOR2X4    10.6   86     225    4693
mul_26_43/n0025Dtn0331/S  CI->S   F       ADDFHX4   5.2    82     269    4962
FF_O2_reg[0]/D            <<< -   F       DFFHQX2     -     -       0    4962
#--------------------------------------------------------------------------------
```

Endpoint Reports

In addition to detailed reports, endpoint reports can also be generated (using the `report_timing -path_type endpoint` option). These reports allow you to include additional endpoint related information that can be useful in debugging or identifying certain issues. For example, an endpoint report can show the number of buffers and inverters on a path.

A sample report is shown below:

```
> report_timing -path_type endpoint

#-----------------------------------------

# ID    Slack      Endpoint Group View
#(ns)
#-----------------------------------------
 1    -0.034255    out      clk   view3
 2    -0.034255    out      clk   view5
 3    -0.034255    out      clk   view4
 4     0.389435    f0/D     clk   view3
 5     0.389435    f0/D     clk   view4
 6     0.389435    f0/D     clk   view5
 7     0.398548    f1/D     clk   view5
 8     0.398548    f1/D     clk   view4
 9     0.398548    f1/D     clk   view3
#----------------------------------------
```

The endpoint reports and detailed reports are mutually exclusive.

Use of Informational Flags

Informational tags are used for providing additional details. The following informational tags are supported:

(i)	Net is ideal
(P)	Instance is preserved / don’t_touch
(p)	Instance is preserved / don’t_touch but may be resized

The output format uses the flag keys on the timing report and adds a description at the end of the report for reference. A sample output is as shown below:

```
#------------------------------------------------------------------------------
# Timing Point  Delay (ns) Latch  Window   Arrival   Arc   Flags   Adjustment
# (ns)
#------------------------------------------------------------------------------
f0/CP           -           -    1.000000            CP       -      -
f0/Q            0.254100    -    1.254100            CP->Q    -      -
b32/Z           0.598446    -    1.852546            I->Z     -      -
...
b4/Z            0.086940    -    2.502181            I->Z     -      -
f1/D            0.000000    -    2.502181            D        <<<    -
#------------------------------------------------------------------------------

