# Voltus Stylus Common UI Migration Guide

**Product Version 21.10**
**May 2021**

© 2021 Cadence Design Systems, Inc. All rights reserved.
Printed in the United States of America.

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

**Trademarks:** Trademarks and service marks of Cadence Design Systems, Inc. (Cadence) contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence's trademarks, contact the corporate legal department at the address shown above or call 1-800-862-4522.

All other trademarks are the property of their respective holders.

**Restricted Print Permission:** This publication is protected by copyright and any unauthorized use of this publication may violate copyright, trademark, and other laws. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. This statement grants you permission to print one (1) hard copy of this publication subject to the following conditions:

1. The publication may be used solely for personal, informational, and noncommercial purposes;
2. The publication may not be modified in any way;
3. Any copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement; and
4. Cadence reserves the right to revoke this authorization at any time, and any such use shall be discontinued immediately upon written notice from Cadence.

**Disclaimer:** Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. The information contained herein is the proprietary and confidential information of Cadence or its licensors, and is supplied subject to, and may be used only by Cadence's customer in accordance with, a written agreement between Cadence and its customer. Except as may be explicitly set forth in such agreement, Cadence does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from use of such information.

**Restricted Rights:** Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor.

# Contents

# About This Manual

This document applies to and provides information about Stylus Common User Interface.

# How This Document Is Organized

This manual is organized into chapters that cover broad areas of Stylus Common UI functionality. Each chapter contains topics that may address one or more of the following areas:

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

# 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

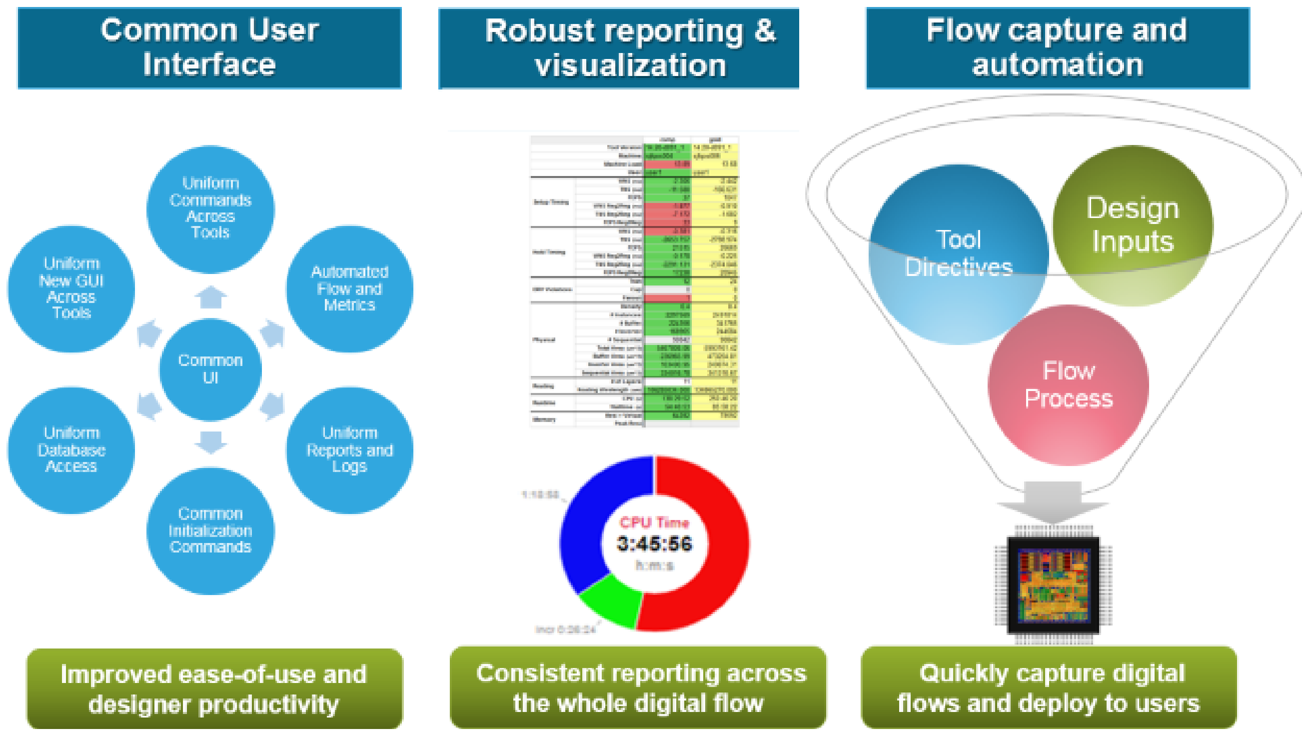
- Voltus Stylus Common UI Text Reference Manual
  Describes the Voltus Stylus Common UI text commands, including syntax and examples.
- Stylus Common UI Database Object Information
  Provides information about Stylus Common UI database objects.

1

# Overview

- Stylus Introduction
- Common Database Access Methods in All Tools
- Common Initialization Flow with Common MMMC File
- Common GUI
- Common Timing Reports
- New Flow Kit
- Unified Metrics
- Consistent Command Names and Usage Across Tools
- Uniform Logging

# Stylus Introduction




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

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 simplifies command naming and aligns common implementation methods across Cadence digital and signoff tools. For example, the processes of design initialization, database access, command consistency, and metric collection have all been streamlined and simplified. In addition, updated and shared methods have been added to run, define, and deploy reference flows. These updated interfaces and reference flows increase productivity by delivering a familiar interface 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.

⚠ Common UI is limited access for early adopters. You should contact Cadence to obtain extra support if you are interested in using Common UI.

The key features of Common UI are covered below.

# Common Database Access Methods in All Tools

You can now use a single command, `get_db`, to query and filter all database attributes. `get_db` replaces `dbGet`, `get_ccopt_property`, and various other `get` methods. It includes `get_property` timing attributes, although `get_property` is still retained for SDC usage.

The `set_db` and `reset_db` commands are the companion commands to set values.

