Fusion Compiler™ Data Model User Guide

Version T-2022.03-SP3, July 2022

SYNOPSYS®

Feedback

Contents

About This User Guide

The Synopsys Fusion Compiler tool provides a complete netlist-to-GDSII or netlist-to-OASIS® design solution, which combines proprietary design planning, physical synthesis, clock tree synthesis, and routing for logical and physical design implementations throughout the design flow.

This user guide is for design engineers who use the library manager tool and the Fusion Compiler implementation tool to prepare libraries and implement designs.

To use the Fusion Compiler tool, you need to be skilled in physical design and synthesis, and be familiar with the following:

- Physical design principles
- The Linux or UNIX operating system
- The tool command language (Tcl)

This preface includes the following sections:

New in This Release

Information about new features, enhancements, and changes, known limitations, and resolved Synopsys Technical Action Requests (STARs) is available in the Fusion Compiler Release Notes on the SolvNetPlus site.

Related Products, Publications, and Trademarks

For additional information about the Fusion Compiler tool, see the documentation on the Synopsys SolvNet® online support site at the following address:

https://solvnetplus.synopsys.com

Feedback

You might also want to see the documentation for the following related Synopsys products:

- Design Compiler®
- IC Validator
- PrimeTime® Suite

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
`Courier`	Indicates syntax, such as `write_file`.
`Courier italic`	Indicates a user-defined value in syntax, such as `write_file` *`design_list`*
`Courier bold`	Indicates user input—text you type verbatim—in examples, such as `prompt>` **`write_file top`**
`Purple`	• Within an example, indicates information of special interest. • Within a command-syntax section, indicates a default, such as `include_enclosing = true | false`
[ ]	Denotes optional arguments in syntax, such as `write_file [-format` *`fmt`*`]`
...	Indicates that arguments can be repeated as many times as needed, such as *`pin1 pin2 ... pinN`*.
|	Indicates a choice among alternatives, such as `low | medium | high`
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Bold	Indicates a graphical user interface (GUI) element that has an action associated with it.
Edit > Copy	Indicates a path to a menu command, such as opening the **Edit** menu and choosing **Copy**.
Ctrl+C	Indicates a keyboard combination, such as holding down the Ctrl key and pressing C.

Feedback

Customer Support

Customer support is available through SolvNetPlus.

Accessing SolvNetPlus

The SolvNetPlus site includes a knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. The SolvNetPlus site also gives you access to a wide range of Synopsys online services including software downloads, documentation, and technical support.

To access the SolvNetPlus site, go to the following address:

https://solvnetplus.synopsys.com

If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to sign up for an account.

If you need help using the SolvNetPlus site, click REGISTRATION HELP in the top-right menu bar.

Contacting Customer Support

To contact Customer Support, go to https://solvnetplus.synopsys.com.

1

Libraries and Blocks

A *block* is a container for physical and functional design data. A *library* is a collection of related blocks, together with technology information that applies to the block collection. A chip design consists of one or more blocks, often stored in different *design libraries*. A design library uses instances of blocks defined in lower-level libraries, called *reference libraries*. A design library can serve as a reference library for another design library.

The following topics describe the usage of libraries and blocks:

Design Libraries

A *design library* stores all of the information about a design, including the associated technology information and the reference library setup information. In addition to the technology and reference library setup information, a design library contains various versions of the designs stored in the design library. A version of a design is referred to as a *block*. The block name can be the same as or different than the top module name of the design. For example, for a design with a top module named my_design, you could have blocks named my_design_preplace, my_design_postplace, and my_design_postcts.

Feedback

For each block, the design library can contain one or more of the following views:

- Design view

  The design view contains the layout information for the block.

- Frame view

  The frame view is an abstraction of the layout view that contains only the information needed for placement and routing. The exclusion of unnecessary data reduces the database size and routing time. For information about generating frame views, see Generating Frame Views in the *Library Manager User Guide*.

- Outline view

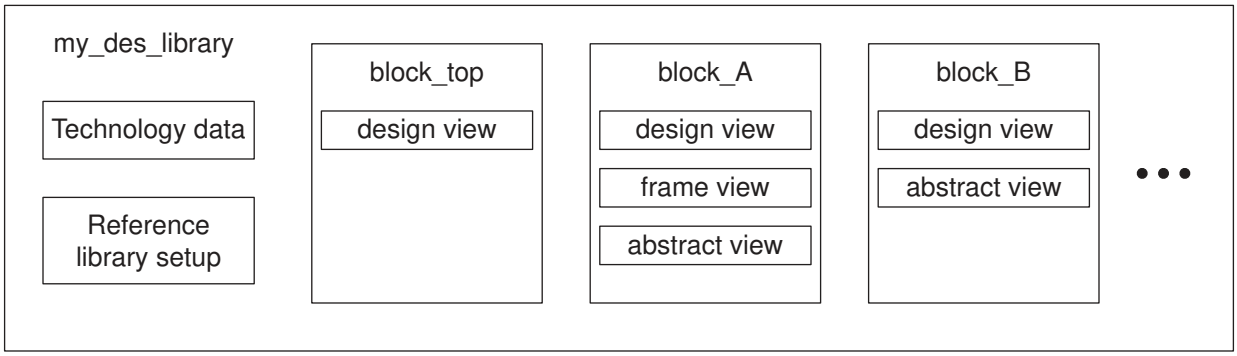
  The outline view is used for floorplan creation for very large hierarchical blocks. It contains the hierarchy information, but no leaf cells or nets. For information about creating and using the outline view, see Creating Dense and Sparse Outline Views for Large Designs in the *Fusion Compiler Design Planning User Guide*.

- Abstract view

  The abstract view is a lightweight representation of the block that contains only the information needed to perform top-level global placement, timing, and other tasks. For information about creating and using the abstract view, see the *Fusion Compiler™ Design Planning User Guide*.

Figure 1 shows a high-level view of the contents of a typical design library.

Figure 1 Design Library Contents



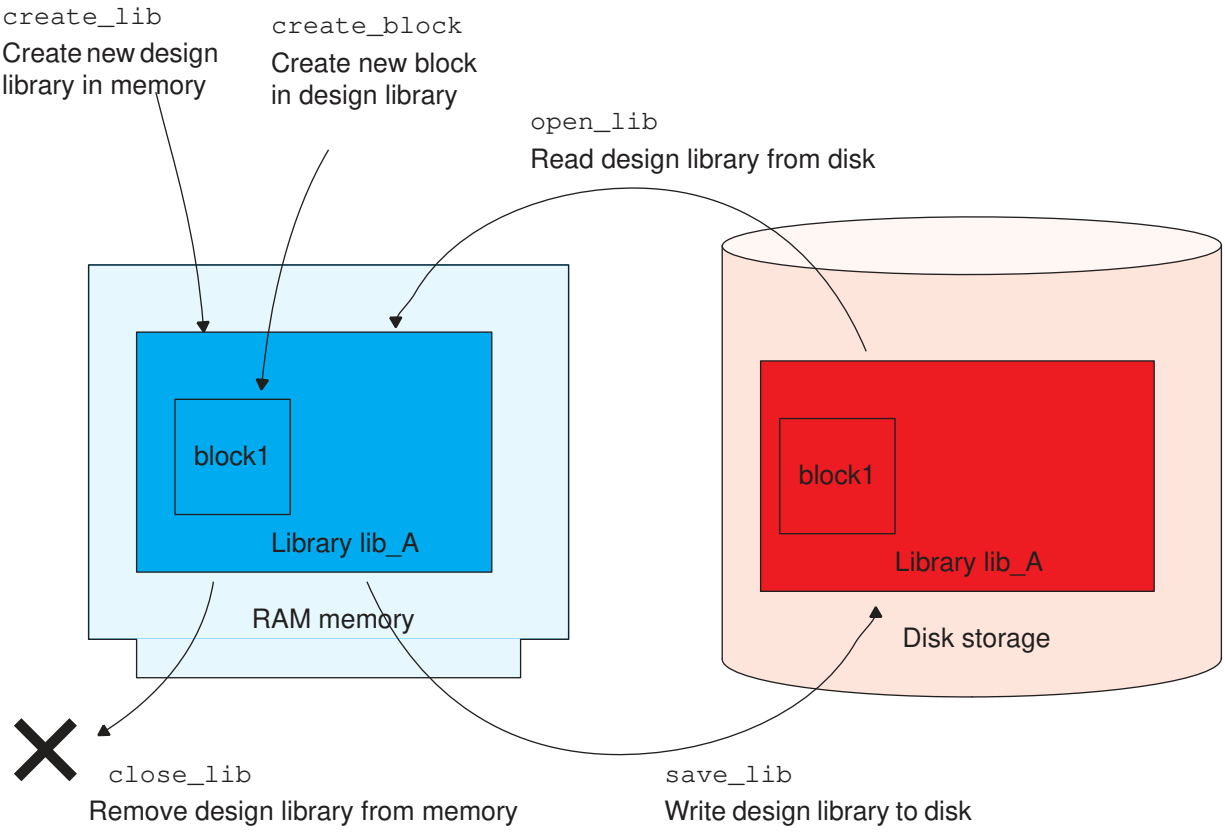
A design library can contain one or more different designs, or a complete hierarchical design that includes all views of the design. You can store the parts of a design in a single design library or in multiple design libraries that share the same technology information.

The *current library* is the default design library affected by library-related commands. By default, the last library opened is the current library. You can explicitly set the current library by using the `current_lib` command.

Feedback

The `create_lib` command creates a new design library and optionally specifies the associated technology data and reference library setup. You create blocks in a design library by using the `create_block` command. The `save_lib` command saves an open design library to disk and the `open_lib` command opens a saved design library for block access, as shown in the following figure.

Figure 2 Creating, Opening, Editing, Saving, and Closing a Library



Each design library has a *reference library list* that points to other libraries containing blocks referenced in the design library. In other words, a block that you edit in the design library contains *instances* of blocks defined in other libraries. The referenced blocks can be leaf-level library blocks in cell libraries or soft macros, intellectual property (IP) blocks, or other types of blocks contained in other design libraries.

The following table briefly describes the commands that operate on design libraries.

Feedback

Table 1 Commands That Operate on Design Libraries

Command	Description
`close_lib`	Closes a library, removing it from memory
`copy_lib`	Copies a design library on disk
`create_lib`	Creates a design library in memory
`current_lib`	Sets or gets the current library
`move_lib`	Moves a library on disk
`open_lib`	Opens a saved library for viewing or editing
`report_lib`	Reports library information
`save_lib`	Saves a design library to disk
`get_libs`	Creates a collection of the libraries loaded in memory
`report_ref_libs`	Reports the reference libraries of a library loaded in memory
`set_ref_libs`	Sets the reference libraries for a library loaded in memory

For more information about working with design libraries, see

- Relative and Absolute Paths to Design Libraries
- Creating a Design Library
- Opening a Design Library
- Setting the Current Design Library
- Querying a Design Library
- Saving a Design Library
- Design Library Recovery Data
- Closing a Design Library
- Design Library Open Count

See Also

- Reference Library List
- Loading the Technology Data

Feedback

- Blocks
- Library Packaging

Relative and Absolute Paths to Design Libraries

When you create, open, query, save, or close a design library, you can specify an absolute path, a relative path, or no path at all with the library name, as in the following examples.

```
# No path, opens lib in search_path
fc_shell> open_lib lib_A
...

# Relative path, current directory
fc_shell> open_lib ./lib_A
...

# Relative path, parent of current directory
fc_shell> open_lib ../lib_A
...

# Relative up 2 levels, down 1
fc_shell> open_lib ../../mylibs/lib_A
...

# Absolute path to lib
fc_shell> open_lib /remote/home/mylibs/lib_A
...
```

This is how the tool finds the specified library:

- No path – The tool looks in the directories specified by the `search_path` variable, in the order that the directories are listed in the variable.
- Relative path – The tool looks in the specified directory relative to the current working directory (`.`) or its parent directory (`..`).
- Absolute path – The tool looks in the specified absolute path, starting with the slash character (`/`).

See Also

- Relative and Absolute Paths to Reference Libraries
- Library Packaging

Feedback

Creating a Design Library

To create a design library, use the `create_lib` command. When you run this command to create a new design library, you must specify the library name. Slash (/) and colon (:) characters are not allowed in library names.

For example, to create a new design library named my_libA, use the following command:

```
fc_shell> create_lib my_libA
{my_libA}
```

By default, the `create_lib` command does not save the design library to disk. To save the library to disk when you create it, set the `lib.setting.on_disk_operation` application option to `true` before running the `create_lib` command. You can also save the design library to disk by using the `save_lib` command, as described in Saving a Design Library.

You can optionally specify the following items:

- The reference libraries

  The reference libraries contain the leaf-level blocks such as standard cells and hard macros. To specify the reference library list when you create the design library, use the `-ref_libs` option. For more information, see Specifying a Design Library's Reference Libraries.

- The technology data

  The technology data specifies the process technology information such as measurement units, layers, unit tile dimensions, and routing rules. To provide the technology data when you create the design library, use one of the following methods:

  - Load a technology file by using the `-technology` option. This is the preferred method for providing the technology data.
  - Reference a technology library by using the `-use_technology_lib` option.

  For more information, see Loading the Technology Data.

- The scale factor

  The scale factor specifies the number of database units per micron. The scale factor must be identical for a design library and all of its reference libraries. If you specify reference libraries, their scale factor is the default scale factor for the design library.

  For information about setting the scale factor, see Specifying the Scale Factor.

Feedback

See Also

- Relative and Absolute Paths to Design Libraries
- Opening a Design Library
- Reference Library List
- Loading the Technology Data
- Reading a Hierarchical Design Into a Single Design Library
- Reading a Hierarchical Design Into Multiple Design Libraries

Specifying the Scale Factor

The design data in the database is stored in internal units. The scale factor of a design library determines the internal distance unit which represents the smallest distance that can be represented in the database. The Fusion Compiler tool converts the user-specified data to internal units. The default user-specified unit is a micron, but you can change this default.

The tool uses the scale factor to convert floating point numbers into integers when storing data in the cell libraries and design libraries. When the tool saves a floating point number, it multiplies the number by the scale factor and then rounds the number to an integer. When the tool retrieves the number, it divides the number by the scale factor. For example, if the scale factor is 1000, the tool stores a value of 0.0032 as 3 (0.0032 * 1000 = 3.2, which rounds to 3). The tool retrieves this value as 0.003 (3 / 1000).

If the scale factor is not large enough, the precision of specified coordinates might be affected. For example, assume that you create a net shape with the following command:

```
fc_shell> create_shape -shape_type rect -layer M1 \
   -boundary {{0.500 0.500} {0.5053 0.5053}}
```

With a scale factor of 1000, the bounding box is stored as {500 500} {505 505}, instead of {5000 5000} {5053 5053}. With a scale factor of 10000, the bounding box is stored as {5000 5000} {5053 5053}.

The scale factor for a design library must be compatible with the session's scale factor which the tool derives and sets based on the scale factor of the existing design library and its reference libraries, and it must be a multiple of the length precision of the technology file associated with the design library.

- To determine the scale factor for a reference library, query its `scale_factor` attribute.
- To determine the length precision of the technology file, check its `lengthPrecision` attribute.

Feedback

While creating a new library, if you specify the reference libraries, the tool determines the scale factor of the new library based on the scale factor of the reference libraries. If all the reference libraries have the same scale factor, the new library is created with the same scale factor. If the reference libraries have different scale factors, the scale factor of the new library is the LCM (lowest common multiple) of the scale factor of all reference libraries. By default, the tool creates the new libraries with a scale factor of 10000, which allows distances down to 1 Angstrom to be represented. The following example shows that the tool creates the design library with 4000 scale factor using the reference libraries with scale factors of 2000 and 4000:

```
fc_shell> create_lib test.nlib \
   -ref_libs {mytest_hvt_std_s4000.nlib mytest_hvt_std_2000.ndm}
fc_shell> get_attribute [get_lib test.nlib] scale_factor
#4000
fc_shell> get_attribute [get_lib mytest_hvt_std_2000] scale_factor
#4000
```

In this case, the tool sets the session's scale factor to 4000.

In the same session, for a design library with a scale factor of 10000, you can open and use the reference library with a scale factor of 2000. This is because the reference library's scale factor is compatible with the session's scale factor, as shown in the following example:

```
fc_shell> open_lib test.nlib
fc_shell> get_attribute [get_lib test.nlib] scale_factor
#10000
fc_shell> set_ref_lib -library test.nlib \
   -add {mytest_*_2000.ndm}
```

The recommended scale factor is the technology length precision as specified by the `lengthPrecision` attribute in the technology file. To use the technology length precision as the scale factor, set the `lib.setting.use_tech_scale_factor` application option to `true` before running the `create_lib` command.

```
fc_shell> set_app_options \
   -name lib.setting.use_tech_scale_factor -value true
```

If neither the default scale factor nor the technology length precision is appropriate for your design library, you can explicitly specify the scale factor by using the `-scale_factor` option with the `create_lib` command.

Reading a Hierarchical Design Into a Single Design Library

The following script reads a hierarchical design into a single design library (see Figure 3). The input netlists are in Verilog format and the input layout information is in DEF format. You must read in the netlists in hierarchical order, from bottom to top.

```
create_lib mydesign -ref_libs {std_cell_lib.ndm} \
   -technology mytech.tf
```

Feedback

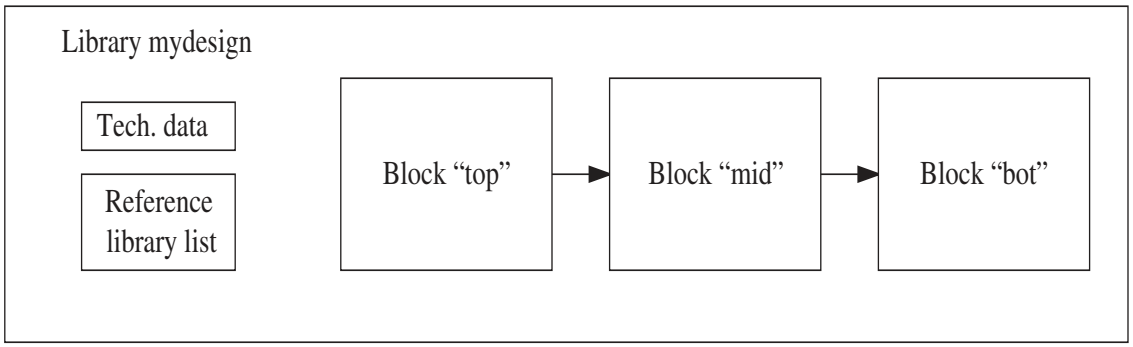
```
read_verilog bot.v       # Implicitly creates new block "bot"
read_def bot.def
...

read_verilog mid.v       # Implicitly creates new block "mid"
read_def mid.def
...

read_verilog top.v       # Implicitly creates new block "top"
read_def top.def
...

save_lib
```

Figure 3 Hierarchical Data Stored in a Single Design Library



See Also

- Design Libraries
- Reference Library List
- Specifying a Design Library’s Reference Libraries

Reading a Hierarchical Design Into Multiple Design Libraries

The following script reads a hierarchical design into different design libraries for different levels of the design (see Figure 4). The input netlists are in Verilog format and the input layout information is in DEF format.

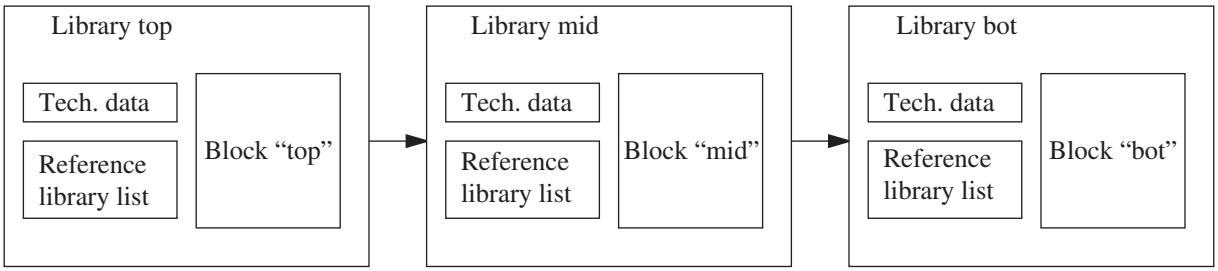
```
# Create library bot and read the design
create_lib bot -ref_libs {std_cell_lib.ndm} \
 -technology mytech.tf
read_verilog bot.v          # Implicitly creates new block "bot"
read_def bot.def
...
save_lib
```

Feedback

```
# Create library mid and read the design
create_lib mid -ref_libs {std_cell_lib.ndm bot} \
 -technology mytech.tf
read_verilog mid.v          # Implicitly creates new block "mid"
read_def mid.def
...
save_lib

# Create library top and read the design
create_lib top -ref_libs {std_cell_lib.ndm bot mid} \
 -technology mytech.tf
read_verilog top.v          # Implicitly creates new block "top"
read_def top.def
...
save_lib
```

Figure 4 Hierarchical Design Data Stored in a Multiple Libraries



See Also

- Design Libraries
- Reference Library List
- Specifying a Design Library’s Reference Libraries

Opening a Design Library

To open an existing design library saved on disk, use the `open_lib` command:

```
fc_shell> open_lib my_libA
Information: Loading library file '/usr/lib/StdCells.ndm' (FILE-007)
Information: Loading library file '/usr/lib/RAMs.ndm' (FILE-007)
Information: Loading library file '/usr/lib/PhysicalOnly.ndm' (FILE-007)
{my_libA}
```

The tool opens the specified design library, makes that library the current library, and opens all of its associated reference libraries. Opening a design library means loading it into memory and making its blocks accessible. To reduce memory usage, you can restrict the process, voltage, and temperature combinations loaded into memory from the cell

Feedback

libraries by using the `set_pvt_configuration` command, as described in Loading Timing Information Based on Operating Corners. To enable interpolation during timing analysis and optimization, create scaling groups by using the `define_scaling_lib_group` command. For details, see Creating Scaling Groups in the *Fusion Compiler Timing Analysis User Guide*.

Note:

Before opening a design library, ensure that the `link_library` variable setting is the same as when the design library was created or last opened. Otherwise, the tool will rebuild the cell libraries in its reference library list.

By default, the `open_lib` command opens the design library in read/write mode and the associated reference libraries in read-only mode. You can override the default read/write modes by using the `-read` and `-ref_libs_for_edit` options with the `open_lib` command.

See Also

- Relative and Absolute Paths to Design Libraries
- Creating a Design Library
- Saving a Design Library
- Closing a Design Library
- Reference Library List
- Configuring Cell Libraries

Setting the Current Design Library

The *current library* is the default library affected by library-related commands. By default, the library most recently opened is the current library. You can explicitly set any open library to be the current library by using the `current_lib` command:

```
fc_shell> current_lib my_libA
{my_libA}
```

To determine which library is the current library, use the `current_lib` command by itself:

```
fc_shell> current_lib
{my_libA}
```

See Also

- Creating a Design Library
- Opening a Design Library

Feedback

Querying a Design Library

You can get the following types of information about the open libraries:

- The current library – use the `current_lib` command.

```
fc_shell> current_lib
{my_design}
```

- The libraries loaded in memory – use the `get_libs` command.

```
fc_shell> open_lib lib_A
Information: Loading library file '... libA' ...
Information: Loading library file '... ref_lib1' ...
Information: Loading library file '... stdhvt.ndm' ...
{lib_A}

fc_shell> open_lib lib_B
Information: Loading library file '... libB' ...
Information: Loading library file '... ref_lib2' ...
Information: Loading library file '... stdhvt.ndm' ...
{lib_B}

# Get all open libs
fc_shell> get_libs
{lib_A lib_B ref_lib1 ref_lib2 stdhvt.ndm}

# Get the libs opened explicitly
fc_shell> get_libs -explicit
{lib_A lib_B}

# Get the libs opened as ref libs
fc_shell> get_libs -implicit
{ref_lib1 ref_lib2 stdhvt.ndm}
```

- The blocks in a library – use the `get_blocks` or `list_blocks` command.

```
fc_shell> get_blocks -all
{lib_A:block1.design lib_A:block2.design lib_B:blocka.design
lib_B:blockb.design}

fc_shell> list_blocks
Lib lib_A /path/libs/lib_A tech current
 -> 0 block1.design Apr-20-16:36
 +> 0 block2.design Apr-20-16:38 current
Lib lib_B /path/libs/lib_B tech
```

```
 -> 0 blocka.design Apr-18-14:02
 -> 0 blockb.design Apr-18-14:20

fc_shell> list_blocks lib_B
Lib lib_B /path/libs/lib_B tech
 -> 0 blocka.design Apr-18-14:02
 -> 0 blockb.design Apr-18-14:20
```

The characters at the beginning of each row of the block list indicate the following:

```
 + block open
 - block not open
 > block selected to resolve link references
 * block modified and not yet saved
```

The word `tech` at the end of the `Lib` line indicates that the library has technology data, and the word `current` means it is the current library.

- The associated reference libraries – use the `report_ref_libs` command.

```
fc_shell> current_lib lib_B
{lib_B}
fc_shell> report_ref_libs
...
Library: lib_B
...
   Name           Path          Location
-------------------------------------------------
*+ ref_lib1       ../../libs    /path/libs/ref_lib1
*+ ref_lib2       ../../libs    /path/libs/ref_lib2
*+ stdhvt         ../../libs    /path/libs/std/stdhvt.ndm
```

The characters at the beginning of each row of the library list indicate the following:

```
 * library currently open
 + library has technology data
 - library name and location not available
```

- The associated process technology data – use the `get_techs` command.

```
fc_shell> get_techs
{tech28nm1p tech13nm2x}
fc_shell> get_techs -of_objects lib_A
{tech28nm1p}
```

- The parasitic technology (defined by a TLUPlus file) associated with a library – use the `report_lib` command.

```
fc_shell> report_lib -parasitic_tech my_tech_lib
...
Library: my_tech_lib
...
Full name: /path/tech-info/.../my_tech_lib:my_tech
```

Feedback

```
File name: /path/tech-info/.../my_tech_lib
Design count: 0
No timing data

Parasitic tech data:
----------------------------------------
 Parasitic tech name:             my_tech_Cmax
 Parasitic itf technology name:   my_tech_Cmax
 Parasitic tech type:             TLUPLUS
 Parasitic source file name:      /path/Tlup/...
 ...
```

See Also

- Relative and Absolute Paths to Design Libraries
- Opening a Design Library
- Setting the Current Design Library
- Saving a Design Library
- Closing a Design Library

Saving a Design Library

When you create a design library or edit its contents, the changes are stored only in memory. To save a design library to disk, use the `save_lib` command.

```
fc_shell> current_lib
{lib_A}
fc_shell> save_lib
Saving library 'lib_A'
1
fc_shell> save_lib lib_B
Saving library 'lib_B'
1
fc_shell> save_lib -all
Saving all libraries...
5
```

The `save_lib` command saves all blocks in the design library that have been modified and not yet saved. Be sure to save a new or edited library before you close it.

Feedback

You can save a design library in compressed format to reduce the file size by using one of the following methods:

- To save all open blocks in a library in compressed format, use the `-compress` option with the `save_lib` command.
- To save all design data in the current tool session in compressed format, set the `lib.setting.compress_design_lib` application option to `true`. If you set the application option back to `false` (the default), any design libraries created from that point are saved in uncompressed format. However, previously compressed libraries are preserved in their compressed format.

To save previously compressed libraries to uncompressed libraries, set the `is_compressed` attribute on the library and all of its blocks to `false` before saving them.

Note:

The tool compresses design libraries only. Cell libraries cannot be compressed.

See Also

- Relative and Absolute Paths to Design Libraries
- Creating a Design Library
- Opening a Design Library
- Setting the Current Design Library
- Closing a Design Library

Design Library Recovery Data

You can automatically save design library recovery data to disk. This recovery data consists of all those blocks that are modified in memory, but not yet saved. In the event that the tool unexpectedly terminates, you can load and edit the last automatically saved snapshot of this data. The tool automatically creates recovery data in the following ways:

- Time-based save – Modified blocks are automatically saved at specified, regular intervals of time after you execute a command.
- Recovery save – The tool attempts to save modified data when it terminates unexpectedly. This will succeed most of the time, but is not guaranteed. The recovery save is triggered only when the library is in a valid state.

When the tool automatically saves your work, this snapshot and your session log file are saved to a location that you specify. Automatically saved data is identified by the name of the base library and process ID. This automatically saved data contains only the modified designs of the base library. For each library, only one automatically saved snapshot of data

is present at a time. After the data is automatically saved, the previously saved data is removed from the disk. Recovery-saved data is an exception to this rule.

See Also

- Configuring Time-Based Save
- Configuring Recovery Save
- Including Automatically Saved Data in a Library Package
- Reporting Automatically Saved Data
- Restoring Automatically Saved Data
- Removing Automatically Saved Data

Configuring Time-Based Save

You can save modified blocks at regular intervals. To enable the tool to automatically save your work during the current session, run the `start_auto_save` command. You can specify the frequency of the tool automatically saving your work, location where the tool saves the automatically saved data, amount of disk space allocated for storing recovery data, whether you want to save the environment settings, and whether you want to keep the automatically saved data when you exit the session.

- To control how frequently the tool saves your data, specify a length of time (in seconds) with the `-frequency` option. The minimum value is 1800 seconds (30 minutes).

  The following example instructs the tool to automatically save the data every 45 minutes:

  ```
  fc_shell> start_auto_save -frequency 2700
  Auto-saving started
  ```

  If you do not specify a time interval, the tool saves your design every 30 minutes.

- To control where the tool saves automatically saved data, specify a location with the `-location` option.

  The following example instructs the tool to save automatically saved data to an auto_save subdirectory:

  ```
  fc_shell> start_auto_save -frequency 2700 -location ./auto_save
  ```

  If you do not specify a location, the tool saves automatically saved data to the current working directory.

- To limit the amount of disk space available to store your automatically saved data, specify a limit with the `-allowed_disk_space` option. The units must be specified in

gigabytes or megabytes. When the specified disk space limit is reached, the tool stops automatically saving data.

The following example allocates 2.2 MB of disk space to store automatically saved data:

```
fc_shell> start_auto_save -frequency 2700 -location ./auto_save \
   -allowed_disk_space 2.2MB
```

For more information about the `start_auto_save` settings, see the `start_auto_save` man page.

To stop automatically saving recovery data during a session, run the `stop_auto_save` command.

By default, the automatically saved data of your session is saved to disk. To discard the automatically saved data when you close a design library, use the `-auto_saved_data clear` option of the `close_lib` command.

```
fc_shell> close_lib libD -auto_saved_data clear
```

See Also

- Design Library Recovery Data
- Configuring Recovery Save
- Including Automatically Saved Data in a Library Package
- Reporting Automatically Saved Data
- Restoring Automatically Saved Data
- Removing Automatically Saved Data

Configuring Recovery Save

One of the ways in which modified design library data is automatically saved is when the tool receives an unexpected termination signal.

The following application options control how modified design library data is automatically saved when the tool receives an unexpected termination signal:

- `design.enable_recovery_save`

  This application option controls whether recovery data is saved or not.

  By default, this application option is `false`. To enable this feature, set it to `true`. When the option is set to `true`, recovery data is saved. When it is `false`, recovery data is not saved.

Feedback

- `design.recovery_save_dir_path`

  This application option specifies the location where the recovery data is saved.

  By default, this application option is set to the current working directory.

See Also

- Design Library Recovery Data
- Configuring Time-Based Save
- Including Automatically Saved Data in a Library Package
- Reporting Automatically Saved Data
- Restoring Automatically Saved Data
- Removing Automatically Saved Data

Including Automatically Saved Data in a Library Package

You can include or exclude automatically saved data when you write a library package using the `write_lib_package` command. By default, automatically saved data is included in the library package.

The following examples show how you can include or exclude automatically saved data when you write a library package:

```
fc_shell> write_lib_package libD.pkg -auto_saved_data include

fc_shell> write_lib_package libD.pkg -auto_saved_data exclude
```

See Also

- Design Library Recovery Data
- Reporting Automatically Saved Data
- Restoring Automatically Saved Data
- Removing Automatically Saved Data

Reporting Automatically Saved Data

To report information about the automatically saved recovery data, use the `report_auto_save` command. The report lists the name, size, location, and time-stamp of the automatically saved data. It also specifies whether the data was automatically saved as a recovery save.

Feedback

By default, the `report_auto_save` command reports information about the automatically saved data of the current library. To list information about the automatically saved data of a different library, specify the name or path of that library with the `-library` option.

For example, to list the information about the automatically saved data of the libD design library, specify the `-library` option with the `report_auto_save` command:

```
fc_shell> report_auto_save -library libD
*********************************************************
Original Library : libD

Auto-saved library path                 : /remote/home/mylibs/libD_1914
Auto-saved library timestamp            : Wed Apr 29 00:54:44 2020
Auto-saved library size                 : 2912391 bytes
Auto-saved library is_recovery_save     : false
Auto-saved library context              : create_net
```

When no library is specified, the command reports the information about the automatically saved data for the current library.

See Also

- Restoring Automatically Saved Data
- Removing Automatically Saved Data

Restoring Automatically Saved Data

Restoring automatically saved data is not automatic. You must initiate recovery of the automatically saved data. You can report, open, and remove automatically saved data of an opened or closed library.

The `recover_auto_save` command restores the automatically saved data of the current library.

```
fc_shell> recover_auto_save

Information: Loading library file
 '/remote/home/mylibs/libD_1914' (FILE-007)
```

You can also recover the automatically saved data using the `-recover` option of the `open_lib` command:

```
fc_shell> open_lib libD -recover

Information: Loading library file
 '/remote/home/mylibs/libD_1914' (FILE-007)
```

Feedback

See Also

- Reporting Automatically Saved Data
- Removing Automatically Saved Data

Removing Automatically Saved Data

To remove automatically saved recovery data from disk, use the `remove_auto_save` command.

```
fc_shell> remove_auto_save
```

By default, the `remove_auto_save` command removes the automatically saved data of the current library. To remove automatically saved data of a different library, specify the name or path of that library with the `-library` option. The following example removes the automatically saved library from the libD design library:

```
fc_shell> remove_auto_save -library libD
```

See Also

- Reporting Automatically Saved Data

Closing a Design Library

When you no longer need access to data in a library, you can close it. Be sure to save the changes in the library before you close it.

To close a library, use the `close_lib` command.

```
fc_shell> current_lib
{lib_A}
fc_shell> close_lib
Closing library 'lib_A'
1
fc_shell> close_lib lib_B
Closing library 'lib_B'
1
fc_shell> close_lib -all
Closing all libraries...
1
```

To deliberately close an edited library and discard the changes:

```
fc_shell> close_lib -force
Closing library 'lib_A'
1
```

Feedback

Caution:

By default, when you close a design library by using the `close_lib` command, the tool does not save the open blocks and does not issue a warning about unsaved design changes. To save new versions for all open blocks in a library, use the `-save_designs` option with the `close_lib` command.

If the library *open count* is 1 or more after you execute the `close_lib` command, the library remains open.

To remove an unwanted design library from disk, first use the `remove_blocks` command to remove all the blocks in the library. Note that removed blocks cannot be recovered. Then use operating system commands to remove the unwanted library directory.

You can save a design library in compressed format before you close it by specifying the `-compress` and `-save_designs` options with the `close_lib` command. The tool compresses design libraries only. Cell libraries cannot be compressed.

See Also

- Relative and Absolute Paths to Design Libraries
- Creating a Design Library
- Opening a Design Library
- Setting the Current Design Library
- Saving a Design Library
- Design Library Open Count

Design Library Open Count

The *open count* of a design library is an integer specifying the number of times the library has been opened. The tool monitors the open count to keep the library open as long as you need access to it, based on a matching number of `open_lib` and `close_lib` commands used on the library.

When you open a closed library or create a new library, its open count is set to 1. Each time the same library is reopened, its open count is incremented by 1:

```
fc_shell> open_lib lib_C
Information: Loading library file ...
{lib_C}
...
fc_shell> open_lib lib_C
Information: Incrementing open_count of library 'lib_C' to 2. (LIB-017)
...
fc_shell> open_lib lib_C
```

Feedback

```
Information: Incrementing open_count of library 'lib_C' to 3. (LIB-017)
...
```

Opening a design library also opens its associated reference libraries, which increments the open count for each reference library as well.

The `close_lib` command decrements a library's open count by 1. If the resulting new count is 1 or more, the library remains open; or if the new count is 0, the library is closed and removed from memory.

```
fc_shell> close_lib lib_C
Information: Decrementing open_count of library 'lib_C' to 2. (LIB-018)
1
...
fc_shell> close_lib lib_C
Information: Decrementing open_count of library 'lib_C' to 1. (LIB-018)
1
...
fc_shell> close_lib lib_C
Closing library 'lib_C'
1
```

To determine the open count of a library without affecting the count:

```
fc_shell> get_attribute [get_libs lib_C] open_count
3
```

To close a library irrespective of its open count:

```
fc_shell> close_lib lib_C -purge
Closing library 'lib_C'
1
```

Using the `save_lib` command does not affect the open count.

Be sure to save a new or edited library before you close it. The tool does not warn you about unsaved data before it closes the library.

See Also

- Creating a Design Library
- Opening a Design Library
- Saving a Design Library
- Closing a Design Library

Feedback

Cell Libraries

A *cell library* is a unified library that contains the logical and physical information for a specific technology and one or more of its library cells. Each library cell is represented as a single object with multiple views that contain various types of information about the cell.

This type of library is used only as a reference library for design libraries; it does not itself have design-specific data, a reference library list, or lower-level libraries. Reference libraries contain basic leaf-level blocks such as standard cells, I/O pads, and hard macros.

You can create cell libraries by performing library preparation in the Library Manager tool as described in the *Library Manager User Guide* or by performing library configuration when you create the design library in the implementation tool, as described in Configuring Cell Libraries.