(P) : Instance is preserved
(p) : Instance is preserved but may be resized
(i) : Net is ideal.
(b) : Timing paths are broken.
```

Supported Fields

By default, timing reports show default fields. Any additional fields can be added (or removed) by using the `Innovus -fields` parameter.

The supported fields and their default status is shown in the table below:

Field	**Description**	**Detailed Report**	**Endpoint Report**

adjustment	Reports generated clock adjustment values. When reporting the detailed path for a generated clock, there are circumstances where the arrival time needs to be adjusted when a `create_generated_clock` assertion is encountered along the path. The adjustment argument can be used to provide a clearer indication of why a "jump" in the arrival time is seen in the path report. When the `create_generated_clock -edge_shift` parameter is specified, the adjustment column will show the amount of shift specified.		
abs_delay_err	Reports the absolute delay difference of Innovus from Spice.		
abs_transition_err	Reports the absolute slew difference of Innovus from Spice.		
annotation	Reports RC parasitic annotations for net arcs, and delay annotations for either gate or net arcs. The following annotations are reported: • SDF: SDF back-annotation. The annotation status reflects the gate arc, not the interconnect arc. • SPEF: This entry appears if parasitic extraction has been done or annotated using SPEF. • LumpedRC: Lumped RC annotation • DlyAssert: Assertion set with set_annotated_delay command • WLM: Wire load model back-annotation • <none>: All others		

aocv_adj_stages	Shows up "Aocv Adjust Stages" column. This column reports the AOCV adjustment stages that indicate the total effective stage count.		
aocv_derate	Reports the graph-based AOCV derating factor used for cells/nets in AOCV mode. In retime mode, this column will report graph-based derating for cells/nets before the AOCV branch point and path-based derating.		
aocv_weight	Reports AOCV weight values for the path elements.		
arc	Reports the arc as described by the from pin, from pin edge, to pin, and to pin edge. For example, the arc from the rising edge of pin A to the falling edge of pin Z is reported as `A -> Z`.	Y	
arrival	Reports the arrival time on the pin.	Y	
arrival_mean	Reports mean of the arrival time on a pin.		
arrival_sigma	Reports standard deviation of the arrival time on a pin.		
cell	Reports the cell name of the given pin's instance.	Y	
delay	Reports the arc delay. If the stolen slack at the arc output pin (output of transparent latches) is not zero, the stolen column is also displayed along with this column.	Y	
delay_mean	Reports the mean of each arc's delay.		

delay_sigma	Reports the standard deviation of each arc's delay. **Note**: In the absence of correlated and uncorrelated components of variation, the individual delay sigma values will not be summed up.		
direction	Reports the pin direction (IN, OUT).		
edge	Reports the edge of a pin (^=R, v=F).	Y	
fanin	Reports the number of source nodes of a net connected to a timing pin.		
fanout	Reports the number of sinks of a net connected to a timing pin.	Y	
flags	Indicates the size_only, dont_use, dont_touch, and ideal attributes.	Y	
hpin	Reports the hierarchical name for a given pin. **Note**: The hpin, pin, and timing_point will print the same values in the report output.		
incr_delay	Reports the incremental delay of a pertaining arc. The software reads the incremental delay from an incremental SDF file using the `read_sdf` command.		
incr_slew	Reports the hierarchical name of a given pin's instance.		
instance	Reports the location (x,y) of an instance. If a design is not placed, this column will be blank. This option displays the `instance` column automatically.	Y	
latch_window	Reports the available window for latch to sample the data.		
load	Reports the total capacitance load on a given pin.		

module	Reports the module name of the corresponding block hierarchy. This hierarchical module name will be reported only when the `-hpin` parameter of the `report_timing` command is specified.		
net	Reports the hierarchical name of a net connected to a given pin.		
pct_delay_err	Reports the percent delay difference of Innovus from Spice.		
pct_transition_err	Reports the percent slew difference of Innovus from Spice.		
phase	Reports the phase name of a pin.		
phys_info	Shows minimum/maximum assigned layer, and non-default rule name information.		
pin	Reports the reference name for the given pin.		
pin_load	Reports the pin load data. The pin load refers to the sum of pin capacitance of all the pins connected to the current net.		
pin_location	Reports the location (x,y) of a pin. If a design is not placed, this column will be blank.		
power_domain	Reports power domain details of the specified path instance.	Y	
required	Reports the required time on a pin.		
retime_delay	Reports the recalculated arc delay based on the path being reanalyzed with the specified retime method.		
retime_delay_mean	Reports the mean of the recalculated arc delay on the path being reanalyzed with the specified retime method.		

retime_delay_sigma	Reports the standard deviation of each parameter of an arc's retimed delay.		
retime_transition	Reports the recalculated slew based on the path being reanalyzed with the specified retime method.		
retime_transition_mean	Reports the mean of each parameter of an arc sink pin's retimed slew.		
retime_transition_sigma	Reports the standard deviation of each parameter of an arc sink pin's retimed slew.		
stage_count	Displays the aggregate stage count of a specific instance in graph-based analysis (GBA) mode, and the actual stage count in path-based analysis (PBA) mode.		
slew	Reports the propagated slew at the given pin.		
transition_mean	Reports mean of each parameter of an arc sink pin's slew.		
transition_sigma	Reports the standard deviation of each parameter of an arc sink pin's slew.		
socv_derate	Reports the SOCV or the spatial derating factor applied on cells or nets in SOCV analysis mode.		
spice_arrival	Reports the Spice arrival time at the timing point.		
spice_delay	Reports the Spice delay of a timing arc.		
spice_transition	Reports the Spice slew of a timing point.		
stolen	Reports the slack stolen (or the time given to the previous stage) at the given pin. If the slack is not zero and the `delay` column is specified, then this column is displayed by default (but only visible on the output of transparent latches).		

timing_point	Reports the hierarchical names for the given points.	Y	
total_derate	Reports the total derating factor applied on the corresponding cell or nets that reflect the combined impact of all the derates (OCV/AOCV/spatial) applied on each stage of the timing path.	Y	