For more information, refer to the Database Access and Handling with get_db and set_db chapter of the *Stylus Common UI Migration Guide*.

# Common Initialization Flow with Common MMMC File

The initialization flow is now the same for all the tools and uses the same MMMC file.

A new `timing_condition` object has been added to the existing MMMC syntax in the MMMC file. This object is required by Common UI and makes it possible to:

- Remove all timing data from power_intent files
- Bind power_domains to the MMMC timing data more easily.

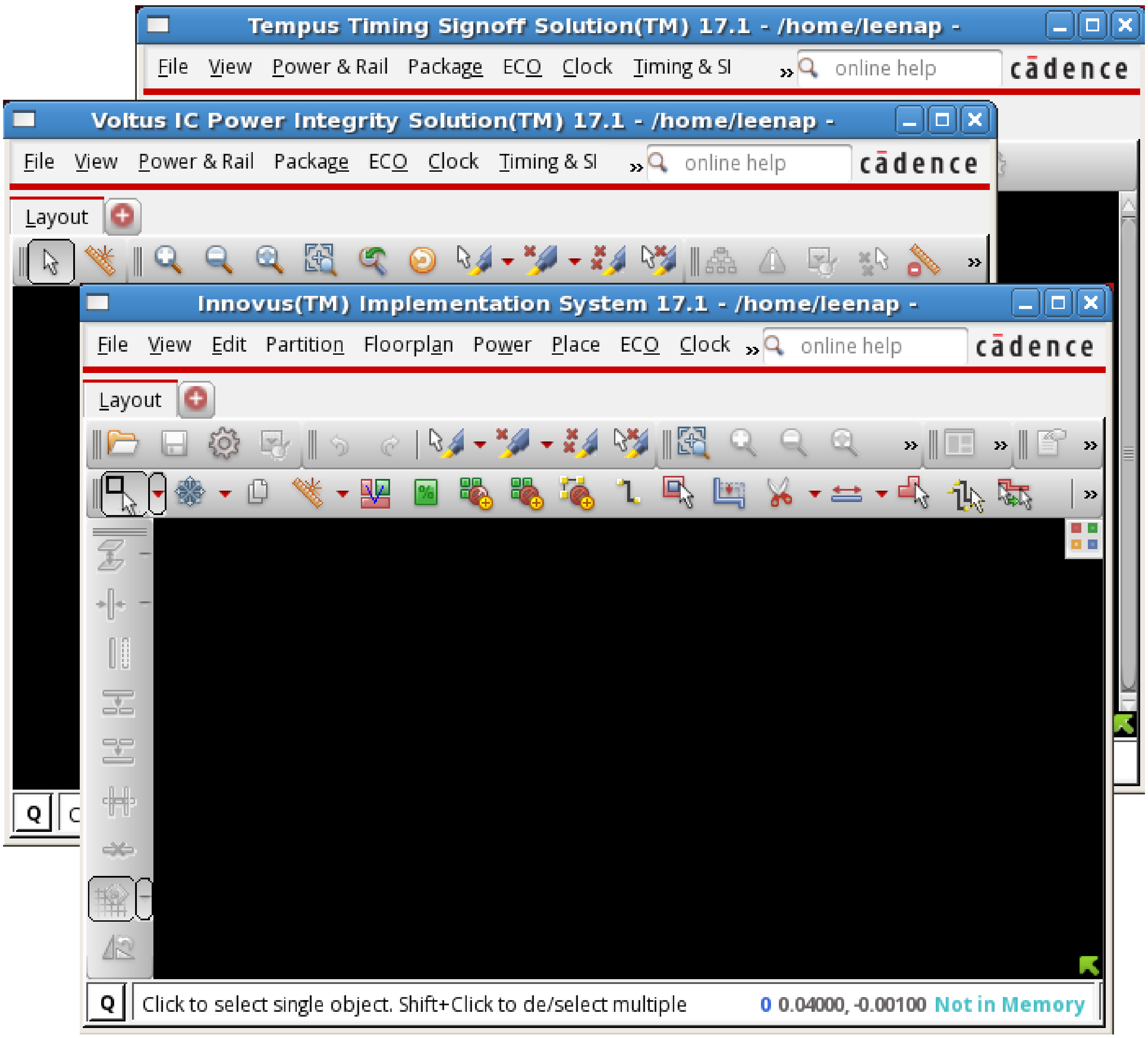
For more information, refer to the Design Initialization chapter of the *Stylus Common UI Migration Guide* and the following initialization-related commands in the *Text Command Reference*:

- `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.



# Common Timing Reports

In Common UI, the `report_timing` command has been enhanced to:

- Allow fast identification of issues related to clock definition, optimization, and constraints.
- Facilitate analysis and debugging. Enhancements include easier issue identification, cut and paste option, similar reports for different tools, and some customization features.
- Produce a more 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.

# New Flow Kit

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

For more information, refer to the Flow chapter of the *Stylus Common UI Migration Guide*.

# Unified Metrics

Common UI enables you to generate unified and holistic metrics, which make it easier to summarize and compare results and lead to effective debugging. For information on new metric commands and attributes, refer to the Unified Metrics chapter of the *Stylus Common UI Migration Guide*.

# Consistent Command Names and Usage Across Tools

Common UI uses consistent command naming conventions across tools.

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

  You can find the Common UI equivalent of any legacy command by using the `get_common_ui_map` command. For example, you can find the Common UI equivalent of the legacy `assignBumps` command as follows:

```
innovus 1> get_common_ui_map assignBump

assignBump                              assign_bumps

-area                                   -area

-constraint_file                        -constraint_file

...

-maxDistance                            not_mapped
```

#not_mapped means this option is obsolete and not in Common UI

```
-multiBumpToMultiPad                    -multi_bumps_to_multi_pads
```

...

For example, the legacy command

```
innovus 1> assignBump -multiBumpToMultiPad
```

is

```
innovus 1> assign_bumps -multi_bumps_to_multi_pads
```

in Common UI.

You can use the `eval_legacy` command to run a legacy command in the legacy Tcl interpreter. In normal usage, this command is not required. However, it may be needed in cases where some legacy usage is not yet available in Common UI. This command is intended as a temporary workaround as users transition to Common UI and will be removed in a future release.