When you open a design library, the tool implicitly opens its associated cell libraries. To open a cell library directly, specify the complete library name, including any extension, such as .ndm. For all other commands that operate on a cell library, specify the library name without the extension. For example,

```
fc_shell> open_lib /path/nlibs/std32hvt.ndm
Information: Loading library file '/path/nlibs/std32hvt.ndm' (FILE-007)
{std32hvt}
...
fc_shell> get_libs
{lib_A lib_B std32hvt}
...
fc_shell> current_lib std32hvt
{std32hvt}
fc_shell> close_lib std32hvt
Closing library 'std32hvt'
1
```

See Also

- Reference Library List
- Specifying a Design Library’s Reference Libraries
- Library Packaging

Reference Library List

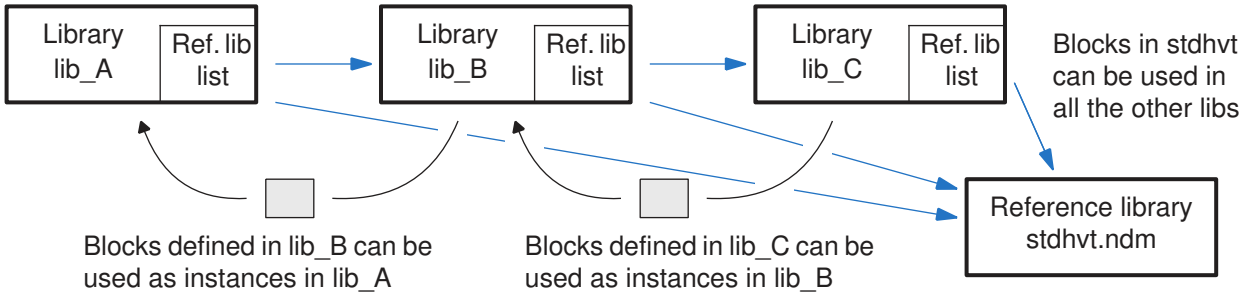
When you build a design, the building blocks for the design, whether library cells, hard macros, or other blocks, must be in the libraries associated with the design library. These associated libraries can be cell libraries or other design libraries. They are called reference libraries of the design library and are specified in the design library's *reference library list*.

Feedback

The entries in a reference library list each describe a one-way, one-level relationship between two libraries in a design hierarchy. If you set lib_B as a reference library of lib_A, you can use blocks in lib_B as instances in lib_A, but not the other way around.

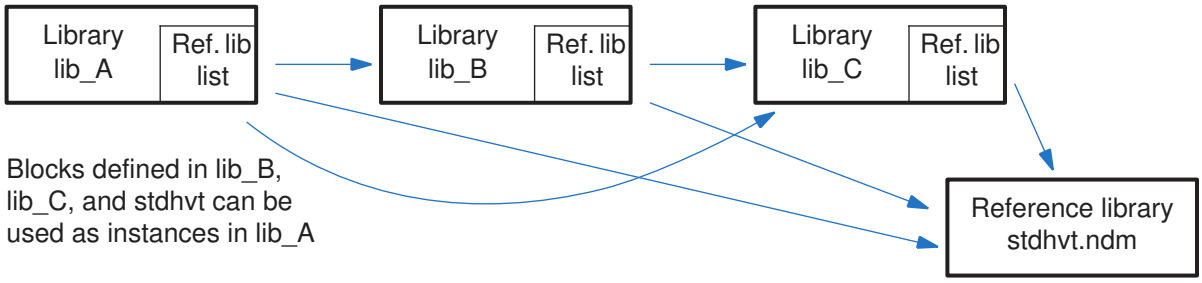
You can set lib_B to be a reference library of lib_A, and set lib_C to be a reference library of lib_B, as shown in the following figure.

Figure 5 Reference Library Lists



In that case, you can use blocks from lib_C as instances in lib_B, and use blocks in lib_B as instances in lib_A. However, you cannot directly use blocks in lib_C as instances in lib_A unless you also define a reference library relationship directly between lib_A and lib_C, as shown in the following figure.

Figure 6 Reference Library List Spanning Multiple Levels



To avoid naming conflicts between libraries or blocks in different physical hierarchies, set the `use_hier_ref_libs` attribute on the child design libraries to `true`. When you set this attribute on a child library, the `link_block` command uses the reference library list of the child library to resolve references for any blocks referenced from that library down the block hierarchy.

In the following example, the tool uses the reference libraries from the sub_block1 and sub_block2 child libraries when linking the top-level block:

```
fc_shell> set_attribute [get_libs sub_block1] \
     use_hier_ref_libs true
fc_shell> set_attribute [get_libs sub_block2] \
     use_hier_ref_libs true
fc_shell> current_block top
fc_shell> link_block
```

Note, however, that the tool does not honor the `use_hier_ref_libs` attribute if you link a block using a session-scoped reference library list. In this case, sub_block1 and sub_block2 link using the reference libraries in the session-scoped reference library list.

To report the reference libraries of a design library, use the `report_ref_libs` command. For details, see Reporting Reference Libraries

For more information, see

- Specifying a Design Library's Reference Libraries
- Reporting Reference Libraries
- Rebinding Reference Libraries of a Design Library

See Also

- Design Libraries
- Blocks
- Library Packaging

Specifying a Design Library's Reference Libraries

When you specify the reference libraries for a design library, you can specify

- Existing cell libraries, which can be single-file cell libraries generated with releases earlier than N-2017.09, directory-based cell libraries generated with N-2017.09 or later releases, or a mix of single-file and directory-based cell libraries.
- Physical source files, which can be physical libraries, LEF files, or database reference libraries

  If you specify physical source files, the tool creates Fusion Compiler cell libraries, as described in Configuring Cell Libraries, and associates the cell libraries with the design library.

Feedback

- Other design libraries, including their reference library lists

  When you link a design using another design library's reference library list, you are using a session-scoped reference library list. For more information, see Linking a Design With a Session-Scoped Reference Library List.

You can specify the libraries and source files using relative paths, absolute paths, or no paths. For details, see Relative and Absolute Paths to Reference Libraries.

- To specify the reference libraries when you create a new design library, use the `-ref_libs` option of the `create_lib` command.

```
fc_shell> create_lib lib_B \
 -ref_libs {../LIBS/lib_C ../STND/stdhvt.ndm} ...
{lib_B}
```

- To specify the reference libraries for an existing design library, open the library and use the `set_ref_libs` command:

```
fc_shell> open_lib lib_B
Information: Loading library file '/remote/home/mylibs/lib_B'
{lib_B}
fc_shell> set_ref_libs \
   -ref_libs {../LIBS/lib_C ../STND/stdhvt.ndm}
../LIBS/lib_C ../STND/stdhvt.ndm
```

  You can also use the `set_ref_libs` command on a closed design library by using the `-library` option to specify the name of the library:

```
fc_shell> set_ref_libs \
   -ref_libs {../LIBS/lib_C ../STND/stdhvt.ndm} -library design.nlib
../LIBS/lib_C ../STND/stdhvt.ndm
```

  - To add reference libraries to the existing reference library list, use the `-add` option.
  - To remove reference libraries from the existing reference library list, use the `-remove` option.
  - To replace the reference library list, use the `-ref_libs` option.
  - To specify physical source file, you can only use the `-ref_libs` option.

  If you change the reference libraries associated with a design library, update the references of an existing block in the library by using the `link_block -rebind` command. To control the order in which different lower-level block views are selected for rebinding, use the `set_view_switch_list` command.

The reference library link order depends on the order and types of files specified in the `-ref_libs` option. For cell libraries generated from physical source files, the logic library order in the `link_library` variable determines the cell library link order.

Feedback

To report the reference libraries that have been set for a design library, use the `report_ref_libs` command.

See Also

- Design Libraries
- Reference Library List

Relative and Absolute Paths to Reference Libraries

When you specify a reference library or physical source file for a design library using the `set_ref_libs` or `create_lib` command, you can specify a simple name with no path, a relative path, or an absolute path, as shown in the following examples.

```
fc_shell> set_ref_libs \            #Uses search_path
   -ref_libs stdB.ndm -library design.nlib

fc_shell> set_ref_libs \            # Uses search_path
   -ref_libs mylibs/stdB.ndm -library design.nlib

fc_shell> set_ref_libs \    # Relative to design library
   -ref_libs ./lib_A -library design.nlib

fc_shell> set_ref_libs \    # Relative up one level
   -ref_libs ../lib_A

fc_shell> set_ref_libs -ref_libs ../../mylibs/lib_A
   -library design.nlib

fc_shell> set_ref_libs \              # Absolute
   -ref_libs /remote/home/mylibs/lib_a -library design.nlib
```

This is how the tool finds the specified reference library or source file:

- Relative path (recommended) – The tool looks in the specified directory relative to the *design library's* directory (`.`) or the *design library's* parent directory (`..`) . This relative path is stored in the design library, so if you move the library and its reference libraries together to a new location, the tool still automatically finds each reference library located in the same place *relative to the design library*.
- Simple name – The tool looks in the directories specified by the `search_path` variable, in the order that the directories are listed in the variable. This path resolution *binds* the reference library to the design library. If you want to change the bindings based on a

new `search_path` variable setting, run the `set_ref_libs` command with the `-rebind` option. For details, see Rebinding Reference Libraries of a Design Library.

- Absolute path – The tool looks in the specified absolute path, starting with the slash character (`/`). This absolute path is stored in the design library, so you can move the design library while keeping its reference libraries in the same absolute locations.

Note:

Moving a design library and its reference libraries using the Library Packaging feature (`write_lib_package` and `read_lib_package`) moves the reference libraries to a new directory under the restored design library directory and automatically rebinds them. For details, see Restoring Data From a Library Package.

See Also

- Specifying a Design Library's Reference Libraries
- Reference Library List

Configuring Cell Libraries

You configure cell libraries by specifying the physical libraries with the `create_lib` or `set_ref_libs` command.

To configure a cell library,

1. Enable cell library configuration by setting the `lib.configuration.enable` application option to `true`.
2. Specify the search path for the library source files by setting the `search_path` variable.

```
fc_shell> set_app_var search_path ". \
   $ADDITIONAL_SEARCH_LOCATIONS $search_path"
```

3. Specify the logic libraries by setting the `link_library` variable.

```
fc_shell> set_app_var link_library "* $LINK_LIBRARY_FILES"
```

   To specify process labels for the logic libraries, set the `lib.configuration.process_label_mapping` application option. For more information about process labels, see Identifying the Process Associated With a Logic Library.

   By default, if a cell exists only in the logic libraries, the tool issues a warning message and ignores the cell. For information about creating dummy physical cells for these logical-only cells, see Handling Logical-Only Cells.

Feedback

4. Specify the library generation options by setting the `lib.configuration` application options.

   - To specify the central output directory for the generated cell libraries, set the `lib.configuration.central_output_dir` application option.

     If you do not set this application option, the command creates the cell libraries in the directory specified by the `lib.configuration.local_output_dir` application option.

     If you set this option, the tool checks the specified directory for existing cell libraries before generating new cell libraries. For each cell library referenced by the design library,

     - If the tool finds the cell library in the central location, it uses that cell library.
     - If the tool does not find the cell library in the central location, it assembles a new cell library in the directory specified by the `lib.configuration.local_output_dir` application option.

     To minimize the disk space requirement when you are working with a design team, ensure that the entire team uses the same setting for this application option and includes this location in the search path, as specified by the `search_path` variable. In general, the project lead creates the cell libraries in a central location and the rest of the team uses the generated cell libraries.

   - To specify the local output directory for the generated cell libraries, set the `lib.configuration.local_output_dir` application option.

     If you do not set this application option, the command creates the cell libraries in a directory named CLIBS under the current working directory.

     If you do not want to save the cell libraries generated in the current session after you exit the tool, either to save disk space or to ensure that you always have the latest cell libraries, set the `lib.configuration.remove_local_output_dir` application option to `true`. When this application option is `true`, the tool deletes the directory specified by the `lib.configuration.local_output_dir` application option before it exits.

     Note:

     If the `lib.configuration.remove_local_output_dir` application option is `true` and the block requires cell libraries to be stored in the local output directory, the tool rebuilds these libraries when you open the design library in a new session if the `lib.configuration.enable` application option is `true`.

- To specify a configuration script with library manager commands used to customize the library configuration process, set the `lib.configuration.default_flow_setup` application option.

  If you set this application option, the `create_lib` command sources this script at the start of the library configuration process.

- To specify assembly scripts with library manager commands used to assemble the cell libraries, set the `lib.configuration.assembly_scripts` application option.

  This application option maps an assembly script to each physical library specified in the `-ref_libs` option. If you set this application option, the `create_lib` command sources these scripts to generate the cell libraries instead of using the default flow.

- To display messages issued by the library manager, set the `lib.configuration.display_lm_messages` application option to `true`.

- If you are using LEF files for the physical source and need to convert the LEF site names to technology file site names, set the `lib.configuration.lef_site_mapping` application option.

- If you are using database reference libraries for the physical source, you must specify the IC Compiler executable with the `lib.configuration.icc_shell_exec` application option.

- To enable distributed processing using the Synopsys Common Distributed Processing Library (CDPL), set the `lib.configuration.cdpl_host` application option, as described in Using Distributed Processing When Configuring Cell Libraries.

5. Specify the physical source files by using the `-ref_libs` option with the `create_lib` or `set_ref_libs` command.

   You must specify a technology file by using the `-technology` option with the `create_lib` command; otherwise, the tool cannot create the cell libraries. The `set_ref_libs` command gets the technology data from the design library.

   For example, to create a design library using the mytech.tf technology file and the physical libraries specified by the $PHYSICAL_LIB_FILES variable, use the following command:

   ```
   fc_shell> create_lib -technology mytech.tf \
      -ref_libs $PHYSICAL_LIB_FILES mydesign
   ```

   You can specify any combination of physical libraries, LEF files, and database reference libraries. You can also specify Fusion Compiler cell libraries; for the library generation process, these are used only to exclude logic libraries from library generation.

If the `lib.configuration.remove_local_output_dir` application option is `true`, the tool stores the names of the physical source files in the design library so they can be used to rebuild the cell libraries when you open the design library in a new session.

For details about how the `create_lib` or `set_ref_libs` command generates the cell libraries, see Cell Library Generation.

Handling Logical-Only Cells

To create dummy frame views for logical-only cells and include them in the generated cell library, set the `lib.configuration.create_missing_macro_frame` application option to `true`.

When this application option is `true`, for each logical-only cell, the tool uses the height of the default site definition to estimate the width and height of the cell based on whether the cell is combinational or sequential and the number of cell pins. The estimated cell width is a multiple of the width of the default site definition; the estimated cell height is a multiple of the height of the default site definition. All input pins are assumed to be on the left side of the cell; all output and inout pins are assumed to be on the right side of the cell.

Cell Library Generation

To create the cell libraries, use the `create_lib`, `set_ref_libs`, or `open_lib` command

1. Sources the script file specified by the `lib.configuration.default_flow_setup` application option, if any.
2. Assembles the cell libraries by running either the default flow or the scripts specified in the `lib.configuration.assembly_scripts` application option.

   The default flow includes the following steps:

   a. Sets the design mismatch configuration to `auto_fix`.

   ```
   set_current_mismatch_config auto_fix
   ```

   b. Creates a library workspace for the exploration flow.

   ```
   create_workspace -flow exploration myworkspace
   ```

   c. Loads the technology file into the library workspace.

   The technology data source depends on the command that initiates the cell library generation.

   - For the `create_lib` command, the tool loads the technology file specified with the `-technology` option.
   - For the `set_ref_libs` and `open_lib` commands, the tool loads the technology file from the design library.

d. Loads the logic libraries and physical source files into the library workspace.

   The logic libraries are specified with the `link_library` variable. The physical source files are specified with the `-ref_libs` option.

   Note:

   If the `-ref_libs` option includes Fusion Compiler cell libraries, the tool loads only the logic libraries that are not used in those cell libraries into the library workspace.

e. Analyzes the library source files by running the `group_libs` command.

f. Validates and commits the exploration flow library workspace by running the `process_workspaces` command.

   The tool saves the cell libraries in the directory specified by the `lib.configuration.local_output_dir` application option.

See Also

- Creating a Design Library
- Specifying a Design Library's Technology File
- Specifying a Design Library's Reference Libraries

Using Distributed Processing When Configuring Cell Libraries

Distributed processing is available via the Common Distributed Processing Library (CDPL), which is included with every Fusion Compiler installation. Several utility programs are available to help you set up and monitor distributed processes. For more information about CDPL and the utility programs, see the *DP Manager User Guide* and *DP Manager Frequently Asked Questions* documents in the cdpl/doc directory under the Fusion Compiler installation directory.

To enable and configure distributed processing, set the `lib.configuration.cdpl_host` application option.

Feedback

The following protocols are supported:

- SH (shell process on the local host)

  To use SH, specify the maximum number of processes for the local host.

  For example, to specify a maximum of 16 processes on the local host, use the following command:

  ```
  fc_shell> set_app_options -name lib.configuration.cdpl_host \
     -value "-hosts localhost:16"
  ```

  You can omit the localhost name and specify just the maximum number of processes, as shown in the following example:

  ```
  fc_shell> set_app_options -name lib.configuration.cdpl_host \
     -value "-hosts :16"
  ```

- RSH (remote shell)

  To use RSH, specify the hosts and the maximum number of processes for each host.

  For example, to specify a maximum of two processes on machine1 and four processes on machine2, use the following command:

  ```
  fc_shell> set_app_options -name lib.configuration.cdpl_host \
     -value "-hosts machine1:2, machine2:4"
  ```

- SGE (Univa Grid Engine, formerly Sun Grid Engine)

  To use the SGE qsub command, use the following command:

  ```
  fc_shell> set_app_options -name lib.configuration.cdpl_host \
     -value SGE
  ```

  To specify additional qsub options, specify the options after SGE when setting the application option. You must use double quotation marks around the entire string. For example,

  ```
  fc_shell> set_app_options -name lib.configuration.cdpl_host \
     -value "SGE -l mem_free=10G"
  ```

Alternatively, you can use a hosts file to specify the distributed processing environment, as described in Using a Hosts File for Distributed Processing.

Feedback

Using a Hosts File for Distributed Processing

A hosts file defines the distributed processing environment by specifying the machines in the compute farm and the way jobs are launched.

To use a hosts file to specify the distributed processing environment, specify the name of the hosts file as the value of the `lib.configuration.cdpl_host` application option. For example,

```
fc_shell> set_app_options -name lib.configuration.cdpl_host \
   -value myhostsfile
```

The hosts file is an ASCII file in which each line provides information about one entity. The format for each line is:

flag | *hostname* | *num_slots* | *tmpDir* | *mode* | *command*

Comment lines begin with a pound sign (#).

The following examples show hosts files for the supported protocols:

- The SH protocol distributes processing over multiple cores on the same machine. The *hostname* field is not required, but in this example, localhost is entered as a reminder:

  ```
  # hosts file for SH
  1|localhost |8|/remote/users/tmp |SH| sh
  ```

- RSH infrastructure with two host machines and allowing a total of 10 worker slots (4 on one machine and 6 on the other)

  ```
  # hosts file for RSH
  1|engr_lab-x9|4|/remote/users/tmp |RSH| rsh
  1|engr_lab-x2|6|/remote/users/tmp |RSH| rsh
  ```

- SGE compute farm

  The host name field is empty for SGE farms.

  ```
  # hosts file for SGE
  1| |-1| |SGE| qsub -P bnormal -l mem_free=1G
  ```

- SGE compute farm in which the number of launched jobs is limited to 3 and a directory for temporary files is specified

  ```
  # hosts file for SGE
  1| |3| /remote/users/tmp |SGE| qsub -P bnormal arch=glinux
  ```

For detailed information about the hosts file syntax, see the *DP Manager User Guide*.

See Also

- Using Distributed Processing When Configuring Cell Libraries

Feedback

Updating Cell Libraries for Newer Tool Versions

When you move to a newer tool version, you might want to update the schema version or timing views of the cell libraries to take advantage of the latest tool features. The following topics describe how to perform these updates:

- Updating the Schema Version of a Cell Library
- Updating the Timing Views for Signoff Fusion

Updating the Schema Version of a Cell Library

To update the physical information in cell libraries when you move to a newer version of the Fusion Compiler tool, run the `set_ref_libs` command. When you run this command, the tool automatically generates updated cell libraries using the latest library schema.

To update cell libraries with the `set_ref_libs` command,

1. Specify the library generation options by setting the `lib.configuration` application options, as described in Configuring Cell Libraries.
2. Specify the physical source files by using the `set_ref_libs` command.

For example, to update the cell libraries associated with the mydesign design library, use the following command:

```
fc_shell> set_ref_libs -ref_libs $PHYSICAL_LIB_FILES \
   -library mydesign.nlib
```

For details about how the `set_ref_libs` command generates the cell libraries, see Cell Library Generation.

Updating the Timing Views for Signoff Fusion

Signoff Fusion requires timing views generated with version O-2018.06 or later of the library manager. To update older cell libraries,

1. Enable automatic updating of older timing views by setting the `lib.configuration.update_timing_view_for_pt_delay` application option to `true`.
2. Specify the library generation options by setting the `lib.configuration` application options, as described in Configuring Cell Libraries.
3. Open the design library by using the `open_lib` command.

For example, to update the timing views of the cell libraries associated with the mydesign design library, use the following commands:

```
fc_shell> set_app_options \
   -name lib.configuration.update_timing_view_for_pt_delay \
   -value true
fc_shell> open_lib mydesign
```

Linking a Design With a Session-Scoped Reference Library List

By default, the tool links the current block and its subblocks using the reference library list in the current block's design library. However, you can link the current block using another block library's reference library list, in this context called a session-scoped reference library list.

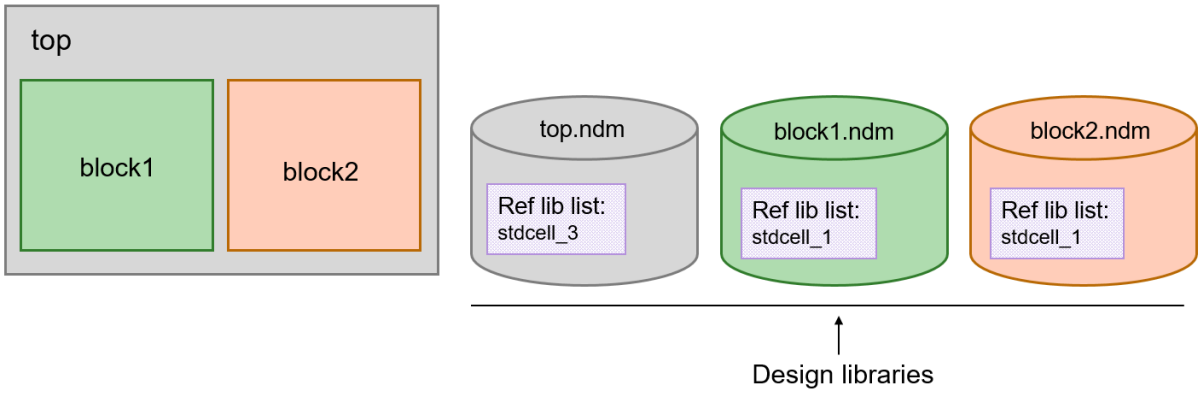
When you set a session-scoped reference library list, the `link_block` command links the current block using the session-scoped reference library list instead of the reference library list in the block's own design library. This allows you to update a block or its references without altering the block library's existing reference library list.

Setting a Session-Scoped Reference Library List

To set a session-scoped reference library list, use the `link.use_session_ref_libs` application option to specify the design library whose reference library list you want to use for that session. This design library must be open when you set the application option.

Suppose you have a design named top, which contains block1 and block2 subblocks.

Feedback



By default, the tool links each block using the reference library in each block library's reference library list:

```
fc_shell> open_lib top.ndm

fc_shell> open_block top
fc_shell> link_block   #link using top.ndm's stdcell_3 ref lib

fc_shell> current_block block1
fc_shell> link_block   #link using block1.ndm's stdcell_1 ref lib

fc_shell> current_block block2
fc_shell> link_block   #link using block2.ndm's stdcell_2 ref lib
```

To link all the blocks using the top.ndm library's reference library list, open the library and specify top.ndm with the `link.use_session_ref_libs` application option.

```
fc_shell> open_lib top.ndm
fc_shell> set_app_options -name link.use_session_ref_libs \
   -value top.ndm
```

Now top.ndm's reference library list, which contains just the stdcell_3 reference library, is the session-scoped reference library list. As a result, the `link_block` command links each block using top.ndm's stdcell_3 reference library:

```
fc_shell> open_block top
fc_shell> link_block   #link using top.ndm's stdcell_3 ref lib

fc_shell> current_block block1
fc_shell> link_block   #link using top.ndm's stdcell_3 ref lib

fc_shell> current_block block2
fc_shell> link_block   #link using top.ndm's stdcell_3 ref lib
```

Note that the `link.use_session_ref_libs` application option affects both explicit linking performed by the `link_block` command and implicit linking performed by commands that require a linked design, such as the `write_gds` and `write_oasis` commands.

Feedback

The application option does not, however, affect the linking of blocks that are open in separate sessions. To link these blocks using a session-scoped reference library list, set the `link.use_session_ref_libs` application option in each session.

If the session-scoped reference library list is missing a reference block or library cell referenced by a hierarchical block, the `link_block` command reports an unresolved reference and fails to link the block. The tool does not use the block library's own reference library list to resolve the reference.

When you set a session-scoped reference library list, the `report_ref_libs` command reports the reference libraries in the session-scoped reference library list, regardless of the current library, unless you specify a different library with the `-library` option.

Modifying a Session-Scoped Reference Library List

To modify a session-scoped reference library list, use the `set_ref_libs` command. The command modifies the session-scoped reference library list, regardless of the current library, unless you specify a different library with the `-library` option.

Because a session-scoped reference library list is a reference to, not a copy of, the specified library's reference library list, any changes to that library's reference library list are automatically reflected in the session-scoped reference library list.

To change the session-scoped reference library list in the current session, open the library whose reference library list you want to use as the new session-scoped reference library list and reset the value of the `link.use_session_ref_libs` application option. Simply resetting the value of the application option does not affect blocks that have already been linked. To relink these blocks using the new session-scoped reference library list, use the `link_block` command.

Caution:

If the session-scoped reference library list is missing a reference block or library cell referenced by a hierarchical block, the `link_block` command reports an unresolved reference and fails to link the block. The tool does not use the block library's own reference library list to resolve the reference.

Saving a Session-Scoped Reference Library List

By default, when you save a design library with the `save_lib` command, the session-scoped reference library list is not saved into the library. To save a session-scoped reference library list into one or more open design libraries, set the `design.save_session_ref_libs` application option to `true` before saving the libraries. This saves the session-scoped reference library list to the specified libraries' `ref_libs` attribute on disk. When you close and reopen the libraries, the tool links their blocks with the session-scoped reference library list.

Feedback

Caution:

When you save a session-scoped reference library list into a design library, the `save_lib` command overwrites the library's existing reference library list with the session-scoped reference library list.

The following example sets top.ndm's reference library list as a session-scoped reference library list and saves the list into the block1.ndm and block2.ndm design libraries:

```
# Open the libraries and set top.ndm's reference library
# list as the session-scoped reference library list
open_lib top.ndm
open_lib block1.ndm
open_lib block2.ndm
set_app_options -name link.use_session_ref_libs -value top.ndm

# Enable the tool to save the session-scoped reference library
# list into the open design libraries
set_app_options -name design.save_session_ref_libs -value true

# Save top.ndm's reference library list into the block1.ndm and
# block2.ndm design libraries
save_lib -all
close_lib -all
```

When you open the block1.ndm or block2.ndm library and report the library's reference libraries, you will see that the reference library list now contains top.ndm's stdcell_3 reference library:

```
fc_shell> open_lib block2.ndm
...
fc_shell> report_ref_libs
...
Library: block2
...
Name            Path                Location
--------------------------------------------------------------
*+ stdcell_3    ../../libs          /path/libs/std/stdcell_3
```

When you close a library whose reference library list is designated as a session-scoped reference library list, the tool resets the value of the `link.use_session_ref_libs` application option to an empty string. If you did not save the session-scoped reference library list into your open libraries and you subsequently relink their blocks, the tool links the blocks using the libraries' original reference library lists.

Reporting Reference Libraries

To report the reference libraries of a design library, use the `report_ref_libs` command:

```
fc_shell> create_lib lib_A -ref_libs \
  {../libs/SCLL.ndm ../libs/SCHH.ndm ../BLOCKS/MACROS}
```

Feedback

```
{lib_A}
fc_shell> report_ref_libs
...
   Name    Path                Location
   ------------------------------------------------------------------
*+ SCLL    ../libs/SCLL.ndm    /remote/project10/test1/libs/SCLL.ndm
*+ SCHH    ../libs/SCHH.ndm    /remote/project10/test1/libs/SCHH.ndm
*  MACROS  ../BLOCKS/MACROS    /remote/project10/test1/BLOCKS/MACROS
    "*" = Library currently open
    "+" = Library has technology information
```

The report shows the Name, Path (as originally specified), and Location (absolute path) of each reference library in the design library's reference library list.

The entry in the Path column is the simple name, relative path, or absolute path originally specified for the reference library.

If you set a session-scoped reference library list, the `report_ref_libs` command reports the reference libraries in the session-scoped reference library list, regardless of the current library, unless you specify a different library with the `-library` option.

See Also

- Specifying a Design Library's Reference Libraries
- Reference Library List

Loading Timing Information Based on Operating Corners

By default, the tool loads the timing information from all logic libraries included in the reference libraries. To include only the timing information that matches the valid operating corners (process, voltage, and temperature values) for your design, define a PVT configuration for the design library. The PVT configuration applies only to the current session and must be specified before you open the design library.

Note:

The PVT configuration is applied when loading the cell libraries. It does not apply to cell libraries that are already open when you run the `set_pvt_configuration` command.

A PVT configuration is a sequence of rules, each of which specifies the process labels, process numbers, voltages, and temperatures to match. If you want all the data that matches a specific set of operating corners, a single rule is sufficient. If you want to match a specific subset of many operating corners, you must specify multiple rules.

Feedback

To define a PVT configuration rule, use the `set_pvt_configuration` command.

- To create a new rule, use the `-add` option.

  By default, the command uses the rule_*n* naming convention. To specify the rule name, use the `-name` option.

- To modify an existing rule, use the `-rule` option to specify the rule name.
- To add filters to a rule, use one or more of the following options: `-process_labels`, `-process_numbers`, `-voltages`, and `-temperatures`.
- To remove filters from a rule, use the `-clear_filter` option.

For example, to create a PVT configuration rule that loads the timing data only for voltages of 0.65 or 1.32 volts and a temperature of 25 degrees for the mylib design library, use the following commands:

```
fc_shell> set_pvt_configuration -add \
   -voltages {0.65 1.32} -temperatures {25}
fc_shell> open_lib mylib
```

Rebinding Reference Libraries of a Design Library

A design library's associated reference libraries contain lower-level blocks used as instances in the design library's blocks. You associate or *bind* a reference library with a design library by using `-ref_libs` option of the `create_lib` command or by using the `set_ref_libs` command. If you specify a simple name for the reference library, the command resolves the path by using the `search_path` variable.

If you make a change that invalidates the reference library list, you need to *rebind* the reference libraries. If necessary, first set the `search_path` variable appropriately, then use the `set_ref_libs -rebind` command:

```
fc_shell> current_lib
{lib_A}
fc_shell> set_app_var search_path {. ../MYLIBS ../NLIBS}
. ../MYLIBS ../NLIBS
fc_shell> set_ref_libs -rebind
../MYLIBS/lib_C ../MYLIBS/lib_D ../NLIBS/stdhvt.ndm}
```

Rebinding a library does not affect the bindings of blocks already existing in the design library. To rebind these blocks using an updated reference library list, use the `link_block` command with the `-rebind` option.

See Also

- Specifying a Design Library's Reference Libraries
- Relative and Absolute Paths to Reference Libraries

Feedback

- Reporting Reference Libraries
- Reporting Unbound Objects
- Reference Library List

Configuring Cell Libraries

You configure cell libraries by specifying the physical libraries with the `create_lib` or `set_ref_libs` command.

To configure a cell library,

1. Enable cell library configuration by setting the `lib.configuration.enable` application option to `true`.
2. Specify the search path for the library source files by setting the `search_path` variable.

   ```
   fc_shell> set_app_var search_path ". \
      $ADDITIONAL_SEARCH_LOCATIONS $search_path"
   ```

3. Specify the logic libraries by setting the `link_library` variable.

   ```
   fc_shell> set_app_var link_library "* $LINK_LIBRARY_FILES"
   ```

   To specify process labels for the logic libraries, set the `lib.configuration.process_label_mapping` application option. For more information about process labels, see Identifying the Process Associated With a Logic Library.

   By default, if a cell exists only in the logic libraries, the tool issues a warning message and ignores the cell. For information about creating dummy physical cells for these logical-only cells, see Handling Logical-Only Cells.

Feedback

4. Specify the library generation options by setting the `lib.configuration` application options.
   - To specify the central output directory for the generated cell libraries, set the `lib.configuration.central_output_dir` application option.

     If you do not set this application option, the command creates the cell libraries in the directory specified by the `lib.configuration.local_output_dir` application option.

     If you set this option, the tool checks the specified directory for existing cell libraries before generating new cell libraries. For each cell library referenced by the design library,

     - If the tool finds the cell library in the central location, it uses that cell library.
     - If the tool does not find the cell library in the central location, it assembles a new cell library in the directory specified by the `lib.configuration.local_output_dir` application option.

     To minimize the disk space requirement when you are working with a design team, ensure that the entire team uses the same setting for this application option and includes this location in the search path, as specified by the `search_path` variable. In general, the project lead creates the cell libraries in a central location and the rest of the team uses the generated cell libraries.
   - To specify the local output directory for the generated cell libraries, set the `lib.configuration.local_output_dir` application option.

     If you do not set this application option, the command creates the cell libraries in a directory named CLIBS under the current working directory.

     If you do not want to save the cell libraries generated in the current session after you exit the tool, either to save disk space or to ensure that you always have the latest cell libraries, set the `lib.configuration.remove_local_output_dir` application option to `true`. When this application option is `true`, the tool deletes the directory specified by the `lib.configuration.local_output_dir` application option before it exits.

     Note:

     If the `lib.configuration.remove_local_output_dir` application option is `true` and the block requires cell libraries to be stored in the local output directory, the tool rebuilds these libraries when you open the design library in a new session if the `lib.configuration.enable` application option is `true`.

Feedback

- To specify a configuration script with library manager commands used to customize the library configuration process, set the `lib.configuration.default_flow_setup` application option.

  If you set this application option, the `create_lib` command sources this script at the start of the library configuration process.

- To specify assembly scripts with library manager commands used to assemble the cell libraries, set the `lib.configuration.assembly_scripts` application option.

  This application option maps an assembly script to each physical library specified in the `-ref_libs` option. If you set this application option, the `create_lib` command sources these scripts to generate the cell libraries instead of using the default flow.

- To display messages issued by the library manager, set the `lib.configuration.display_lm_messages` application option to `true`.

- If you are using LEF files for the physical source and need to convert the LEF site names to technology file site names, set the `lib.configuration.lef_site_mapping` application option.

- If you are using database reference libraries for the physical source, you must specify the IC Compiler executable with the `lib.configuration.icc_shell_exec` application option.

- To enable distributed processing using the Synopsys Common Distributed Processing Library (CDPL), set the `lib.configuration.cdpl_host` application option, as described in Using Distributed Processing When Configuring Cell Libraries.

5. Specify the physical source files by using the `-ref_libs` option with the `create_lib` or `set_ref_libs` command.

   You must specify a technology file by using the `-technology` option with the `create_lib` command; otherwise, the tool cannot create the cell libraries. The `set_ref_libs` command gets the technology data from the design library.

   For example, to create a design library using the mytech.tf technology file and the physical libraries specified by the $PHYSICAL_LIB_FILES variable, use the following command:

   ```
   fc_shell> create_lib -technology mytech.tf \
      -ref_libs $PHYSICAL_LIB_FILES mydesign
   ```

   You can specify any combination of physical libraries, LEF files, and database reference libraries. You can also specify Fusion Compiler cell libraries; for the library generation process, these are used only to exclude logic libraries from library generation.

Feedback

If the `lib.configuration.remove_local_output_dir` application option is `true`, the tool stores the names of the physical source files in the design library so they can be used to rebuild the cell libraries when you open the design library in a new session.

For details about how the `create_lib` or `set_ref_libs` command generates the cell libraries, see Cell Library Generation.

Handling Logical-Only Cells

To create dummy frame views for logical-only cells and include them in the generated cell library, set the `lib.configuration.create_missing_macro_frame` application option to `true`.

When this application option is `true`, for each logical-only cell, the tool uses the height of the default site definition to estimate the width and height of the cell based on whether the cell is combinational or sequential and the number of cell pins. The estimated cell width is a multiple of the width of the default site definition; the estimated cell height is a multiple of the height of the default site definition. All input pins are assumed to be on the left side of the cell; all output and inout pins are assumed to be on the right side of the cell.

Cell Library Generation

To create the cell libraries, use the `create_lib`, `set_ref_libs`, or `open_lib` command

1. Sources the script file specified by the `lib.configuration.default_flow_setup` application option, if any.
2. Assembles the cell libraries by running either the default flow or the scripts specified in the `lib.configuration.assembly_scripts` application option.

   The default flow includes the following steps:

   a. Sets the design mismatch configuration to `auto_fix`.

   ```
   set_current_mismatch_config auto_fix
   ```

   b. Creates a library workspace for the exploration flow.

   ```
   create_workspace -flow exploration myworkspace
   ```

   c. Loads the technology file into the library workspace.

   The technology data source depends on the command that initiates the cell library generation.

Feedback

- For the `create_lib` command, the tool loads the technology file specified with the `-technology` option.
- For the `set_ref_libs` and `open_lib` commands, the tool loads the technology file from the design library.

d. Loads the logic libraries and physical source files into the library workspace.

The logic libraries are specified with the `link_library` variable. The physical source files are specified with the `-ref_libs` option.

Note:

If the `-ref_libs` option includes Fusion Compiler cell libraries, the tool loads only the logic libraries that are not used in those cell libraries into the library workspace.

e. Analyzes the library source files by running the `group_libs` command.

f. Validates and commits the exploration flow library workspace by running the `process_workspaces` command.

The tool saves the cell libraries in the directory specified by the `lib.configuration.local_output_dir` application option.