user_derate	Reports the derating scale factors set with the `set_timing_derate` command. **Note:** The timing system cannot provide an accurate derating factor for the lumped gate and wire delay when the `set_timing_derate` factors for `-cell_delay` and `-net_delay` are not equivalent. In this case, an asterisk (*) is reported in the SSI Derate column.		
when_cond	Reports the "when" condition of the arc specified in the library.		
wire_load	Reports the wire load.		
wlmodel	Reports the kind of parasitic used by a pin or net, such as SPEF. If the wire load model is derived from a Liberty file, the information is reported in the following format: `Wireload model (.lib` *`model_name library_name`*`)`		

10

Uniform Logging

Command logging plays a vital role in the debug process. The Stylus Common UI provides uniform logging across products by logging all commands in the log file, irrespective of whether they are issued interactively or through startup files and scripts.

In Common UI, logging is verbose by default for all commands, except those originating through user procs. The following table shows how logging works for commands issued using various methods:

Command origin	Logging type	Log Output Example
Interactive typing	Always logged directly in log file	`@innovus 1> report_timing`

Sourcing files	Verbose logging, by default
	
	To turn off verbose logging for all source file requests, set the `source_verbose` attribute to `false`
	
	To turn off verbose logging for a specific source file request,
	use `source -quiet` *`file.tcl`*
		`@innovus 1> source scripts/setup.tcl`
		
		`#@ Begin verbose source /proj/scripts/setup.tcl`
		`@file 1: set_db design_process_node 16`
		`...`
		`Applying the recommended capacitance filtering threshold values for 16nm process node: total_c_th=0, relative_c_th=1 and coupling_c_th=0.1.`
		
		…
		`...`
		`@file 2: set_db opt_fix_hold_verbose true`
		`@file 3: set_db route_design_skip_analog true`
		`#@ End verbose source /proj/scripts/setup.tcl`
		`1 true`

Running flow_steps	Verbose logging, by default
	
	To turn off verbose logging, set the `flow_verbose` attribute to `false`
		`@innovus 1> create_flow_step -name timing {`
		`report_timing`
		`}`
		`@innovus 2> create_flow -name rpt {flow_step:timing}`
		`@innovus 3> run_flow -flow rpt`
		`...`
		`#@ Begin verbose flow_step rpt.timing`
		`@flow 1: report_timing`
		`###############################################################`
		`# Generated by: Cadence Innovus 20.10-b015_1`
		`# OS: Linux x86_64(Host ID noi-leenap1)`
		`# Generated on: Tue Feb 11 16:48:49 2020`
		`# Design: carve`
		`# Command: report_timing`
		`#################################################################`
		`...`
		`#@ End verbose flow_step rpt.timing`

Calling procs	Non-verbose logging, by default
	To enable verbose logging, use the `set_proc_verbose` command
		Default log output example
		`@innovus 1> proc hi {} {puts Hello}`
		`@innovus 2> hi`
		`Hello`
		Output with `set_proc_verbose`
		`@innovus 1> define_proc foo {} { place_opt_design; report_timing }`
		`@innovus 2> set_proc_verbose foo`
		`@innovus 3> foo`
		`#@ Begin verbose proc foo`
		`@proc 1: place_opt_design`
		`#% Begin place_opt_design`
		`...`
		`...`
		`#% End place_opt_design`
		`@proc 2: report_timing`
		`##############################################################`
		`# Generated by: Cadence Innovus 17.10-d119_1`
		`# OS: Linux x86_64(Host ID sj-jasond)`
		`# Generated on: Wed Mar 15 10:37:56 2017`
		`# Design: dtmf_recvr_core`
		`# Command: report_timing`
		`###############################################################`
		`#@ End verbose proc foo`

Issued through GUI	Similar to interactive typing	On selecting *Timing -> Report Timing* from the GUI menu, log file will show: `@innovus 1> report_timing`

Product startup files	Verbose logging, by default
	To turn off verbose logging, set the shell envar `CDS_STYLUS_SOURCE_VERBOSE` to `0`, which will set `source_verbose` to `false` at startup
		If you add the following to the `./.cadence/innovus/innovus.tcl` startup file:
		`puts "inside [info filename]"`
		the log file, by default, shows:
		`#@ Loading startup files ...`
		`@innovus 1> source .cadence/innovus/innovus.tcl`
		`#@ Begin verbose source .cadence/innovus/innovus.tcl`
		`@file 1 : puts “inside [info filename]”`
		`inside ./cadence/innovus/innovus.tcl`
		`#@ End verbose source .cadence/innovus/innovus.tcl`

Product startup options (`-files` and `-execute`)	Verbose logging, by default
	
	To turn off verbose logging, set the shell envar `CDS_STYLUS_SOURCE_VERBOSE` to `0`, which will set `source_verbose` to `false` at startup
		If the file `run.tcl` contains the following:
		`proc test {} {puts “This is a test”}`
		`test`
		
		and the product is started with:
		`innovus -stylus -files run.tcl`
		
		the log file, by default, shows:
		`#@ Loading startup files ...`
		`@innovus 1> source .cadence/innovus/innovus.tcl`
		`#@ Begin verbose source .cadence/innovus/innovus.tcl`
		`@file 1 : puts “inside [info filename]”`
		`inside ./cadence/innovus/innovus.tcl`
		`#@ End verbose source .cadence/innovus/innovus.tcl`
		`#@ Processing -files option`
		`@innovus 2> source run.tcl`
		`#@ Begin verbose source /proj/scripts/run.tcl`
		`@file 1: proc test {} {puts “This is a test”}`
		`@file 2: test`
		`This is a test`
		`#@ End verbose source /proj/scripts/run.tcl`

11

Uniform Startup

In Stylus Common UI, the bootstrap files are loaded in a consistent order across all tools - Genus, Innovus, Joules, Modus, Tempus, and Voltus. The startup sequence is listed in the table below:

#	Startup File	Example
1	`~/.cadence/<tool>/{gui.pref.tcl, gui.color.tcl, workspaces}` in the home directory if the GUI is enabled.	`~/.cadence/innovus/{gui.pref.tcl, gui.color.tcl, workspaces}`
2	`.cadence/<tool>/{gui.pref.tcl, gui.color.tcl, workspaces/}` in the current directory if the GUI is enabled.	`.cadence/innovus/{gui.pref.tcl, gui.color.tcl, workspaces/}`
3	`<install_dir>/etc/<tool>/<tool>.tcl` in the installation directory. **Note**: For Tempus and Voltus, `etc/<tool>` is `etc/ssv` because they share the same installation. However, both these tools use separate Tcl file names - `tempus.tcl` and `voltus.tcl`, respectively.	`<install_dir>/etc/innovus/innovus.tcl`
4	`~/.cadence/<tool>/<tool>.tcl` in the home directory.	`~/.cadence/innovus/innovus.tcl`
5	`./.cadence/<tool>/<tool>.tcl` in the current directory.	`./.cadence/innovus/innovus.tcl`
6	`./<tool>_startup.tcl` in the current directory.	`./innovus_startup.tcl`

Note: The `-stylus` usage sets the `source_verbose` root attribute to `true` at startup. The GUI preference init files (1 and 2 above) ignore this setting, but the non-GUI preference initialization files above (3, 4, 5, and 6) will use verbose logging by default. If the shell envar `CDS_STYLUS_SOURCE_VERBOSE` exists and has a boolean value, then the `source_verbose` attribute will be set to the envar value at startup. So you can turn off verbose logging at startup by using the following command:

```
setenv CDS_STYLUS_SOURCE_VERBOSE 0
```

Related Information

- `innovus` command in the *Text Command Reference*

12

Unified Metrics

Unified metrics provides functionality to:

- Capture metric data from various commands
- Collect and stores metrics in a structured format
- Save/restore metrics including transitions between tools
- Generate HTML reports and dumps exchange format files
- Compare HTML reports across multiple runs

Metric Infrastructure

In Common UI, the metric infrastructure provides a common set of commands to capture metrics, create snapshot, query metrics, and generate reports. These commands are consistent across all tools that support unified metrics. Once metrics are captured in a snapshot, they are persistent in the database and can be passed between the various tools in the flow, thus creating a cohesive history of the flow.

Metrics are set by various reporting commands such as `report_timing` and `report_analysis_summary` . However, once set, these metrics are only 'pending'. They are not persistent until they are captured in a snapshot using the `create_snapshot` command. To calculate and capture metrics, you can use `create_snapshot -categories` *`list of categories`*. This ensures that all metrics of a given category are captured in the snapshot even if the user executed a sequence of reporting commands.

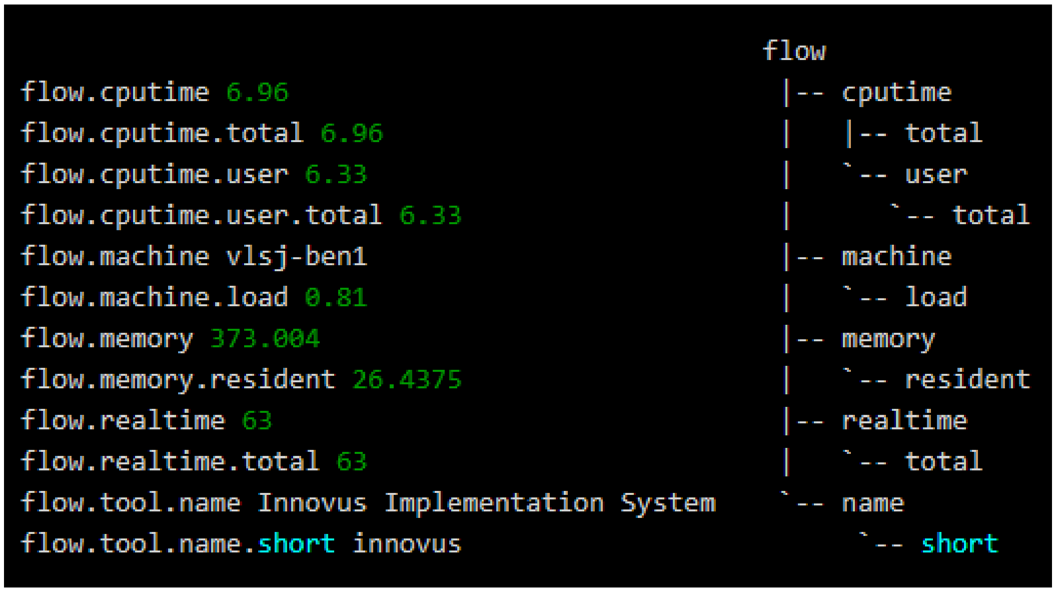
Once set, metrics can be queried using the `get_metric` command. Note that pending metrics can be queried using `get_metric -pending`.

A metric is the core representative of a piece of data. It has a name and a value. The value should have a unit specified wherever applicable. This allows `report_metric` to scale the values, the "display unit" so that same metrics captured by different tools, using different units are displayed uniformly in the HTML reports.

Metric Names

Metric names are standardized across the tools so that the same name will contain the same data.

This means that each tool will populate a metric with a value that represents the same semantic value. This value can be compared across the runs. The names look hierarchical with the use of the ".". Use of "." allows grouping common metrics through their names. For example, consider the following metric names and how they could be represented as a hierarchy.