The following commands run the `setFillerMode` and `addFiller` commands in the legacy Tcl interpreter.

```
set my_prefix fill
eval_legacy "setFillerMode -corePrefix $my_prefix"
eval_legacy "addFiller ..."
```

The following uses a TCL variable defined in the current interpreter and passes it to a legacy command:

```
set inst CTS_ccl_BUF_clk_G0_L2_245

eval_legacy “dbGet top.insts.name $inst”
```

The following uses `get_db` to find a particular instance and then pass that object's name to the legacy `dbGet` command. The argument is inside "", so both of the `[get_db.. ]` proc substitutions, and the `$inst` are evaluated inside the CUI interpreter first, before passing the result to the `dbGet` command in the legacy interpreter.

```
eval_legacy "dbGet -p1 top.insts.name [get_db [get_db insts $inst] .name]"
```

- All global values are now root attributes and accessed the same way across tools.

  The previous `setXXXMode` options and global Tcl vars are replaced with root attributes that are accessed/modified with `get_db`/`set_db`/`reset_db`. You can check the mapping of these legacy mode commands by using the `get_common_ui_map` command. For example, you can check the mapping for the legacy `setDesignMode` command as follows:

  ```
  innovus 2> get_common_ui_map setDesignMode

  setDesignMode                          set_db

  -addPhysicalCell                       add_filler_cell_name_style

  -expressRoute                          design_express_route

  -flowEffort                            design_flow_effort


  ...
  ```

  For example, the legacy command
  `innovus 1> setDesignMode -flowEffort high`
  is
  `innovus 1> set_db design_flow_effort high`
  in Common UI.

  You can also check the mapping for any legacy Tcl global in the same way:

  ```
  innovus 2> get_common_ui_map dbgLefDefOutVersion

  dbgLefDefOutVersion                    write_def_lef_out_version
  ```

  For example, the legacy command
  `innovus 1> set dbgLefDefOutVersion 5.7`
  is
  `innovus 1> set_db write_def_lef_out_version 5.7`
  in Common UI.

  You can use the `convert_legacy_to_common_ui` command to convert old legacy Tcl scripts into Common UI Tcl scripts. Although this command does not handle 100% of the legacy commands, it does handle most of the legacy commands and adds comments were manual checking or fixing is needed.

- 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 simpler selection, deselection, and deletion mechanism for objects.
  - In Common UI, a single command, `select_obj`, can be used to select different types of objects. The `select_obj` command replaces the following legacy selection commands:
    ```
    select_bump
    selectBusGuide
    selectBusGuideSegment
    selectGroup
    selectInst
    selectInstByCellName
    selectInstOnNet
    selectIOPin
    selectModule
    selectNet
    selectObjByProp
    selectPGPin
    selectPin
    selectRouteBlk
    ```
    Instead of selecting objects of a specific type, if you want to select objects within a specific area or defined by a specific location, you can use the `gui_select` command in Common UI.
  - Similarly, a single command, `deselect_obj`, can be used to deselect different types of objects. The `deselect_obj` command in Common UI replaces the following legacy deselection commands:
    ```
    deselect_bump
    deselectBusGuide
    deselectGroup
    deselectInst
    deselectInstByCellName
    deselectInstOnNet
    deselectIOPin
    deselectModule
    deselectNet
    deselectPin
    ```
    Instead of deselecting objects of a specific type, if you want to deselect all objects or objects within a specific area or defined by a specific location, you can use the `gui_deselect` command in Common UI.
  - The `delete_obj` command can replace some of the legacy `delete*` commands. The

following obj_types can be deleted through this command: bump bus_guide, custom_line, custom_rect, custom_text, group, marker, net_group, pin_guide, pin_group, pin_blockage, place_blockage, resize_blockage, route_blockage, row, special_wire, special_via, and text.

Note that several `delete_*` commands have special options, so they have been retained in Common UI.

# 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 Startup and Logging chapter of the *Innovus Stylus Common UI Migration Guide*.

2

# Design Initialization

# 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.

Earlier, various tools used different initialization strategies. For example, Genus used an incremental approach, with each command being an autonomous operation. Once the libraries were loaded, the user could query the libraries. On the other hand, Innovus uses a setup or load strategy where variables are first defined pointing to the various design elements. Then, an initialization command is issued and all the elements are loaded at once. Tempus supports versions of both these strategies.

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.<br><br>This step is optional because Synthesis and STA do not always require physical data. It is also optional for implementation to support timing regressions.<br><br>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.<br><br>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.<br><br>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.<br><br>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.<br><br>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. | |

# Considerations

- Synthesis Considerations
- Implementation Considerations
- STA Considerations

# 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 Innovus legacy usage, all loading of data is deferred until init_design. 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.

# Supported Flows and Tools

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 read_power_intent | | √ | |
| MMMC only | No physical | √ | √ | √ |
| Simple timing | Only read_libs and read_sdc | √ | | √ |
| Simple physical | Only read_libs, read_physical, read_qrc, and read_sdc | √ | | |