See Also

- Creating a Design Library
- Specifying a Design Library's Technology File
- Specifying a Design Library's Reference Libraries

Loading the Technology Data

Each design library needs process technology information such as measurement units, layers, unit tile dimensions, and routing rules. This type of information comes from a technology file. The tool reads the data from the technology file and stores that information in a library. The technology file syntax is described in the *Synopsys Technology File and Routing Rules Reference Manual*.

A design library can get its technology information either directly from a technology file or indirectly through another library called a *technology library*. A technology library contains only technology data and does not contain design data or leaf-level reference library cells. The direct method is the preferred method. The two methods are summarized in the following figures.

Feedback

Figure 7 Direct Method of Specifying a Library's Technology Data

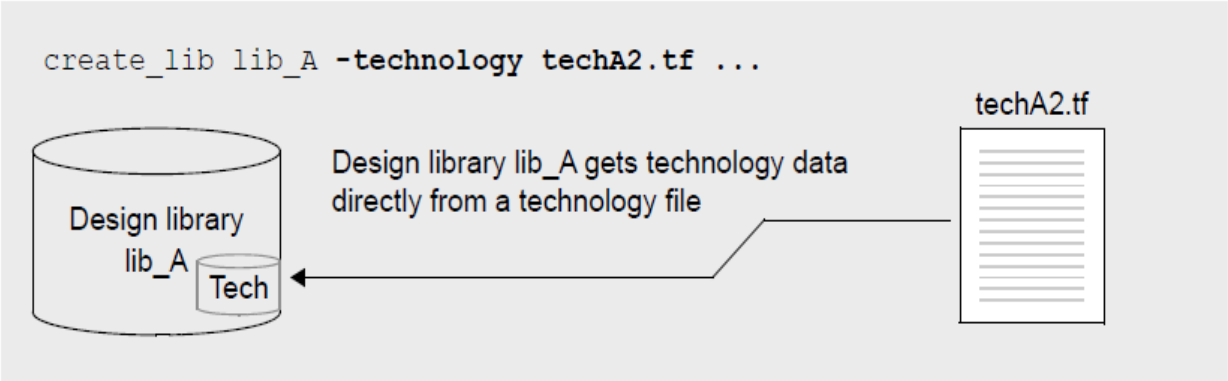
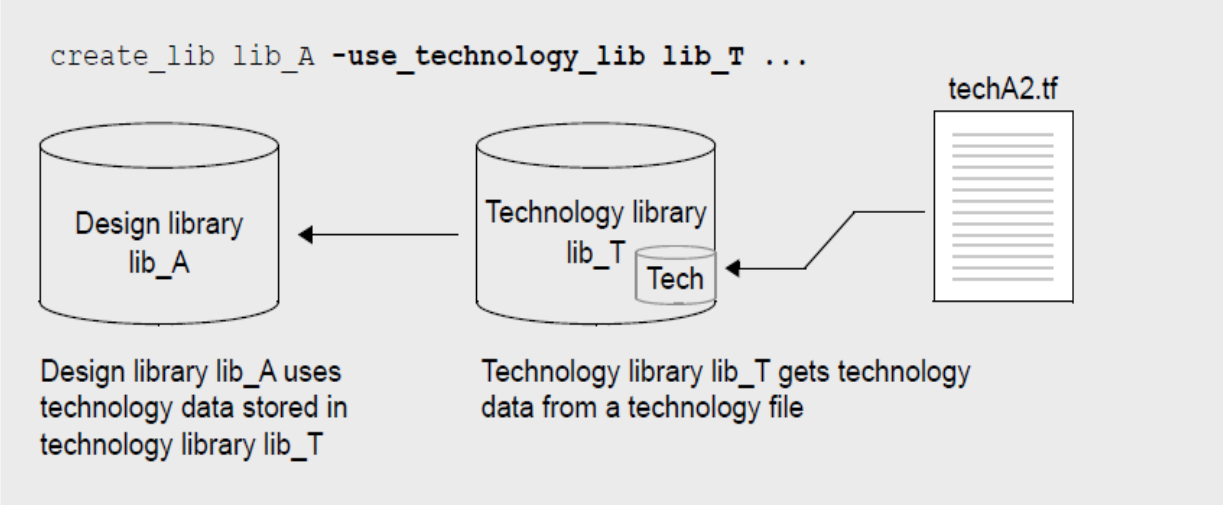


Figure 8 Indirect Method of Specifying a Library's Technology Data



To create a technology library, use the `create_workspace` or `create_lib` command, as shown in the following example:

```
fc_shell> create_lib lib_T -technology techA2.tf
```

Note:

The technology file does not define all technology information required by the Fusion Compiler tool, such as site symmetry and the routing direction for each layer. You must provide this information as described in Completing the Technology Data in the *Library Manager User Guide*.

To report the source of the technology data associated with a library, use the `report_ref_libs` command.

Feedback

If you load technology data into a design library that already contains technology data, the new technology data replaces the existing technology data, whether from a previous technology file, a technology library, or specified with Tcl commands. The new technology data must be upwardly compatible with the existing technology data, as explained in Updating the Technology Data.

You can determine the consequences of loading new technology data into an existing design library by using the `report_tech_diff` command.

For more information, see

- Specifying a Design Library's Technology File
- Specifying a Design Library's Technology Library
- Updating the Technology Data

See Also

- Reference Library List
- Specifying a Design Library's Reference Libraries

Specifying a Design Library's Technology File

To specify the technology file for a new design library, use the `-technology` option with the `create_lib` command:

```
fc_shell> create_lib my_lib -technology /usr/TECH/my-tech.tf
Information: Loading technology file
   '/usr/TECH/my-tech.tf' (FILE-007)
{my_lib}
```

To update the technology file for an existing library, use the `read_tech_file` command:

```
fc_shell> open_lib my_lib
{my_lib}
fc_shell> read_tech_file /usr/TECH/my-tech2.tf
Information: Replacing technology file '/usr/TECH/my-tech.tf'
   with '/usr/TECH/my-tech2.tf' (LIB-037)
1
```

In either case, the command reads the technology file and stores the technology data in the library. After that, the library no longer needs or uses the technology file. However, be sure to keep the technology file for reference and for future updates.

You can specify the technology file using an absolute path, relative path, or a simple name. If you use a simple name, the tool looks for the technology file in the directories defined by the `search_path` variable.

Feedback

When you read a technology file, the tool performs syntax and semantic checks on the contents of the technology file. The technology file checker has two modes:

- User mode (the default)

  In this mode, the tool downgrades the message severity for suspected errors for the general user.

- Developer mode

  In this mode, the tool increases the message severity for suspected errors for the technology file developer to correct or waive. To enable this mode, set the `file.tech.library_developer_mode` application option to `true`.

See Also

- Loading the Technology Data
- Updating the Technology Data

Specifying a Design Library's Technology Library

To specify the technology library for a new design library, use the `-ref_libs` and `-use_technology_lib` options with the `create_lib` command:

```
fc_shell> create_lib my_lib -ref_libs {lib_T ref_A} \
  -use_technology_lib lib_T ...
```

To update the technology file associated with an existing design library, use the `set_ref_libs` command:

```
fc_shell> set_ref_libs -ref_libs lib_T2 -use_technology_lib lib_T2 \
   -library my_lib.nlib
lib_T2
```

The technology library specified with the `-use_technology_lib` option must be a library containing technology data previously read into it from a technology file, and it must be a library in the `-ref_libs` list.

Note:

If the `-ref_libs` option contains physical source files, you must specify a technology file, as described in Specifying a Design Library's Technology File, rather than a technology library.

Feedback

See Also

- Loading the Technology Data
- Updating the Technology Data
- Reference Library List

Updating the Technology Data

When you update the technology data for a design library, the tool verifies that the new technology data is compatible with the existing technology data, and if so, replaces the existing technology data with the new technology data.

When updating the technology data, the tool performs checks for the following objects:

- Unit tiles

  The new technology file must contain at least the same `Tile` definitions, with the same dimensions, as the old technology file. The new technology can define additional tiles.

  Caution:

  If the new technology data is missing unit tiles defined in the existing technology data, the tool issues error messages and does not load the updated technology data.

- Layers

  If the new technology data is missing layers defined in the existing technology data, any objects on these layers become unbound. To report unbound objects after updating the technology data, use the `report_unbound -shape` command. If you later update the technology data to redefine the missing layers, the unbound objects are rebound to the layers.

  Caution:

  If missing layers are referenced in the technology data, as in the case of a reference to a `ContactCode` section that is defined in a missing layer, the tool issues error messages and does not load the updated technology data. If missing layers are referenced in design rule definitions, however, the tool issues warning messages and ignores the design rules but loads the updated technology data.

  If the layer definitions change in the updated technology data, objects remain bound to layers based on the layer number, rather than the layer name. For example, assume that the existing technology data defines the METAL10 layer as layer number 63. If the new technology data defines the M10 layer as layer number 63, any objects on layer number 63 are now bound to the M10 layer.

Feedback

- Contact codes

  The new technology data must contain at least the same `ContactCode` definitions as the old technology data, with the same settings for the contact name and the `cutLayer`, `lowerLayer`, and `upperLayer` attribute settings.

  Caution:

  If the new technology data is missing contact codes defined in the existing technology data, the tool issues error messages and does not load the updated technology data.

  The new technology can define additional contact codes, and the design rules for the existing contact codes do not need to be the same as in the old technology.

See Also

- Loading the Technology Data

Blocks

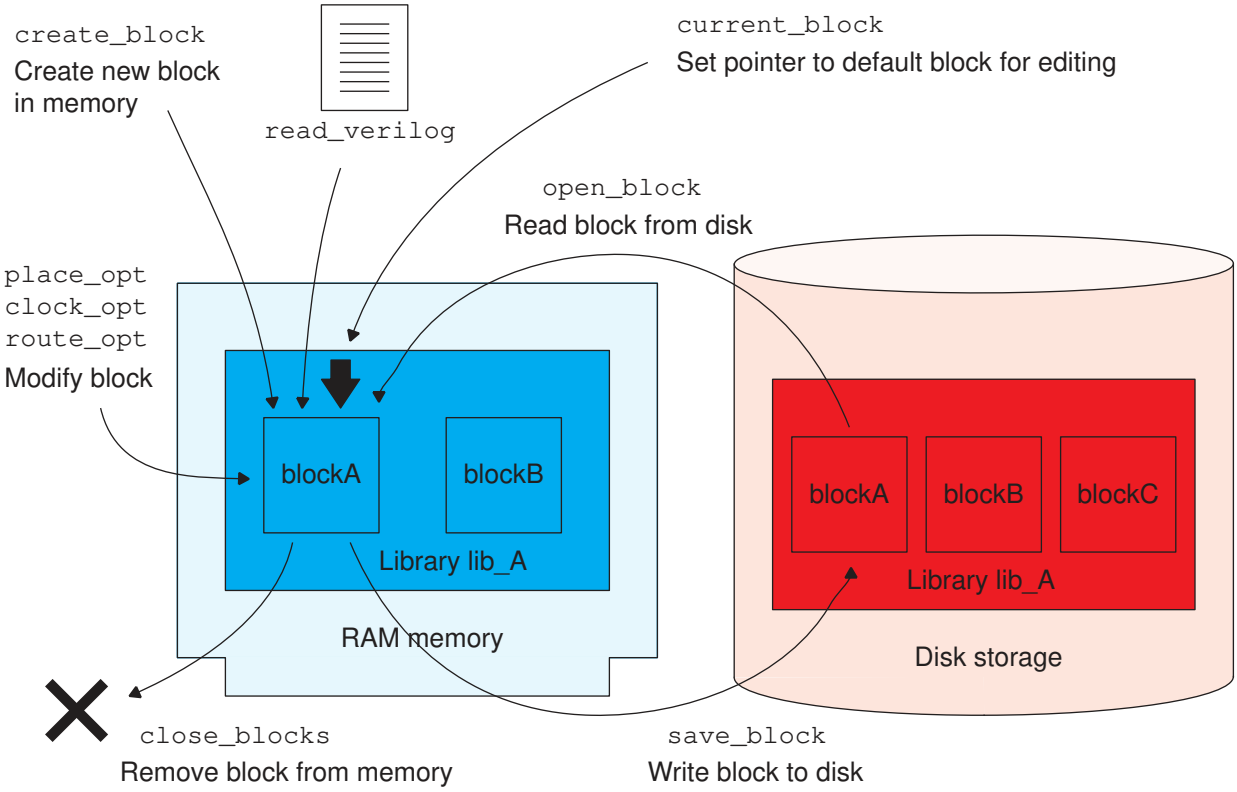
A *block* is a container for physical and functional design data. A typical project consists of the following steps to create and edit each block, at each hierarchical level:

1. Read the design netlist using the `read_verilog` command. This implicitly creates a new block, like using the `create_block` command.
2. Read other design information and constraints by using commands such as `load_upf`, `read_sdc`, and `read_def`.
3. Perform physical implementation by using the `place_opt`, `clock_opt`, and `route_opt` commands.
4. Save the block by using the `save_block` command.

You create, query, and edit blocks in memory. To save a new or edited block to disk, use the `save_block` command; or to read a block from disk into memory, use the `open_block` command, as shown in Figure 9.

Feedback

Figure 9 Creating, Opening, Editing, Saving, and Closing a Block



A block can have multiple *views*, each view containing an alternative representation of the same design:

- Design view – a complete physical view that contains the full design information of the cell
- Frame view – a limited physical view that contains only the information needed to perform placement and routing
- Abstract view – an interface-only block representation used in complex hierarchical designs
- Outline view – a simplified representation used for floorplanning

The following table briefly describes the commands that operate on blocks.

Table 2 Commands That Operate on Blocks

Command	Description
`close_blocks`	Closes a block, removing it from memory

Feedback

Table 2 Commands That Operate on Blocks (Continued)

Command	Description
`copy_block`	Copies a block to a new block in the same or different design library
`create_block`	Creates a new block in memory
`current_block`	Sets or gets the current block
`get_blocks`	Creates a collection of the blocks loaded in memory
`link_block`	Resolves the references in a block
`list_blocks`	Lists the blocks (stored on disk) in the specified design libraries
`move_block`	Moves a block to a new block, design library, or view
`open_block`	Opens a saved block for viewing or editing
`rebind_block`	Rebinds the references in a block
`remove_blocks`	Removes a block from memory and disk
`reopen_block`	Changes the open mode (read-only or edit) of an opened block
`save_block`	Saves a block to disk

For more information about blocks, see:

- Block Naming Conventions
- Block Labels
- Block Views
- Creating a Block
- Opening a Block
- Setting the Current Block
- Querying Blocks
- Saving a Block
- Copying a Block
- Block Open Count
- Block Types

Feedback

See Also

- Design Libraries
- Reference Library List
- Cell Libraries
- Library Cells
- Library Packaging

Block Naming Conventions

You specify a block using the following format:

```
[libName:][blockName][/labelName][.viewName]
```

where *blockName* is required when you use the `create_block` and `open_block` commands.

For example,

```
fc_shell> open_block my_lib:MUX2/ver1.design
```

The following default naming conventions apply to a specified block:

- Default library: current library
- Default block: current block
- Default label: none
- Default view
  - For the `open_block` and `create_block` commands, the default view is the design view.
  - For the `copy_block`, `move_block`, `remove_blocks`, `rename_block`, and `save_block` commands,
    - If you do not specify the source block, the command performs the task on the view of the current block.
    - If you specify the source block without a view, the command performs the task on all available views of the source block.

Feedback

For example, if the current block is the design view of the `my_lib:MUX2` block, the following commands save only the design view of the block:

```
save_block
save_block MUX2.design
save_block my_lib:MUX2.design
```

The following examples show commands you can use to save all views of the block:

```
save_block MUX2
save_block my_lib:MUX2
save_block MUX2 -as NEW_MUX2
```

The current block is the default block affected by block-related commands. The last block opened is the current block by default. You can explicitly set the current block by using the `current_block` command.

You can create and access a block in the current library using a simple name:

```
create_block MUX2
save_block MUX2
close_blocks MUX2
open_block MUX2
```

If you want to use a more complex block name, append a *label* using a slash character:

```
fc_shell> create_block MUX3/ver1
```

Block labels are useful for maintaining different block versions in hierarchical designs.

Note that you cannot use a period character to add a block name extension:

```
fc_shell> create_block MUX3.ver1
Error: Invalid block name 'MUX3.ver1'. (DES-008)
...
```

The period character is reserved for designating the *view* name: .design, .frame, .abstract, or .outline.

For more information about block naming conventions, see

- Block Labels
- Block Views
- Timing Views in Design Libraries

Block Labels

When you create a block, you can optionally specify a label as part of the block name. You can use labels to make different versions of the same block. Each version is considered a

Feedback

separate block. For a hierarchical design, you can effectively manage different versions of the full hierarchy by using different labels for different design versions.

If used, the label follows a slash character:

```
[libName:][blockName][/labelName][.viewName]
```

For example, to specify “ver1” as a block label:

```
fc_shell> open_block my_lib:MUX2/ver1.design
```

The default label is an empty string. In other words, there is no default label. You can use labels only for designs in design libraries, not library cells in cell libraries.

To save a block on disk using a new or different label, use the `save_block` command with the `-label` option. You can save different versions of a block while maintaining the current block setting in the tool:

```
current_block blkZ
save_block -label v1
save_block -label v2
current_block
```

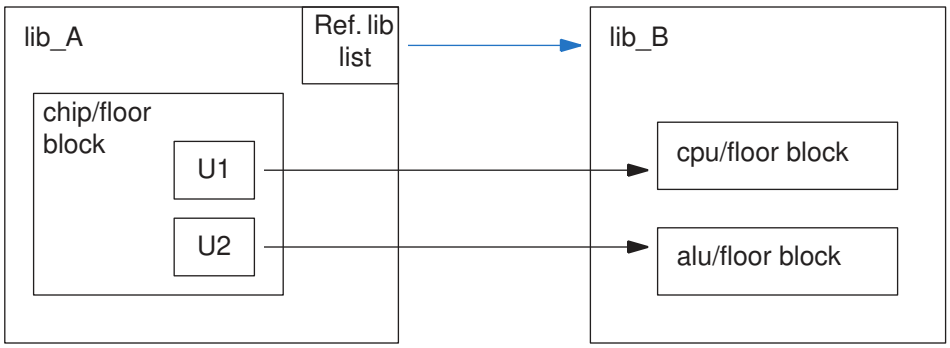
See Also

- Block Views
- Block Naming Conventions

Using Labels in a Hierarchical Design

Suppose that you have a top-level design named chip/floor in a design library named lib_A. The block contains instances U1 and U2, which represent lower-level blocks named cpu/floor and alu/floor in reference library lib_B, as shown in the following figure. All the blocks are at the floorplanned stage.

Figure 10 Hierarchical Design Using Block Names With Labels



Feedback

You decide to proceed to the placement stage, and you want to keep a copy of the design at the floorplanned stage for possible modification later. You save the chip/floor block to a new block named chip/placed by using the `save_block -hierarchical` command:

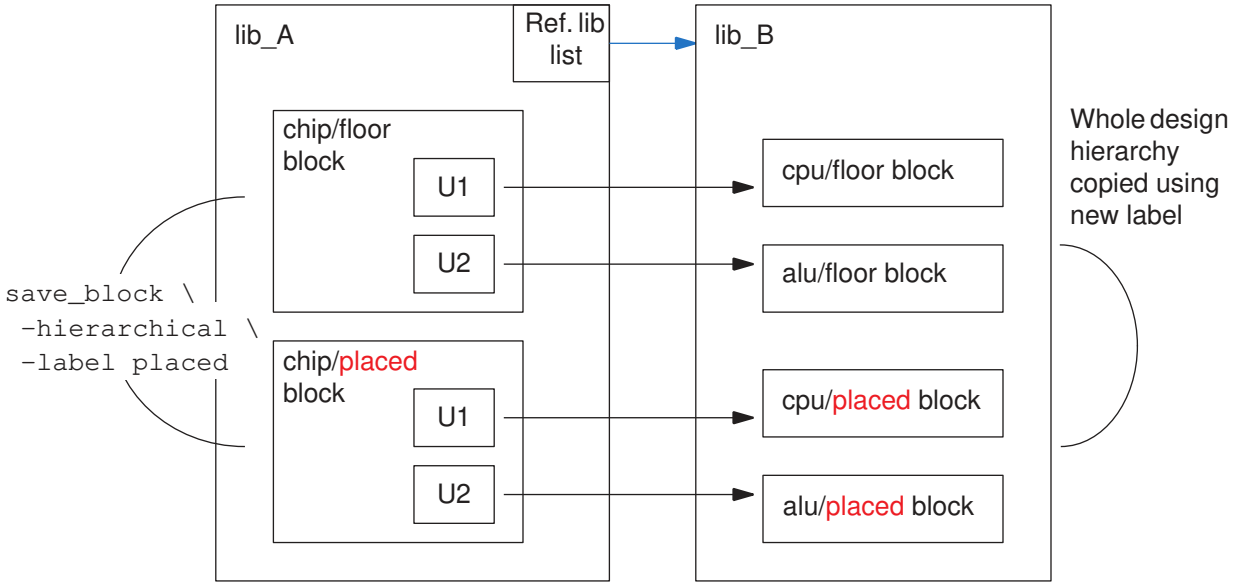
```
fc_shell> current_block
fc_shell> save_block -hierarchical -label placed
```

or equivalently:

```
fc_shell> save_block -hierarchical -as chip/placed
...
```

This effectively copies the chip/floor block to a new block, including its full hierarchy, creating new blocks in both the current library and the lower-level reference library. Then you can edit the new top-level and new lower-level blocks, leaving the original blocks unchanged, as shown in the following figure.

Figure 11 Copying a Hierarchical Design With Labels



See Also

- Block Labels

Feedback

Creating a Label When Reading a Verilog Netlist

When you read a Verilog netlist with the `read_verilog` command, the command creates a new block to contain the netlist. If you want the new block to have a label, you can do either of the following:

- Specify the label name together with the design name:

```
fc_shell> read_verilog mychip.v -design chip/floor
...
```

- Set the `file.verilog.default_user_label` application option to the desired label name before you read in the Verilog netlist:

```
fc_shell> set_app_options \
  -name file.verilog.default_user_label -value floor
file.verilog.default_user_label floor
fc_shell> read_verilog mychip.v -design chip
Information: Reading Verilog into new design 'chip/floor' in lib ...
...
```

See Also

- Block Labels
- Block Naming Conventions

Block Views

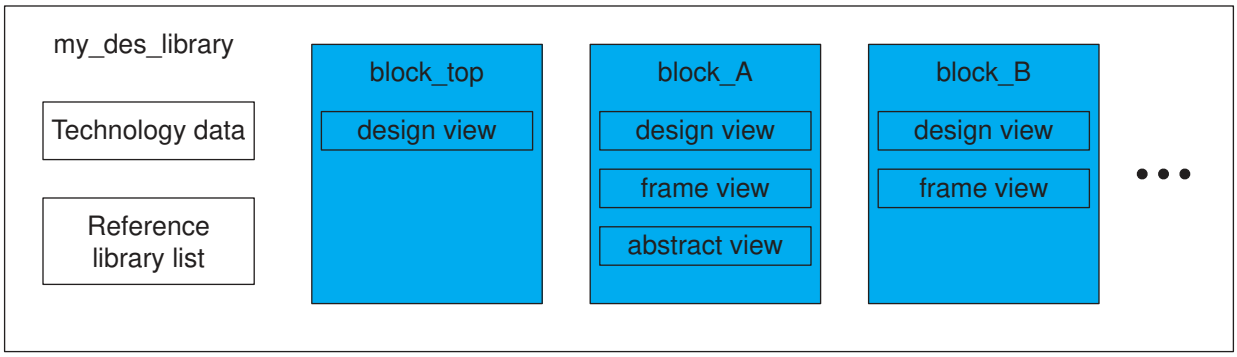
A block can have multiple *views*, with each view containing an alternative representation of the block used for a specific purpose. The Fusion Compiler infrastructure supports the following types of views for blocks in design libraries:

- Design view (`.design`) – a complete physical view that contains the full design information of the cell, including placed block instances and routed nets. This is the default view type.
- Frame view (`.frame`) – a limited physical view that contains only the information needed to perform placement of the block as an instance and routing to the ports of the instance: the block outline, pins, via regions, and routing blockages.
- Abstract view (`.abstract`) – a simplified view that contains only the interface information of a subdesign, used for placement and timing analysis at the next higher level of the design. For details, see the *Fusion Compiler™ User Guide*.
- Outline view (`.outline`) – a simplified view of a large child block that contains only the hierarchy information, without nets or leaf-level library cells, used for floorplan creation. For details, see the *Fusion Compiler™ Design Planning User Guide*.

Feedback

The following figure shows how the block views are stored in a typical design library.

Figure 12 Design Library Contents



The following additional types of views are supported for library-cell blocks in cell libraries:

- Timing view (`.timing`) – a functional description of the timing, logic, and power characteristics of a library cell. The implementation tool uses this information for timing analysis, power analysis, and optimization.
- Layout view (`.layout`) – a physical-only view of the shapes in a library cell, which contains the same information as the GDSII description of the cell, not including connectivity and pin information. The library manager tool uses this view as a data source for generating the design view for a library cell, and the implementation tool uses this view for mask generation.

Note:

The layout view (`.layout`) applies only to leaf-level library cells. This is different from the “layout view” shown in GUI windows when you view a design. The GUI “layout view” shows the full chip layout, which is actually a display of the design view of the block.

When you create, open, save, or close a block, you specify the block view as the last item in the full block name, after the period character:

```
[libName:][blockName][/labelName][.viewName]
```

For example, the following command opens the frame view of block NAND2/ver1:

```
fc_shell> open_block NAND2/ver1.frame
```

Feedback

If you do not specify a view name,

- The `create_block` and `open_block` commands open the design view of the block.
- The `copy_block`, `move_block`, `remove_blocks`, `rename_block`, and `save_block` commands operate on all available views of the block.

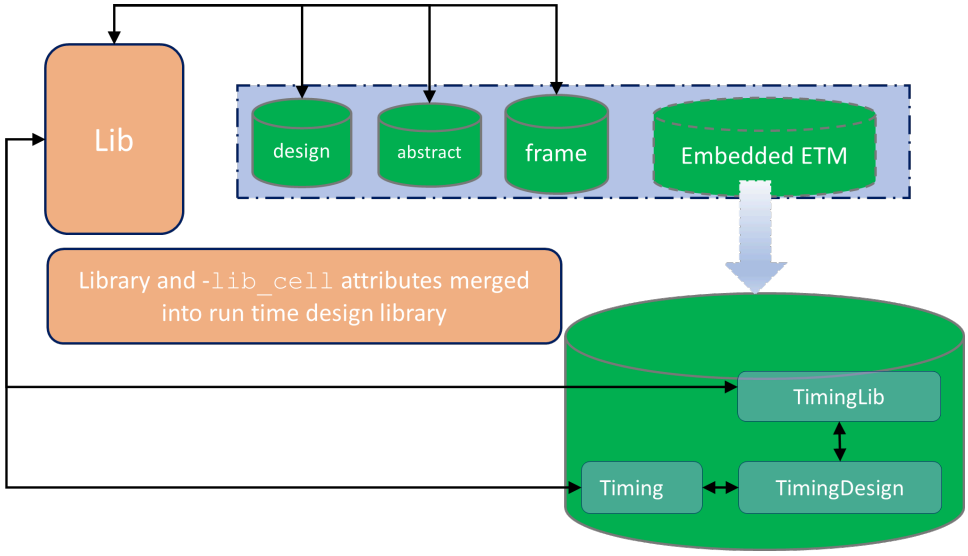
For information about how the implementation tool resolves the hierarchical references of a design (binding a block to its lower-level reference blocks) and how the tool specifies the order in which different views are selected for binding, see Resolving Block References.

See Also

- Block Labels
- Block Types
- Library Cells
- Blocks

Timing Views in Design Libraries

After a block meets its timing requirements, the detailed internal timing of the full netlist is stored in the design library. The detailed internal timing of the full netlist can often be replaced by a simplified Extracted Timing Model (ETM), which is much smaller than the full netlist view and requires significantly less runtime and memory to analyze. You can use the `create_timing_view` command to create a new ETM library or update an existing one. An ETM created using this command is imported as a timing view into the block design library, as shown in the following diagram.



Feedback

It is easy to use and update an ETM created using the `create_timing_view` command because the ETM library is embedded in the design library data, and the timing and frame views are decoupled.

When you run the `extract_model` command on the design library, it creates an ETM library, which contains both the timing view and frame view, and is embedded as a standalone reference library by the Library Manager tool. Therefore, it takes a longer time to update an ETM created using the `extract_model` command.

- To create a new timing view, use the `create_timing_view` command as shown in the following example:

```
fc_shell> create_timing_view -block block1/label1 -db_files "\
   block1_mode1_bc_1_10V_m40c_etm.db \
   block1_modeX_bc_1_10V_m40c_etm.db \
   block1_mode1_wc_0_90V_125C_etm.db \
   block1_modeX_wc_0_90V_125C_etm.db" \
   -mode_labels "mode1 modeX mode1 modeX" \
   -process_labels "bc bc wc wc"
```

  This example creates a timing view for the block “block1/label1” using the four .db files, 2 modes, 2 corners.

- To use an existing ETM, use the `create_timing_view` command, as shown in the following example

```
fc_shell> create_timing_view -block block1/label1 \
   -etm block1_etm.ndm
```

- Use the `create_frame` command to update a frame view without changing the timing view.
- The original name used to create the ETM library is overridden with the design library’s name.
- The library and `lib_cell` attributes on the ETM library are read and merged under the scope of the design library.
- Use the `change_view` command to switch between the full netlist view design view and the timing view.
- Use the `open_lib` or `open_block` command to load the ETM library in the design library.
- Use the `close_block` or `remove_block` command to close the ETM library associated with this timing view. The `remove_block` command also removes the ETM library attachment from the disk.

Feedback

- You cannot run the following commands on the ETM timing view:
  - The `save_block` command, because the timing view is read-only and can only be updated with the `create_timing_view` command.
  - The `save_block -as` or `copy_block` command to copy it to another block, because each design library with an ETM timing view can only have a single block.
  - The `rename_block` command, because each design library can have only one ETM timing view.

Creating a Block

To create a new block for holding design data, use the `create_block` command:

```
fc_shell> create_block MUX2
Information: Creating block 'MUX2.design' in library 'my_lib'. (DES-013)
{my_lib:MUX2.design}
```

This command creates the block in memory and sets it as the current block. To save the new block, use the `save_block` command.

To create a new block that overwrites an existing block of the same name:

```
fc_shell> create_block -force MUX2
```

The `read_verilog` command implicitly creates a new block to contain the Verilog netlist read by the command, so there is no need for the `create_block` command in that situation.

See Also

- Block Naming Conventions
- Opening a Block
- Saving a Block
- Copying a Block
- Closing a Block
- Reading a Verilog Netlist
- Blocks

Feedback

Opening a Block

To open an existing saved block for viewing or editing, use the `open_block` command:

```
fc_shell> open_block my_lib:MUX2
Opening block 'my_lib:MUX2.design'
{my_lib:MUX2.design}
```

The tool opens the specified block and sets it as the current block. If you specify the library name with the block name (for example, `my_lib:MUX2`), the tool also opens that library, if it is not already open, and sets it as the current library.

To open a block in read-only mode:

```
fc_shell> open_block -read my_lib:MUX2
```

See Also

- Block Naming Conventions
- Creating a Block
- Saving a Block
- Copying a Block
- Closing a Block
- Blocks

Setting the Current Block

The *current block* is the default block affected by block-related commands. By default, the block most recently opened is the current block. To explicitly set a block to be the current block, use the `current_block` command:

```
fc_shell> current_block MUX2
{my_lib:MUX2.design}
```

The tool sets the specified block as the current block. The block must be already open. If you specify the library name together with the block name (for example, `my_lib:MUX2`), the tool also sets that library as the current library.

To determine which block is the current block, use the `current_block` command by itself:

```
fc_shell> current_block
{my_lib:MUX2.design}
```

Feedback

See Also

- Block Naming Conventions
- Creating a Block
- Opening a Block
- Querying Blocks
- Saving a Block
- Copying a Block
- Closing a Block
- Blocks

Querying Blocks

You can get the following types of information about the open blocks:

- The current block – use the `current_block` command.

```
fc_shell> current_block
{my_lib:MUX2.design}
```

- The blocks in the library – use the `list_blocks` command.

```
fc_shell> list_blocks
Lib lib_A /path/lib_dir/lib_A current
  +> 0 MUX2.design Apr-30-21:01
  +> 0 MUX3.design Apr-30-21:29 current
Lib lib_B /path/lib_dir/lib_B
  +  0 ADDR1.design Apr-29-20:54
3

fc_shell> list_blocks lib_B
Lib lib_B /path/lib_dir/lib_B
  +  0 ADDR1.design Apr-29-20:54
1
```

By default, the `list_blocks` command lists all blocks in all open libraries, but not blocks in associated reference libraries. Use the `list_blocks` command options to override the default behavior:

```
# List blocks in library lib_A only
fc_shell> list_blocks libA
...

# List blocks in ref libs also
fc_shell> list_blocks -ref_libs
```

Feedback

```
...

# List lib_cell blocks also
fc_shell> list_blocks -lib_cells
...
```

- Create a block collection – use the `get_blocks` command.

```
# Get open blocks in current lib
fc_shell> set my_blks1 [get_blocks]
{lib_A:MUX2.design lib_A:MUX3.design}

# Get blocks in open libs
fc_shell> set my_blks2 [get_blocks -all]
{lib_A:MUX2.design lib_A:MUX3.design lib_B:ADDR1.design}

# Search string
fc_shell> set my_blks2 [get_blocks *ADDR* -all]
{lib_B:ADDR1.design}
```

See Also

- Block Naming Conventions
- Querying a Design Library
- Opening a Block
- Setting the Current Block
- Saving a Block
- Copying a Block
- Closing a Block
- Blocks

Resolving Block References

The implementation tool resolves the hierarchical references of a design when you run the `link_block` command or when the tool performs linking implicitly. The process of associating a block to its lower-level reference blocks is called *binding*.

To specify the order in which different views are selected for binding, use the `set_view_switch_list` command. You can set the view type priority at the block level, at the library level, and globally.

To query the current order of priority, use the `get_view_switch_list` command. The default order is `{abstract design frame outline}`.

Feedback

To rebind an existing block, use the `link_block -rebind` command:

```
fc_shell> link_block -rebind
Using libraries: lib_A lib_B tech32lvt
Visiting block lib_A:block1.design
Design 'block1' was successfully linked.
```

To change the references of leaf cells, modules, or blocks of a mapped netlist, use the `set_reference` command.

The `set_reference` command changes the references of

- Cells within the same block and across different blocks
- Modules and the corresponding submodules

Note:

When you change the reference of a cell to a module or black-box type, you must run the `link_block -force` command after the `set_reference` command.

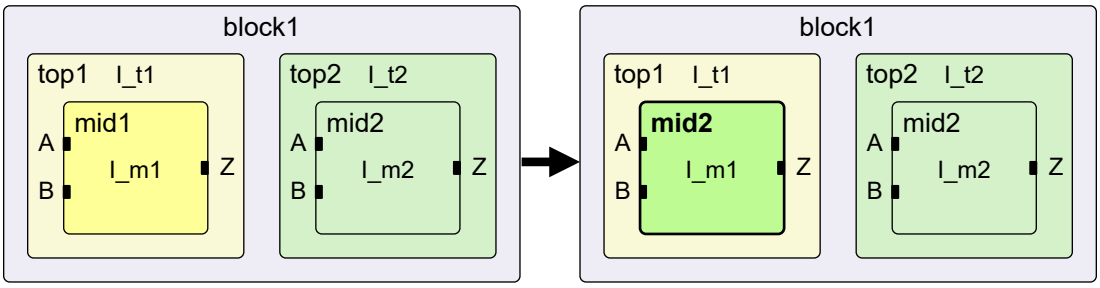
See Also

- Block Naming Conventions
- Querying a Design Library
- Opening a Block
- Setting the Current Block
- Saving a Block
- Copying a Block
- Closing a Block
- Block Views

Changing Module References

The following command changes the reference of module instance I_t1/I_m1 to module top2/mid2:

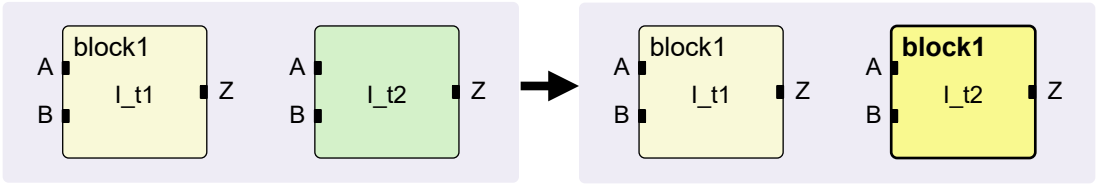
```
fc_shell> set_reference [get_cells I_t1/I_m1] \
   -to_module [get_modules top2/mid2]
```



Changing Block References

The following command changes the reference of block instance I_t2 to block1:

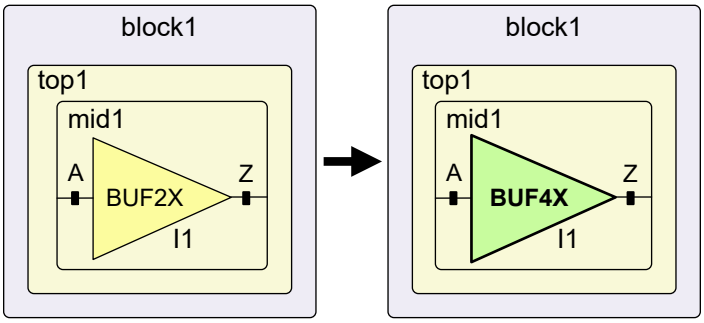
```
fc_shell> set_reference [get_cells I_t2] \
   -to_block [get_blocks block1]
```



Changing Cell References

The following command changes the reference of cell instance top1/mid1/I1 to cell library/BUF4X:

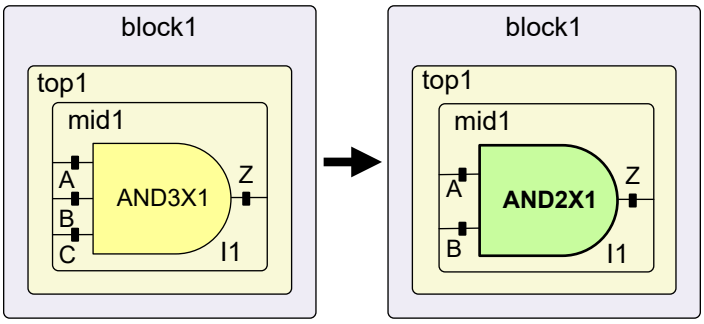
```
fc_shell> set_reference [get_cells top1/mid1/I1] \
   -to_block [get_lib_cells library/BUF4X]
```



Changing Cell References Ignoring Net Connections

The following command changes the reference of cell instance top1/mid1/I1 to cell library/AND2X1 without the required net connections; that is, ignoring the original three input pin connections.