```
                                                  flow
flow.cputime 6.96                                  |-- cputime
flow.cputime.total 6.96                            |   |-- total
flow.cputime.user 6.33                             |   `-- user
flow.cputime.user.total 6.33                       |      `-- total
flow.machine vlsj-ben1                             |-- machine
flow.machine.load 0.81                             |   `-- load
flow.memory 373.004                                |-- memory
flow.memory.resident 26.4375                       |   `-- resident
flow.realtime 63                                   |-- realtime
flow.realtime.total 63                             |   `-- total
flow.tool.name Innovus Implementation System       `-- name
flow.tool.name.short innovus                            `-- short
```

Metric Values

Metric values can be of many types but are most often a number. Wherever possible, the unit should be defined so that the metric display can always show the values in the same units. This will prevent different tools from producing numbers in different units.

Metric Object Definition

The following table lists the metric object definition in Common UI:

Attribute	Type	Description
name	String	Name of the metric
value	String	Value of the metric
visibility	Enum	Default, user, hidden, internal
default_unit	String	Default unit for the metric (assumed for value)
unit	String	Unit to use for reporting
threshold	Float	Threshold for numeric metrics
threshold_function	Enum	How to interpret the threshold (greater_than, less_than)

precision	Int	Number of decimal places
tcl	TCL	List of TCL commands or proc to generate (return) the metric that is of the format { <value> <unit>}

Metric Commands

The following table shows the metric commands in Common UI:

Command	Description
`define_metric`	Allows the user to define new metrics
`create_snapshot`	Creates a snapshot of all the defined metrics
`get_metric`	Gets the current value of a metric or metrics
`read_metric`	Loads a previously saved metric file
`write_metric`	Writes out metrics in various file formats
`report_metric`	Generates reports for the specified metric

Setting Metrics

The TCL command for setting a metric is `set_metric`.

- The following command would set the value of the worst negative setup slack (all path groups) to 0.100 ns.