# Steps and Associated Commands for Common MMMC Configuration

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

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

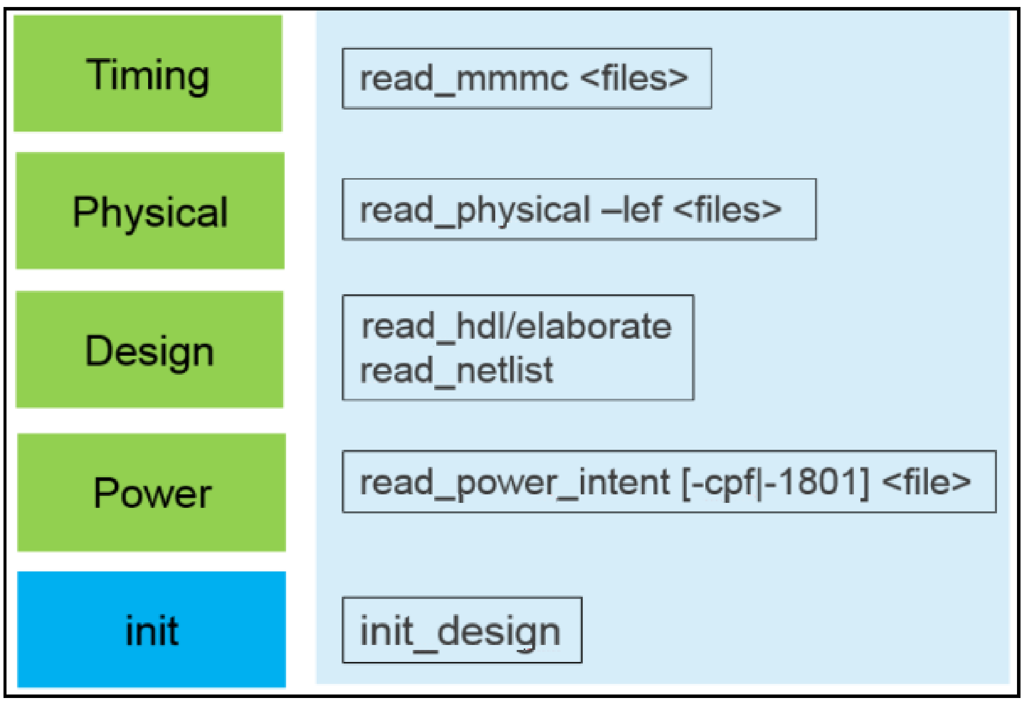
| Command | Genus | Innovus | Tempus |
|---|---|---|---|
| Timing | read_mmmc or read_libs | read_mmmc | read_mmmc |
| Physical | read_physical or read_qrc | read_physical | read_physical |
| Design | read_hdl/elaborate or read_netlist | read_netlist | read_netlist |
| Floorplan | read_def | read_floorplan or read_def | |
| Power | read_power_intent | read_power_intent | read_power_intent |

| | | | |
|---|---|---|---|
| Constraints (simple mode) | `read_sdc` | | `read_sdc` |
| Initialization | `init_design` | `init_design` | `init_design` |

# Initialization Flow - MMMC Mode

- Timing
- Physical
- Design
- Power
- Init

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




# Initialization Flow - 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.

# Initialization Flow - MMMC Mode - Physical Step

During the Physical Step, the physical libraries, OA or LEF, are loaded into the tool's database using the `read_physical` command. The database will be loaded with the specified objects, and will be available for query.

# Initialization Flow - MMMC Mode - 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 & elaborate

- This command pair is used for Synthesis. This step is identical to the current Synthesis operation.
- 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.

# Initialization Flow - MMMC Mode - 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.

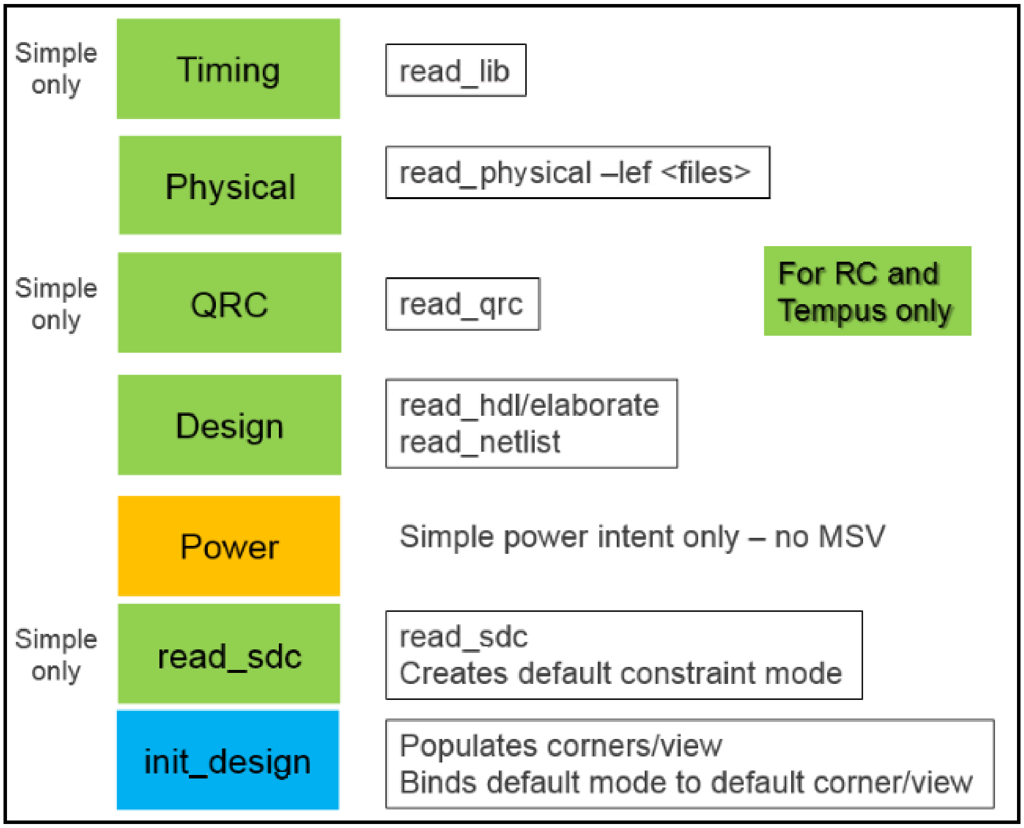
# Initialization Flow - MMMC Mode - Init Step

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.

# Initialization Flow - Simple

To support a simple non-MMMC initialization, there will be a simple initialization sequence consisting of simple commands. However, the underlying infrastructure and attributes are the same. 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
- Physical
- Design
- Power
- SDC
- Init




# Initialization Flow - Simple - Timing Step

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

# Initialization Flow - Simple - Physical Step

In the Physical Step, LEF or OA databases are loaded via `read_physical` 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.