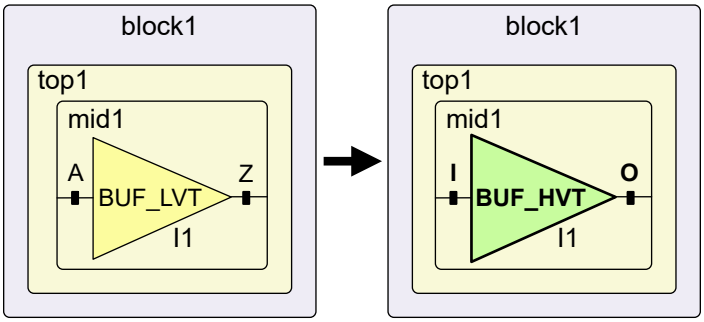
```
fc_shell> set_reference [get_cells top1/mid1/I1] \
   -to_block [get_lib_cells library/BUF4X] -pin_rebind force
```



Changing Cell References With Pin Mapping

The following command changes the reference of cell instance top1/mid1/I1 to cell library/BUF_HVT with pin the required pin mapping:

```
fc_shell> set_reference [get_cells top1/mid1/I1] \
   -to_block [get_lib_cells library/BUF_HVT] -pin_map {{A I} {Z O}}
```



Saving a Block

By default, a block you create, edit, or copy exists only in memory and is lost when you end the tool session. To save a block to disk, use the `save_block` command:

```
fc_shell> save_block MUX2
```

By default, the command saves only the specified block and ignores any lower-level blocks. To also save the lower-level blocks of the specified block, use the `-hierarchical` option:

```
fc_shell> save_block MUX2 -hierarchical
```

To save more than one block, use the `-blocks` option:

```
fc_shell> save_block -blocks [get_blocks]
```

Note:

You cannot use the `-as` option with the `-blocks` option.

Feedback

To save a block with a new name without affecting the original saved block or current block setting, use the `-as` option:

```
fc_shell> save_block MUX2 -as MUX2B
Information: Saving 'lib_A:MUX2.design' to 'lib_A:MUX2B.design' ...
```

To save a block with a new or different label without affecting the original saved block or current block settings:

```
fc_shell> save_block MUX2 -label ver1
Information: Saving 'lib_A:MUX2.design' to 'lib_A:MUX2/
 ver1.design' ...
```

You can save a block in compressed format to reduce the file size by using one of the following methods:

- To compress only the current view of the current block, specify the `-compress` option with the `save_block` command.
- To compress all views of the current block and subblocks, specify the `-hierarchical` and `-compress` options with the `save_block` command.

See Also

- Creating a Block
- Opening a Block
- Copying a Block
- Closing a Block
- Blocks

Copying a Block

To copy a block to a new block in the same library and view or a different library and view, use the `copy_block` command. If you do not explicitly specify a view for the source block and the destination block, all available views of the source block are copied to the destination block, and the design view of the destination block is returned. The block's source library does not need to be open. The tool opens the library if it is not already open.

By default, copied blocks are stored in memory only. To list the blocks stored in memory, run the `list_blocks` command. To save the copied block to disk, set the `design.on_disk_operation` application option to `true` before running the `copy_block` command. The default is `false`.

Feedback

The following example copies Top block to Top2 block for subsequent block exploration and saves the Top2 block to disk. The destination library and view are taken from the original block:

```
fc_shell> set_app_options design.on_disk_operation true
fc_shell> copy_block -from_block Top -to_block Top2
{lib:Top2.design}
```

See Also

- Creating a Block
- Opening a Block
- Saving a Block
- Closing a Block
- Blocks

Closing a Block

When you no longer need access to a block, you can close it with the `close_blocks` command:

```
fc_shell> close_blocks MUX2
Closing block 'lib_A:MUX2.design'
1
```

To automatically save a block before closing it:

```
fc_shell> close_blocks -save MUX2
...
```

Alternatively, you can first save the block by using the `save_block` command.

To close a block and deliberately discard recent changes to the block:

```
fc_shell> close_blocks -force MUX2
```

If the block *open count* is 1 or more after you execute the `close_lib` command, the block remains open.

To remove unwanted blocks from disk, use the `remove_blocks` command. Note that removed blocks cannot be recovered.

Do not attempt to add, move, or delete block files directly using operating system commands, as doing so can corrupt the database.

Feedback

See Also

- Block Naming Conventions
- Creating a Block
- Opening a Block
- Setting the Current Block
- Saving a Block
- Copying a Block
- Block Open Count
- Blocks

Block Open Count

The *open count* of a block is an integer specifying the number of times the block has been opened. The tool monitors the open count to keep the block open as long as you need access to it, based on the number of `open_block` and `close_blocks` commands used on the block. The tool assumes that you want the block to be closed when the number of `close_blocks` commands equals the number of `open_block` command

When you use the `create_block` or `open_block` command for the first time, the block's open count is set to 1. Each time the same block is reopened, its open count is incremented by 1:

```
fc_shell> open_block MUX3
Opening block 'lib2:MUX3.design'
...
fc_shell> open_block MUX3
Information: Incrementing open_count of block 'lib2:MUX3.design' to 2...
{lib2:MUX3.design}
...
fc_shell> open_block MUX3
Information: Incrementing open_count of block 'lib2:MUX3.design' to 3...
{lib2:MUX3.design}
...
```

Opening a hierarchical block also opens its associated lower-level blocks, which increments the open count for each of the lower-level blocks as well.

The `close_blocks` command decrements the block's open count by 1. If the new count is 1 or more, the block remains open; or if the new count is 0, the block is closed and removed from memory.

```
fc_shell> close_blocks MUX3
Information: Decrementing open_count of block 'lib2:MUX3.design' to 2...
```

Feedback

```
...
fc_shell> close_blocks MUX3
Information: Decrementing open_count of block 'lib2:MUX3.design' to 1...
...
fc_shell> close_blocks MUX3
Closing block 'lib2:MUX3.design'
1
```

To close a block irrespective of the open count:

```
fc_shell> close_blocks MUX3 -purge
Closing block 'lib2:MUX3.design'
```

To determine the open count of a block without affecting the count, use the `get_attribute` command:

```
fc_shell> get_attribute [get_blocks MUX3] open_count
3
```

Using the `save_block` command does not affect the open count.

To determine whether a block is already open:

```
fc_shell> open_block -check MUX3
Opening block 'my_lib:MUX3.design'
{my_lib:MUX3.design}
```

If the block is already open, it stays open without any change to the block's open count. If the block is not already open, the command returns an error message.

See Also

- Block Naming Conventions
- Creating a Block
- Opening a Block
- Saving a Block
- Copying a Block
- Closing a Block
- Blocks

Block Types

A block contains both physical layout data and functional design information to support synthesis, analysis, and optimization of the physical design.

Feedback

With respect to the design hierarchy, there are two basic types of blocks: *designs*, which are the top-level and intermediate-level blocks, and *library cells*, which are the leaf-level blocks. Designs are stored in *design libraries*, whereas library cells are stored in *cell libraries*.

A block can have multiple *views*, with each view containing an alternative representation of a design. Each type of view serves a specific purpose.

For more information, see

- Designs
- Library Cells
- Block Labels
- Block Views

See Also

- Design Libraries
- Reference Library List
- Creating a Block
- Opening a Block
- Saving a Block
- Copying a Block
- Closing a Block
- Blocks

Designs

A *design* is a representation of a circuit that consists of block instances and connections. A typical way to build a design is to read a Verilog netlist into a new block and have the Fusion Compiler tool perform placement and routing based on the netlist contents.

The lower-level blocks used to build a design can be library cells, macros (hierarchical cells that represent lower-level designs), or a combination of both types of cells. A design can itself be used as a macro instance inside another design at a higher level of hierarchy.

Feedback

See Also

- Design Libraries
- Reference Library List
- Specifying a Design Library's Reference Libraries
- Block Labels
- Blocks

Library Cells

A *library cell* (lib_cell object) is a representation of a leaf-level function such as logic gate or I/O pad. A library cell model includes physical information such as layer shapes, pins, and routing blockages; and functional information such as logic, timing, and power parameters. Library cells are stored in *cell libraries*.

The following commands operate on library cells:

- `get_lib_cells` – Creates a collection of library cells
- `report_lib_cells` – Generates a report on specified library cells
- `set_lib_cell_purpose` – Specifies how a collection of library cells can be used, either including or excluding purposes such as clock tree synthesis, optimization, or hold fixing

To list or query library cells and their attributes:

```
fc_shell> list_blocks -lib_cells
...
fc_shell> get_blocks -of_objects [get_libs stdhvt] -lib_cells
...
fc_shell> list_attributes -application -class lib_cell
...
fc_shell> get_attribute -objects [get_lib_cells stdhvt/DFF1] \
   -name area
7.8785
fc_shell> report_attributes -application -class lib_cell \
  [get_lib_cells stdhvt/DFF1A]
...
```

A library cell model typically consists of different block *views* used for different purposes:

- *design* view for storing the physical and functional details
- *frame* view for placement and routing

- *timing* view for storing the timing, power, and noise characteristics of the cell
- *layout* view for library cell preparation and mask generation, containing only the geometric data obtained from a GDSII file

For information about preparing library cells and cell libraries, see the *Library Manager User Guide*.

See Also

- Design Libraries
- Reference Library List
- Cell Libraries
- Block Views
- Blocks

Sparse Libraries

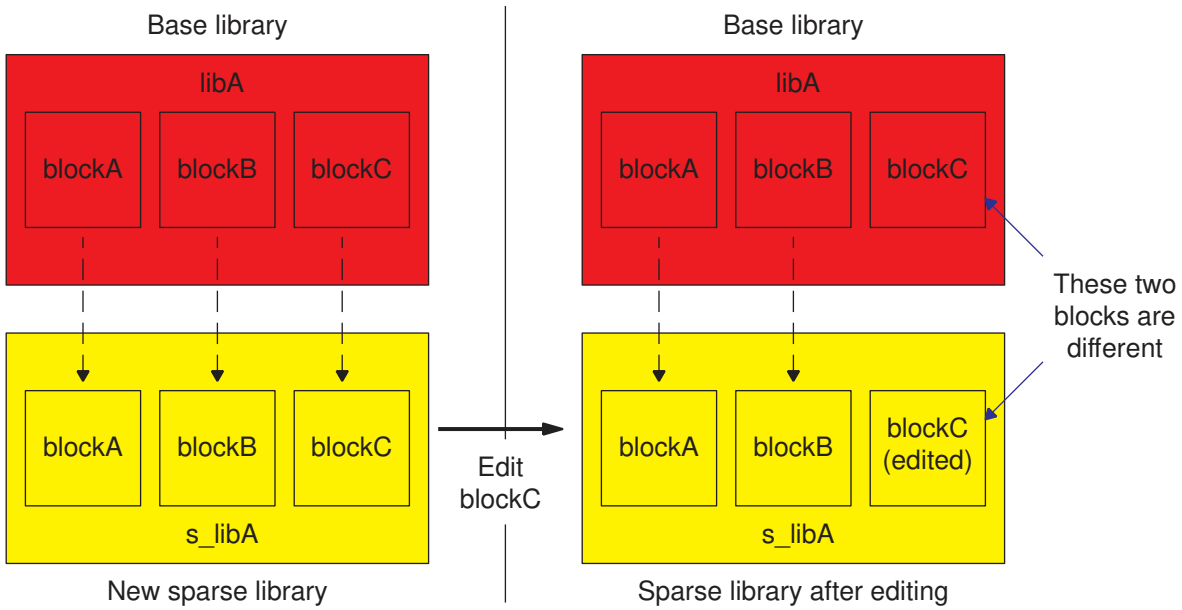
When multiple users or projects share a common set of design data, you can use *sparse libraries* to efficiently share the common data. A single central *base library* contains the stable data shared between different users or projects, while multiple sparse libraries build upon that data by modifying the blocks in the base library or by adding new blocks.

You create a sparse library by using the `create_lib` command with the `-base_lib` option to specify the related base library. The sparse library initially uses the technology data, reference library list, and blocks of the base library. After you create a sparse library, you can modify its contents without affecting the base library.

Feedback

The following figure shows how the tool maintains the contents of a sparse library.

Figure 13 Base Library and Sparse Library Relationships



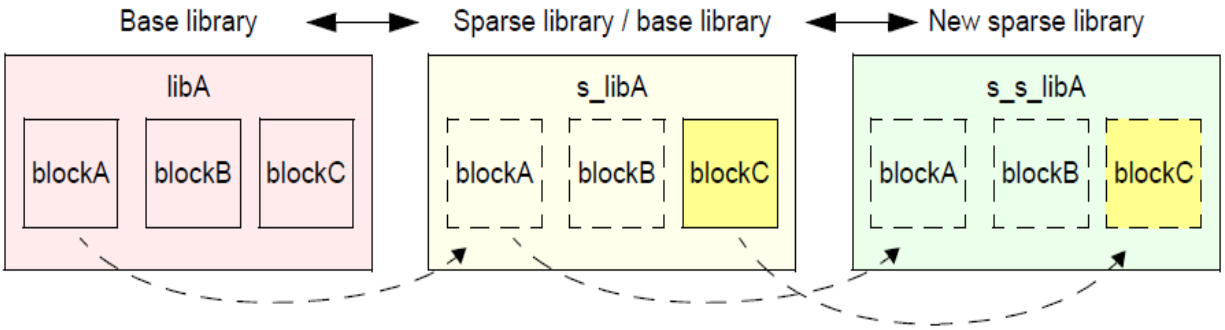
When you create a new sparse library, it points to the contents in the related base library, so its contents are exactly the same as the base library. When you edit a block in the sparse library, such as blockC in the figure, that block becomes different from the one in the base library, so that block is saved locally in the sparse library directory.

Multiple users can create multiple sparse libraries associated with the same base library. These users concurrently share the same data in the base library, without any chance of corrupting the shared database. All changes they make to their sparse libraries affect only their own libraries, not the base library.

The base library associated with a sparse library can itself be a sparse library, as shown the following figure. Library s_libA is a sparse library to libA but also a base library to the new sparse library s_s_libA. To find the blocks of the sparse library s_s_libA, the tool traverses the chain of sparse-base libraries until it finds a locally saved block.

Feedback

Figure 14 Sparse Library Used as a Base for Another Sparse Library



For more information, see

- Creating a Sparse Library
- Remote and Local Blocks
- Reverting Blocks in a Sparse Library
- Synchronizing a Sparse Library to a Base Library
- Synchronizing a Base Library to a Sparse Library

Creating a Sparse Library

To create a new sparse library, use the `create_lib` command with the `-base_lib` option to specify the related base library:

```
fc_shell> create_lib -base_lib libA s_libA
...
Information: Creating Sparse View library 's_libA' with base library
 'libA'. (NDM-103)
...
{s_libA}
```

The base library can be either open or closed. If closed, it must be accessible in the paths specified by the `search_path` variable. If you specify a simple name, the command uses the `search_path` variable to search for the base library and uses the first one that it finds. The base library can itself be a sparse library.

The new sparse library initially contains the same blocks, reference library list, and technology data as its base library. When you add, edit, or remove blocks, the tool stores the changes in the local library directory.

Feedback

To save a sparse library, use the `save_lib` command, just like any library:

```
fc_shell> save_lib
Saving library 's_libA'
1
```

To determine whether a library is sparse, query its `is_sparse` attribute:

```
fc_shell> get_attribute -name is_sparse [get_libs libA]
false
fc_shell> get_attribute -name is_sparse [get_libs s_libA]
true
```

To determine a sparse library's base library, query its `base_lib` attribute:

```
fc_shell> get_attribute -name base_lib [get_libs s_libA]
libA
```

See Also

- Sparse Libraries
- Remote and Local Blocks
- Synchronizing a Sparse Library to a Base Library
- Synchronizing a Base Library to a Sparse Library

Remote and Local Blocks

You can use commands such as `create_block`, `open_block`, `save_block`, `copy_block`, and `reopen_block` to edit the contents of a sparse library, just like any library.

A block saved and maintained in the base library called is a *remote block*, whereas a block saved and maintained in the sparse library directory is a called a *local block*. Opening or copying a remote block in a sparse library is an in-memory operation. When you save the block, the block becomes local and is saved in the local sparse library directory. The local block is maintained separately from the remote block.

If you save a hierarchical remote block, that block and its lower-level blocks are all made local and saved in the local sparse library directory.

To determine whether an open block is remote or local, query the block's `is_remote` attribute:

```
fc_shell> open_block BLK1 # Open remote block stored in base lib
...
fc_shell> create_block BLK5 # Create local block in sparse lib
...
fc_shell> get_attribute -name is_remote [get_blocks BLK1]
true
```

Feedback

```
fc_shell> get_attribute -name is_remote [get_blocks BLK5]
false
fc_shell> save_block BLK1  # Saving BLK1 makes it local
...
fc_shell> get_attribute -name is_remote [get_blocks BLK1]
false
```

To create a collection of all local blocks in the current library:

```
fc_shell> get_blocks -filter "is_remote == false"
{BLK1 BLK5}
```

To create a collection of all open sparse libraries:

```
fc_shell> get_libs -filter "is_sparse == true"
{s_libA x_libA y_libA}
```

Commands that operate on blocks in a sparse library only affect that library; they have no effect on blocks in the base library. For example, if you remove a remote block from a sparse library using the `remove_blocks` command, that block ceases to exist in the sparse library and can no longer be opened there, even though the same block still exists in the base library.

See Also

- Sparse Libraries
- Creating a Sparse Library
- Reverting Blocks in a Sparse Library
- Synchronizing a Sparse Library to a Base Library
- Synchronizing a Base Library to a Sparse Library

Reverting Blocks in a Sparse Library

To *revert* a block means to discard the local block saved in a sparse directory and go back to using the identically named block in the base library. To revert one or more blocks, use the `revert_blocks` command. This command works only when the blocks are closed, as shown in the following example.

```
fc_shell> create_lib -base_lib libA s_libA
...
Information: Creating Sparse View library 's_libA' with base library
 'libA'. (NDM-103)
...
fc_shell> open_block BLK1
Information: Updating library catalog of Sparse View library 's_libA'
...
```

Feedback

```
Information: Adding block 'BLK1.design' to Sparse View library
 's_libA'. (NDM-105)
...
fc_shell> save_block BLK1
...
fc_shell> get_attribute -name is_remote [get_blocks BLK1]
false
fc_shell> close_blocks BLK1 # Must close block before reverting
...
fc_shell> revert_blocks BLK1
...
Information: Reverted design 's_libA:BLK1.design' and removed it ...
1
fc_shell> open_block BLK1  #Reopen remote block stored in base lib
...
fc_shell> get_attribute -name is_remote [get_blocks BLK1]
true
```

Note that *reverting* is different from *removing* a block. Removing a block discards the local block and also makes the original remote block inaccessible.

If you accidentally remove a block when you meant only to revert it, you can recover the block from the base library by rebasing the sparse library with the `set_base_lib` command, without affecting other local blocks in the sparse library. For details, see Synchronizing a Sparse Library to a Base Library.

See Also

- Sparse Libraries
- Creating a Sparse Library
- Remote and Local Blocks
- Synchronizing a Sparse Library to a Base Library
- Synchronizing a Base Library to a Sparse Library

Synchronizing a Sparse Library to a Base Library

In some situations, you might need to *rebase* a sparse library, which means to update the association between the sparse library and its base library without affecting the local blocks in the sparse library. For example,

- The base library has been renamed or moved.
- The technology data, reference library list, or blocks in the base library have been changed.

Feedback

- You removed (instead of reverted) a block from the sparse library, and you need to recover the original block from the base library.
- You want to associate a sparse library with a different base library.

To rebase a sparse library, first close all the open blocks in the library, then use the `set_base_lib` command to specify the new or modified base library:

```
fc_shell> current_lib s_libA              # Existing sparse library
{s_libA}
fc_shell> close_blocks *                  # Close all open blocks
...
fc_shell> set_base_lib -library s_libA -base_lib libB
...
Information: Rebasing Sparse View library 's_libA' to base library
 'libB'. (NDM-103)
...
Information: Updating library catalog of Sparse View library 's_libA'
 from base library 'libB'. (NDM-104)
...
 -- marking removal of design: s_libA:BLK1.design missing in base-lib:
 'libB'
 -- marking addition of design: libB:BLKZ.design
Information: Removing block 's_libA:new_block.design' from Sparse View
 library 's_libA'. (NDM-106)
Information: Adding block 'BLKZ.design' to Sparse View library
 's_libA'. (NDM-105)
```

Rebasing a sparse library updates the contents of the library based on the contents of the new or updated base library. Remote blocks no longer existing in the base library are removed from the sparse library, and any new remote blocks are added to the sparse library. Local blocks in the sparse library are not affected, irrespective of the new base library contents.

You can specify the path to the base library as a simple name, a relative path, or an absolute path. If you specify a simple name, the command uses the `search_path` variable to search for the base library and uses the first one that it finds.

To query the path to the base library from a sparse library, look at the sparse library's `base_lib_path` attribute:

```
fc_shell> get_attribute -name base_lib_path [get_libs s_libA]
/remote/path/user/data/tmp/libA
```

Setting the `base_lib_path` attribute using the `set_attribute` command has the same effect as using the `set_base_lib` command.

See Also

- Synchronizing a Base Library to a Sparse Library

Feedback

Synchronizing a Base Library to a Sparse Library

A base library typically contains stable data that can be shared among multiple sparse libraries, and the related sparse libraries contain only blocks that differ from the common base. However, in some situations, you might want to move data from a sparse library to the base library so that the other sparse libraries can have access to that data.

The Fusion Compiler tool does not provide commands specifically for updating a base library with data from a sparse library. However, you can manually update the base library with data taken from a sparse library.

For example, suppose you have a base library named libA containing a block BLK1, and a sparse library named s_libA containing a newer version of BLK1. To update the base library with the newer version of BLK1, use the following script to copy the newer version back to the base library.

```
open_lib s_libA              # Open sparse library
open_lib -edit libA          # Open base library for editing
copy_block -from_block s_libA:BLK1 -to_block libA:BLK1 # Copy block
save_lib libA                # Save modified base library
```

See Also

- Synchronizing a Sparse Library to a Base Library

File Attachments

You can attach any type of file to a block or library. The attached file becomes part of the block or library. You can use this feature to store any important information related to the block or library such as design specifications, Tcl scripts, or signoff documents.

The following commands support the file attachment feature:

- `add_attachment` – Attaches a file to a specified block or library; copies the file into the block or library.
- `report_attachments` – Reports the attachments on a block or library.
- `open_attachment` – Opens a file handle for the attachment, like the Tcl `open` command, allowing you to read or restore the file.
- `remove_attachments` – Removes one or more attached files from a block or library.

For details, see the man pages for the `add_attachment`, `report_attachments`, `open_attachment`, and `remove_attachments` commands.

Feedback

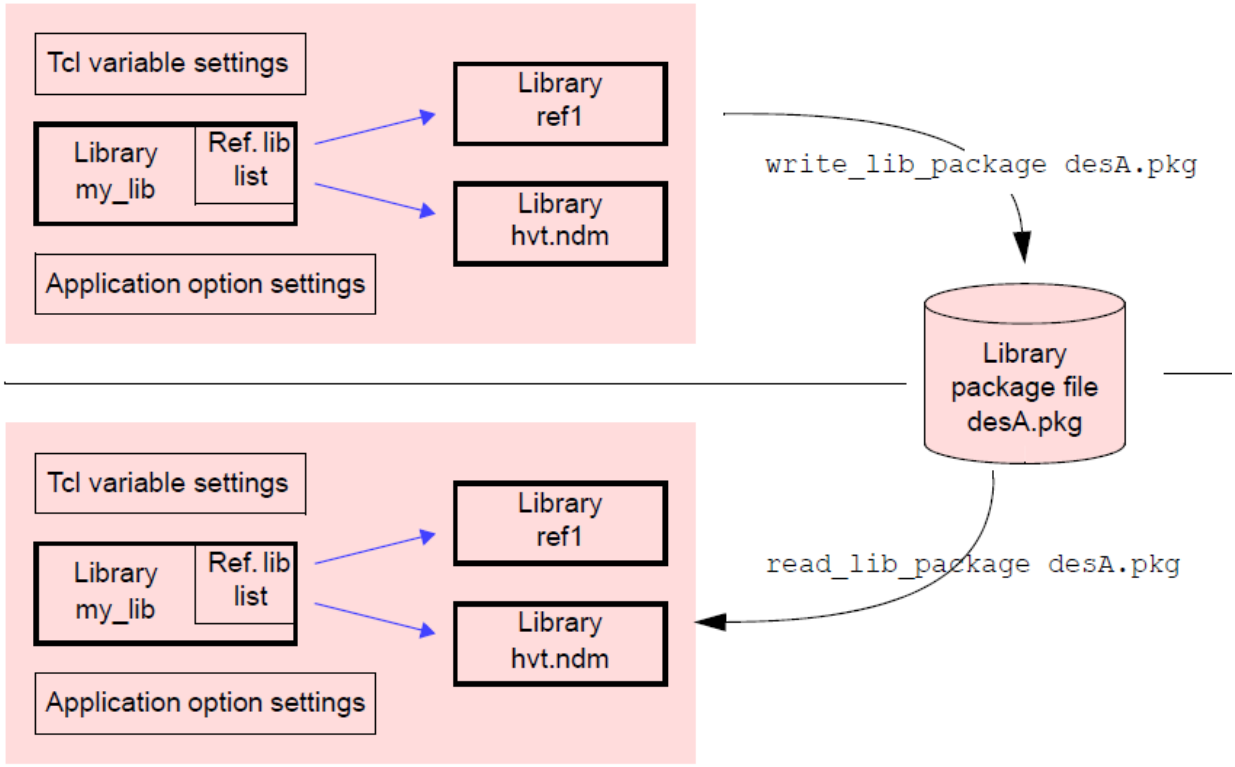
Library Packaging

To transfer or archive a chip design, you can put all of the library data and environment settings into a single file called a *library package*:

- The `write_lib_package` command puts the library data and environment settings into a compressed library package file.
- The `read_lib_package` command restores the original data and environment settings from the library package file.

The following figure shows how to create and restore a library package.

Figure 15 Creating a Library Package and Restoring the Library Data



For details, see

- Creating a Library Package
- Restoring Data From a Library Package

Feedback

Creating a Library Package

To create a library package, use the `write_lib_package` command:

```
# Checks current library
fc_shell> current_lib
{my_lib}

# Creates package for current library
fc_shell> write_lib_package desA.pkg
1
```

By default, the `write_lib_package` command creates a library package file that contains the current design library, its reference libraries, and environment settings such as Tcl variables, application options, and the current block.

The command automatically compresses the library package, reducing disk usage. You can disable compression by setting the `lib.setting.compress_lib_package` application option to `false` before running the `write_lib_package` command.

The `write_lib_package` command has options to

- Create a package for a library other than the current library (`-library`)
- Specify which blocks (subdesigns and library cells) to include in the package

  By default, the `write_lib_package` command includes all linked blocks in the package.

  - To include all blocks, both linked and unlinked, use the `-include_all_blocks` option.
  - To include all linked blocks plus unlinked blocks with specific views and labels, use the `-include_views` and `-include_labels` options.
  - To include only a specified subset of the blocks, use the `-blocks` option.

  To exclude specific views and labels of the included unlinked blocks, use the `-exclude_views` and `-exclude_labels` options.
- Exclude specified reference libraries from the package (`-exclude_ref_libs`)
- Exclude all library cells that are not used in the top-level design from reference libraries (`-exclude_unused_libcells`).

  Before using this option, ensure that the excluded library cells are not used later in the flow. Fusion reference libraries do not support the use of this option.
- Include database files used in reference libraries (`-include_db_files`).

Feedback

Using this option ensures that all database files used in the reference libraries are included in the package. When unpacking, the database files are placed in a subdirectory named “dbfiles” in the unpacked library folder.

- Report the command progress and application options being saved (`-verbose`)

The command writes the library package file to the current working directory. The file name can have any extension, such as .pkg, or no extension at all.

See Also

- Library Packaging
- Restoring Data From a Library Package
- Design Libraries
- Blocks

Restoring Data From a Library Package

To restore the data stored in a library package, use the `read_lib_package` command:

```
fc_shell> read_lib_package /path/pkgs/desA.pkg
Information: Loading library file '/path/working2/my_lib' (FILE-007)
Information: Loading library file '/path/working2/my_lib/reflibs/ref1'
Information: Loading library file '/path/working2/my_lib/reflibs/hvt.ndm'
Opening block 'my_lib:top.design'
1
fc_shell> current_lib
{my_lib}
fc_shell> report_ref_libs

...
   Name  Path                          Location
   ----------------------------------------
*  ref1  my_lib/reflibs/ref1          /path/working2/my_lib/reflibs/ref1
*+ hvt   my_lib/reflibs/hvt.ndm       /path/working2/my_lib/reflibs/hvt.ndm
    "*" = Library currently open
    "+" = Library has technology information
1
```

By default, the `read_lib_package` command restores the library to the current working directory, unpacks and restores the reference libraries, and invokes the saved environment settings such as Tcl variables, application options, open blocks, and current block.

Feedback

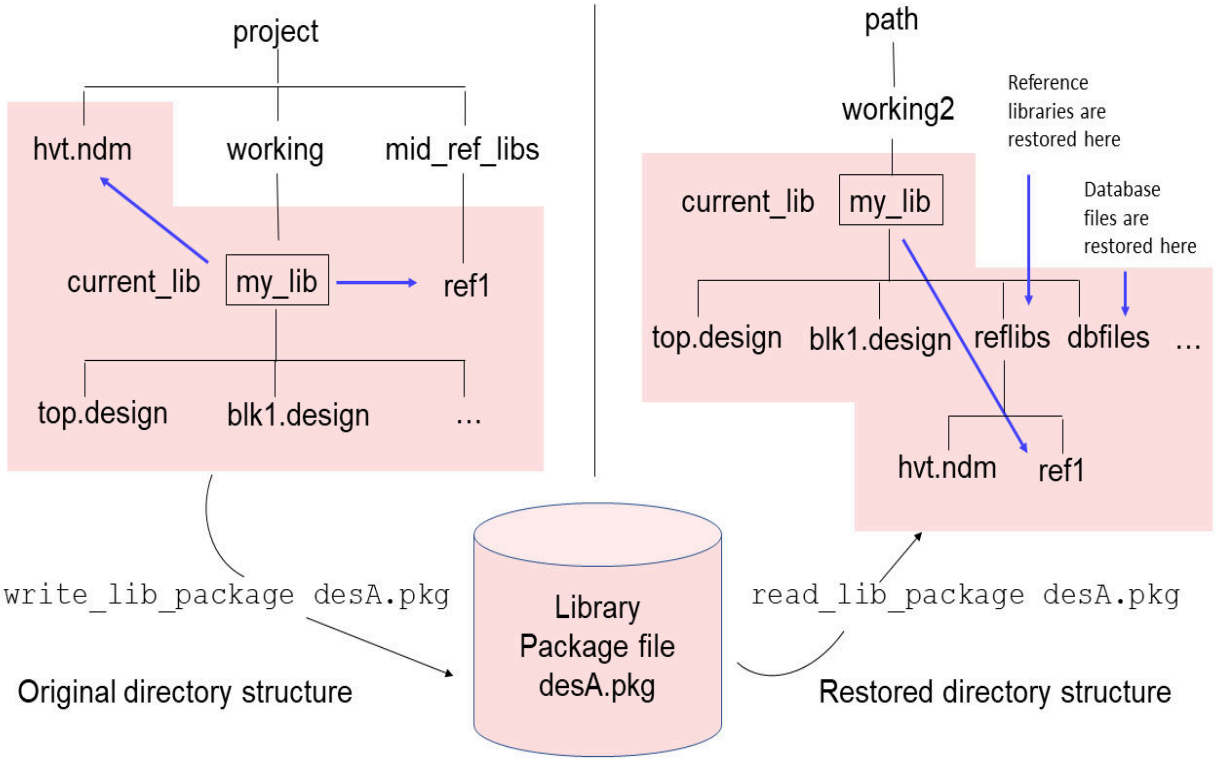
The `read_lib_package` command has options to:

- Specify a destination directory for writing the library (`-destination`)
- Overwrite an existing library in the destination directory (`-overwrite`)
- Skip the unpacking step if the library is already present on disk (`-replay`)

The `-replay` option and the `-overwrite` option are mutually exclusive.

The unpacking of the libraries in the reference library list does not preserve the original absolute or relative locations of those libraries. Instead, the tool restores all of the reference libraries to a single subdirectory named “reflibs” under the directory where the design library is restored, as shown in the following figure. The database files are unpacked and restored to a subdirectory named “dbfiles” under the directory where the design library is stored. This folder is then appended to the search path.

Figure 16 Original and Restored Reference Library Locations



In the restored design library, the library list correctly points to the new locations of the restored reference libraries. You can begin working on the restored design immediately, without concern for the changed reference library locations.

Feedback

See Also

- Library Packaging
- Creating a Library Package
- Design Libraries
- Blocks

2

Data Import and Export

When you work with a design in the Fusion Compiler tool, you open and edit the design view of a block stored in a design library.

You can read and save the design data as described in the following topics:

Feedback

Reading a Verilog Netlist

The starting point for physical implementation is a design netlist in the Verilog hardware description language. The Fusion Compiler tool reads one or more structural gate-level Verilog netlists and uses this information as the basis for placement and routing.

To read a Verilog netlist into the implementation tool,

1. Read the Verilog netlist files for the design by using the `read_verilog` command:

   ```
   fc_shell> read_verilog ../my_data/my_des.v
   ```

   For netlists that are encrypted according to the IEEE 1735 standard, set the `file.verilog.ieee_1735_decryption` application option to `true` before reading the files.

   The command implicitly creates a new block named *top_module_name*.design and reads the Verilog netlist data into the block.

   The command identifies the top-level module as the only module in the Verilog file not used as an instance by any other modules. If the file has multiple modules not used as instances, you need to identify the one that is the top-level module:

   ```
   fc_shell> read_verilog -top my_top ../my_data/my_des.v
   ```

   To explicitly specify a new block name other than *top_module_name*.design:

   ```
   fc_shell> read_verilog -top my_top \
      -design desA ../my_data/my_des.v
   ```

2. (Optional) Check the netlist for connectivity errors and design violations by using the `check_netlist` command:

   ```
   fc_shell> check_netlist -summary
   ```

3. If you are not using a UPF file to specify the power infrastructure, and if you want to explicitly specify the power and ground supply port and net names (the default names are VDD and VSS), specify the names as application options. For example,

   ```
   set_app_options -name mv.pg.default_power_supply_port_name \
      -value VD1
   set_app_options -name mv.pg.default_power_supply_net_name \
      -value VD1
   set_app_options -name mv.pg.default_ground_supply_port_name \
      -value VS1
   set_app_options -name mv.pg.default_ground_supply_net_name \
      -value VS1
   ```

Feedback

4. Link the block by using the `link_block` command:

```
fc_shell> link_block
...
Design 'desA' was successfully linked
1
```

   The linking process resolves the cell references by searching the reference libraries in the order specified by the design library's reference library list.

To save the block to disk, use the `save_block` command.

To write out the Verilog netlist for the block, use the `write_verilog` command.

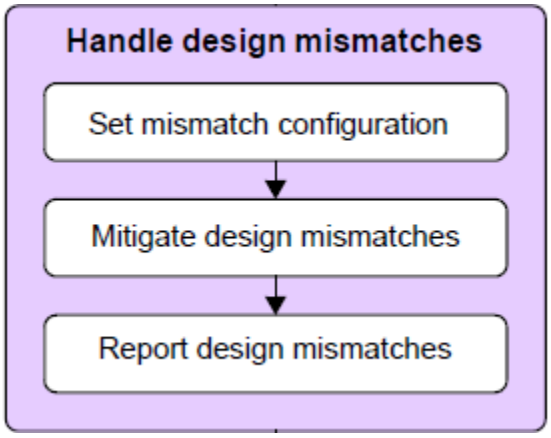
See Also

- Creating a Design Library
- Opening a Design Library
- Saving a Block
- Saving a Design in ASCII Format
- Changing Object Names for Verilog Output

Allowing Incomplete or Inconsistent Design Data

During early design development, frequent design updates can cause incomplete or inconsistent data, such as pin mismatches between a block-level design and the reference to it from a top-level design.

By default, the tool cannot link any block with inconsistent or mismatching data. To enable the tool to successfully link a block even if it has certain types of design mismatches, set a mismatch configuration for the block. When you set a mismatch configuration, the tool can ignore or fix different types of mismatching data, based on the mismatch configuration, and continue the linking process. You can report the details of the mismatches in the design.



Blocks linked with mismatching data can be used only for feasibility analysis.

This topic discusses netlist mismatch types. For information about library mismatch types, see “Allowing Incomplete or Inconsistent Library Data” in the *Library Manager User Guide*.

Mismatch Configurations and Types

The tool can identify mismatches between instances and reference objects (modules or library cells), as listed in Table 3. You can set the following predefined mismatch configurations to allow or repair mismatching data:

- `default` (default)

  This predefined configuration does not allow any mismatching data. When mismatches are encountered, the tool issues error and warning messages such as the following:

  ```
  Warning: Unable to resolve reference to 'sub_design2' first referenced
   from module 'test_ref_name_case'. (LNK-005)
  ```

- `auto_fix`

  This predefined configuration allows the linker to repair all the mismatch types shown in Table 3 using their default repair process. To report these default repair processes, use the `report_mismatch_configs -all` command. You can specify a different repair process to use for this configuration.