  ```
  % set_metric -name timing.setup.wns -value { 0.100 ns}
  ```

- The following commands would set the value of the total area of the design to 100 um^2.

  ```
  % set_metric -name design.area -value {100 um^2}
  ```

- You can get the previously set metric using the `get_metric` command.

  ```
  % get_metric -name timing.setup.wns -pending
  0.100 ns
  ```

- The following will capture these metrics into a snapshot

  ```
  % create_snapshot -name test1
  ```

  Now, these metrics are persistent:

  ```
  % get_metric timing.setup.wns
  0.100ns
  ```

- This metric can now be set again

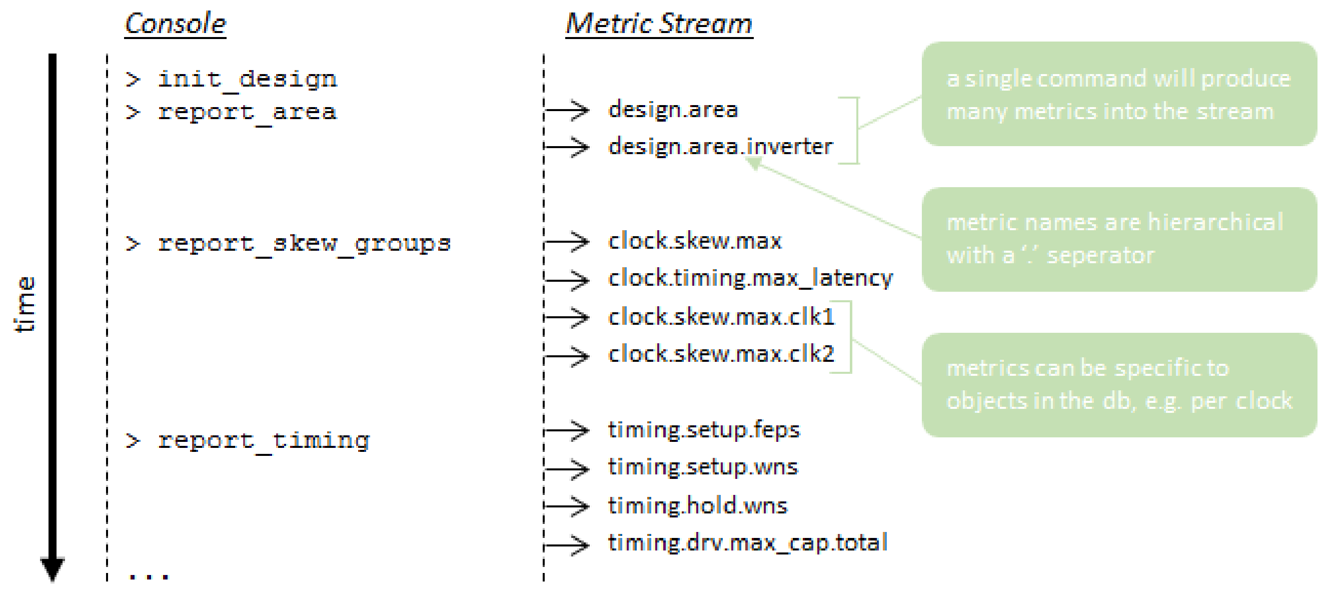
```
% set_metric -name timing.setup.wns -value { -0.200 ns}
% create_snapshot -name test2
% get_metric timing.setup.wns
-0.200ns
```

- To query the previous value of this metric, do the following:

```
% get_metric -names test1 timing.setup.wns
0.100ns
```

Streaming Metrics

The UM-enabled tools will automatically produce data as it is available. For example, the diagram below shows the execution of some of the TCL commands and the example metrics that are produced into the metric stream. If the same metric name is produced, only the latest value is remembered by the tool, that is, the metrics will naturally overwrite each other.



Related Information

- Metric Snapshots

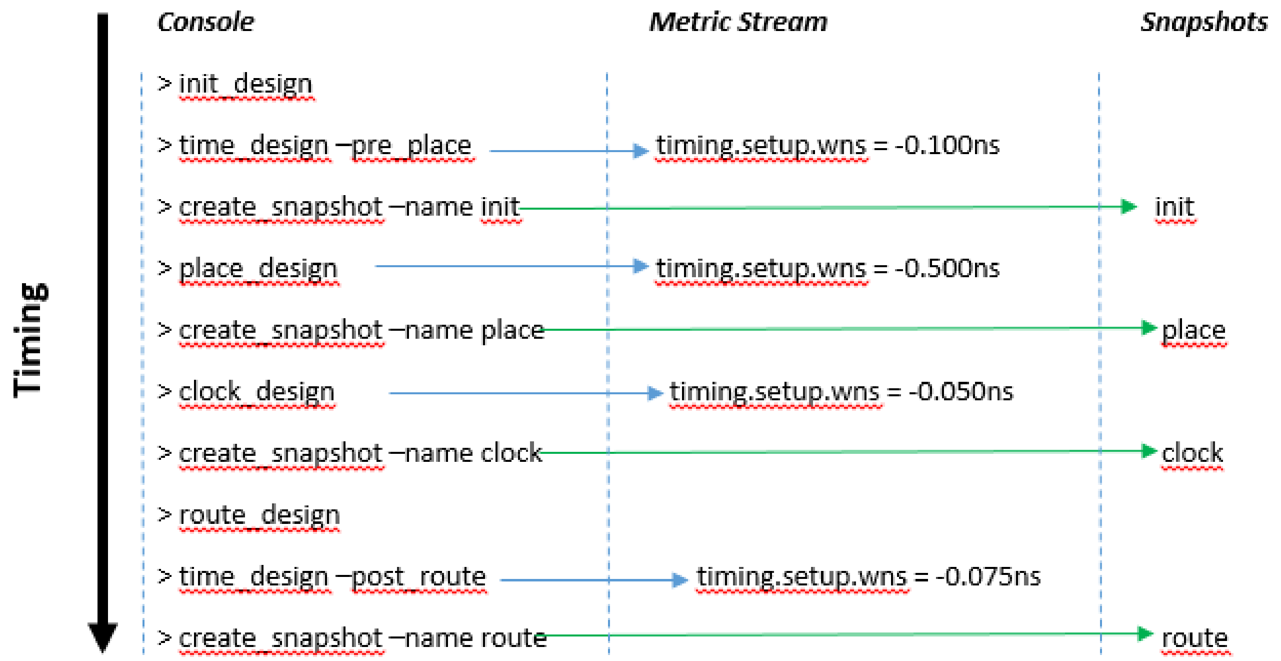
13

Metric Snapshots

The metric stream is collected and structured with the use of snapshots. A snapshot records all the current values of the metrics and stores them in the database. You can create a snapshot by using the `create_snapshot` command.

```
create_snapshot -name my_first_snapshot
```

You can use snapshots to check what happened during the flow:


Console
Metric Stream
Snapshots
Timing
> init_design
> time_design –pre_place
timing.setup.wns = -0.100ns
> create_snapshot –name init
init
> place_design
timing.setup.wns = -0.500ns
> create_snapshot –name place
place
> clock_design
timing.setup.wns = -0.050ns
> create_snapshot –name clock
clock
> route_design
> time_design –post_route
timing.setup.wns = -0.075ns
> create_snapshot –name route
route

In the above example, four snapshots, init, place, clock, and route are created. For example:

Snapshot	Setup WNS
clock	-0.050 ns
init	-0.100 ns
place	-0.500 ns

route	-0.075 ns

To ensure that all desired metrics are calculated and captured, the `create_snapshot -categories` command can be used. Available categories are design, power, setup, clock, hold, route, and flow. The timing (i.e setup/hold) and power categories run the necessary reporting commands, so these categories take additional run time.

Snapshots Hierarchy

You can also use snapshots to produce a hierarchy, which allows you to check the data of a specific step. You can do this by managing the *snapshot stack* to control the current depth of the snapshots being made.

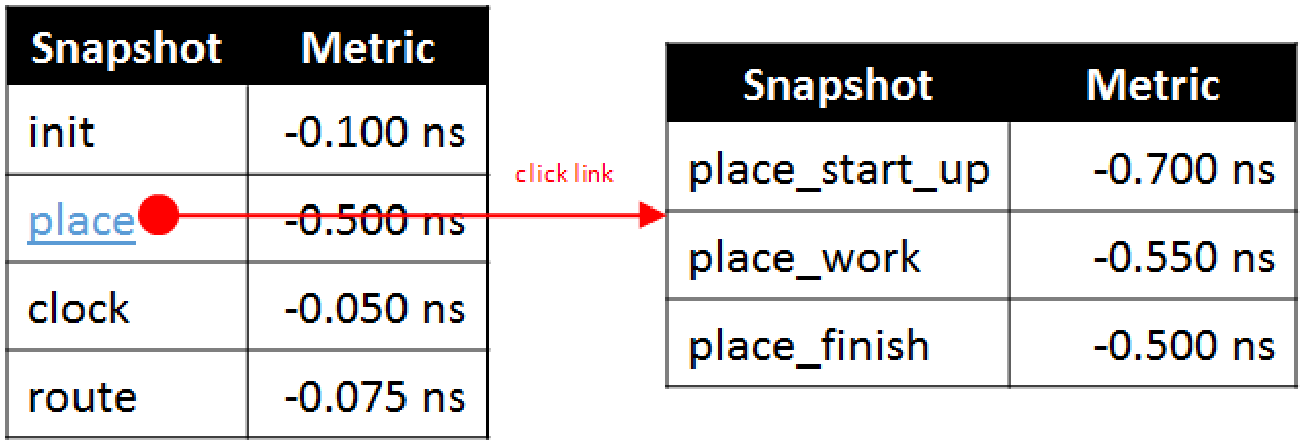
For example the following code would produce the following tables:

```
> set_metric -name timing.setup.wns -value {-0.100 ns}
> create_snapshot -name init
> push_snapshot_stack
> set_metric -name timing.setup.wns -value {-0.700 ns}
> create_snapshot -name place_start_up
> set_metric -name timing.setup.wns -value {-0.550 ns}
> create_snapshot -name place_work
> set_metric -name timing.setup.wns -value {-0.500 ns}
> create_snapshot -name place_finish
> pop_snapshot_stack
> create_snapshot -name place
> set_metric -name timing.setup.wns -value {-0.050 ns}
> create_snapshot -name clock
> set_metric -name timing.setup.wns -value {-0.075 ns}
> create_snapshot -name route
```

The above code creates the following snapshot hierarchy:

```
run
  |--- init
  |--- place
  | |--- place_start_up
  | |--- place_work
  | `--- place_finish
  |--- clock
  `--- route
```

In the html file, a table is produced with a link for the "place" snapshot. When you click this link, it will show the lower hierarchy steps that were included through the use of the stack.



Snapshot	Metric
init	-0.100 ns
place	-0.500 ns
clock	-0.050 ns
route	-0.075 ns

Snapshot	Metric
place_start_up	-0.700 ns
place_work	-0.550 ns
place_finish	-0.500 ns

Note: While using the stack it is important to remember that the system is working with a stream of data. Therefore, when you push, a lower level of hierarchy is created, and when you pop the stack, the generated child snapshots are included in the ***next*** snapshot to be created.

Metric/Snapshot Inheritance

When a hierarchy is produced, the parent snapshot automatically *inherits* metric values from the last child snapshot. For example, consider the following:

```
> set_metric -name m -value 1
> create_snapshot -name before_parent
> push_snapshot_stack
> set_metric -name m -value 2
> create_snapshot -name child_A
> set_metric -name m -value 3

> create_snapshot -name child_B
> pop_snapshot_stack
> create_snapshot -name parent
> set_metric -name m -value 4
> create_snapshot -name after_parent
```

The above code creates the following snapshot hierarchy:

```
run
  |--- before_parent (m=1)
  |--- parent (m=3 *inherited)
  | |--- child_A (m=2)
  | `--- child_B (m=3)
  `--- after_parent (m=4)
```

In this example, the snapshot "parent" will inherit the value for M that was captured by snapshot "child_B", that is, '3'.

Values are only inherited if that metric is not explicitly set during the parent. For example, consider the slight modification of the above example:

```
> set_metric -name m -value 1
> create_snapshot -name before_parent
> push_snapshot_stack
> set_metric -name m -value 2
> create_snapshot -name child_A
> set_metric -name m -value 3

> create_snapshot -name child_B
> pop_snapshot_stack
> set_metric -name m -value PARENT_VALUE; ###### <--- added line
> create_snapshot -name parent
> set_metric -name m -value 4
> create_snapshot -name after_parent
```

This would result in the value of 'PARENT_VALUE' being captured for the 'm' metric in the 'parent' snapshot.

Related Information

- Unified Metrics