# Initialization Flow - Simple - 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.

# Initialization Flow - Simple - 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.

# Initialization Flow - Simple - 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.

# Initialization Flow - Simple - 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.

3

# Database Access and Handling with get_db and set_db



The Stylus Common UI introduces the new commands `get_db`/`set_db` to provide a single and common way to access/query all the database information.

These commands replace the Tempus (Legacy) commands, `dbGet`, `dbSet`, `get_ccopt_property`, and `set_ccopt_property`; and the 100s of DB object commands that have a `db` prefix such as `dbGetInstByName` and `dbSetInstPlacementStatus`. The new 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`; although these commands are still available for SDC compatibility in the Common UI.

The `get_db` and `set_db` commands support various chaining, filtering, and matching options similar to `dbGet` and `dbSet` (in legacy) but with an easier usage model.

For the full command syntax and arguments, see the `get_db` and `set_db` pages in the Text Command Reference document.

# Basic Concepts

Legacy DB object commands are object-specific commands, and `dbGet` access uses `-p1` to get the object pointer while searching a name. `dbGet` also chains through the top or head objects and sometimes through other objects, whereas the `get_db`/`set_db` 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 makes it easy to get objects directly by name. For example,

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

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

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

Some examples of old versus new usage are:

| Legacy | Common UI |
|---|---|
| `dbFindInstsByName i1/*` | `get_db insts i1/*` |
| `dbGet -p1 top.insts.name i1/*` | `get_db insts i1/*` |
| `dbGet -p1 top.fplan.netGroups my_grp` | `get_db net_groups my_grp` |
| `dbGet [dbGet -p1 head.layers.name metal1].width` | `get_db [get_db layers metal1] .width` |
| `get_property [get_pins i1/p1 ] arrival_max_fall` | `get_db [get_db pins i1/p1] .arrival_max_fall` |
| `get_ccopt_property target_max_trans -clock_tree clk1 -early` | `get_db [get_db clock_trees clk1] .cts_target_max_transition_time_early` |

**Note:** `get_db` requires a space before the `.<attribute>` name, while `dbGet` does not allow a space.

# Dual Ported Objects

The 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 never gets converted to a string which is more efficient, but if you do a `puts $var` or return the value to the shell to echo to the `xterm`, it is converted to its string form.

- 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`.

`get_db/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 called top, then the output is something like this:

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 in the table above could be done more easily 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` |

# Significant Object or Name Changes

Most of the Common UI object names are similar to the legacy object names used by `dbGet` except that they use all lower-case characters, with _ for word separators, and the Common UI names use fewer abbreviations (like `pd` is now `power_domain`, and `bndry` is `boundary`).

Objects or names that were changed substantially are listed below:

| dbGet Object Name | get_db Object Name | Comments |
|---|---|---|
| `topCell` | `design` | A top-level design (the top-level Verilog module). Note that `topCell fplan` and `hinst` attributes are directly in the design object (such as `rows` or `hinsts`); there is no longer a `topCell fplan` object, or the `topCell hinst` object. |
| `head` | `<none>` | Not used. The head attributes (like `layers`) are directly available from the root. |
| `fplan` | `<none>` | Not used. The `topCell.fplan` attributes (such as `rows`, `blockages`, and `bndrys`) are directly in the design object. |
| `libCell` | `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 `base_cells` have multiple `lib_cells` (`.lib` data) attached for each timing corner, unless it is a physical-only `base_cell` like a filler cell that has no `lib_cell`.<br><br>So every inst has a `base_cell`, but every `base_cell` does not have a `lib_cell`. |
| `term(libCell)` | `base_pin` | A logical pin on a library `base_cell`. |

| | | |
|---|---|---|
| term(topCell) | port | A logical port on the design. The design and base_cell usage was separated into two different objects, because they have many attributes that only apply to one of the two cases. |
| <none> | lib_cell | A Liberty (.lib) defined library cell. There can be many lib_cells with different timing data for one base_cell. |
| <none> | lib_pin | A Liberty (.lib) defined library cell pin. There can be many lib_pins with different timing data for one base_pin. |
| instTerm | pin | A logical pin on an inst. |
| hinstTerm | hpin | A logical pin on the outside of an hinst (it connects to hnets outside the hinst). |
| hTerm | hport | A logical port on the inside of an hinst (it connects to hnets inside the hinst). |
| <none> | analysis_view<br>clock<br>library<br>library_set<br>opcond<br>rc_corner<br>timing_condition<br>arc<br>lib_arc<br>timing_point<br>timing_path<br>path_group | Timing objects accessed with SDC commands, such as get_pins and get_property, are available. |

| | | |
|---|---|---|
| <none> | clock_tree<br>clock_tree_source_group<br>clock_spine<br>flexible_htree<br>skew_group | ccopt objects accessed with legacy `get_ccopt_property` are available. |
| prop | <direct access> | A property value is now directly accessed using the property name. So instead of:<br>dbGet [dbGet -p1 $inst.props.name my_prop_name].value<br>you can use:<br>get_db $inst .my_prop_name |
| viaRuleGenerate | via_def_rule | A via definition rule (like a LEF VIARULE GENERATE statement). |
| via | via_def | A via definition. |
| viaInst | via | An instance of a via. |
| sViaInst | special_via | An instance of a special via. |
| <none> | obj_type<br>attribute | The DB schema is available with these objects. This replaces the legacy `dbSchema` command. You can do this instead:<br>get_db obj_types #shows all obj_types<br>get_db attributes *slack* #shows all attributes that have slack in their name<br>help -attribute *slack* #shows formatted help output for all attributes with slack in their name |

# Help and Documentation

For information on all the objects and attributes, see the Stylus Common UI Database Object Information manual.

You can also 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 DB

You can use pattern matching for an attribute name to see multiple attribute values for many objects at the same time. This replaces the legacy `dbGet .??` usage.

So instead of this legacy usage to see all the attributes and values of every pin (`instTerm`) in `$my_pins`, run:

```
dbGet $my_pins.??
```

You can use:

```
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
...
```

# Chaining

Object chaining enables you to link to the related objects allowed like `dbGet`.