  When the `auto_fix` configuration is enabled, the tool issues information messages about any repairs it performs. For example,

  ```
  Information: Changed reference of instance 'U0' from 'sub_design2' to
   'sub_DESIGN2'. (LNK-044)
  ```

Feedback

- Custom configurations

  Instead of enabling the configurations above, you can define your own mismatch configuration, as described in Creating a Custom Mismatch Configuration.

For detailed information about the behavior of the predefined `default` and `auto_fix` mismatch configurations, use the `report_mismatch_configs -all` command.

Note:

The Fusion Compiler tool can report mismatches between logic libraries and physical libraries, but you need to use the library manager tool to fix the mismatches.

The tool considers the netlist mismatch types in the following table. For information about library mismatch types, see “Allowing Incomplete or Inconsistent Library Data” in the *Library Manager User Guide*.

Table 3 Netlist Mismatch Types

Mismatch type	Description
`bus_bit_naming`	A reference cell and its instance use different bus notation; a bus might be bit-blasted in the reference cell but defined as a bus in the instance, or vice versa. Supported repairs: • `bus_bit_blast_naming`: Reconstructs the buses using the style specified in the `link.bit_blast_naming_style` application option.
`bus_width_mismatch`	A reference module and its instance might have different bus widths for the same bus port. Supported repairs: • `record_bus_width_extra_ports`: Records the mismatch in bus width. If the bus width in the reference module is smaller than the bus width on the instance, the mismatch record is set to `not_repaired`. If the bus width in the reference module is larger than the bus width on the instance, the extra pins on the instance are considered unconnected, and the mismatch record is set to `repaired`.
`empty_logic_module`	A cell instance references an empty logic module. Supported repairs: • `create_macro`: Creates a black box macro for the empty module so that the module can be physically represented during floorplanning and optimization.

Feedback

Table 3 Netlist Mismatch Types (Continued)

Mismatch type	Description
missing_logical_reference	A cell instance's reference cell (library cell or module) is missing. Supported repairs: • create_blackbox: Creates a physical black box for the cell, which enables the missing cell to be placed in the floorplan and retained. • record_unbound_instance: Records the unresolved reference. The tool ignores the mismatch and does not perform any repairs.
missing_port	Ports are missing between an instance and its reference object (module or library cell). Supported repairs: • delete_extra_port: Deletes extra ports from the cell instance.
port_name_case	A port name mismatch exists between an instance and its reference object (module or library cell) due to case-sensitivity. Supported repairs: • record_port_name_case_insensitivity: Updates the design to match the reference design and records the mismatch.
port_name_synonym	A port name mismatch exists between an instance and its reference object (module or library cell) when a port name synonym is defined. Supported repairs: • record_port_name_synonym: Updates the instance to use the port name in the reference object and records the mismatch.
reference_name_case	A cell name mismatch exists between an instance and its reference object (module or library cell) due to case-sensitivity. Supported repairs: • record_cell_name_case_insensitivity: Updates the design to match the reference design and records the mismatch.

bus_bit_naming

The bus_bit_naming mismatch type indicates that a bus is bit-blasted in a reference cell but defined as a bus in the reference cell's instance, or vice versa.

Feedback

In the following example,

- In the `test` module's `sub_design` instance, ports A and C use bus notation (`.A(a)` and `.C(c)`), while port B uses bit-blast notation (`.\B<1> (b[1]), .\B<0> (b[0])`), as highlighted.
- In the `sub_design` reference, port B uses bus notation (`input [1:0] B;`), while port A uses bit-blast notation (`input \A(1) , \A(0) ;`), as highlighted.
- In the `test` module's `SRAM1RW4x16` instance, port A uses bit-blast notation (`.\A<1> (a[1]), ./A<0> (a[0])`), as highlighted. In the `SRAM1RW4x16` reference, port A uses bus notation (`bus (A)`).

Instances in the top-level module:

```
module test (a, b, c);
   input [1:0] a, b;
   output [1:0] c;
   sub_design U0 (.A(a) , .\B<1> (b[1]), .\B<0> (b[0]), .C(c) );
   SRAM1RW4x16 U1 (.CE(c), .\A<1> (a[1]), .\A<0> (a[0]) /*...*/ );
endmodule
```

Subdesign:

```
module sub_design (\A(1) , \A(0) , B, C);
   input \A(1) , \A(0) ;
   input [1:0] B;
   output [1:0] C;
   input assign C = { \A(1) , \A(0) } | B;
endmodule
```

Library cell:

```
cell (SRAM1RW4x16) {
   pin (CE) {...} bus (A) {...}
}
```

To mitigate the mismatch, specify the bus format according to the style of the reference cell by setting the `link.bit_blast_naming_style` application option. For example,

```
fc_shell> set_app_options \
   -name link.bit_blast_naming_style \
   -value { %s(%d) %s<%d> }
```

As a result, the tool connects the ports correctly, preserving the port of the reference design as shown in the following example:

```
module test (a, b, c);
   input [1:0] a, b;
   output [1:0] c;
   sub_design U0 (.\A(1)(a[1]), .\A(0)(a[0]), .B(b), .C(c));
```

```
   SRAM1RW32x50 U1 (.CE(CE), ..., .A(A));
endmodule
```

See Also

- Mismatch Configurations and Types

bus_width_mismatch

The `bus_width_mismatch` mismatch type indicates that there is a difference between the width of a bus port defined in a reference module and its instance.

If the bus width in the reference module is smaller than the bus width on the instance, the mismatch record is set to `not_repaired`.

In the following example, the top-level module references the `sub_design` module. The bus port has a width of 4 in the `sub_design` reference. However, it has a width of 7 in the reference module instance U1. The tool creates a mismatch record and sets it to `not_repaired`.

Instances in the top-level module:

```
module test (ain, cout);
   input [6:0] ain;
   output cout;
   sub_design U1 (.a(ain) , .c(cout));
endmodule
```

Subdesign:

```
module sub_design (a , b, c);
   input [3:0] a;
   input b;
   output c;
   assign c = |a ;
endmodule
```

If the bus width in the reference module is larger than the bus width on the instance, the extra pins on the instance are considered unconnected, and the mismatch record is set to `repaired`.

In the following example, the top-level module references the `sub_design` module. The bus port has a width of 4 in the `sub_design` reference. However, it has a width of 3 in the reference module instance U2. The tool creates a mismatch record and sets it to `repaired`.

Instances in the top-level module:

```
module test (ain, cout);
   input [2:0] ain;
   output cout;
```

Feedback

```
   sub_design U2 (.a(ain) , .c(cout));
endmodule
```

Subdesign:

```
module sub_design (a , b, c);
   input [3:0] a;
   input b;
   output c;
   assign c = |a ;
endmodule
```

To enable the `bus_width_mismatch` mismatch, set the `link.do_bus_width_mismatch` application option to `true`. For example,

```
fc_shell> set_app_options -name link.do_bus_width_mismatch \
   -value true
```

See Also

- Mismatch Configurations and Types

empty_logic_module

The `empty_logic_module` mismatch type indicates that a cell instance references an empty module.

In the following example, the top-level module references the empty sub_design module.

Instances in the top-level module:

```
module test (a, b, c);
   input [1:0] a, b;
   output [1:0] c;
   sub_design U0 (.A(a), .B(b), .C(c) );
endmodule
```

Subdesign:

```
module sub_design (A, B, C);
   input [1:0] A, B;
   output [1:0] C;

endmodule
```

When you use the `default` mismatch configuration, an empty logic module results in a soft error. The tool continues linking the design and records the mismatch in the database.

When you enable the tool to repair empty logic modules in your design, the tool creates black box macros with dummy timing and physical information for the empty modules, with port directions retained from the modules. The size of each macro is estimated based on pin count and the size of other cells in the cell library. The macro's `is_empty` attribute is

Feedback

set to `true`, and the macro is placed alongside other cells in the block. The tool records the mitigation in the database.

You can query the black box macros in your design with the `get_cells` command:

```
fc_shell> get_cells -hierarchical* \
   -filter "design_type==macro&&is_empty==true"
```

See Also

- Mismatch Configurations and Types

missing_logical_reference

The `missing_logical_reference` mismatch type indicates that a cell instance's reference cell, such as a library cell or module, is missing.

For example, the following netlist contains the missing_design unresolved reference, as highlighted:

```
module test (a, b, c);
   input a, b;
   output [1:0] c;
   //missing_design is missing
   missing_design U1 (.A(a), .B(b), .C(c));
endmodule
```

You can choose one of the following two ways to resolve the missing reference:

- Specify the `create_blackbox` strategy

  ```
  fc_shell> set_attribute \
     [get_mismatch_types missing_logical_reference] \
     current_repair(user_config) create_blackbox
  ```

  The command creates a physical black box (soft macro) for the missing cell to be placed in the floorplan

Feedback

- Specify the `record_unbound_instance` strategy

```
fc_shell> set_attribute \
   [get_mismatch_types missing_logical_reference] \
   current_repair(user_config) record_unbound_instance
```

The tool records the unresolved reference but does not perform any repairs, as shown in the following report:

```
prompt > report_design_mismatch -verbose
...
==============================
Design : test
==============================
Mismatch Type                Total Count      Repaired        Unrepaired
-------------------------------------------------------------------------------------------------
missing_logical_reference    1                0               1


Mismatch Type : missing_logical_reference

ObjName      ObjType   State          Repair Strategy           Inst    MMObjName
-------------------------------------------------------------------------------------------------------------------------------------
sub_design1  design    not_repaired   record_unbound_instance   1       missing_logical_reference_test_missing_design
```

See Also

- Mismatch Configurations and Types

missing_port

The `missing_port` mismatch type indicates that ports are missing between an instance and its reference object (module or library cell).

In the following example, the MP pin is missing from the subdesign and library cell definitions, so the number of ports differs between the top-level module and the reference cells, as highlighted.

- Instances in the top-level module:

```
module top (a, b, c);
   input [1:0] a, b;
   output [1:0] c;
   sub_design U0 (.MP(mp[0]), .A(a[0]), .B(b[0]), .C(c[0]));
   AND2X1_HVT U1 (.MP(mp[1]), .A1(a[1]), .A2(b[1]), .Y(c[1]));
endmodule
```

- Subdesign:

```
module sub_design (A, B, C);
   input A, B;
   output C;
```

```
   assign C = A|B;
endmodule
```

- Library cell:

```
cell (AND2X1_HVT) {
   pin (A1){...} pin (A2){...} pin (Y){...}
}
```

As a result, the extra port is removed from the top-level module and the logic connected to the port is preserved.

```
sub_design U0 (.A(a[1:0]), .B(b[1:0]), .C(c[1:0]));
AND2X1_HVT U1 (.A1(a[2]), .A2(b[2]), .Y(c[2]));
```

See Also

- Mismatch Configurations and Types

port_name_case

The `port_name_case` mismatch type indicates a port name mismatch between an instance and its reference object (module or library cell) due to case sensitivity.

If the `action` attribute is set to `repair`, the tool ignores the case differences, records the mismatch, and links the design using the reference design port names. In the following example, the `.bp` and `.a2` instances use different cases in the netlist than they do in the subdesign and library cell, so a port name mismatch exists between the netlist and the reference cells, as highlighted.

- Instances in the top-level module:

```
module test (a, b, c);
   input [1:0] a, b;
   output [1:0] c;
   sub_design U0 (.AP(a[0]), .bp(b[0]), .CP(c[0]));
   AND2X1_HVT U1 (.A1(a[1]), .a2(b[1]), .Y(c[1]));
endmodule
```

- Subdesign:

```
module sub_design (AP, BP, CP);
   input AP, BP;
   output CP;
   assign CP = AP|BP;
endmodule
```

- Library cell:

```
cell (AND2X1_HVT) {
   pin (A1){...} pin (A2){...} pin (Y){...}
}
```

As a result, the instances of the modules or cells are updated using the reference design port names.

```
sub_design U0 (.AP(a[0]), .BP(b[0]), .CP(c[0]));
AND2X1_HVT U1 (.A1(a[1]), .A2(b[1]), .Y(c[1]));
```

See Also

- Mismatch Configurations and Types

port_name_synonym

The `port_name_synonym` mismatch type indicates a port name mismatch between an instance and its reference object (module or library cell) when a port name synonym is defined.

In the following example, the XX and YY ports in the netlist have different names in the subdesign (BP) and library cell (A2), as highlighted.

- Instances in the top-level module:

```
module test (a, b, c);
   input [1:0] a, b;
   output [1:0] c;
   sub_design U0 (.AP(a[0]), .XX(b[0]), .CP(c[0]));
   AND2X1_HVT U1 (.A1(a[1]), .YY(b[0]), .Y(c[1]));
endmodule
```

- Subdesign:

```
module sub_design (AP, BP, CP);
   input AP, BP;
   output CP;
   assign C = AP|BP;
endmodule
```

- Library cell:

```
cell (AND2X1_HVT) {
   pin (A1){...} pin (A2){...} pin (Y){...}
}
```

To allow the port name mismatch, define port name synonym rules for the mismatched ports by using the `set_pin_name_synonym` command. The following commands set the XX synonym for the BP port and the YY synonym for the A2 port:

```
fc_shell> set_pin_name_synonym BP XX
fc_shell> set_pin_name_synonym A2 YY
```

The tool records the mismatch and links the design using the synonym rules:

```
sub_design U0 (.AP(a[1:0]), .BP(b[1:0]), .CP(c[1:0]));
AND2X1_HVT U1 (.A1(a[2]), .A2(b[2]), .Y(c[2]));
```

See Also

- Mismatch Configurations and Types

reference_name_case

The `reference_name_case` mismatch type indicates a cell name mismatch between an instance and its reference object (module or library cell) due to case sensitivity.

In the following example, the `AND2X1_HVT` library cell and its instance in the netlist are case-different, so a cell name mismatch exists between the netlist and the reference cell, as highlighted. Similarly, a case mismatch exists between the `SUB_design` module and its instance in the netlist, as highlighted.

- Instances in the top-level module:

```
module top (a, b, c);
   input [1:0] a, b;
   output [1:0] c;
   sub_design U0 (.A(a[0]), .B(b[0]), .C(c[0]));
   and2x1_hvt U1 (.A1(a[1]), .A2(b[0]), .Y(c[1]));
endmodule
```

- Subdesign:

```
module SUB_design (A, B, C);
   input A, B;
   output C;
   assign C = A|B;
endmodule
```

- Library cell:

```
cell (AND2X1_HVT) {
   cell_footprint : "AND2";
}
```

Ignoring case differences, the tool records the mismatches and links the design using the reference names.

```
SUB_design U0 (.A(a[1:0]), .B(b[1:0]), .C(c[1:0]));
AND2X1_HVT U1 (.A1(a[2]), .A2(b[2]), .Y(c[2]));
```

See Also

- Mismatch Configurations and Types

Feedback

Controlling Case Sensitivity During Linking

By default, the tool uses the uppercase and lowercase letter matching rules of the HDL language to link modules and cells. The least-sensitive matching rule is applied to resolving references of a design. For example, a VHDL reference is not case-sensitive and a Verilog reference is case-sensitive during linking; however, a mixed language reference is case-insensitive.

To enable the tool to resolve references ignoring case sensitivity during linking, set the `link.force_case` application option to `case_insensitive` before linking the design. For example,

```
fc_shell> set_app_options -name link.force_case \
   -value case_insensitive
```

Note:

You must set this application option before linking the design. This setting is effective only for the `default` configuration and is ignored by the `auto_fix` configuration.

Setting a Mismatch Configuration

By default, the tool performs checks for mismatched data using the `default` mismatch configuration, which does not allow any mismatches. To set a different mismatch configuration,

1. Specify the configuration with the `set_current_mismatch_config` command.

   You can specify the `auto_fix` configuration or a custom configuration.

   Note:

   If you want to set a custom mismatch configuration, you must first create the mismatch configuration (see Creating a Custom Mismatch Configuration).

   The following example sets the mismatch configuration to the predefined `auto_fix` configuration:

   ```
   fc_shell> set_current_mismatch_config auto_fix
   ```

2. If you specify the `auto_fix` configuration and want to change the default repair process for one or more mismatch types, set the `current_repair` attribute on these mismatch types to the desired repair process.

   You can get a list of available repairs for a mismatch type by querying the mismatch type's `available_repairs` attribute.

Feedback

The following example sets the mismatch configuration to the predefined `auto_fix` configuration and specifies the repair process to use when the tool encounters the `missing_logical_reference` mismatch type:

```
fc_shell> set_current_mismatch_config auto_fix

fc_shell> get_attribute \
   [get_mismatch_types missing_logical_reference] \
   available_repairs
{create_blackbox record_unbound_instance}

fc_shell> set_attribute \
   [get_mismatch_types missing_logical_reference] \
   current_repair(auto_fix) record_unbound_instance
```

See Also

- Mismatch Configurations and Types
- Creating a Custom Mismatch Configuration

Creating a Custom Mismatch Configuration

Instead of enabling a predefined mismatch configuration, you can create your own mismatch configuration.

To create a custom mismatch configuration,

1. Create a name for the configuration using the `create_mismatch_config` command.

   By default, the netlist mismatch types are treated as error conditions. To specify a different mismatch configuration as the basis for the new user-defined configuration, use the `-ref_config` option.

   The following example creates a custom mismatch configuration named user_def:

   ```
   fc_shell> create_mismatch_config user_def
   ```

2. Specify how the tool should handle a specific mismatch type within this configuration by setting the mismatch type's `action` attribute.

   Specify a value of `accept`, `error`, `repair`, or `ignore` for the `action` attribute:

   - `accept`

     The tool mitigates the mismatch but does not record it.

   - `repair`

     The tool mitigates the mismatch and records it.

- `error`

  The tool does not mitigate the mismatch but issues an error message.

- `ignore`

  The tool ignores the mismatch, neither issuing an error message nor mitigating the mismatch.

Note that only some of these actions are available for certain mismatch types. To query the actions available for a mismatch type, use the `report_mismatch_configs` command.

The following example sets the `action` attribute for the `missing_logical_reference` and `missing_port` mismatch types to `repair` for this specifc configuration:

```
fc_shell> set_attribute \
   [get_mismatch_type missing_logical_reference] \
   action(user_def) repair

fc_shell> set_attribute [get_mismatch_type missing_port] \
   action(user_def) repair
```

3. If you set the `action` attribute to `repair`, specify the repair process to use by setting the mismatch type's `current_repair` attribute.

   To determine the supported repairs for a mismatch type, query the mismatch type's `available_repairs` attribute. Table 3 describes many of the mismatch types and their available repairs.

   The following example sets the repair process for the `missing_logical_reference` and `missing_port` mismatch types for this configuration:

   ```
   fc_shell> set_attribute \
      [get_mismatch_type missing_logical_reference] \
      current_repair(user_def) create_blackbox

   fc_shell> set_attribute [get_mismatch_type missing_port] \
      current_repair(user_def) delete_extra_port
   ```

After you have created a custom mismatch configuration, you can set the mismatch configuration as described in Setting a Mismatch Configuration. In this example, once you set the user_def configuration and link your design, the tool repairs any `missing_logical_reference` or `missing_port` mismatches that it enounters and flags all other mismatches with error or warning messages.

Feedback

See Also

- Mismatch Configurations and Types
- Setting a Mismatch Configuration

Reporting Mismatch Configurations

You can report the current mismatch configuration settings or all the available mismatch types and their settings for a configuration, as described in the following table:

Report	Command
Report the name of the current mismatch configuration, such as `default` or `auto_fix`	`get_current_mismatch_config`
Report all the available mismatch types and their settings for a mismatch configuration	`report_mismatch_configs`
Report all available mismatch configurations	`report_mismatch_configs -all`
Restrict the report to specific mismatch configurations	`report_mismatch_configs -config_list` *list*
Report the mismatches identified and fixed by the tool during block linking	`report_design_mismatch`

For more information about reporting the mismatches identified by the tool, see Reporting Design Mismatches.

Mitigating Design Mismatches

To mitigate design mismatches, use the `link_block` or `set_top_module` command. When linking the design, these commands identify mismatches and perform the action specified by the current mismatch configuration.

See Also

- Setting a Mismatch Configuration
- Reporting Design Mismatches

Feedback

RTL Designs

For RTL designs, use the `analyze` and `elaborate` commands followed by the `set_top_module` command to read and link the design. The tool mitigates design mismatches when executing the `set_top_module` command. For example,

```
fc_shell> analyze -format verilog ${rtl_files}
fc_shell> elaborate ${top_design}
fc_shell> set_top_module ${top_design}
```

Verilog Netlists

For Verilog netlists, use the `read_verilog` and `set_top_module` commands to read and link the design. The tool mitigates design mismatches when executing the `set_top_module` command. For example,

```
fc_shell> read_verilog ${rtl_files} -top ${top_design}
fc_shell> set_top_module ${top_design}
```

Reporting Design Mismatches

You can generate a report of all the mismatches in the design by using the `report_design_mismatch` command. The command shows a summary of mismatch types and repairs, as well as the details of each mismatch type with its mismatch objects and repair strategy.

The tool can report design mismatches between

- Instances and reference objects (modules or library cells)
- Logic libraries and physical libraries (mitigation is performed by the library manager). For information about library mismatch types, see “Allowing Incomplete or Inconsistent Library Data” in the *Library Manager User Guide*.

For example,

- To report all instances related to a mismatch object, use the `-verbose` option.

  ```
  fc_shell> report_design_mismatch -verbose
  ```

- To report a mismatch object, use the `report_attributes` command.

  A mismatch object contains information for a specific mismatch type, such as repair strategy and reference objects.

  ```
  fc_shell> report_attributes -application \
     [get_mismatch_objects missing_logical_reference*]
  ```

- To get a collection of mismatch objects, use the `get_mismatch_objects` command.

Feedback

The following example returns a collection of empty logic modules in your design:

```
fc_shell> get_mismatch_objects \
   -mismatch_type empty_logic_module
```

- To return a collection of cell instances related to one or more mismatch objects, use the `get_cells` and `get_mismatch_objects` commands together.

  The following example returns a collection of cell instances related to the `missing*` mismatch objects:

  ```
  fc_shell> get_cells -of_objects [get_mismatch_objects missing*]
  ```

Limitations

The following types of mismatches are mitigated but not reported when the configuration is set to `auto_fix`.

Mismatch Type	No report for mitigation between
`reference_name_case`	RTL and libraries, RTL and RTL
`missing_port`	RTL and libraries, RTL and RTL

Handling Design Data Using the Early Data Check Manager

The Early Data Check Manager allows you to check designs for issues early in the design cycle. You can configure a set of design checks, policies, and strategies that are specific to design requirements, such as multivoltage, or application requirements, such as placement. These checks are enforced when the tool encounters different types of incorrect, incomplete, and inconsistent data. The tool provides comprehensive reports on the data checks. The general flow is as follows:

1. Use the `set_early_data_check_policy` command to define the violation handling policy for data checks.
2. Proceed with your tool flow. The tool detects and responds to violations throughout the flow.
3. Use the `report_early_data_checks` command to obtain a report about all violations or specific data checks.
4. Use the `get_early_data_check_records` command to get a Tcl collection of violations that can be used in other commands.

Feedback

5. Use the `write_early_data_check_config` command to save the settings in a file for future use.
6. Use the `remove_early_data_check_records` command to clear the data in preparation for another iteration.

You can view the predefined checks, policies, and strategies.

See Also

- Early Data Checks, Policies, and Strategies
- Setting the Policy for Early Data Checks
- Reporting Early Data Check Records
- Generating a Report of Early Data Check Records
- Writing the Configuration as a Tcl Script
- Removing Early Data Check Records

Early Data Checks, Policies, and Strategies

The tool can identify issues and violations in data during the early stages of design. You can set policies and strategy configurations for the predefined checks to allow, tolerate, or repair data. The following tables describe the predefined checks for

- Multivoltage designs (Table 4)
- Design planning (Table 5)
- Placement (Table 6)
- Legalization (Table 7)
- Optimization (Table 8)
- Top-Level closure (Table 9)

Table 4 Multivoltage Checks, Policies, and Strategies

Check	Supported policies	Available strategies	Description
`mv.supply.unknown_net_type`	`error`, `tolerate`		Checks the type of the supply nets
`mv.supply.inconsistent_resolution_type`	`error`, `tolerate`		Checks the inconsistent resolution type on the supply nets

Feedback

Table 4 Multivoltage Checks, Policies, and Strategies (Continued)

Check	Supported policies	Available strategies	Description
`mv.pst.port_state_redefined`	`error`, `tolerate`		Checks whether the specified port state is already defined either on the same supply or an equivalent supply
`mv.pst.conflict_power_state`	`error`, `tolerate`		Checks whether two conflicting power states with the same name are defined
`mv.pst.undefined_supply`	`error`, `tolerate`		Checks whether a supply port or net listed in the `create_pst` command is not defined in the UPF
`mv.pst.state_undefined_in_block`	`error`, `tolerate`		Checks whether a power state defined in a higher-level power state table is not included in the power state table of a lower-level scope
`mv.pst.conflict_supply_state`	`error`, `tolerate`		Checks whether two different supply states are specified in the power state table for supply nets or ports that are connected together
`mv.pst.power_state_undefined`	`error`, `tolerate`		Checks whether a port or power state for a supply is used in a power state table, but has not been defined
`mv.hier.conflict_switch_strategies`	`error`, `tolerate`		Checks whether a domain from a top-only UPF and a block UPF, which cover the same element, cannot be merged because the two domains have conflicting UPF settings

Feedback

Table 4 Multivoltage Checks, Policies, and Strategies (Continued)

Check	Supported policies	Available strategies	Description
mv.strategy.invalid_element	error, tolerate		Checks whether all elements specified with the set_repeater, set_isolation, set_level_shifter commands are illegal
mv.strategy.conflict_strategy	error, tolerate		Checks whether the same combination of filters (-source, -sink, -applies_to, -diff_supply_only, -applies_to_boundary) are applied on the domain or specified elements twice
mv.strategy.invalid_lib_cell	error, tolerate		Checks whether all library cells specified using the map_isolation_cell, map_level_shifter_cell, or user_interface_cell commands are illegal
mv.strategy.unsupported_in_bias	error, tolerate		Checks whether a UPF command option is incompatible inside a bias block
mv.upf.invalid_terminal_boundary	error, tolerate		Checks whether the terminal_boundary design attribute is set on a cell that is not a root element of a power domain
mv.upf.invalid_bias_block	error, tolerate		Checks whether the enable_bias design attribute is true on a block that is under a non-bias block

Feedback

Table 4 Multivoltage Checks, Policies, and Strategies (Continued)

Check	Supported policies	Available strategies	Description
mv.cells.iso_missing_csn	error, repair, tolerate	To repair, infer supply net connections using one of the following rules: • infer_from_primary • infer_from_sink • infer_from_control_signal	Checks whether an isolation cell has a supply net connection
mv.cells.ls_missing_csn	error, repair, tolerate	To repair, infer supply net connections using one of the following rules: • infer_from_primary • infer_input_from_source_output_from_sink	Checks whether a level-shifter cell has a supply net connection
mv.cells.ret_missing_csn	error, repair, tolerate	To repair, infer supply net connections using one of the following rules: • infer_from_primary • infer_from_most_always_on_supply	Checks whether a retention cell has a supply net connection
mv.cells.psw_missing_csn	error, repair, tolerate	To repair, infer supply net connections using one of the following rules: • infer_from_primary • infer_from_most_always_on_supply	Checks whether a power-switch cell has a supply net connection

Feedback

Table 4 Multivoltage Checks, Policies, and Strategies (Continued)

Check	Supported policies	Available strategies	Description
`mv.cells.ao_missing_csn`	`error`, `repair`	To repair, infer supply net connections using one of the following rules: • `infer_from_primary` • `infer_from_driver_or_load`	Checks whether an always-on cell has a supply net connection
`mv.cells.other_missing_csn`	`error`, `repair`	To repair, infer supply net connections using the `infer_from_primary` rule	Checks whether a cell has a supply net connection on a non-primary PG pin
`mv.cells.unmapped_gtech_cell`	`error`, `repair`		Checks whether unmapped WVGTECH cells are removed after compile
`mv.va.missing_voltage_area`	`error`, `repair`	To repair, infer supply net connections using one of the following rules: • `infer_for_synthesis_only` • `infer_va`	Checks whether the voltage area setup is not ready for physical implementation
`mv.buf.mv_violation`	`error`, `tolerate`		Checks whether isolation or voltage violations exist

Table 5 Design Planning Checks, Policies, and Strategies

Check	Supported policies	Available strategies	Description
`plan.floorplan.missing_layer_routing_direction`	`error`, `repair`		Checks whether the routing directions of metal layers are missing

Table 5 Design Planning Checks, Policies, and Strategies (Continued)

Check	Supported policies	Available strategies	Description
`plan.floorplan.core_offset_conflict_enclosure_rules`	`error`, `repair`, `tolerate`		Checks whether the floorplan core offset conflicts with enclosure rules
`plan.floorplan.core_area_length_conflict_width_rules`	`error`, `repair`, `tolerate`		Checks whether the floorplan core area conflicts with width rules
`plan.floorplan.missing_block_boundary`	`error`, `repair`		Checks whether the block boundary is missing
`plan.pg.strategy_via_rule`	`error`, `tolerate`		Checks whether the strategy via rule specified in the `compile_pg` command is correct as per the strategy specified

Table 6 Placement Checks, Policies, and Strategies

Check	Supported policies	Available strategies	Description
`place.coarse.missing_scan_def`	`error`, `tolerate`		Checks whether a valid scan chain definition is present
`place.coarse.invalid_power_setup`	`error`, `tolerate`		Checks whether the multivoltage setup is complete
`place.coarse.infeasible_constraints`	`error`, `tolerate`		Checks whether the physical constraints are feasible

Feedback

Table 6 Placement Checks, Policies, and Strategies (Continued)

Check	Supported policies	Available strategies	Description
`place.coarse.over_utilization`	`error`, `tolerate`	If you tolerate over-utilization, select one of the following: • `use_blocked_area` - Place cells on top of the blocked area to spread the cells out • `avoid_blocked_area` - Place cells on top of each other, but on top of the blocked area	Checks whether all physical regions have sufficient area to accommodate the assigned cells
`place.coarse.unremovable_overlap`	`error`, `tolerate`		Checks whether cell overlap can be removed
`place.coarse.invalid_bounds`	`error`, `tolerate`		Checks whether bounds are well defined
`place.coarse.invalid_region`	`error`, `tolerate`		Checks whether any physical region inferred from voltage areas, placement bounds, or both has issues
`place.coarse.invalid_ir_drop_setup`	`error`, `tolerate`		Checks whether the IR drop analysis setup is complete
`place.coarse.invalid_timing`	`error`		Checks whether timing analysis for timing-driven placement has issues
`place.coarse.incomplete_timing_data`	`error`, `tolerate`		Checks whether the required data for timing-driven placement is complete and correct
`place.coarse.unfixed_macros`	`error`, `tolerate`		Checks whether macros have placement status of fixed

Feedback

Table 7 Legalization Checks, Policies, and Strategies

Check	Supported policies	Available strategies	Description
`place.legalize.inconsistent_coloring_data`	`error`, `tolerate`		Checks whether the coloring data is consistent between layer tracks and library cell pins
`place.legalize.no_filler_cells`	`error`, `tolerate`		Checks whether the library has filler cells
`place.legalize.no_1x_filler_cell`	`error`, `tolerate`		Checks whether the library has a filler cell of size 1
`place.legalize.invalid_lib_cell_orientation`	`error`, `tolerate`		Checks whether the orientation of library cells and site rows match
`place.legalize.unaligned_lib_cells_pins`	`error`, `tolerate`		Checks whether library cells have their pins aligned with the routing tracks
`place.legalize.invalid_pg_rail_shape_use`	`error`, `tolerate`		Checks whether power and ground rails have the correct `shape_use` attribute setting
`place.legalize.over_utilization`	`error`, `tolerate`		Checks whether all physical regions have sufficient area to accommodate the assigned cells
`place.legalize.illegal_cells_at_end`	`error`, `tolerate`		Checks whether there are illegal cells placed at the end of legalization
`place.legalize.unplaced_cells_at_start`	`error`, `tolerate`		Checks whether cells are (coarse) placed at the start of legalization
`place.legalize.cell_outside_core_area`	`error`, `tolerate`		Checks whether any cell outside the core area remains unlegalized

Feedback

Table 7 Legalization Checks, Policies, and Strategies (Continued)

Check	Supported policies	Available strategies	Description
`place.legalize.zero_pass_rate_cells`	`error`, `tolerate`		Checks whether any cell has a library cell with a zero pass rate; if it does, the cell remains unlegalized
`place.legalize.lib_cell_incorrect_site_data`	`error`, `tolerate`		Checks whether any cell has a library cell with incorrect site data; if it does, the cell is ignored during legalization
`place.legalize.invalid_app_option`	`error`, `tolerate`		Checks whether cell spreading and ordering is turned off during advanced legalization
`place.legalize.illegal_symmetry`	`error`, `tolerate`		Checks whether site symmetry is set to `x` or `y`; a site symmetry value of `y` is critical to advanced legalization
`place.legalize.missing_cell_orientation`	`error`, `tolerate`		Checks whether cell instances have missing or illegal orientations
`place.legalize.route_layer_direction`	`error`, `tolerate`		Checks whether the routing layer direction is specified

Table 8 Optimization Checks, Policies, and Strategies

Check	Supported policies	Available strategies	Description
`opt.cells.incorrect_pvt`	`error`, `tolerate`		Checks whether the library cells of multivoltage cells (isolation/level-shifter/enable level-shifter/retention) are used with incorrect PVT settings
`opt.clock_latency_estimation.missing_clock_routing_rules`	`tolerate`		Checks whether clock routing rules have been defined

Feedback

Table 8 Optimization Checks, Policies, and Strategies (Continued)

Check	Supported policies	Available strategies	Description
`opt.clock_latency_estimation.missing_clock_repeaters`	`tolerate`		Checks whether clock repeaters have been configured
`opt.sanity_check.max_trans`	`tolerate`		Checks whether there are maximum transition constraints in the design
`opt.sanity_check.invalid_buffer_inverter`	`error`		Checks whether there are valid library buffers or inverters for optimization in the design
`opt.sanity_check.link`	`error`		Checks whether the design has any unresolved cell references
`opt.sanity_check.mv_data_ready_check`	`error`		Checks whether critical multivoltage settings such as data or voltage area setup are ready for the `compile_fusion`, `clock_opt`, and `route_opt` commands
`opt.sanity_check.layer_exist`	`error`		Checks whether the design has a routing layer
`opt.sanity_check.icv_binary`	`error`, `tolerate`		Checks whether the IC Validator executable path is set correctly when metal fill is enabled within `route_opt`

Feedback

Table 9 Top-Level Closure Checks, Policies, and Strategies

Check	Supported policies	Supported references	Available strategies	Description
hier.block.missing_frame_view	error, repair, tolerate	MID, BOT	Re-create frame views from the bound view	Checks whether the frame view is missing for a hierarchical block
hier.block.reference_missing_port_location	error, repair, tolerate	MID, BOT	Assign a location for the block pin on the block boundary based on connectivity	Checks whether the location of a physical hierarchy boundary pin is missing
hier.block.reference_port_outside_boundary	error, repair, tolerate	MID, BOT	Reassign a location for the block pin on the block boundary based on connectivity	Checks whether the location of a physical hierarchy boundary pin is outside the physical hierarchy boundary
hier.block.reference_missing_port	error, repair, tolerate	MID, BOT	Create missing ports in the reference block and update its location	Checks whether a port is missing in the physical hierarchy reference
hier.block.instance_bound_to_frame	error, tolerate	TOP		Checks whether a physical hierarchy instance is bound to a frame view
hier.block.instance_with_design_type_macro	error, tolerate	TOP		Checks whether a physical hierarchy instance is bound to a macro
hier.top.estimated_corner	error, tolerate	TOP		Checks whether the estimated corner is present at top level
hier.block.missing_leaf_cell_location	error, repair, tolerate	MID, BOT	Assign an approximate location inside the block boundary for the leaf cell (legality not considered)	Checks whether the location of a leaf cell of a physical hierarchy is missing

Feedback

Table 9 Top-Level Closure Checks, Policies, and Strategies (Continued)

Check	Supported policies	Supported references	Available strategies	Description
hier.block.leaf_cell_outside_boundary	error, repair, tolerate	MID, BOT	Reassign the leaf cell location to the nearest location inside the block boundary (legality not considered)	Checks whether the location of a leaf cell of a physical hierarchy is outside the physical hierarchy boundary
hier.block.missing_child_block_instance_location	error, repair, tolerate	MID	Assign an approximate location for the lower-level subblock based on connectivity inside the block boundary (overlap removal not considered)	Checks whether the location of a child physical hierarchy instance is missing in the specified physical hierarchy
hier.block.child_block_instance_outside_boundary	error, repair, tolerate	MID	Move the subblock to the nearest location inside the block boundary (overlap removal not considered).	Checks whether the location of a child physical hierarchy instance is outside the specified physical hierarchy boundary
hier.block.port_mismatch_between_views	error, repair, tolerate	MID, BOT	Bound view is treated as golden and the frame view is recreated to match the bound view	Checks whether there is a mismatch of ports between the bound view and the frame view of a physical hierarchy
hier.block.abstract_missing_design_view	error, tolerate	MID, BOT		Checks whether the design view for an abstract is missing

Feedback

Table 9 Top-Level Closure Checks, Policies, and Strategies (Continued)

Check	Supported policies	Supported references	Available strategies	Description
hier.block.instance_unlinked	error	TOP		Checks whether the physical hierarchy reference is not linked properly to the top-level design due to a view change from frame or ETM to abstract or design
hier.block.abstract_type_non_timing	error	MID, BOT		Checks whether the abstract type is not timing
hier.block.abstract_target_use_non_implementation	error	MID, BOT		Checks whether the target use of an abstract is not implementation
hier.block.missing_core_area	error	MID, BOT		Checks whether the core area for a physical hierarchy is missing
hier.block.unmapped_logic	error	MID, BOT		Checks whether unmapped logic is present in the physical hierarchy

See Also

- Handling Design Data Using the Early Data Check Manager
- Setting the Policy for Early Data Checks
- Reporting Early Data Check Records
- Generating a Report of Early Data Check Records
- Writing the Configuration as a Tcl Script
- Removing Early Data Check Records

Feedback

Setting the Policy for Early Data Checks

To set or modify the policy and strategy settings for data checks, use the `set_early_data_check_policy` command. You can use the following options to configure the policy:

- `-checks` – Specifies a predefined list of data checks. It supports an asterisk wildcard character (*).
- `-policy` – Specifies the type of policy depending on what is supported by the check. This can be
  - `error` – The tool does not mitigate the violation, but issues an error message.
  - `tolerate` – The tool makes predictable and explainable assumptions to mitigate the violation. No changes are made to the design or setup.
  - `repair` – The tool mitigates the violation using one or more repair strategies and records it. Repair can include changes in design and setup.
  - `strict` – The tool sets the first matching policy in this order – error, tolerate, repair.
  - `lenient` - The tool sets the first matching policy in this order – repair, tolerate, error.

  Note:

  If you specify both the checks and policies, the configuration supports all the policies described in this section, depending on what the application-specific check supports. However, if you specify only the policies, but not the checks, the configuration supports only the `strict` and `lenient` policies.

  You can use the `if_not_exists` option with the `-policy` option to set a policy on the current block only if the block does not already have a policy set.
- `-strategy` – Specifies the strategy to apply for a check.
- `-references` – Specifies the policy and strategy for reference subblocks.

The command supports the `-policy` and `-strategy` options with the `-checks` option. The `-strategy` option is optional when both the `-checks` and `-policy` options are specified.

The following example sets the policy as `lenient` for all checks:

```
fc_shell> set_early_data_check_policy -policy lenient
Config set successfully.
1
fc_shell> report_early_data_checks -policy
Design             Check                                    Policy         Strategy
-------------------------------------------------------------------------------------
-------------

test:test.design   mv.buf.mv_violation                      tolerate
```

```
test:test.design   mv.cells.ao_missing_csn                repair
 infer_from_primary
test:test.design   mv.cells.iso_missing_csn               repair
 infer_from_primary
test:test.design   mv.cells.ls_missing_csn                repair
 infer_from_primary
...
test:test.design   opt.clock_latency_estimation.missing_clock_repeaters
                                                          tolerate
test:test.design   opt.clock_latency_estimation.missing_clock_routing_rules
                                                          tolerate
test:test.design   place.coarse.incomplete_timing_data
                                                          tolerate
test:test.design   place.coarse.infeasible_constraints
                                                          tolerate
...
test:test.design   place.legalize.unaligned_lib_cells_pins
                                                          tolerate
test:test.design   place.legalize.unplaced_cells_at_start
                                                          tolerate
test:test.design   place.legalize.zero_pass_rate_cells
                                                          tolerate
test:test.design   plan.floorplan.core_area_length_conflict_width_rules
                                                          repair
test:test.design   plan.floorplan.core_offset_conflict_enclosure_rules
                                                          repair
```

The following example shows how the tool handles missing SCANDEF when a `strict` policy is specified:

```
fc_shell> set_early_data_check_policy -policy strict \
  -checks place.coarse.missing_scan_def
1
fc_shell> create_placement
Information: Policy for early data check 'place.coarse.missing_scan_def' is
 'error'. (EDC-001)
Error: No valid scan def found. (PLACE-042)
Information: Ending 'create_placement' (FLW-8001)
fc_shell> report_early_data_checks -checks place.coarse.missing_scan_def
Check                            Policy    Strategy      Fail Count
--------------------------------------------------------------------------------
place.coarse.missing_scan_def    error                   1
--------------------------------------------------------------------------------
```

The following example shows how the tool handles missing SCANDEF when a `lenient` policy is specified:

```
fc_shell> set_early_data_check_policy -policy lenient \
  -checks place.coarse.missing_scan_def
1
fc_shell> create_placement
Information: Policy for early data check 'place.coarse.missing_scan_def' is
 'tolerate'. (EDC-001)
Continuing without valid scan def.
Start transferring placement data.
Creating placement from scratch.
coarse place 0% done.
…
```

Feedback

```
coarse place 100% done.

fc_shell> report_early_data_checks -checks place.coarse.missing_scan_def
Check                          Current Policy     Available Strategies
 Supported Policies  Help
-------------------------------------------------------------------------------
-------------------------------------
place.coarse.missing_scan_def    tolerate           []
 {error tolerate }   Check whether a valid scan chain definition is present
-------------------------------------------------------------------------------
-------------------------------------
```

The following example modifies the reference subblock using the `-references` option:

```
fc_shell> set_early_data_check_policy -policy repair \
  -checks hier.block.missing_leaf_cell_location -references {sub2}
```

For more information about the `set_early_data_check_policy` command settings, see the man page.

See Also

- Handling Design Data Using the Early Data Check Manager
- Early Data Checks, Policies, and Strategies
- Reporting Early Data Check Records
- Generating a Report of Early Data Check Records
- Writing the Configuration as a Tcl Script
- Removing Early Data Check Records

Reporting Early Data Check Records

You can report the policy and strategy set for all checks or for a specific set of checks by using the `report_early_data_checks` command. The following options configure the report:

- `-policy` – Reports configuration details for all checks. It shows the selected policy and strategy for each check. Using the `-hierarchical` option with the `-policy` option reports the configuration of the reference subblocks.
- `-checks` – Reports the current policy, supported policies, and strategies for each check. Using the `-hierarchical` option with the `-checks` option reports the checks of the reference subblocks.
- `-hierarchical` – Traverses the hierarchy and reports the check, policy, strategy, and fail count for each block in the hierarchy.

Feedback

- `-ems_report` - Reports an enhanced messaging system (EMS) summary of failed checks, policies, and strategies for the checked objects when used with the `-verbose` option.
- `-verbose` – Reports a summary of all failed checks, policies, strategies applied, and checked objects with comments. Using the `-hierarchical` option with the `-verbose` option reports the records of all failed checks, policies, strategies applied, and checked objects from subblocks. Using the `-ems_report` option with the `-verbose` option reports the EMS summary of the failed checks, policies, strategies for the checked objects.

The following example shows detailed information about the current design configuration:

```
fc_shell> report_early_data_checks -policy -hierarchical
Design                    Check                                Policy         Strategy
------------------------------------------------------------------------------------
-------

test:test.design    mv.buf.mv_violation                        tolerate
test:test.design    mv.cells.ao_missing_csn                    repair
 infer_from_primary
test:test.design    mv.cells.iso_missing_csn                   repair
 infer_from_primary
test:test.design    mv.cells.ls_missing_csn                    repair
 infer_from_primary
...
test:test.design    opt.clock_latency_estimation.missing_clock_repeaters
                                                               tolerate
test:test.design    opt.clock_latency_estimation.missing_clock_routing_rules
                                                               tolerate
test:test.design    place.coarse.incomplete_timing_data
                                                               tolerate
test:test.design    place.coarse.infeasible_constraints
                                                               tolerate
...
unit_des.nlib:top.design  hier.block.instance_bound_to_frame                 error

unit_des.nlib:top.design  hier.block.instance_unlinked                       error

unit_des.nlib:top.design  hier.block.instance_with_design_type_macro         error

unit_des.nlib:top.design  hier.top.estimated_corner                          error
------------------------------------------------------------------------------------
------
blk                       hier.block.abstract_missing_design_view            error

blk                       hier.block.abstract_target_use_non_implementation  error

blk                       hier.block.abstract_type_non_timing                error

blk                       hier.block.leaf_cell_outside_boundary              error
------------------------------------------------------------------------------------
--------
```

Feedback

The following example shows a verbose report with the `-verbose` and `-hierarchical` options:

```
fc_shell> report_early_data_checks -hierarchical -verbose
Design Check                                  Policy  Strategy Checked Object
 Comment
-------------------------------------------------------------------------------
------------------------
blk    hier.block.missing_leaf_cell_location repair          eco_cell
 Instance location updated
                                                                            to
 X: 150145 Y: 69620
-------------------------------------------------------------------------------
------------------------
```

The following example shows how to create an EMS report and then view the report with the `-verbose` and `-ems_report` options:

```
fc_shell> create_ems_database dcm_report.ems
fc_shell> report_early_data_checks -ems_report -verbose
**************************************

Report : report_early_data_checks

Design : Top

Version: S-2021.06-DEV

Date   : Sun Apr 25 16:10:19 2021

**************************************

Check                            Policy    Strategy                  Checked
 Objects     Comment

-------------------------------------------------------------------------------
-------

link.netlist.bus_bit_naming    repair    bus_bit_blast_naming
 test.nlib/U/tx_1

link.netlist.bus_bit_naming    repair    bus_bit_blast_naming
 test.nlib/U/tx_0

-------------------------------------------------------------------------------
-------

1

fc_shell> report_ems_database


 -------------------------------------------------------------------------------
----------------------------
```

```
  Rule      Type   Count   Message

 -------------------------------------------------------------------------
---------------------------

  DCM-002   Info   2       Early data check %Check failed for object
 %Checked_Object in des...


 -------------------------------------------------------------------------
---------------------------

Total messages: 2, Errors: 0, Warnings: 0, Info: 2

Messages

--------

Information:

Early data check link.netlist.bus_bit_naming failed for object
 test.nlib/U/tx_1 in
design Top. (DCM-002)

Information:

Early data check link.netlist.bus_bit_naming failed for object
 test.nlib/U/tx_0 in
design Top. (DCM-002)

1
```

If you use the `report_early_data_checks` command without any options, you get a report of the checks, policies, strategy, and fail count for the block.

See Also

- Handling Design Data Using the Early Data Check Manager
- Early Data Checks, Policies, and Strategies
- Setting the Policy for Early Data Checks
- Generating a Report of Early Data Check Records
- Writing the Configuration as a Tcl Script
- Removing Early Data Check Records

Feedback

Generating a Report of Early Data Check Records

You can generate a report of check records from a block by using the `get_early_data_check_records` command. Using the `-hierarchical` option with this command enables to you get check records for all subblocks.

The report generates check records in the following format:

check_name@object_name

For example,

```
fc_shell> get_early_data_check_records -hierarchical
{place.legalize.no_filler_cells@design:top.design
 place.legalize.unplaced_cells_at_start@and1
place.legalize.unplaced_cells_at_start@and2
 place.legalize.unplaced_cells_at_start@and3
place.legalize.unplaced_cells_at_start@and4
 place.legalize.unplaced_cells_at_start@and5
place.legalize.unplaced_cells_at_start@flop0
 place.legalize.unplaced_cells_at_start@flop1
...
place.legalize.unplaced_cells_at_start@flop88
 place.legalize.unplaced_cells_at_start@flop89
place.legalize.unplaced_cells_at_start@flop90
 place.legalize.unplaced_cells_at_start@flop91
place.legalize.unplaced_cells_at_start@flop92
 place.legalize.unplaced_cells_at_start@flop93
```

You can use the `-filter` option to filter the results of this command by various parameters such as object class and check name.

For example,

```
fc_shell> get_early_data_check_records \
   -filter "check_name == place.coarse.over_utilization"
   {place.coarse.over_utilization@design:top.design}
```

For more information about all the supported options for the `get_early_data_check_records` command, see the man page.

See Also

- Handling Design Data Using the Early Data Check Manager
- Early Data Checks, Policies, and Strategies
- Setting the Policy for Early Data Checks
- Reporting Early Data Check Records

Writing the Configuration as a Tcl Script

To write the early data check configuration as a Tcl script, use the `write_early_data_check_config` command:

The `-output` option specifies the name of the output file into which the script is written.

For example,

```
fc_shell> write_early_data_check_config -output config.tcl
```

See Also

- Handling Design Data Using the Early Data Check Manager
- Early Data Checks, Policies, and Strategies
- Setting the Policy for Early Data Checks
- Reporting Early Data Check Records
- Generating a Report of Early Data Check Records
- Removing Early Data Check Records

Removing Early Data Check Records

You can remove all the check records of the current block by using the `remove_early_data_check_records` command. You can also remove specific check records by specifying the check record name. Using the `-hierarchical` option with this command removes all the check records for each subblock in the hierarchy.

For example, the following command removes all the records from the current block and its subblocks:

```
fc_shell> remove_early_data_check_records -hierarchical
Records removed successfully !!
```

You can use the `-filter` option with the `get_early_data_check_records` command to filter the records based on various criteria, and then remove them using the `remove_early_data_check_records` command.

For example, the following command removes all the repaired records:

```
fc_shell> remove_early_data_check_records \
   [get_early_data_check_records -filter "policy==repair"]
```

Feedback

```
Records removed successfully !!
```

See Also

- Handling Design Data Using the Early Data Check Manager
- Early Data Checks, Policies, and Strategies
- Setting the Policy for Early Data Checks
- Reporting Early Data Check Records
- Generating a Report of Early Data Check Records
- Writing the Configuration as a Tcl Script

Saving a Design in ASCII Format

In addition to saving a design as a block in a design library, you can also save the design as a set of ASCII text files for transfer to third-party tools.

These are the types of ASCII design data files and the commands to generate them:

- Verilog netlist

  ```
  fc_shell> write_verilog -hierarchy all my_design.v
  ```

- DEF file (Design Exchange Format physical design description)

  ```
  fc_shell> write_def my_design.def
  ```

- UPF file (IEEE 1801 Unified Power Format power intent specification file)

  ```
  fc_shell> save_upf my_design.upf
  ```

- SDC files (Synopsys Design Constraints files)

  ```
  fc_shell> write_sdc -output my_design.sdc
  ```

  You can also use the `write_script` command to write a script that contains more complete design and constraint setup information than what is written to the SDC file.

In the Fusion Compiler front-end mode, you can use the `write_ascii_files` command to export the design information in ASCII format. The command writes out your design's netlist data, floorplan data, constraint data, UPF data, and more, depending on the data present. The command also writes out your current session's configuration data, such as your library setup information and application option settings, to preserve the state of the tool.

Feedback

You must specify a directory for the output files with the `-output` option. Figure 17 shows an example directory structure created by the `write_ascii_files` command for a design named top, with the top-level directory, OUT/FC.ASCII, supplied to the `-output` option.

Figure 17 Directory structure for a design named top

```
OUT/FC.ASCII/
├── top.cell_expansion.exp
├── top.fc_script.tcl
├── top.floorplan
│   ├── floorplan_compare_data.txt
│   ├── floorplan.def
│   ├── floorplan.tcl
│   ├── fp.tcl
│   └── mapfile
├── top.global_app_options.tcl
├── top.global_app_options.tcl___synenc_proc_
├── top.library_setup.tcl
├── top.MCMM
│   ├── corner_SC1.tcl
│   ├── design.block_app_options.tcl
│   ├── design.block_app_options.tcl___synenc_proc_
│   ├── design.tcl
│   ├── mode_SC1.tcl
│   ├── scenario_SC1.tcl
│   └── top.tcl
├── top.scan.def
├── top.saif.map
├── top.rp.def
├── top.upf
├── top.v
└── top.write_ascii_files.log
```

The output directory always contains a *design_name*.fc_script.tcl file, the top-level script that loads the design back into the tool, and an execution log file, *design_name*.write_ascii_files.log. You can specify the data to be written out, such as modes, corners, and design objects, with the `write_ascii_files` command options. You can also specify options for library creation, UPF modes, and more. For detailed information about the command options, see the `write_ascii_files` man page.

Saving and Restoring Application Option User Values Between Sessions

Not all application option settings are persistent across tool sessions. The tool has simplified the flow for saving all the application option settings that you have specified into an encrypted file and restoring them in another session, so that there is consistency in the application option settings between sessions.

The new flow is as follows:

1. Write the application option values to a file by using the `write_app_options` command.

   The `write_app_options` command writes out the application option settings that you have specified, including any defaults, to an encrypted file.

   In the following example, the tool writes out the application option values, including any defaults, to a file called appOptionSetting.data in the encoded binary format.

   ```
   fc_shell> write_app_options -output appOptionSetting.data
   ```

   By default, the file is saved to the current directory. To place the file in another location, specify the full path to the location along with the file name.

   To omit the defaults, use the `-non-default` option with the `write_app_options` command.

2. In another session, read the file generated from step 1, which contains the application option values, by using the `read_app_options` command.

   In the following example, the tool reads from the appOptionSetting.data file that resides in the current directory and applies the application option values to the current block.

   ```
   fc_shell> read_app_options appOptionSetting.data
   ```

   If the file is located in any location other than the current directory, you can specify the full path to the file.

To specify the scope of the data to be read from the file, use the `-scope` option. The valid values for this option are:

- `global` - Only global application option values are read and set.
- `user_default` - Only user defaults of block scope application options are read and set.
- `block` - Only user values of block scope applications options are read and set.

The `read_app_options` command loads only the nondefault application option setting.

Error Checking While Reading Application Option Values Across Sessions

The tool validates the following when you read the application option values across sessions:

- If the tool version from which you wrote is different from the tool version from which you are reading the data, the values are not applied and warnings are displayed.
- If the value is invalid, the value is not set and a warning is displayed.
- If there are no current blocks, block-scoped values are not set and warnings are displayed.

The tool displays the application options for which values are not set.

Writing a Design in GDSII or OASIS Stream Format

You can write out the physical data of a design in GDSII or OASIS stream format for tapeout or for exporting the data to third-party tools. The `write_gds` and `write_oasis` commands are available in both the implementation tool (fc_shell) and library manager tool (icc2_lm_shell).

To write out the current block in GDSII format:

```
fc_shell> write_gds my_design.gds
```

To write out a specified block in GDSII format:

```
fc_shell> write_gds -library libA -design blockB my_design.gds
```

To write out the current block in GDSII format and map the Fusion Compiler layers to GDSII layers as specified in a layer mapping file:

```
fc_shell> write_gds -layer_map tech12.map my_design.gds
```

Feedback

To write out the entire hierarchical design in GDSII format, including the current block and all of its lower-level blocks:

```
fc_shell> write_gds -hierarchy all my_design.gds
```

To write out the design up to a specific level of hierarchy, use the `-child_depth` option with the `write_gds` or `write_oasis` command. In the following example, only the top block is written out in GDSII format:

```
fc_shell> write_gds -child_depth 0 my_design.gds
```

To generate the metal layers below the cut metal when there is only one routing or pin shape, including nonzero spacing blockage, abutting the cut metal, use the `-connect_below_cut_metal` option with the `write_gds` or `write_oasis` command:

```
fc_shell> write_gds -connect_below_cut_metal my_design.gds
```

By default, the `write_gds` or `write_oasis` command generates the metal below the cut metal only when there are two metal shapes abutting the cut metal. To make the `write_gds` or `write_oasis` command enable half metal generation, set the `file.gds.enable_half_metal_generation` application option to `true` before running the `write_gds` or `write_oasis` command.

By default, the `write_gds` or `write_oasis` command writes out the current block of the current library and maintains the original layer numbers as it converts the Fusion Compiler data to stream format. The `write_gds` and `write_oasis` command options let you specify:

- The name of the library, design, and view to be written out
- The mapping of Fusion Compiler database layers to stream-format layers
- The mapping of net, instance, and pin names in the conversion to stream format
- The manner of writing out pin objects (text or geometry)
- The handling of mask-shifted multiple-patterning data
- The layers, hierarchical blocks, and fill data to be included in the stream output
- The merging of design data with existing stream files

You can improve the performance of the `write_gds` and `write_oasis` commands by setting the following application options to `true`:

- `file.gds.enable_new_writer`
- `file.oasis.enable_new_writer`

Both these application options are enabled by default.

Feedback

The write_gds and write_oasis commands have many more options not described here. For details, see the man page for the command. For information about the mapping of tool layers to stream-format layers, see Fusion Compiler Layer Mapping File.

Reducing the GDSII and OASIS File Size

You can reduce the file size when you write out GDSII and OASIS files by using via arrays, via matrixes, or PG pattern shapes.

Compressing PG Vias Into Via Arrays

You can reduce the file size of GDSII and OASIS files by compressing individual PG vias into via arrays, instead of creating a record for each via in the stream file:

- To compress PG vias into via arrays when generating GDSII layout files, set the file.gds.pg_via_arrays application option to true before running the write_gds command:

```
fc_shell> set_app_options -name file.gds.pg_via_arrays \
   -value true
fc_shell> write_gds my_design.gds
```

- To compress PG vias into via arrays when generating OASIS layout files, set the file.oasis.pg_via_arrays application option to true before running the write_oasis command:

```
fc_shell> set_app_options -name file.oasis.pg_via_arrays \
   -value true
fc_shell> write_oasis my_design.oasis
```

Both application options are false by default. This feature supports PG vias of the following types only: simple vias, custom vias, and simple array vias.

To write out the via properties when you compress PG vias into via arrays, use the -via_property option when you run the write_gds or write_oasis command. If you read a GDSII or OASIS file that includes via properties back into the Fusion Compiler tool, you can create a custom via by specifying the -via_property option with the read_gds or read_oasis command and using the same -via_property value that you used to write out the via properties.

Using Via Matrixes or PG Pattern Shapes

You can significantly reduce the GDSII and OASIS file size and improve runtime by using via matrixes in blocks that contain a large number of PG vias. Via matrixes are compact array representations of individual via definitions that provide an efficient method for instantiating large regular patterns of vias in a block, resulting in a reduction in database size and faster processing by the tool.

Feedback

You can also reduce the GDSII and OASIS file size by using PG pattern shapes in blocks that contain a large number of PG wire shapes on individual layers. PG pattern shapes are compact array representations of individual PG wire shapes. As with via matrixes, you can instantiate PG pattern shapes in a block to reduce database size.

For information about working with via matrixes, see “Using Via Matrixes” in the *Fusion Compiler Design Planning User Guide*. For information about working with PG pattern shapes, see “Using PG Pattern Shapes” in the *Fusion Compiler Design Planning User Guide.*

Reading and Writing LEF Data

You can import and export library data in Library Exchange Format (LEF) so that you can use Fusion Compiler data with external tools. The LEF format defines the elements of an IC process technology and associated library of cell models, including layer, via, placement site type, and macro cell definitions.

Exporting a Cell Library to LEF

To export a complete cell library named S32hvt to a LEF file, use the following `write_lef` command:

```
fc_shell> write_lef -library S32hvt S32hvt.lef
```

The generated LEF file contains technology information and physical data for all cells in the S32hvt library.

Exporting a Block to LEF

To export a block named chipABC to a LEF file, use the following `write_lef` command:

```
fc_shell> write_lef -design chipABC chipABC.lef
```

The generated LEF file contains the technology information of the design library that contains the specified block. If the specified block is a frame view, the command writes the cell LEF. If the specified block is a design view, the command writes the cell LEF for all reference designs instantiated in the block as frame views.

When you export data to LEF, you can optionally specify which section types and `PROPERTY` statements to write out, and whether to write out macros. For details, see `write_lef` man page.

Importing LEF Library Data

In the library manager tool, you can read library data in LEF format by using the `read_lef` command. For example,

```
icc2_lm_shell> read_lef S32std.lef -library my_std_lib
```

The `read_lef` command also has options to convert site names, to control the merging of data into existing cell libraries, to specify which section types and `PROPERTY` statements to use, and to specify whether to read macros. For details, see the *Library Manager User Guide* and the `read_lef` man page.

Reading and Writing DEF Data

The Design Exchange Format (DEF) defines the elements of an IC design that are related to the physical layout, including the placement and routing information, design netlist, and design constraints. It contains the design-specific information for a circuit and is a representation of the design at any point during the layout process. You can import and export library data in DEF format to use Fusion Compiler data with external tools.

You can significantly reduce the DEF file size and improve runtime by using via matrixes in blocks that contain a large number of PG vias. Via matrixes are compact array representations of individual via definitions that provide an efficient method for instantiating large regular patterns of vias in a design, resulting in a reduction in database size and faster processing by the tool. For information about working with via matrixes, see Using Via Matrixes the *Fusion Compiler Design Planning User Guide*.

Exporting a Block to DEF

To write out the current block to a DEF file, including the physical layout, netlist, and design constraints, use the `write_def` command. If the current block is not linked, it is automatically linked and then written out.

The `write_def` command supports multithreading and uses the number of cores specified by the `set_host_options -max_cores` command. For information about multithreading, see Enabling Multicore Processing in the *Fusion Compiler User Guide*.

The following example writes a compressed DEF file using 1000 units per micron:

```
fc_shell> write_def -compress gzip -units 1000 top.def
```

The `write_def` command also supports writing out the current block to an encrypted DEF file.

The following example writes the current block to an encrypted DEF file:

```
fc_shell> write_def -encrypt design.def
```

The following example writes a DEF file that is encrypted and compressed:

```
fc_shell> write_def  -encrypt -compress gzip design.def
```

Feedback

The following example writes a DEF file that includes the floorplan data and excludes all other optional constructs and object types:

```
fc_shell> write_def -include {rows_tracks bounds \
     cells blockages} fp.def
```

The following example writes a DEF file that includes the selected object:

```
fc_shell> write_def -objects [get_site_rows unit_row_*]  row.def
```

The following example writes a DEF file that includes specific cell types with a specific physical status:

```
fc_shell> write_def \
   -cell_types {corner end_cap pad macro lib_cell} \
   -include_physical_status {placed} cell.def
```

The following example writes a DEF file that includes routed signal nets or clock nets only:

```
fc_shell> write_def -routed_nets -net_types {signal clock} net.def
```

The following example writes design information across all levels of the physical hierarchy to a DEF file:

```
fc_shell> write_def -traverse_physical_hierarchy hier.def
```

The following example specifies pairs of site names to match the floorplan's site name with the site name in the technology data:

```
fc_shell> write_def -convert_sites { {CORE unit} \
     {unitGA unitGA} } top.def
```

The following example writes a DEF file that includes the via definitions defined in the technology file:

```
fc_shell> write_def -include_tech_via_definitions top.def
```

Importing DEF Data

To read design information from DEF files into the current block or a specified block, use the `read_def` command:

```
fc_shell> read_def my_def.def
```

The `read_def` command also reads an encrypted DEF file that is created using the `write_def -encrypt` command.

If you use relative path names to specify the DEF files, the command locates the files based on the `search_path` variable.

Feedback

The following example reads new cells into the design:

```
fc_shell> read_def -add_def_only_objects cells top.def
```

The following example reads only cells, nondefault routing rules, and nets from the DEF file:

```
fc_shell> read_def -include {cells routing_rules nets} top.def
```

The following example reads a full-chip DEF file into a hierarchical design:

```
fc_shell> read_def -traverse_physical_hierarchy hier.def
```

The `read_def` command can also remove all existing physical annotations from the current block before reading the DEF files and specify the mapping of DEF site names. For details, see the `read_def` man page.

Application Options

The following table describes the application options you can use to import and export DEF data. The application option settings are global, with the exception of the blocked-scoped `file.def.gzip_effort` application option.

Table 10 Application Options in the DEF Interface

Application option	Type	Default	Description
`file.def.append_shape_and_terminal`	Boolean	`true`	Controls whether the `read_def` command appends net wiring shapes and terminals
`file.def.gzip_effort`	Enumerated	`gzip_fast`	Controls the compression effort when writing a compressed DEF file with the `write_def` command.
`file.def.fill_purpose_map`	list of strings	{}	Controls how the DEF fill shape use values are mapped to Fusion Compiler shape purposes
`file.def.maintain_via_ladders`	Boolean	`false`	Controls whether the `write_def` and `read_def` commands write and read additional information about via ladders

Feedback

Table 10 Application Options in the DEF Interface (Continued)

Application option	Type	Default	Description
`file.def.non_default_width_wiring_to_net`	Boolean	`false`	Controls whether the `write_def` command writes nets with a nondefault width
`file.def.pg_via_arrays`	Boolean	`true`	Controls whether the `write_def` command writes all regularly spaced vias on power and ground nets in compact via array format
`file.def.rule_based_name_matching`	Boolean	`true`	Controls whether the `read_def` command uses rule-based name matching.
`file.def.scan_cells_in_netlist_order`	Boolean	`false`	Controls whether the `write_def` command outputs floating scan cells in the netlist order
`file.def.set_pg_mask_fixed`	Boolean	`false`	Controls whether the `read_def` command sets all power and ground shape and via masks as fixed
`file.def.support_property_definitions`	Boolean	`false`	Controls whether the `read_def` and `write_def` commands preserve the DEF property definitions and values
`file.def.wrong_way_wiring_to_special_net`	Boolean	`false`	Controls whether the `write_def` command writes wiring with wrong-way default widths to the DEF SPECIALNETS section instead of the NETS section

Fusion Compiler Layer Mapping File

The `write_gds`, `read_gds`, `write_oasis`, and `read_oasis` commands convert physical design data between a stream format and the Fusion Compiler database format. By default, these commands preserve the layer numbers during the conversion process.

Feedback

To change the layer numbers or other properties during the conversion process,

- Create a *layer mapping file* to specify the corresponding properties of the layers in stream format and the Fusion Compiler database.
- Use the `-layer_map` option of the `write_gds`, `read_gds`, `write_oasis`, or `read_oasis` command to specify the name of the mapping file.

Each line in the layer mapping file specifies a layer in the technology file and corresponding layer in the GDSII or OASIS file, as shown in the following examples.

Table 11 Layer Mapping File Entry Examples

Line in layer mapping file	Mapping effects for GDSII/OASIS output and input
`56 10`	The `write_gds` or `write_oasis` command converts all geometries on layer 56 in the Fusion Compiler database to layer 10 in the stream file. The `read_gds/read_oasis` command converts all geometries on layer 10 in the stream file to layer 56 in the Fusion Compiler database.
`data 56:4:power 10:2`	The `write_gds` or `write_oasis` command converts all geometric (nontext) power-related data on layer 56, purpose 4, in the Fusion Compiler database to layer 10, data type 2 in the stream file. The `read_gds` or `read_oasis` command ignores the `data` keyword. It converts all data on layer 10, data type 2 in the stream file, to layer 56, purpose 4 in the Fusion Compiler database.

The following layer mapping file shows some more of the available mapping features.

Example 1 Layer Mapping File Example

```
; Disable new layer creation
read_always false
; Map layer 10 with data type 2 in stream file to power net shapes
; on layer 56 with a purpose of 4 in the Fusion Compiler database
data 56:4:power 10:2
; Map layers 31 and 32 with data type 0 in stream file to layer 31
; with a purpose of 0 in the Fusion Compiler database
data 31:0 31:0
data 31:0 32:0
```

For backward compatibility with the IC Compiler tool, you can use a layer mapping file written for the IC Compiler database format.

Feedback

Fusion Compiler Layer Mapping Syntax

Each line in the layer mapping file specifies a layer in the technology file and the corresponding layer in the GDSII or OASIS stream file using the following syntax:

```
[object_type] tf_layer[:tf_purpose][:use_type][:mask_type]
      stream_layer[:stream_data_type]
```

Each line can contain the following items:

- *object_type* – The types of objects mapped by this rule when writing the stream file

  Valid values are `data` (all nontext objects), `text`, and `all`. The default is `all`.

- *tf_layer* – The layer number in the technology file, a required item
- *tf_purpose* – The purpose number in the technology file

  Valid values are either an integer or the `drawing` keyword.

- *use_type* – The Fusion Compiler usage of the geometries in the design

  Valid values are `power`, `ground`, `signal`, `clock`, `boundary`, `hard_placement_blockage`, `soft_placement_blockage`, `routing_blockage`, `area_fill`, and `track`.

  For a multi-die design, the *use_type* argument supports usage of following manufacturing objects:

  - `sealring_boundary,sealring_name`

    `sealring_name` is the name of the `seal_ring` object.

  - `assembly_die_boundary`
  - `stitching_zone`

  For more information, see Working With Manufacturing Shapes.

- *mask_type* – The multiple-patterning mask constraint of the geometries

  Valid values for metal layers are `mask_one`, `mask_two`, `mask_three`, and `mask_same`. Via layers support the additional values `MASK_FOUR` through `MASK_FIFTEEN`.

- *stream_layer* – The layer number in the stream file, a required item
- *stream_data_type* – The layer data type number in the stream file

Any text in a line that comes after a semicolon character (`;`) is considered a comment and has no effect on layer mapping.

Feedback

Layer Mapping File Statements

The layer mapping file supports the following optional statements to control the behavior of the `write_gds`, `read_gds`, `write_oasis`, and `read_oasis` commands:

```
read_always true|false [-mapped_only] [-ignore_missing_layers]
blockage_as_zero_spacing true|false
cell_prop_attribute attribute_value
net_prop_attribute attribute_value
pin_prop_attribute attribute_value
```

These statements control the handling of extra or missing layer data, the interpretation of blockage layers, and the optional storage and retrieval of cell, net, and pin names in the GDSII or OASIS data file. Table 12 and Table 13 describe how these statements affect the behavior of the `write_gds`, `read_gds`, `write_oasis`, and `read_oasis` commands.

Table 12 Layer Mapping File Statements to Control the Reading of Layer Data

Statement	Description
`read_always true`	The `read_gds` or `read_oasis` command loads all layers defined in the external data file. For any layers not defined in the technology file, the command creates new layers. This is the default behavior.
`read_always false`	The `read_gds` or `read_oasis` command fails and issues an error message if the external data file contains layers not defined in the technology file.
`read_always false -mapped_only`	The `read_gds` or `read_oasis` command reads only those layers specified in the layer mapping file and ignores all other layers.
`read_always false -ignore_missing_layers`	The `read_gds` or `read_oasis` command reads only those layers defined in the technology file and ignores all other layers.
`blockage_as_zero_spacing false`	The `read_gds` or `read_oasis` command does not mark blockages read from the external data file as zero-spacing blockages, so the blockages follow normal spacing rules. This is the default behavior.
`blockage_as_zero_spacing true`	The `read_gds` or `read_oasis` command marks blockages read from the external data file as zero-spacing blockages.

Feedback

Table 13 Layer Mapping File Statements to Maintain Cell, Net, and Pin Names

Statement	Description
`cell_prop_attribute` *`attribute_value`*	The `write_gds` or `write_oasis` command uses the Fusion Compiler cell instance name to set the name associated with the specified cell attribute number in the stream file. Using this feature preserves the cell names when you read the stream data back into the Fusion Compiler database in the library manager tool.
`net_prop_attribute` *`attribute_value`*	The `write_gds` or `write_oasis` command writes out each net name as a property attribute of the specified value for each geometric shape in the stream file. Using this feature preserves the net names associated with shapes in the design when you read the stream data back into the Fusion Compiler database, reducing the need for connectivity tracing.
`pin_prop_attribute` *`attribute_value`*	The `write_gds` or `write_oasis` command writes out each pin name as a property attribute of the specified value for each pin in the stream file. It also writes out the pin name as text on the same layer and datatype as the pin geometry. Using this feature preserves the pin names when you read the stream data back into the Fusion Compiler database, reducing the need for connectivity tracing.

Guidelines for Writing and Reading GDSII and OASIS

The following usage guidelines apply to the layer mapping file and the usage of the `write_gds`, `read_gds`, `write_oasis`, and `read_oasis` commands.

- If conflicting mapping rules are specified in the layer mapping file, the last instance in the file is used and the earlier ones are ignored.
- You can use the asterisk character (`*`) to represent all layer numbers, all purpose numbers, all usage types, or all mask types.
- You can use either the layer mapping file or the write/read stream command options to specify the property attribute numbers used for maintaining the cell, net, and pin names in the stream file. If you use both methods, the layer mapping file definitions have priority.

  For example, consider the following lines in a layer mapping file:

```
; layer mapping file object property options
cell_prop_attribute 1 ; read/write cell names as property
                      ; attribute 1
net_prop_attribute 2  ; read/write net names as property
                      ; attribute 2
pin_prop_attribute 3  ; read/write pin names as property
                      ; attribute 3
```

These lines override the corresponding options in the write_gds, read_gds, write_oasis, and read_oasis commands, such as the following:

```
fc_shell> write_gds -layer_map my_layers.map \
  -instance_property 4 -net_property 5 -pin_property 6 \
  my_design.gds
```

- By default, the write_gds or write_oasis command uses the cut data type numbers defined in the technology file. For example,

```
Layer "VIA2"
  cutTblSize      = 3
  cutNameTbl      = (V1SM, V1BAR, V1LRG)     # cut names
  cutWidthTbl     = (0.05, 0.05, 0.10)       # cut width
  cutHeightTbl    = (0.05, 0.10, 0.10)       # cut height
  cutDataTypeTbl  = (5, 10, 15)              # cut data types
  ...
```

  In this example, the write_gds or write_oasis command maps the V1SM, V1BAR, and V1LRG cuts to data types 5, 10, and 15, respectively. If you do not want this mapping, use the -ignore_cut_datatype_tbl_mapping option with the write_gds or write_oasis command.

- By default, the write_gds or write_oasis command converts all layer data types to 0 for layers not listed in the layer mapping file, or when no layer mapping file is used. To retain the layer data type numbers in these situations, use the -keep_data_type option with the write_gds or write_oasis command.

- By default, the write_gds or write_oasis command writes out text objects in the R0 orientation. To modify the orientation of pin text according to the route access direction, do the following:

```
fc_shell> set_app_options \
   -name file.gds.rotate_pin_text_by_access_direction \
   -value true
fc_shell> set_app_options \
   -name file.oasis.rotate_pin_text_by_access_direction \
   -value true
```

  The read_gds and read_oasis commands can also read any angle cell that has a single reference in the GDS or OASIS format files.

- By default, the write_gds or write_oasis command writes out contact via names using the prefix string "$$". To specify a different prefix string, do the following:

```
fc_shell> set_app_options \
   -name file.gds.contact_prefix -value "MY_PREFIX"
fc_shell> set_app_options \
   -name file.oasis.contact_prefix -value "MY_PREFIX"
```

- Different fill cells with the same name can exist in different subblocks. By default, the `write_gds` and `write_oasis` commands append the name of the top-level block to fill cell names to avoid a name conflict.

  To prevent the `write_gds` command from appending the top-level block name to a fill cell, set the `file.gds.prefix_for_fill` application option to `false`:

  ```
  fc_shell> set_app_options -name file.gds.prefix_for_fill \
     -value false
  ```

- You can save a layer mapping file in a design library by using the `set_layer_map_file` command. In that case, the saved file is used as the default mapping file when you perform layout validation with the IC Validator tool or parasitic extraction with the StarRC tool. For details, see the man page for the command.

For information about other options for writing or reading stream files and using the layer mapping file, see the applicable man page.