**Note:** `dbGet` requires no space before the "." (`dbGet $nets.name`), but `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 count 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)
```

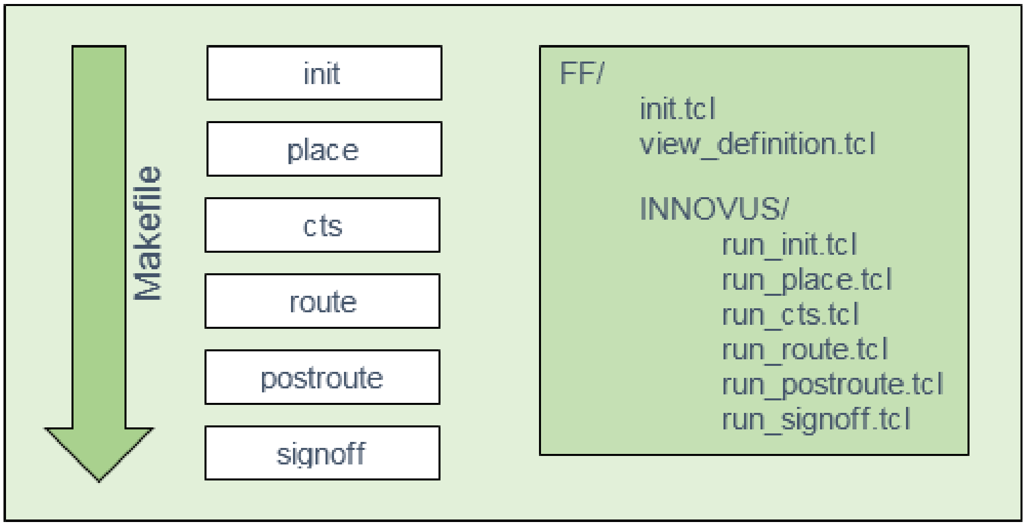
4

# Flow



## Overview

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.

See the following sub-sections for more details:

# Flowkit

With the flowkit, the previous step scripts (standalone Tcl files) 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:

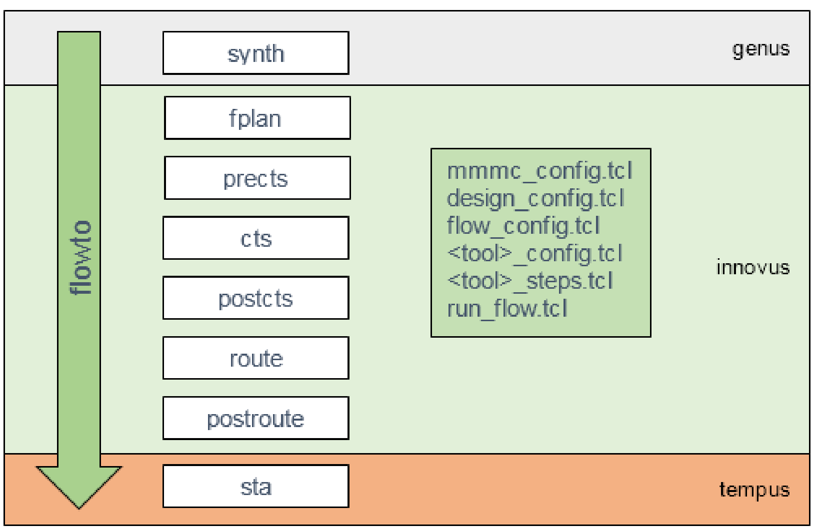
```
create_flow -name block {flow:fplan flow:prects flow:cts flow:route flow:postroute}
```

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 (Genus, Innovus, or Tempus). For a multi-tool flow (as depicted below), the block flow would be pre-pended with `flow:synth` (genus) and post-pended with `flow:sta` (tempus).

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

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

# The Flowkit Stylus Flow




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 flowkit `cts` flow is defined as the following sequence of steps:

```
create_flow -name cts -tool tempus \
{flow_start add_clock_spec build_clock_tree add_tieoffs flow_finish}
```

You can create a flow step that contains the plug-in commands (if necessary, convert the legacy plug-in commands to Common UI commands and 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 a new executable called flowtool, as show below, so make is no longer necessary.

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 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 Common UI (and hence the flowkit) requires additional migration of the view definition file. In particular, the new MMMC syntax has a new object called the timing condition. Here is an example of converting a legacy MMMC file to CUI syntax:

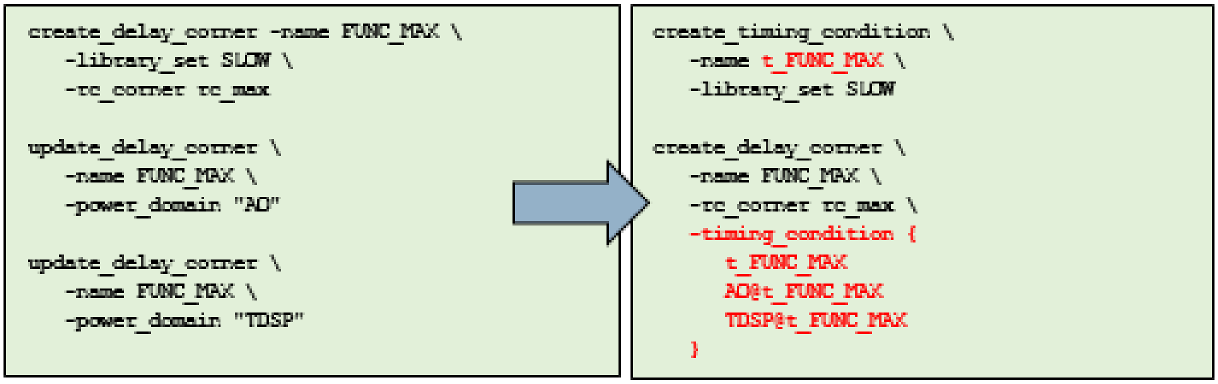
```
create_rc_corner \
   -name max_125 \
   -T 125 \
   -qx_tech_file QRCTechFile

create_delay_corner \
   -name slow_max_125 \
   -rc_corner max_125 \
   -library_set slow_libs
...
```

```
create_rc_corner \
   -name max_125 \
   -temperature 125 \
   -qrc_tech QRCTechFile

create_timing_condition \
   -name slow_condition \
   -library_set slow_libs

create_delay_corner \
   -name slow_max_125 \
   -rc_corner max_125 \
   -timing_condition slow_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 new syntax for power domain binding:




The command `update_delay_corner` is no longer necessary because the power domains are bound directly to a timing condition using the `@` syntax.