IC Compiler Layer Mapping File Syntax

For backward compatibility with the IC Compiler tool, you can use a layer mapping file written for the IC Compiler database format. For example, for the `write_gds` command, specify the layer mapping file format in one of the following ways:

```
fc_shell> write_gds -layer_map map_file_name \
   -layer_map_format icc_default ...

fc_shell> write_gds -layer_map map_file_name \
   -layer_map_format icc_extended ...
```

The `icc_default` setting uses a layer mapping file written for a database library in default layer mode, which uses layer numbers 1 through 255. The `icc_extended` setting uses a layer mapping file written for a database library in extended layer mode, which uses layer numbers 1 through 4095.

There are two types of IC Compiler layer mapping files - Conventional layer mapping and double-patterning layer mapping.

Conventional Layer Mapping

In an IC Compiler conventional layer mapping file, each line shows a data object type, database layer, and resulting layer in the output stream. This is the general syntax:

```
MilkywayObjType[MilkywayNetType][PinCode]
     MilkywayLayer[:MilkywayDataType] StreamLayer[:StreamDataType]
```

The first character in the line is the code for the type of data object to be translated. Use A for all, T for text, or D for data. The optional second character specifies the net type, such as S for signal, P for power, or G for ground. An optional third character specifies the pin

code. The database layer is a name or an integer. The stream layers and data types are integers.

By default, net text and pin text are mapped into the same layer as the associated net or pin. However, you can specify a different layer for net text or pin text in the layer mapping file.

The following example demonstrates the stream-out layer mapping syntax.

```
T METAL:2 3:5    ; converts text on the database layer
                 ; METAL data Type 2 to GDSII Stream
                 ; layer #3 data type 5
                 ; Note: If you stream back into
                 ; the database without using
                 ; the layer map file, this will
                 ; switch your text on layer METAL
                 ; to layer METAL5

TS 16 31:6       ; converts text associated with
                 ; signal nets on database layer #16
                 ; to GDSII Stream layer #31 and
                 ; data type 6

TP METAL3 16     ; converts text associated with
                 ; power on database layer METAL3
                 ; to GDSII Stream layer #16

TG 28 16         ; converts text associated with
                 ; ground on database layer #28
                 ; to GDSII Stream layer #16

D METAL2 45      ; converts data on database layer
                 ; METAL2 to GDSII Stream layer #45

DS 45:2 18:5     ; converts data associated with
                 ; signal nets on database layer #45
                 ; data Type 2 to GDSII Stream layer #18
                 ; data Type 5

A METAL6 3:4     ; converts text and data on
                 ; database Layer METAL6 to GDSII
                 ; Stream layer #3 and datatype 4

A METAL4 -       ; minus sign (hyphen) prevents transfer of all
                 ; text and data on database layer METAL4
```

You can add a comment to the file by inserting a semicolon. Text is ignored from the semicolon to the end of the line. A hyphen character in the stream layer position prevents the transfer of all text and data in the specified database object and layer.

Feedback

A colon is the delimiter between the stream layer and the optional stream data type. For backward compatibility, you can use a space character alone or a colon followed by white space as the delimiter.

Table 14 describes the items used in each line of the layer mapping file.

Table 14 IC Compiler Conventional Layer Mapping File Syntax

Variable	Description
MilkywayObjType	The code for the type of object in the data to be translated: A for all types T for text D for data
MilkywayNetType	The code for the net type of the object in the data to be translated: S for signal P for power G for ground C for clock U for Up conduction layer (upper layer in a via). Example: metal2 in an m1/m2 via D for Down conduction layer (lower layer in a via). Example: metal1 in an m1/m2 via X for power and ground wires or contacts with a “signal” or “tie-off” routing type A for all net types Note that U and D are used only for via and flattened via objects. They override any other net type set for the layer when the layer is in a via. If `MilkywayNetType` is omitted, the tool maps all data of the specified object type to the specified stream layer. If `MilkywayNetType` is included, the tool examines every object of the object type to determine its net type and maps it to the layer you specify. If an object has no net or a net has no type, the tool assumes that it is a signal net. If a layer file contains contradictory lines, the last line overrides any previous line. For example, if an earlier line of a layer file specifies mapping for a particular object or net type and a later line specifies mapping for the same object type but no net type, the tool translates all data of the specified object type as defined by the later line.
PinCode	Optional. Can be used when only the Pin geometry or Terminal (top-level pin) geometry in the database needs to be translated: T for terminal geometries of the current designP for terminal geometries of the current design, as well as the physical pin geometries of the child cells, depending on the child depth option. Note: PinCode P and T only work when *MilkywayNetType* is set to A. Otherwise, the PinCode rule is ignored.

Feedback

Table 14 IC Compiler Conventional Layer Mapping File Syntax (Continued)

Variable	Description
MilkywayLayer	Specifies the name or number, from 1 to 255 (1 to 4095 in extended layer mode), of a layer to be translated and defined in the technology file.
MilkywayDataType	The data type of the data to be translated: an integer from 0 to 4095.
StreamLayer	Specifies the number of the stream layer to which the objects of `MilkywayObjType` on `MilkywayLayer` are translated. Use an integer from 0 to 32767 or use a hyphen character (-) to ignore the database object and layer during translation.
StreamDataType	Specifies the number of the stream data type to which objects of `MilkywayObjType` and `MilkywayNetType`, if given, on `MilkywayLayer` are translated. Use an integer from 0 to 32767.

The `write_gds` or `write_oasis` command writes geometries, such as blockages, into the stream file by using the reserved layer mapping in the layer mapping file according to the following table.

Table 15 Database Layer Numbers and Blockage Usage

Database layer number (default)	Database layer number (extended)	Blockage layer
212	4025	polyBlockage
216	4029	metal4Blockage
217	4030	via3Blockage
218	4031	metal1Blockage
219	4032	metal2Blockage
220	4033	metal3Blockage
223	4036	polyContBlockage
224	4037	via1Blockage
225	4038	via2Blockage

For example, to write metal1 blockages to the stream file, use `A 218 31:2` in the layer mapping file.

Feedback

Double-Patterning Layer Mapping

In an IC Compiler double-patterning layer mapping file, each line shows a database object type, database layer, and resulting layer in the output stream. This is the general syntax:

```
MilkywayMaskType
     MilkywayLayer[:MilkywayDataType] StreamLayer[:StreamDataType]
```

Table 16 IC Compiler Double-Patterning Layer Mapping File Syntax

Variable	Description

MilkywayMaskType	The code for the type and subtype of mask in the data to be translated:
	`M` for `same_mask`
	`M1` for `mask_one`
	`M2` for `mask_two`
	`M3` for `mask_three`
	The character `M`, by itself, performs mapping of the layer without changing the double-patterning subtype mask attribute. The codes `M1`, `M2`, and `M3` perform mapping for double-patterning subtypes 1, 2, and 3, respectively.

MilkywayLayer	Specifies the name or number, from 1 to 255 (1 to 4095 in extended layer mode), of a layer to be translated and defined in the technology file.
MilkywayDataType	The data type of the data to be translated: an integer from 0 to 4095.
StreamLayer	Specifies the number of the stream layer to which the objects of `MilkywayObjType` on `MilkywayLayer` are translated. Use an integer from 0 to 32767 or use a hyphen character (-) to ignore the database object and layer during translation.
StreamDataType	Specifies the number of the stream data type to which objects of `MilkywayObjType` and `MilkywayNetType`, if given, on `MilkywayLayer` are translated. Use an integer from 0 to 32767.

To write database layer 10 using the `mask_one` mask type to GDSII layer 200, datatype 120, for example, use the following line in the layer mapping file:

```
M1 10 200:120
```

In the double-patterning mapping syntax, you can use an asterisk wildcard character (*) to represent all database layers and all stream layers that have a specific datatype.

For example, to write all database layers that have the `same_mask` mask type, mapping each layer to the same GDSII layer number, datatype 108, use the following line in the layer mapping file:

```
M * *:108
```

Feedback

See Also

- IC Compiler System Layer Mapping
- Translating Between IC Compiler and Fusion Compiler Layer Mapping Formats

IC Compiler System Layer Mapping

The `write_gds` or `write_oasis` command interprets each reserved system layer mapping in the IC Compiler layer mapping file as two mappings – One for the system layer and the other for the normal layer.

Each line of the layer mapping file converts to two lines because the reserved system layer mapping contains two meanings. For example,

`A 255 255 0` - IC Compiler format layer mapping with the `icc_default` option

This IC Compiler format layer mapping converts to

`*:*:boundary 255:0` - System layer boundary mapping in the Fusion Compiler format

and

`255:*:* 255:0` - Normal layer mapping in the Fusion Compiler format

Table 17 lists the IC Compiler system layers.

Table 17 IC Compiler System Layer Numbers and Layer Names

Database layer number (default)	Database layer number (extended)	Layer name
188	4001	RegionBlockage
189	4002	IGRPath
190	4003	routeGuide
191	4004	placeGuide
192	4005	OverlapCheck
193	4006	placeSRConst
194	4007	polyContOvrSize
195	4008	via1OvrSize
196	4009	via2OvrSize
197	4010	via3OvrSize

Feedback

Table 17 IC Compiler System Layer Numbers and Layer Names (Continued)

Database layer number (default)	Database layer number (extended)	Layer name
198	4011	via4OvrSize
199	4012	MacroBlockage
200	4013	via11
201	4014	metal12
202	4015	polyOvrSize
203	4016	metal1OvrSize
204	4017	metal2OvrSize
205	4018	metal3OvrSize
206	4019	metal4OvrSize
207	4020	PRboundary
208	4021	metal7
209	4022	via7
210	4023	metal8
211	4024	via8
212	4025	poly
213	4026	Barrier
214	4027	metal9
215	4028	via9
216	4029	metal4
217	4030	via3
218	4031	metal1
219	4032	metal2
220	4033	metal3
221	4034	PlaceBlockage

Feedback

Table 17 IC Compiler System Layer Numbers and Layer Names (Continued)

Database layer number (default)	Database layer number (extended)	Layer name
222	4035	SoftPlaceBlk
223	4036	polyCont
224	4037	via1
225	4038	via2
226	4039	horChannel
227	4040	verChannel
228	4041	pinBlockage
229	4042	hardFence
230	4043	polyContRegion
231	4044	via1Region
232	4045	via2Region
233	4046	via3Region
234	4047	via4Region
235	4048	hilite
236	4049	metal10
237	4050	via10
238	4051	metal11
239	4052	metal5
240	4053	metal6
241	4054	via4
242	4055	via5
243	4056	via6
244	4057	chanGate
245	4058	pinGuide

Feedback

Table 17 IC Compiler System Layer Numbers and Layer Names (Continued)

Database layer number (default)	Database layer number (extended)	Layer name
246	4059	metal13
247	4060	metal14
248	4061	metal15
249	4062	via12
250	4063	via13
251	4064	via14
252	4065	coreRegion
253	4066	stdCellRow
254	4067	stdCellRegion
255	4095	boundary

See Also

- IC Compiler Layer Mapping File Syntax
- Translating Between IC Compiler and Fusion Compiler Layer Mapping Formats

Translating Between IC Compiler and Fusion Compiler Layer Mapping Formats

The IC Compiler and Fusion Compiler tools have different layer mapping file formats. You can translate a layer mapping file written for the IC Compiler or Fusion Compiler tool database format into one of the following formats:

- (`icc2`), which is used by the Fusion Compiler tool
- IC Compiler default layer mode (`icc_default`), which uses system layers 188-255
- IC Compiler extended layer mode (`icc_extended`), which uses system layers 4001-4095

In the Fusion Compiler format file, the database layer can be a valid layer number or an asterisk wildcard character (`*`).

Feedback

The system layers from the layer mapping file in the IC Compiler layer mapping formats (either default or extended layer) are translated into a technology layer per mask name or layer name. The mask name has priority over the layer name. If no corresponding technology layer matches the mask name or layer name, the Fusion Compiler layer number is turned into an asterisk wildcard character (`*`), which means that it is turned into any layer.

To convert a layer mapping file, use the `write_layer_map` command with the following syntax:

```
write_layer_map
   [-format format]
   [-original_format format]
   -original original_layer_map_file_name
   output_layer_map_file_name
```

In the `-format` option, specify the format of the output layer map file. This can be `icc2`, `icc_default`, or `icc_extended`.

In the `-original_format` option, specify the format of the original layer map file. This can be `icc2`, `icc_default`, or `icc_extended`.

In the `-original` option, specify the name of the layer mapping file to be translated to a new layer mapping file in the specified format.

For example, to convert the icc.map file, which is in the `icc_default` format, to the outIcc2.map file in `icc2` format, use the following command:

```
fc_shell> write_layer_map -original icc.map ./outIcc2.map
```

See Also

- IC Compiler Layer Mapping File Syntax
- IC Compiler System Layer Mapping
- Limitations When Translating Between Layer Mapping Formats

Feedback

Limitations When Translating Between Layer Mapping Formats

IC Compiler to Fusion Compiler Layer Mapping Formats

When translating `MilkywayNetType` from the IC Compiler layer mapping formats (either default or extended layer) to the Fusion Compiler layer mapping format, the `write_layer_map` command skips the following codes:

- `U` for Up conduction layer (upper layer in a via)
- `D` for Down conduction layer (lower layer in a via)
- `X` for power and ground wires or contacts with a signal or tie-off routing type

Fusion Compiler to IC Compiler Layer Mapping Formats

- When converting the use type from the Fusion Compiler layer mapping format to the IC Compiler layer mapping formats, the `write_layer_map` command skips the following use types:
  - `area_fill`
  - `track`
  - `bridge_shape`
  - `keepout_region`
  - `instance_specific_mask`
  - `same_net_feedthrough_tsv`
  - `lo_voltage`
  - `hi_voltage`
  - “`routing_blockage,*`”

    Note:

    Denotes the `routing_blockage` use type with a wildcard subtype
  - “`route_guide,*`”

    Note:

    Denotes the `route_guide` use type with a wildcard subtype

Feedback

- When converting the mask type from the Fusion Compiler layer mapping format to the IC Compiler layer mapping formats, the IC Compiler layer mapping formats support only those mask rules that have the following mask types:

  - `same_mask`
  - `mask_one`
  - `mask_two`
  - `mask_three`

  Mask rules that have other mask types are skipped.

- The `write_layer_map` command skips the following statements when translating from the Fusion Compiler layer mapping format to the IC Compiler layer mapping formats:

```
[read_always  { true | false [-mapped_only |
 -ignore_missing_layers ] } ]
[blockage_as_zero_spacing {true|false}]
[cell_prop_attribute attribute_number]
[net_prop_attribute attribute_number]
[pin_prop_attribute attribute_number]
[via_matrix_prop_attribute  attribute_number]
[via_prop_attribute  attribute_number]
```

See Also

- IC Compiler Layer Mapping File Syntax
- IC Compiler System Layer Mapping
- Translating Between IC Compiler and Fusion Compiler Layer Mapping Formats

Changing Object Names

The Fusion Compiler tool follows certain naming conventions for ports, cells, and nets. When you export a design to a new format such as Verilog, DEF, LEF, or GDSII, you might need to have the object names follow naming conventions of the external tool that are different from Fusion Compiler naming conventions.

Before you export a design, you can use the `change_names` command to change the names of cells, nets, and ports so that the new names follow the required naming conventions. The Fusion Compiler tool has a built-in capability to convert object names to conform to Verilog requirements. You can also create your own naming conversion rules.

Feedback

See Also

- Changing Object Names Using Custom Rules
- Changing Object Names for Verilog Output

Changing Object Names Using Custom Rules

Using the `define_name_rules` and `change_names` commands, you can change the names of ports, cells, and nets in the design to conform to specified naming conventions. The name-changing process can perform the following types of changes:

- Force the name of any net connected to a port to match the port name
- Force all port, cell, and net names to be unique across all objects
- Change all occurrences of a given string in object names to a new string
- Restrict the first character, last character, or all characters of object names to a given list of characters
- Prohibit the usage of a given list of characters as the first character, last character, or all characters of object names
- Prohibit the usage of a given list of words as object names
- Restrict the number of characters in each name to a specified maximum limit
- Specify a list of objects whose names must not be changed

  The objects specified for exclusion from the name change operation using the `-dont_touch` option of the `change_names` command are considered in the following order:

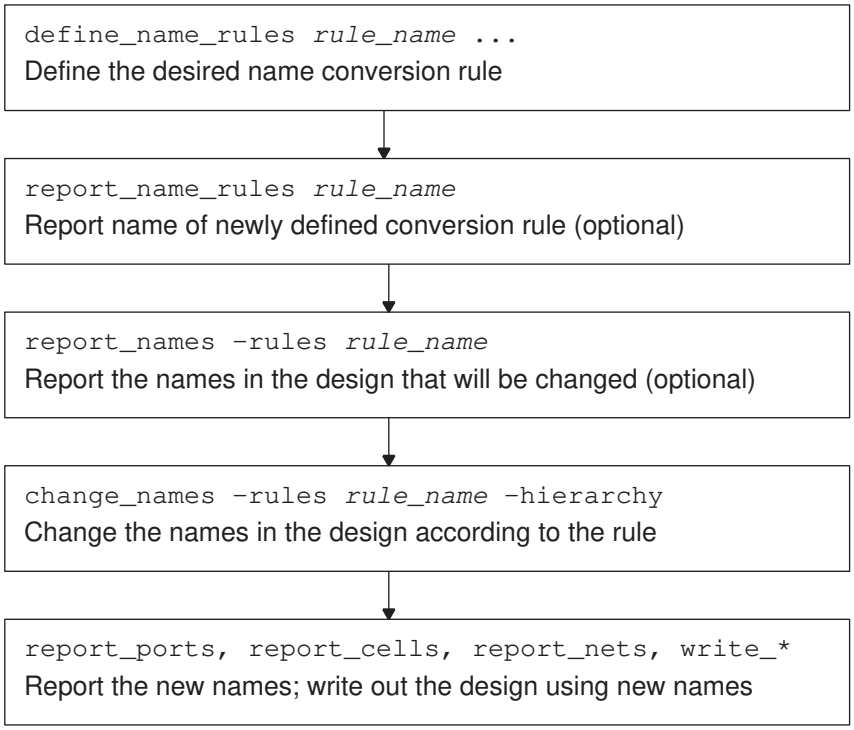
  1. Module
  2. Port Bus
  3. Port
  4. Cell Bus
  5. Cell
  6. Net Bus
  7. Net

Feedback

For example, if "a" is specified as the object to be excluded from the name change operation, the `-dont_touch` option first checks whether "a" is a module, then whether "a" is a port, and so on. During the check, if there is a net named "a" and port named "a", the port named "a" is excluded from the name change operation.

The following diagram shows the typical flow for changing the names of objects in the design.

Figure 18 Typical Flow for Changing Object Names



The following example changes each occurrence of the string “address” or “data” to “ADR” or “D”, respectively, in all port names.

1. Define the name conversion rule.

```
fc_shell> define_name_rules UCports -type port \
   -map {{"address","ADR"},{"data","D"}}
1
```

This defines a rule called “UCports” (uppercase ports), which maps the string “address” to “ADR” and the string “data” to “D” in port names.

2. (Optional) Report the rule just defined and verify its behavior.

```
fc_shell> report_name_rules UCports
```

Feedback

```
...
Rules Name: UCports
    ...

    Object Max  Repl Rem
    Type   Len  Char Chrs Prefix Allowed Chars
    ------------------------------------------------------------------
    Port   none '_'  no   P      Mapping String: "{"address","ADR"},
                                                 {"data","D"}"
    Cell   none '_'  no   U
    Net    none '_'  no   N
1
```

3. (Optional) Report the names that would be changed by applying the rule, without actually making the changes, and verify that the proposed changes are correct.

```
fc_shell> report_names -rules UCports
...
        Design          Type     Object           New Name
        ------------------------------------------------------------
        my_design_mp    port     address[27]      ADR[27]
        my_design_mp    port     address[26]      ADR[26]
        my_design_mp    port     address[25]      ADR[25]
        ...
        my_design_mp    port     data[31]         D[31]
        my_design_mp    port     data[30]         D[30]
        my_design_mp    port     data[29]         D[29]
        ...
```

4. Change the names in the design according to the rule.

```
fc_shell> change_names -rules UCports -hierarchy
 Information: Using name rules 'UCports'.
 Information: 62 objects (60 ports, 2 bus ports, 0 cells &
    0 nets) changed in design 'my_design_mp'.
```

Note:

To change the name of a cell bus, set the `design.change_names.include_cell_bus` application option to `true` before running the `change_name` command.

To apply the name changes across multiple levels of the physical hierarchy, use the `-include_sub_blocks` option with the `-hierarchy` option:

```
fc_shell> change_names -rules UCports -hierarchy \
   -include_sub_blocks
...
```

To specify a list of objects on which the name change must not be applied, use the `-dont_touch` option:

```
fc_shell> change_names -rules rule1 -dont_touch [get_cell A]
```

The tool determines the physical hierarchy by using the bound view, which must be the design view. If the bound view is not a design view, the `change_names` command issues an error message. When the `change_names` command modifies names in a physical subblock, it updates the bound view; you must re-create any other views of the physical subblock, such as the frame and abstract views.

5. (Optional) Report the object names and verify that they have been changed correctly.

```
fc_shell> report_ports
...
                        Pin Cap                    Wire Cap
                     Min          Max          Min          Max
Port       Dir    rise/fall    rise/fall    rise/fall    rise/fall
-------------------------------------------------------------------
ADR[0]     out    0.00/0.00    0.00/0.00    0.00/0.00    0.00/0.00
ADR[10]    out    0.00/0.00    0.00/0.00    0.00/0.00    0.00/0.00
ADR[11]    out    0.00/0.00    0.00/0.00    0.00/0.00    0.00/0.00
...
D[0]       inout  0.00/0.00    0.00/0.00    0.00/0.00    0.00/0.00
D[10]      inout  0.00/0.00    0.00/0.00    0.00/0.00    0.00/0.00
D[11]      inout  0.00/0.00    0.00/0.00    0.00/0.00    0.00/0.00
...
```

When you save or export the design, the design data reflects the new names.

To report the names of ports, cells, and nets across all levels of the physical hierarchy, use the `-include_sub_blocks` option when you run the `report_names -hierarchy` command.

For information about the various rules you can use to change port, cell, and net names, see the man page for the `define_name_rules` command.

See Also

- Changing Object Names for Verilog Output

Feedback

Changing Object Names for Verilog Output

Before you export a design in Verilog format by using the `write_verilog` command, use the `change_names` command to change all the object names to conform to Verilog conventions:

```
fc_shell> current_block
{my_lib:my_design/done.design}

fc_shell> change_names -rules verilog -hierarchy
 Information: Using name rules 'verilog'.
 Information: 321 objects (0 ports, 0 bus ports, 0 cells & 321 nets)
   changed in design 'my_design'.
1
fc_shell> write_verilog my_design.v
1
```

To specify a list of modules, ports, cells, nets, port buses, cell buses, and net buses on which the name change must not be applied, use the `-dont_touch` option:

```
fc_shell> change_names -rules verilog -dont_touch \
 [list  [get_nets A] [get_ports B]]
```

To see the details of the Verilog naming rules, use the `report_name_rules` command:

```
fc_shell> report_name_rules verilog
...
Rules Name: verilog
    Collapse name space: true
    Equal port and net names: true
    Equal inout port and net names: true
    Check internal net name: true
    Target bus naming style: %s[%d]
    Remove irregular port bus: true
    Check bus indexing: true
    Check bus indexing use type info: false
    Remove port bus: false
    Case insensitive: false
    Add dummy nets: false
    Reserved words: not defined
...
```

For a full explanation of each type of name change, see the man page for the `define_name_rules` command.

See Also

- Changing Object Names Using Custom Rules

3

Working With Design Data

When you work with a design in the Fusion Compiler tool, you open and edit the design view of a block stored in a design library.

You can query and edit the design data as described in the following topics:

Application Options

You set the tool's *application options* by using the `set_app_options` command. These options control various aspects of tool behavior. For example, to set the maximum allowed coarse placement density for the current block:

```
fc_shell> set_app_options -name place.coarse.max_density /
   -value 0.6
place.coarse.max_density 0.6
```

Feedback

Application options use the following naming convention:

```
category[.subcategory].option_name
```

where *category* (such as `place`) is the tool feature affected by the option and *subcategory* (such as `coarse`) is a refinement of the feature affected by the option.

To get a list of all available application options, use the `get_app_options` command:

```
fc_shell> get_app_options
shell.common.tmp_dir_path shell.common.enable_line_editing
shell.common.line_editing_mode shell.common.collection_result_...
```

To restrict the list of reported application options, provide a pattern string:

```
fc_shell> get_app_options place.*
place.coarse.channel_detect_mode
place.coarse.congestion_analysis_effort
...
fc_shell> get_app_options *lib*
design.edit_read_only_libs link.user_units_from_first_library ...
...
```

Each application option applies either globally or to a specific block:

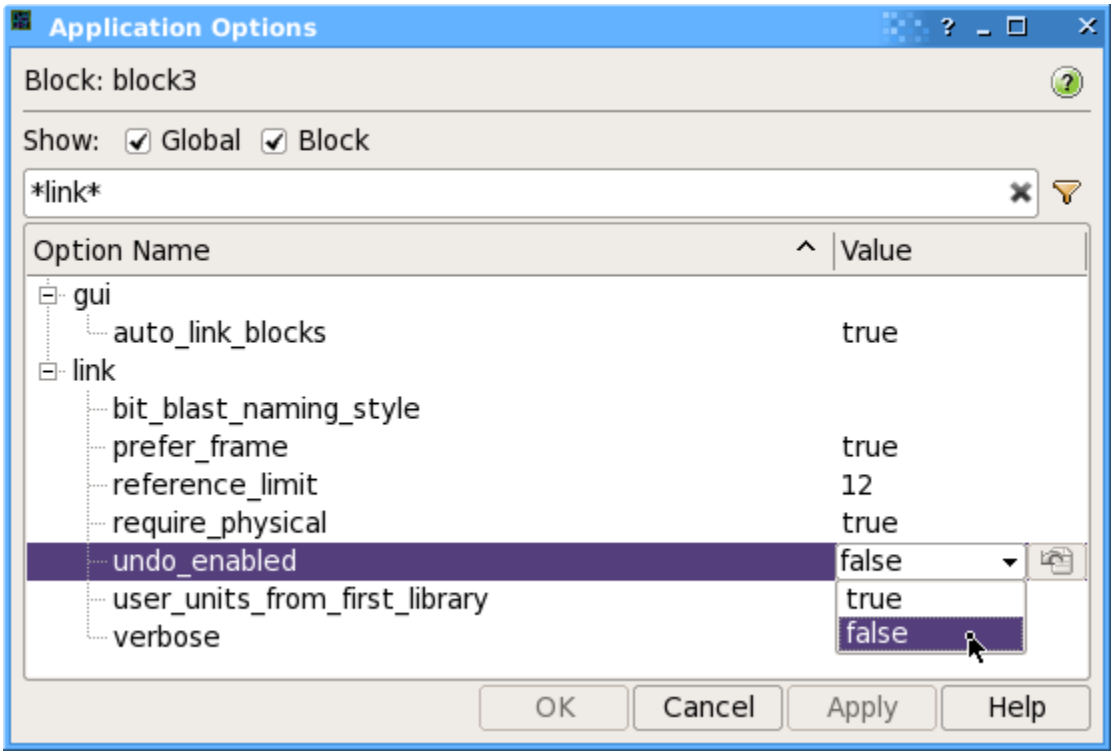
- Global application options apply everywhere within the current session.
- Block-level application options apply only to the block on which they are set. Before you can set a block's application options, the block must be open.

You can determine whether an application option is global or block-level by using the `report_app_options` command.

To use the GUI to view, search, and edit application options, choose File > Application Options. This opens the Application Options dialog box, shown in Figure 19.

Feedback

Figure 19 Application Options Dialog Box



In the Application Options dialog box, to view the man page for the highlighted application option, click the question mark icon in the upper-right corner.

For more information about application options, see

- Setting Application Options
- Querying Application Options
- Saving Application Options
- Comparing Application Option Settings Between Two Tool Sessions
- Checking Application Options in Your Scripts
- Help on Application Options
- Man Pages for Application Options
- User Default for Application Options
- Resetting Application Options

Feedback

Setting Application Options

To set an application option, use the `set_app_options` command:

```
fc_shell> set_app_options -name link.verbose -value true
link.verbose true
```

To set a block-level application option on a block other than the current block:

```
fc_shell> set_app_options -block des_A -name link.verbose \
   -value true
link.verbose true
```

To set multiple application options in a single command:

```
fc_shell> set_app_options -list \
   {link.verbose true link.reference_limit 8}
link.reference_limit 8 link.verbose true
```

Alternatively, if the options belong to the same category, you can do this:

```
fc_shell> set_app_options -category link \
   -list {verbose true reference_limit 8}
link.reference_limit 8 link.verbose true
```

See Also

- Application Options
- Querying Application Options
- Saving Application Options
- User Default for Application Options
- Resetting Application Options

Querying Application Options

To get an application option setting, use the `get_app_option_value` command:

```
fc_shell> get_app_option_value -name link.verbose
true
```

To get a block-level application option setting for a block other than the current block, use the `-block` option.

To get the default setting for an application option, use the `-system_default` option:

```
fc_shell> get_app_option_value -name link.verbose -system_default
false
```

To get all the attributes of an application option, use the `-details` option:

```
fc_shell> get_app_option_value -name link.verbose -details
name link.verbose value_type bool default_value false help
{Verbose messages while linking} status basic minimum_value {}
maximum_value {} allowed_values {}
is_obsolete false is_global_only false
is_read_only false is_persistent true
is_subscripted false is_design_cell_scoped true
is_lib_cell_scoped false
```

To report multiple application option settings, only the global options, or only the user-set options, use the `report_app_options` command:

```
fc_shell> report_app_options link*
...
Name                          Type           Value  ... default  Scope
----------------------------- -------------- ------ ----------  --------
link.bit_blast_naming_style   list_of string {}     ...   ...    block
link.prefer_frame             bool           --     ...   ...    global
link.undo_enabled             bool           --     ...   ...    global
...

fc_shell> report_app_options link* -global
...
Name                    Type    Value  ...  System-default
----------------------- ------- ------ ...  ------------
link.prefer_frame       bool    --     ...  true
link.undo_enabled       bool    --     ...  true
...

# Report user-set options
fc_shell> report_app_options -non_default
...
Name                       Type   Value    ...  System-default  Scope
-------------------------- ------ -------  ---  --------------- -
place.coarse.max_density   float  0.70     ...  0               block
-------------------------- ------ -------  ---  --------------- -
1

# Report all options
fc_shell> report_app_options
...
```

If an application option was set by using a script, the `report_app_options` command reports the script name and line number used to create the application option setting as follows:

```
fc_shell> report_app_options -non_default
...

Name                          Type    Value ... Source
                                            ...
--------------------------- ------- ----- --- -------------------------
abstract.high_fanout_limit integer 603   ... /u/scripts/abstract.tcl:1
abstract.latch_levels      integer 6     ... /u/scripts/abstract.tcl:2
...
```

See Also

- Application Options
- Setting Application Options
- Saving Application Options
- Help on Application Options
- Man Pages for Application Options
- User Default for Application Options
- Resetting Application Options

Saving Application Options

Each application option applies either globally or to a specific block. To save application option settings, use the following guidelines:

- Global application options apply everywhere within the current session. They are not stored and are not carried over from one session to the next. To retain these option settings, you can set them in your .synopsys_fc.setup file.
- Block-level application options apply only to the block on which they are set. They are saved with the block in the design library and are persistent across tool sessions except when they are set by the `set_app_options -as_user_default` command. In

Feedback

this case, they are treated like global application options and are not persistent across tool sessions.

- To determine whether an application option was set using the `-as_user_default` option, run the `report_app_options` command and check the value in the user-default column.
- Use the `write_script` command to save the current application option settings in a script file.

See Also

- Application Options
- Setting Application Options
- Querying Application Options
- Help on Application Options
- Man Pages for Application Options
- User Default for Application Options
- Resetting Application Options

Comparing Application Option Settings Between Two Tool Sessions

You can write out and compare application option settings between two different tool sessions or two different designs with the `write_app_options` and `compare_app_options` commands. The `write_app_options` command writes out a binary data file that contains all application option settings for the current tool session. The `compare_app_options` command reads in two binary data files written by the `write_app_options` command and writes out a detailed report of any application option settings which differ between two data files.

To write out and compare application option settings between two tool sessions,

1. Write out application option settings for the baseline design or tool session with the `write_app_options` command.

   ```
   fc_shell> write_app_options -output session1.data
   ```

2. In another tool session or after loading a different design, write out application option settings for the comparison design or tool session with the `write_app_options` command.

   ```
   fc_shell> write_app_options -output session2.data
   ```

Feedback

Be sure to specify a different filename to avoid overwriting the original baseline data.

3. Compare two application option data files and write out a detailed comparison report with the `compare_app_options` command.

```
fc_shell> compare_app_options -previous_data session1.data \
   -current_data session2.data
```

The `compare_app_options` command reads in data from the two binary data files specified with the `-previous_data` and `-current_data` options, and writes out a single report. The report contains changes to application variables and a section for each change of the application option settings for the following values:

- Default value
- Global value
- Status
- Block value
- Read-only status
- Deprecated status
- Persistent status
- Disabled status

The report's header contains the total number of application options in each file and a summary count of the differences. If a difference exists between the files, the report lists the application option name, the setting in the previous (baseline) data file and the setting in the current data file. The report only shows application options with values that are different between the two data files.

The following example shows differences in settings for block-scoped application options between the session1.data and session2.data files. Note that the `abstract.high_fanout_limit` and two other application options were not set explicitly in the tool when the session1.data file was created; these application options retained their original default value and have no data value displayed in the table. To write the same information to a file as comma-separated value data, use the `-csv file_name` option of the `compare_app_options` command. The `file_name` argument can be a local file name or a full file path.

```
fc_shell> compare_app_options -previous_data session1.data \
   -current_data session2.data
...
Changes in: Block Value                  Status Previous Current
------------------------------------ ------ -------- -------
abstract.high_fanout_limit               basic           603
```

Feedback

```
abstract.latch_levels                    basic           6
extract.high_fanout_threshold            basic           1003
link.reference_limit                     basic  12       15
mv.upf.max_supply_count_without_driver   basic  7        10
opt.common.max_fanout                    basic  40       43
...
```

See Also

- Application Options
- Setting Application Options
- Querying Application Options
- Saving Application Options
- Help on Application Options
- Man Pages for Application Options

Checking Application Options in Your Scripts

You can check your scripts for any outdated or changed application options with the `audit_scripts` command. The command accepts multiple input scripts, checks the scripts for any deprecated, removed, or otherwise changed application options, and reports the results in an annotated version of each script. The annotated scripts are written to a subdirectory named ./audit and contain the same name as the original file.

The `audit_scripts` command evaluates application options that have been specified with the following commands only:

- `set_app_options`
- `reset_app_options`
- `get_app_options`
- `get_app_option_value`
- `report_app_options`
- `help_app_options`

Checking a Script

To check a single script, specify a Tcl file using the `-input` option:

```
fc_shell> audit_scripts -input myscript.tcl
```

To check multiple scripts, specify a directory instead. When you specify a directory, the command performs checks on all Tcl files in the directory.

```
fc_shell> audit_scripts -input scripts
```

Using the `-from` option, you can compare your application options against a specified tool version. The following example checks the application options in myscript.tcl against those in theO-2018.06-SP5 version of the tool:

```
fc_shell> audit_scripts -input myscript.tcl -from 18.06-sp5
```

Note that only some tool versions are available for comparison, and that these versions might change across releases (see the `audit_scripts` man page for information about querying the tool versions available for comparison). By default, if the `-from` option is not specified, the command compares your script against the most current tool version available.

Interpreting the Annotated Results

For each audited script, the `audit_scripts` command writes out an annotated version of that script (with the same name as the input file) to a subdirectory named ./audit. The annotated scripts contain AUDIT_INFO and AUDIT_ERROR comments describing the audit results. These comments are also written to the command line output as information messages.

A portion of an annotated script might look like this:

```
# AUDIT_INFO at line 58: application option
# 'time.awp_compatibility_mode'
# is deprecated.

# AUDIT_ERROR at line 106: application option 'cts.verbose.buf' is
# removed.

# AUDIT_ERROR at line 156: system-default of application option
# 'cts.use_global_route_atree_topology' is changed from true to
# atree.

# AUDIT_ERROR at line 156: data type of application option
# 'cts.use_global_route_atree_topology' is changed from bool to
# enum.
```

AUDIT_INFO comments identify application options

- That have been deprecated
- That have become hidden or unhidden
- Whose scope has changed (global vs block-scoped)
- That can replace application variables

Feedback

AUDIT_ERROR comments identify application options

- That have been removed
- Whose data type has changed
- Whose value, user-default value, or system-default value has changed (via an internal or user script)

Repairing Your Script

To repair any changed application options in your script, you can either

- Manually repair the application options in your original script
- Manually repair the application options in the annotated script, and then copy the annotated script to the location of your original script

Help on Application Options

You can use the `help_app_options` command to get help on using application options:

```
fc_shell> help_app_options
Categories of the registered application options are:
  abstract
  clock_opt.congestion
  clock_opt.flow
  clock_opt.hold
  ...
  link
  ...

fc_shell> help_app_options -category link
link.bit_blast_naming_style  # naming style for bit blasted ...
link.prefer_frame            # Prefer blocks with FRAME view ...
link.reference_limit         # Maximum number of unresolved ...
link.require_physical        # Require sub-blocks to have ...
link.undo_enabled            # Allow link_design command to be ...
link.user_units_from_first_library # Set user units to units of ...
link.verbose                 # Verbose messages while linking

fc_shell> help_app_options -category link -scope block
link.bit_blast_naming_style  # naming style for bit blasted ...
link.reference_limit         # Maximum number of unresolved ...
link.verbose                 # Verbose messages while linking

fc_shell> help_app_options link.undo_enabled
link.undo_enabled            # Allow link_block command to be
                             # undoable

fc_shell> help_app_options link.undo_enabled -verbose
```

Feedback

```
name                            link.undo_enabled
value_type                      bool
default_value                   false
help                            Allow link_block command to be
                                undoable
...                             ...
is_read_only                    false
is_persistent                   false
is_subscripted                  false
is_design_cell_scoped           false
is_lib_cell_scoped              false
```

See Also

- Application Options
- Setting Application Options
- Querying Application Options
- Saving Application Options
- Man Pages for Application Options

Man Pages for Application Options

To view the man page for a specific application option, use the `man` command:

```
fc_shell> man link.reference_limit
...

NAME
       link.reference_limit
         Specifies the maximum number of detailed unresolved
         reference errors display for each block when linking ...
TYPE
        integer

DEFAULT
        12

DESCRIPTION
        In many cases, many link errors have the same root cause ...
        ...
SEE ALSO
        link_block(2)
        get_app_option_value(2)
        ...
```

To view the man page for all application options under a top-level category, use the `man` command to view the man page for *`category`*`_options`:

```
fc_shell> man link_options
   ...
DESCRIPTION
       ...
link.bit_blast_naming_style
    ...
link.prefer_frame
    ...
link.reference_limit
    ...
...
```

See Also

- Application Options
- Setting Application Options
- Querying Application Options
- Saving Application Options
- Help on Application Options

User Default for Application Options

You can optionally set a *user default* for an application option. The user default has higher priority than the *system default* but lower priority than an explicit setting for the option. The user default applies only to the current session and is not saved.