# References

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

- Flowkit Commands and Attributes
- flowtool

# Flowtool

While flowkit is designed to work interactively within a Tcl session, an external command-line tool for running flows, flowtool (equivalent to `make` in Unix), 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 the 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 are 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 runtime, based on the partitioning it has performed.

# References

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

5

# Uniform Startup and Logging



## 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.<br>**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
```

# Uniform Logging

Command logging plays a vital role in the debug process. The Stylus Common UI provides improved and 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<br>To turn off verbose logging for all source file requests, set the `source_verbose` attribute to `false`<br>To turn off verbose logging for a specific source file request, use `source -quiet` *`file.tcl`* | `@innovus 1> source scripts/setup.tcl`<br>`#@ Begin verbose source /proj/scripts/setup.tcl`<br>`@file 1: set_db design_process_node 16`<br>`...`<br>`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.`<br>…<br>`...`<br>`@file 2: set_db opt_fix_hold_verbose true`<br>`@file 3: set_db route_design_skip_analog true`<br>`#@ End verbose source /proj/scripts/setup.tcl`<br>`1 true` |
| Running flow_steps | Verbose logging, by default<br>To turn off verbose logging, set the `flow_verbose` attribute to `false` | `@innovus 1> create_flow_step -name timing {`<br>`report_timing`<br>`}`<br>`@innovus 2> create_flow -name rpt {flow_step:timing}`<br>`@innovus 3> run_flow -flow rpt`<br>`...`<br>`#@ Begin verbose flow_step rpt.timing`<br>`@flow 1: report_timing`<br>`###############################################################`<br>`# Generated by: Cadence Innovus 20.10-b015_1`<br>`# OS: Linux x86_64(Host ID noi-leenap1)`<br>`# Generated on: Tue Feb 11 16:48:49 2020`<br>`# Design: carve`<br>`# Command: report_timing`<br>`###############################################################`<br>`...`<br>`#@ End verbose flow_step rpt.timing` |

<table>
<tr><td>Calling procs</td><td>Non-verbose logging, by default<br><br>To enable verbose logging, use the <code>set_proc_verbose</code> command</td><td>Default log output example<pre>
@innovus 1> proc hi {} {puts Hello}
@innovus 2> hi
Hello
</pre>Output with <code>set_proc_verbose</code><pre>
@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
</pre></td></tr>
<tr><td>Issued through GUI</td><td>Similar to interactive typing</td><td>On selecting Timing -> Report Timing from the GUI menu, log file will show:<pre>
@innovus 1> report_timing
</pre></td></tr>
<tr><td>Product startup files</td><td>Verbose logging, by default<br><br>To turn off verbose logging, set the shell envar <code>CDS_STYLUS_SOURCE_VERBOSE</code> to <code>0</code>,<br>which will set <code>source_verbose</code> to <code>false</code> at startup</td><td>If you add the following to the <code>./.cadence/innovus/innovus.tcl</code> startup file:<pre>
puts "inside [info filename]"
</pre>the log file, by default, shows:<pre>
#@ 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
</pre></td></tr>
</table>

| | | |
|---|---|---|
| Product startup options (`-files` and `-execute`) | Verbose logging, by default<br><br>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:<br>`proc test {} {puts "This is a test"}`<br>`test`<br><br>and the product is started with:<br>`innovus -stylus -files run.tcl`<br><br>the log file, by default, shows:<br>`#@ Loading startup files ...`<br>`@innovus 1> source .cadence/innovus/innovus.tcl`<br>`#@ Begin verbose source .cadence/innovus/innovus.tcl`<br>`@file 1 : puts "inside [info filename]"`<br>`inside ./cadence/innovus/innovus.tcl`<br>`#@ End verbose source .cadence/innovus/innovus.tcl`<br>`#@ Processing -files option`<br>`@innovus 2> source run.tcl`<br>`#@ Begin verbose source /proj/scripts/run.tcl`<br>`@file 1: proc test {} {puts "This is a test"}`<br>`@file 2: test`<br>`This is a test`<br>`#@ End verbose source /proj/scripts/run.tcl` |

6

# Unified Metrics



# Overview

Unified Metrics (UM) is a system integrated with some Cadence tools that automatically delivers data about the design and run to you. UM provides functionality to:

- Stream data from the reports and algorithms.
- Collect and organize the stream data into a structured form.
- Display the data in a Simple Unified Metrics file or the Advanced Unified Metrics server.

Currently, the primary digital flow tools, Genus, Innovus, Voltus, and Tempus are UM-enabled as an integral part of the Stylus Common UI feature across these tools.

# Stylus Common UI Metric Infrastructure

- In Common UI, metrics are captured and stored in a file format common to various tools, such as Innovus and RC.
- There are common sets of metrics that are captured consistently in different tools as well as tool-specific metrics.
- The metrics file/database can be used to generate reports/graphs in various formats and can be compared against other runs.
- Metrics can be queried using the `get_metric` command and are written to a <design>.metrics file in the Innovus database directory as part of `write_db`.
- Using the Common UI metric infrastructure, you can define and capture metric snapshots, using the `create_snapshot` command, at specific points in the flow such that metric extraction is not required, and/or timing/power analysis will be calculated at the point the snapshot is saved.
- Timing/power metrics are reported based on the last calculated value. You need to ensure that the metrics are updated prior to saving the metric snapshot. All commands that save metrics call `create_snapshot` to save metrics in a consistent manner.
- Because this data will get saved as part of the database (during `write_db`), it can be used across multiple runs for building design project summary pages.

# Metrics

A metric is the core representative of a piece of data. A metric has a name and a value. The value should have an SI unit specified wherever possible. 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 {0.100ns}
  ```

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

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

You can get a metric using the `get_metric` command.

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

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

# 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 are 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>} |

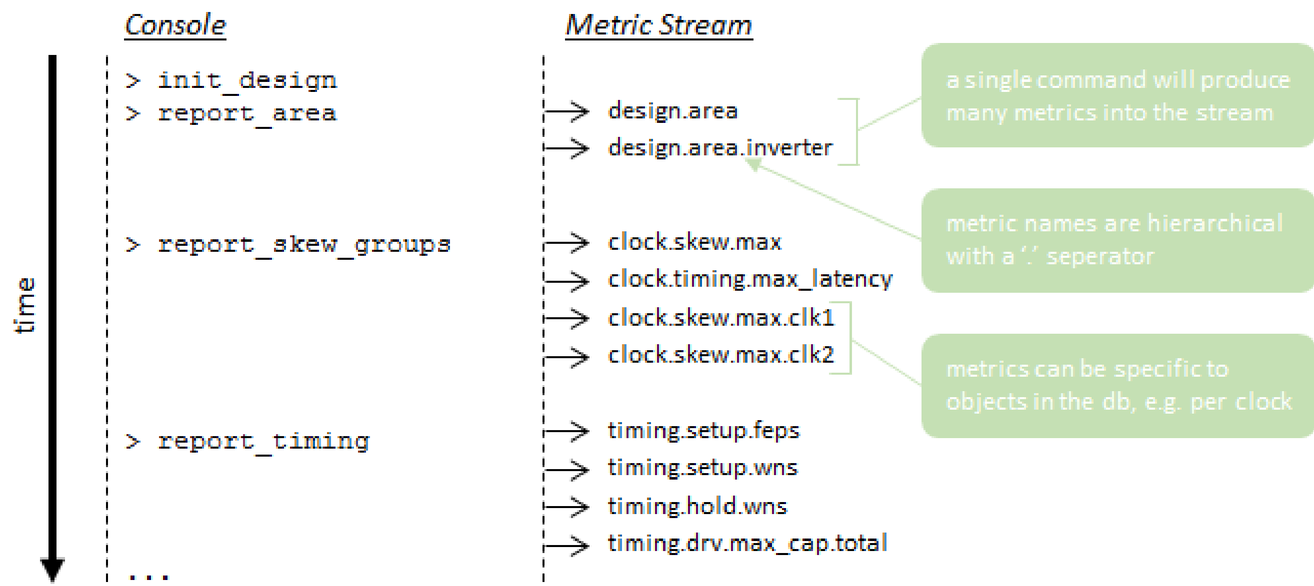
# Metric Commands

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 metrics |

# Streaming Metrics

The UM-enabled tools will automatically produce data as it is available. For example, the diagram below shows the execution (in this case within Innovus) 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.

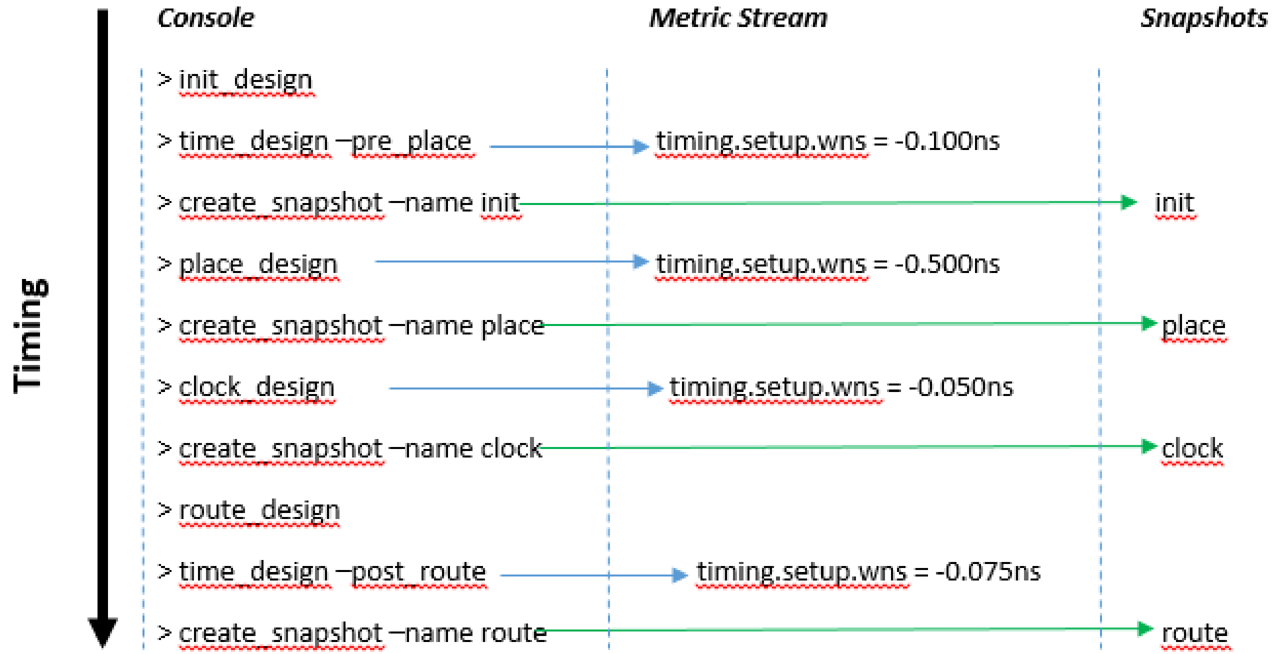



# Snapshots

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:




In the above example, four snapshots, init, place, clock, and route were created. You can check this in the Simple Unified Metrics html file and the Advanced Unified Metrics server in a table. For example:

| Snapshot | Setup WNS |
|---|---|
| clock | -0.050 ns |
| init | -0.100 ns |
| place | -0.500 ns |
| route | -0.075 ns |

# 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 |

click link

| 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.