For example, the system default for the `place.coarse.max_density` application option is 0.0, so that value applies to all blocks by default. To set the user default to a different value, use the `set_app_options` command with the `-as_user_default` option:

```
fc_shell> set_app_options -as_user_default \
   -name place.coarse.max_density -value 0.65
place.coarse.max_density 0.65

fc_shell> report_app_options place.coarse.max_density
...
Name                      Type   Value  User-default  System-default  ...
------------------------- ------ ------ ------------- --------------- ---
place.coarse.max_density  float  --     0.65          0               ...
------------------------- ------ ------ ------------- --------------- ---
```

The new default, 0.65, applies to all blocks in the current session that do not have an explicit value set.

Feedback

You can override the user default for a particular block by setting the `place.coarse.max_density` application option to the desired value for the block:

```
fc_shell> current_block
{lib_A:block3.design}
fc_shell> set_app_options -name place.coarse.max_density \
   -value 0.70
place.coarse.max_density 0.7

fc_shell> report_app_options place.coarse.max_density
...
Name                     Type   Value  User-default  System-default  ...
------------------------ ------ ------ ------------- --------------- ---
place.coarse.max_density float  0.70   0.65          0               ...
------------------------ ------ ------ ------------- --------------- ---
```

The order of priority is “Value” first, then “User-default,” and “System-default” last.

See Also

- Application Options
- Setting Application Options
- Resetting Application Options

Resetting Application Options

To reset an application option so that it has no explicitly assigned value, use the `reset_app_options` command:

```
fc_shell> reset_app_options place.coarse.max_density
1
fc_shell> report_app_options place.coarse.max_density
...
Name                     Type   Value  User-default  System-default  ...
------------------------ ------ ------ ------------- --------------- ---
place.coarse.max_density float  --     0.65          0               ...
------------------------ ------ ------ ------------- --------------- ---
```

The `reset_app_options` command removes the “Value” setting and allows the “User-default” setting to apply (if any), or the “System-default” to apply if there is no user default.

To reset a user default setting, use the `reset_app_options` command with the `-user_default` option:

```
fc_shell> reset_app_options place.coarse.max_density \
   -user_default
1
fc_shell> report_app_options place.coarse.max_density
```

Feedback

```
...
Name                       Type   Value  User-default  System-default  ...
------------------------ ------ ------ ------------- --------------- ---
place.coarse.max_density float  --     --            0               ...
------------------------ ------ ------ ------------- --------------- ---
```

See Also

- Application Options
- Setting Application Options
- User Default for Application Options

Working With Objects

The Fusion Compiler infrastructure supports a full set of design-related objects and Tcl commands to create, query, modify, and remove these objects, including the following types:

- Logical objects such as cells, ports, and nets
- Timing data objects such as modes and corners
- Physical objects such as shapes, vias, tracks, and blockages
- Cell library objects such as libraries, library cells, and library pins

To get the full list of object types, use the following command:

```
fc_shell> get_defined_attributes -return_classes
block bound bound_shape budget_clock budget_path_type ...
bundle_segment bundle cell clock clock_balance_group clock_group ...
track utilization_config via via_def via_region via_rule ...
...
```

The tool supports commands to create, get, and remove specific types of objects, such as:

```
create_block    get_blocks     remove_blocks
create_net      get_nets       remove_nets
create_tech     get_techs      remove_tech
```

You can also operate on objects of mixed types in a collection by using commands such as the following:

- `move_objects` – move objects by a specified amount or to a specified location
- `flip_objects` – flip objects around a specified axis

Feedback

- `rotate_objects` – rotate objects by a specified angle or to a specified orientation
- `remove_objects` – remove objects from the design

Some types of objects have special-purpose commands that apply only to specific object types, such as `report_lib`, `add_to_bound`, `set_lib_cell_purpose`, and `connect_net`.

You can perform tasks such as resizing a cell. For example, to return a collection of equivalent library cells for a specific cell or library cell, use the `get_alternative_lib_cells` command. You can then use the collection to replace or resize the cell. The `size_cell` command allows you to change the drive strength of a leaf cell by linking it to a new library cell that has the required properties.

For a design with multiply instantiated modules, edit the module by using the `edit_module` command, as shown:

```
fc_shell> edit_module [get_module add_block] { \
   set_reference [get_cells U25] -to_block AND2 -pin_rebind none \
   set_reference [get_cells U61] -to_block XOR2 -pin_rebind none}
```

See Also

- Querying Common Design Objects
- Object Attributes
- Querying Objects
- Removing Objects
- Working With Collections

Querying Common Design Objects

Table 18 lists some common design objects and the commands to query them. These objects are provided as examples; there are many more objects not shown in the table. For a complete list of object types, use the `get_defined_attributes -return_classes` command.

Table 18 Commands to Query Some Common Design Objects

Object	Command	Description
Blockages	`get_pin_blockages`	Returns a collection of pin blockages.
	`report_pin_blockages`	Displays information about the pin blockages.
	`get_placement_blockages`	Returns a collection of placement blockages.
	`get_routing_blockages`	Returns a collection of routing blockages.

Feedback

Table 18 Commands to Query Some Common Design Objects (Continued)

Object	Command	Description
Bounds	`get_bounds`	Returns a collection of placement bounds.
	`get_bound_shapes`	Returns a collection of shapes associated with placement bounds.
	`report_bounds`	Displays information about the placement bounds.
Cells	`get_cells`	Returns a collection of cell instances. By default, the tool uses the logic hierarchy to resolve cell names. To use the physical hierarchy to resolve cell names, use the `-physical_context` option.
	`report_cells`	Displays information about the cell instances.
Clocks	`get_clocks`	Returns a collection of clocks.
	`get_clock_groups`	Returns a collection of clock groups.
	`get_generated_clocks`	Returns a collection of generated clocks.
	`report_clocks`	Reports information about clocks.
Core area	`get_core_area`	Returns a collection that contains the core area.
Guides	`get_io_guides`	Returns a collection of I/O guides.
	`report_io_guides`	Displays information about the I/O guides.
	`get_pin_guides`	Returns a collection of pin guides.
	`report_pin_guides`	Displays information about the pin guides.
I/O rings	`get_io_rings`	Returns a collection of I/O rings.
	`report_io_rings`	Displays information about I/O rings.
Library cell	`get_lib_cells`	Creates a collection of library cells from the reference libraries loaded in memory.
	`report_lib_cells`	Reports information about library cells.
	`get_alternative_lib_cells`	Returns a collection of equivalent library cells for the specified cell or library cell.
Modules	`get_modules`	Returns a collection of logic hierarchies in memory, as defined in the Verilog netlist.

Feedback

Table 18 Commands to Query Some Common Design Objects (Continued)

Object	Command	Description
Nets	`get_nets`	Returns a collection of logical nets. To return a collection of physical nets, use the `-physical_context` option.
	`get_shapes`	Returns a collection of shapes associated with nets.
	`report_nets`	Displays information about the nets.
Net buses	`get_net_buses`	Returns a collection of net buses from the current block.
	`report_net_buses`	Displays net bus information within the current block.
Pins	`get_pins`	Returns a collection of logical pins.To return a collection of physical pins, use the `-physical_context` option.
Placement attractions	`get_placement_attractions`	Returns a collection of placement attraction objects in the block.
	`report_placement_attractions`	Reports placement attraction objects in the block.
Ports	`get_ports`	Returns a collection of logical ports. To return a collection of physical ports, use the `-physical_context` option.
	`report_ports`	Displays information about the ports.
Port buses	`get_port_buses`	Returns a collection of port buses from the current block.
	`report_port_buses`	Displays port bus information within the current block.
Power domains	`get_power_domains`	Returns a collection of power domains.
	`get_domain_elements`	Returns a collection of elements associated with power domains.
	`report_power_domains`	Displays information about the power domains.
Power regions	`get_pg_regions`	Returns a collection of power and ground (PG) regions.
	`report_pg_regions`	Displays information about the PG regions.

Feedback

Table 18 Commands to Query Some Common Design Objects (Continued)

Object	Command	Description
Site rows	`get_site_rows`	Returns a collection of site rows.
Supply nets	`get_supply_nets`	Returns a collection of supply nets.
	`get_related_supply_nets`	Returns a collection of related supply nets.
	`report_supply_nets`	Displays information about the supply nets.
Supply ports	`get_supply_ports`	Returns a collection of supply ports.
	`report_supply_ports`	Displays information about the supply ports.
Terminals	`get_terminals`	Returns a collection of terminals.
Tracks	`get_tracks`	Returns a collection of tracks.
	`report_tracks`	Reports the routing tracks for a specified layer or for all layers.
Vias	`get_vias`	Returns a collection of vias.
Voltage areas	`get_voltage _areas`	Returns a collection of voltage areas.
	`get_voltage_area_shapes`	Returns a collection of shapes associated with voltage areas.
	`report_voltage_areas`	Displays information about the voltage areas.

Editing the Netlist

Table 19 lists some common design objects and the commands to edit them. These objects are provided as examples; there are many more objects not shown in the table. For a complete list of object types, use the `get_defined_attributes -return_classes` command.

Table 19 Commands to Work With Some Common Design Objects

Object	Command	Description
Blockages	`create_pin_blockage`	Creates a pin blockage in the current design.
	`remove_pin_blockages`	Removes pin blockages from the current design.
Cells	`create_cell`	Creates a new cell.

Feedback

Table 19 Commands to Work With Some Common Design Objects (Continued)

Object	Command	Description
	`remove_cells`	Removes specific cell instances
	`reparent_cells`	Moves cells from their current location in the logical hiearchy to a new location.
	`size_cell`	Rebinds leaf cells to a new library cell.
	`set_reference`	Changes the reference of an existing cell.
Clocks	`create_clock`	Creates a clock object.
	`remove_clocks`	Removes one or more clocks from the current design.
	`set_clock_groups`	Specifies clock groups that are mutually exclusive or asynchronous with each other.
	`remove_clock_groups`	Removes specific exclusive or asynchronous clock groups from the current design.
Nets	`create_net`	Creates a new net.
	`connect_net`	Connects a net to a pin or port.
	`disconnect_net`	Disconnects a net from a pin or port.
	`remove_nets`	Removes an existing net.
Placement attractions	`create_placement_attraction`	Specifies how cells should be placed relative to each other and relative to the floorplan.
	`remove_placement_attraction`	Removes placement attraction constraints from the block.
	`remove_from_placement_attraction`	Removes cells from an existing placement attraction collection.
	`add_to_placement_attraction`	Assigns cells to an existing placement attraction collection.
Ports	`create_port`	Creates a new port.
	`remove_ports`	Removes ports from the current design.

In some cases, when running hundreds or thousands of ECO commands together, you can reduce the runtime significantly by using the `edit_block` command, which allows a list of ECO commands to run within a single command. This can help to reduce runtime

Feedback

because the tool performs the data computation needed to keep the database consistent only once.

In the following example, the `edit_block` command is used to create a cell and its nets:

```
fc_shell> edit_block {
   create_net {n1 n2}
   create_cell ibuf [get_lib_cells lib1/BUF3]
   connect_net -net n2 ibuf/i
   connect_net -net n1 ibuf/z
}
```

In cases where you source a script containing hundreds or thousands of ECO commands, nest the `source` command within the `edit_block` command.

```
fc_shell> edit_block {
   source -echo edit_commands.tcl
}
```

Object Attributes

Each object type has a list of predefined (also known as application-defined) attributes. You can query these attributes to get information about the design. In some cases, you can modify the attribute settings to control the design implementation.

To list the attributes associated with an object type, use the `list_attributes` command. Specify the `-application` option to report the predefined object attributes. By default, the `list_attributes` command reports only user-defined attributes. The following example lists the predefined attributes for nets:

```
fc_shell> list_attributes -application -class net
...
Properties:
    A - Application-defined
    U - User-defined
    I - Importable from design/library (for user-defined)
    S - Settable
    B - Subscripted

Attribute Name      Object  Type     Properties Constraints
------------------------------------------------------------
antenna_rule_name   net     string   A,S
base_name           net     string   A
bbox                net     rect     A
dont_touch          net     boolean  A,S
full_name           net     string   A
...
```

Feedback

You can also report the attributes associated with an object type by using the `get_defined_attributes` command:

```
fc_shell> get_defined_attributes -application -class net
antenna_rule_name base_name bbox dont_touch full_name is_bbt_object
is_global is_ideal is_in_bundle is_physical is_rdl is_shadow
is_tie_high_net is_tie_low_net max_layer max_layer_mode ...
```

For descriptions of the attributes, see the *object_name*`_attributes` man page for each object type.

Settable Attributes

You can set a “settable” attribute for an object by using the `set_attribute` command. For example,

```
fc_shell> set_attribute -objects [get_nets net16] \
   -name dont_touch -value true
{{net16}}
fc_shell> get_attribute -objects [get_nets net16] \
   -name dont_touch
true
```

An attribute that is not “settable” is read-only; you cannot change it directly by using the `set_attribute` command. For example, the `bbox` (bounding box) attribute of a net is not settable. However, when you edit or optimize a net, the tool automatically updates the `bbox` attribute to reflect the net’s bounding box.

Subscripted Attributes

Subscripted attributes are attributes that have multiple values, each of which is associated with a specific key of the attribute. The key represents the available modes or subtypes of the attribute. A subscripted attribute has the following syntax:

attribute`(`*key*`)`

For example, the `current_repair` attribute for a mismatch configuration is a subscripted attribute whose key is the name of a mismatch configuration. It can be one of the two predefined mismatch configurations, `auto_fix` or `default`, or the name of a user-defined mismatch configuration.

The following example queries the `current_repair` attribute for the `auto_fix` mismatch configuration:

```
fc_shell> get_attribute \
   [get_mismatch_types lib_missing_logical_port] \
   current_repair(auto_fix)
create_placeholder_logic_lib_port
```

Feedback

Cascaded Attributes

For collection-type attributes that point to other design objects, you can query the attributes of that object directly from the current object by using *cascaded attributes*. A cascaded attribute has the following syntax, where the child attribute is an attribute of one or more collection objects returned by the parent attribute:

```
parent_attribute.child_attribute
```

The syntax for the `get_attribute` and `set_attribute` commands with a cascaded attribute:

```
get_attribute $object parent_attribute.child_attribute

set_attribute $object parent_attribute.child_attribute value
```

You can cascade any number of attributes, such as the ones in the following examples:

```
fc_shell> get_attribute [get_vias VIA_S1] \
   owner.top_block.top_module
{ORCA}

fc_shell> set_attribute [get_routing_blockages RB_0] nets.is_user_pg true
{RB_0 RB_0}

fc_shell> get_attribute [get_routing_blockages RB_0] nets.is_user_pg
true true
```

You can remove cascaded attributes by using the `remove_attributes` command as shown in the following example:

```
fc_shell> get_attribute [get_vias VIA_S_0] via_def.cut_pattern
01010000 10101000 01010000 10101000 01010000

fc_shell> remove_attributes [get_vias VIA_S_0] via_def.cut_pattern
{VIA_S_0}

fc_shell> get_attribute [get_vias VIA_S_0] via_def.cut_pattern
Warning: Attribute 'cut_pattern' does not exist on via_def VIA12_5X8
 (ATTR-3)
```

For more information about using cascaded attributes, see Querying Cascaded Attributes.

Defining New Attributes

To define a new attribute, use the `define_user_attribute` command. To set the user-defined attribute to a specific value on a list of objects, use the `set_attribute` command. These user-defined attributes are preserved during optimization if the objects marked with the attributes are not optimized away.

Feedback

The following example defines an attribute named `test_use` and then sets this attribute to `true` on all cells in this level of the hierarchy:

```
# Define a new attribute named test_use
define_user_attribute -classes cell -type string test_use
list_attributes
# Set the value for the test_use attribute
set_attribute -objects [get_cells *] -name test_use -value true
# Report attributes after compile
report_attributes [get_cells *]
```

To define a new attribute that will not be saved to disk, specify the `-non_persistent` option with the `define_user_attribute` command.

```
fc_shell> define_user_attribute \
   -classes pin -name my_att_A -type string \
   -one_of {One Two Three} -non_persistent
1
fc_shell> define_user_attribute \
   -classes pin -name my_att_B -type int \
   -range_min 2
1
```

Keep the following in mind when you use the `define_user_attribute` command:

- The tool can avoid conflicts when reading multiple libraries that have the same user-defined attribute with different data types.
- User-defined attributes on cell libraries and cell library pins are not persistent.

To ensure these attributes are persistent across sessions, define these attributes in the library manager tool during library preparation.

Reporting Attributes

To query the attributes of an object in the design, use the `report_attributes` command:

```
fc_shell> report_attributes -application \
   -class net [get_nets clkm]
...
Design         Object   Type      Attribute Name        Value
-------------------------------------------------------------
my_design_mp   clkm     string    antenna_rule_name
my_design_mp   clkm     string    base_name             clkm
my_design_mp   clkm     rect      bbox                  {357.580 ...
my_design_mp   clkm     boolean   dont_touch            false
my_design_mp   clkm     string    full_name             clkm
my_design_mp   clkm     string    net_type              signal
my_design_mp   clkm     int       number_of_pins        5
...
```

Feedback

To query just a single attribute of an object, use the `get_attribute` command:

```
fc_shell> get_attribute -objects [get_nets clkm] -name net_type
signal
fc_shell> get_attribute -objects [get_nets clkm] \
   -name number_of_pins
5
```

The `-objects` argument can be either a collection or a string. For example,

```
fc_shell> get_attribute -objects [get_nets n123] -name max_layer
M5

fc_shell> get_attribute -objects n123 -name max_layer
M5
```

When you specify the object as a string, the command searches for object types in a specific order. For details, see the man page for the `shell.common.implicit_find_mode` application option. To restrict a search to a specific object type, use the `-class` option.

```
fc_shell> get_attribute -class net -objects a12 -name dont_touch
false
```

Querying Objects

You can get information about objects in the design several different ways. In the GUI display, when you hover over an object, a ToolTip displays basic information about that object. For more information about that object, right-click it and choose Properties.

At the shell prompt, you can use the `report_*` commands (for example, `report_nets`) to report some types of objects, or the `get_*` commands (for example, `get_nets`) to create a collection of objects. You can use a collection as input to another command, or set a variable to a collection for future use:

```
fc_shell> get_nets test*
{test_mode test_se test_si test_si_1 ...}

fc_shell> set my_test_nets [get_nets test*]
{test_mode test_se test_si test_si_1 ...}

fc_shell> report_nets [get_nets test*]
...
Net              Fanout   Fanin   Pins
-----------------------------------------------
test_mode             1       1      2
test_se               1       1      2
test_s1               1       1      2
test_s_1              1       1      2
...
```

Feedback

```
fc_shell> report_nets $my_test_nets
... (generates the same report) ...
```

For more information about collections, see the `collections` man page.

For more information about using the `get_*` commands, see

- Query by Location
- Query by Association
- Query in Physical Context
- Querying Cascaded Attributes

Query by Location

To search for objects at a specific location or within a specified rectangular region, use one of the following methods:

- Use the `-at`, `-intersect`, `-touching`, or `-within` options with the `get_*` command.

  Use this method when you are looking for objects of a single type. Figure 20 shows the behavior of these options. The behavior is the same as in the IC Compiler tool.

- Use the `get_objects_by_location` command.

  Use this method when you are looking for objects of more than one type. Figure 21 shows the default behavior of the `-at`, `-intersect`, `-touching`, and `-within` options with the `get_objects_by_location` command. To change the behavior of these options to match the IC Compiler tool behavior and the `get_*` behavior, set the `design.enable_icc_region_query` application option to `true`.

Feedback

Figure 20 get_* by Location Behavior

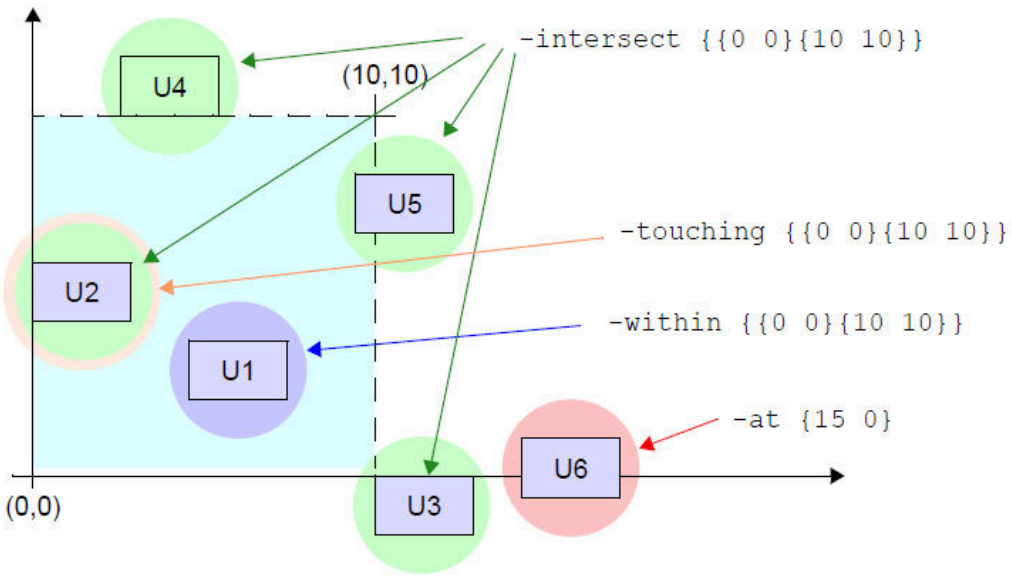
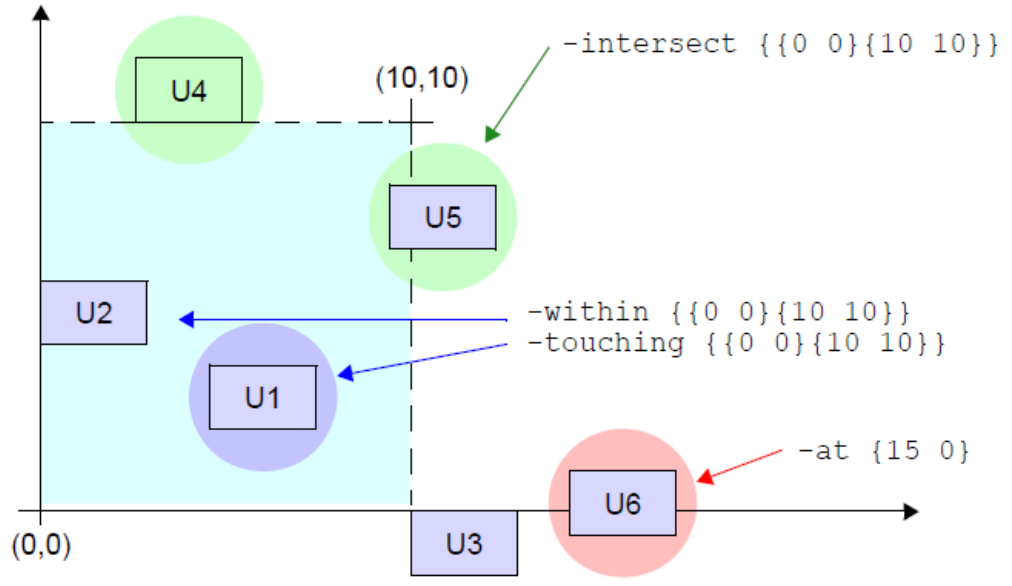


Figure 21 get_objects_by_location Behavior



When you search within a box using the `-within` option, the `get_*` commands get only the objects entirely within the box. By default, the `get_objects_by_location` command gets those objects entirely within the box and those that touch the boundary from inside.

When you search for objects that touch a box using the `-touching` option, the `get_*` commands get only the objects that touch the boundary from inside. By default, the

`get_objects_by_location` command gets the objects entirely within the box and those that touch the boundary from inside.

When you search for objects that intersect a box using the `-intersect` option, the `get_*` commands get the objects that touch or cross the box boundary. By default, the `get_objects_by_location` command gets only the objects that cross the box boundary; those that only touch the boundary from inside or outside are not included.

When you specify an *x y* point using the `-at` option, the `get_*` commands and the `get_objects_by_location` command get all objects at the specified point.

For example, to get all objects at (15,0):

```
fc_shell> get_objects_by_location -at {15 0}
{U6}
```

To get all nets and cells at (0,0):

```
fc_shell> get_objects_by_location -at {0 0} -classes {net cell} \
Warning: Nothing implicitly matched '*' (SEL-003)
```

To get all cells within the box defined by (0,0) and (10,10):

```
fc_shell> get_cells -within {{0 0} {10 10}}
{U1}

fc_shell> get_objects_by_location -within {{0 0} {10 10}} \
  -classes cell
{U1 U2}
```

To get all cells that intersect the boundary of the box defined by (0,0) and (10,10):

```
fc_shell> get_cells -intersect {{0 0} {10 10}}
{U2 U3 U4 U5}

fc_shell> get_objects_by_location -intersect {{0 0} {10 10}} \
  -classes cell
{U4 U5}
```

Query by Association

When you use a `get_*` command, in addition to finding objects by name, you can find objects associated with or connected to specific objects by using the `-of_objects` option. For example, to get the cells connected to a given net:

```
fc_shell> get_cells -of_objects [get_nets net32]
{U83 U75 U18}
```

or to get the nets connected to a given cell:

```
fc_shell> get_nets -of_objects [get_cells U83]
{net24 net32 net91}
```

Feedback

You can use the get_cells command with the -of_objects option to get the cells associated with other cells or with objects such as pins, nets, bounds, I/O guides, I/O rings, voltage areas, voltage area shapes, edit groups, or power strategies.

Query in Physical Context

By default, a get_* command searches for objects in the logical context, using the design netlist. To search for objects in the physical context instead, use the -physical_context option.

For example, the following command gets all nets in the *logical context* (nets in the design netlist):

```
fc_shell> get_nets
{net24 net32 ... }
```

The following command gets all nets in the *physical context* (physical nets in the design view):

```
fc_shell> get_nets -physical_context
{U18/sr0 U18/um18 ... }
```

When you search for an object by name without the -physical_context option, the command matches the specified pattern with the *base name* of the object. Conversely, when you use the -physical_context option, the command matches the specified pattern with the *full name* of the object in the physical hierarchy:

```
fc_shell> get_cells -physical_context *I_SDRAM_IF*
  {I_ORCA_TOP/I_SDRAM_IF/sd_mux_dq_out_11 …}
```

The -physical_context option is available in commands that get logical objects, such as get_cells, get_pins, get_ports, and get_nets; and the report_hierarchy command.

When you use the -physical_context option, a subblock is treated as a single physical cell, so you cannot use this option from the top-level block to search for logical objects in subblocks, except for a command in the following form:

```
fc_shell> get_cells -physical_context -of_objects subblock1
...
```

In this case, the command returns all the physical cells of subblock1.

Querying Cascaded Attributes

For collection-type attributes that point to other design objects, you can query the attributes of that object directly from the current object by using cascaded attributes. For example, the owner attribute of a via points to the via's owner net. You can query an attribute of the owner net directly from the via object with the get_attribute command by cascading the net attribute with the via's owner attribute.

Feedback

To get the owner net of via VIA_S1:

```
fc_shell> get_attribute -objects [get_vias VIA_S1] -name owner
{ISTACK/net12}
```

To get the number of wires in the via owner net:

```
fc_shell> get_attribute -objects [get_vias VIA_S1] \
  -name owner.number_of_wires
3
```

To get all the vias belonging to the owner net:

```
fc_shell> get_vias -filter owner.full_name==ISTACK/net12
{VIA_S1 VIA_S2 VIA_S3 VIA_S4}
```

You can cascade any number of attributes, such as the three in the following example:

```
fc_shell> get_attribute [get_vias VIA_S1] \
   owner.top_block.top_module
{ORCA}
```

Rule-Based Name Matching

When the tool cannot find an exact match for specified objects, you can use the rule-based name matching capability to resolve differences between the specified object names and the name strings of the in-memory object names. The tool provides default name-matching rules for separators and bus notation. You can also define name-matching rules that meet your requirements.

The following application options control rule-based name matching:

- `design.enable_rule_based_query`

  This application option controls rule-based name matching for the `get_cells`, `get_pins`, `get_ports`, `get_nets`, and `find_objects` command. By default, this application option is `false`. To enable this feature, set it to `true`.

- `file.def.rule_based_name_matching`

  This application option controls rule-based name matching for the `read_def` command. By default, this application option is `true`. To disable this feature, set it to `false`.

- `mv.upf.enable_golden_upf`

  This application option enables the golden UPF flow. When this flow is enabled, the `load_upf` command follows rule-based name matching even if the `design.enable_rule_based_query` application option is `false`.

Feedback

If you enable rule-based name matching but do not define specific matching rules, the tool uses the default settings described in the following table:

Rule type	Rule value
Hierarchical separator	`{ / _ . }`
Bus notation	`{[] __ ()}`
Object class	{ *cell port pin net power_domain, power_switch supply_net, supply_set, supply_port* }
Nocase	`false`
Wildcard	`false`
Verbose	`false`

To define specific matching rules, use the `set_query_rules` command, as shown in the following script example. To display the current settings of the matching rules, specify the `-show` option with the `set_query_rules` command.

Script Example

```
# Enable the rule-based name matching capability
set_app_options -name design.enable_rule_based_query -value true
# Specify the matching rules
set_query_rules  -hierarchical_separators {/ _ _xHx_ } \
   -bus_name_notations {[] __ yy}  \
   -wildcard
get_cells instA_instB_0_/C_reg[0]
read_sdc -echo top.sdc
```

Using the matching rules specified in the script, the following table shows an example of query commands and matching queried in-memory design objects:

Query command	In-memory design
`get_cells instA_instB_0/C_reg[0]`	`instA/instB_0__C_reg[0]`
`get_cells instA/instB_0__xHx_C_reg[0]`	`instA/instB_0__C_reg[0]`
`get_cells instA/instB*/*reg[0]`	`instA/instB_0__C_reg[0]`
`get_pins instA/instB[0]_c_regy0y/Q`	`instA/instB_0__C_reg[0]/Q`
`get_cells instA_instB_0_/D`	`instA/instB_0_/D[0]`, `instA/instB_0_/D[1]`, . . .

Feedback

Removing Objects

To remove an object from a block, use a `remove_*` command, such as `remove_cells`:

```
fc_shell> get_cells U11*
{U111 U112 U113 U114 U115}
fc_shell> remove_cells [get_cells U11*]
5
```

To remove objects belonging to different classes, use the `remove_objects` command:

```
fc_shell> remove_objects ADDR*
28
```

To protect objects from accidental removal, set their `physical_status` attribute to `locked`. To override this protection, use the `-force` option to remove the object:

```
fc_shell> get_attribute -objects [get_cells U201] \
   -name physical_status
placed

fc_shell> set_attribute -objects [get_cells U201] \
  -name physical_status -value locked
{U201}

fc_shell> remove_cells [get_cells U201]
Error: One or more objects has locked status. (NDMUI-251)

fc_shell> remove_cells -force [get_cells U201]
1
```

If you remove objects that are included in an existing collection, those objects are removed from the collection as well as from the block.

Supporting Multiple Technologies

A Fusion Compiler design can consist of multiple subdesigns, all of which use a single technology. A 3DIC design can be assembled from multiple subdesigns, and each subdesign can use a different technology.

Fusion Compiler commands are enhanced to allow their arguments to support tech objects, such as `layers` or `via_defs`, from multiple technologies. If a tech object is specified that is not in the technology of the context block or library, the tool issues an error message.

In the context of a 3DIC design, consider the example of the `create_shape` command.

```
fc_shell> create_shape -shape_type polygon -layer M3
```

Feedback

If M3 is a layer that is not present in the technology of the context block in which the shape is created, the tool issues an error message that specifies that the M3 layer is not from the technology of the block.

Note:

To enable 3DIC mode for the tool, set the `design.is_3dic_mode` application option to `true`.

The context block of a command can be

- the current block, when you do not explicitly specify a block
- specified using the `-block`, `-design`, and `-cell` options

The following commands are enhanced to support objects from multiple technologies:

Table 20 Commands That Support Objects From Multiple Technologies

Command	Multiple technology objects supported
`create_rdl_routing_guides`	`-layer`, `-check_overlap_layer`
`create_topological_constraint`	`-start_layers`, `-end_layers`
`create_icovl_cells`	`-routing_blockage_layers`, `-over_icovl_routing_guide_layers`
`create_secondary_pg_placement_constraints`	`-layers`
`create_poly_rect`	`-layers`
`create_pin_guide`	`-layers`
`create_pin_blockage`	`-layers`
`create_grid`	`-layers`
`create_drc_error_shapes`	`-layers`
`create_drc_error`	`-layers`
`create_clips`	`-layers`
`create_differential_group`	`-valid_layers`, `-layer_spacings`
`create_bus_routing_style`	`-valid_layers`
`create_trunk_pin_to_trunk`	`-layer`, `-min_layer`
`create_track`	`-layer`

Feedback

Table 20 Commands That Support Objects From Multiple Technologies (Continued)

Command	Multiple technology objects supported
create_shape_pattern	-layer
create_shape	-layer
create_bump_block	-layer
create_bond_pad_array	-layer
create_dense_tap_cells	-isn_layer
create_pin_constraint	-layers
create_pg_vias	-from_layers, -to_layers
create_keepout_margin	-layers
create_trunk	-layer
create_trunk_shared_track	-layer
create_net_shielding	-disabled_layers
create_rdl_shields	-layers, -reference_layer
create_routing_blockage	-layers
create_routing_guide	-layers
set_block_pin_constraints	-allowed_layers
set_bundle_pin_constraints	-allowed_layers
set_individual_pin_constraints	-allowed_layers
set_programmable_spare_cell_mapping_rule	-dont_overlap_pg_stripe_layer
set_floorplan_length_rules	-from_layers, -to_layers
set_floorplan_location_rules	-from_layers, -to_layers
set_floorplan_spacing_rules	-from_layers, -to_layers
set_virtual_pad	-layer
set_floorplan_area_rules	-layers
set_floorplan_enclosure_rules	-layers

Feedback

Table 20 Commands That Support Objects From Multiple Technologies (Continued)

Command	Multiple technology objects supported
set_floorplan_halo_rules	-layers, -to_layers
set_floorplan_width_rules	-layers
set_grid	-layers
create_via_region	-via_def
set_routing_rule	-min_routing_layer, -max_routing_layer
set_clock_gate_routing_rule	-min_routing_layer, -max_routing_layer
set_clock_routing_rules	-min_routing_layer, -max_routing_layer
set_ignored_layers	-rc_congestion_ignored_layers, -min_routing_layer, -max_routing_layer
remove_ignored_layers	-min_routing_layer, -max_routing_layer
remove_track_constraint	-layer, -track_direction
remove_virtual_pads	-layer
remove_redundant_shapes	-layers
report_first_track_line	-layer
gui_add_missing_vias	-min_layer, -max_layer
gui_check_drc_errors	-layers
check_mib_alignment	-layers
convert_shapes_to_shape_pattern	-layers
convert_vias_to_via_matrix	-via_defs
derive_metal_cut_routing_guides	-site_def
derive_pin_access_routing_guides	-layers
define_antenna_layer_rule	-layer
signoff_report_metal_density	-select_layers
optimize_rdl_routes	-layer
signoff_check_drc_icv_live	-layers

Feedback

Table 20 Commands That Support Objects From Multiple Technologies (Continued)

Command	Multiple technology objects supported
copy_to_layer	-layer
connect_pg_via_ladders	-target_layer
push_rdl_routes	-layer
signoff_calculate_hier_antenna_property	-diffusion_layers, -poly_layers, -contact_layers, -gate_class1_marking_layers, -gate_class2_marking_layers, -gate_class3_marking_layers, -v0_layers_between_m1_m0, -m0_layers_for_poly_connection, -m0_layers_for_diffusion_connection, -contact_layers_between_m0_diffusion, -diff_not_layer_1, -diff_not_layer_2, -cont_not_layer_1, -cont_not_layer_2, -poly_not_layer_1, -poly_not_layer_2
set_track_constraint	-layer
create_routing_rule	-half_shield, -parallel_wire
write_gds	-layers
write_oasis	-layers

Working With Collections

A *collection* is a set of design objects such as cells, nets, or libraries. You create, view, and manipulate collections by using commands provided specifically for working with collections. A regular collection contains only one object type, whereas a *group collection* contains multiple object types.

Typically, you create collections with the `get_*` and `all_*` commands. For example, to create a collection that contains the cells with instance names that begin with *o* and reference an FD2 library cell, use the following command:

```
fc_shell> get_cells {o*} -filter {ref_name == FD2}
{o_reg1 o_reg2 o_reg3 o_reg4}
```

Although the returned result looks like a list, it is not. A collection is referenced by a *collection handle*, which is simply a string value that the tool associates with the collection's internal data structure. When the tool returns a collection during an interactive

Feedback

session, for convenience, it shows the collection contents instead of the collection handle. Collections returned by commands during script execution are not printed.

Most command arguments that accept design objects support collections. You can use the `get_*` commands with the `-of_objects` option to return a collection of objects or a collection of group objects. For example, the following command returns the nets that belong to the MY_GRP group:

```
fc_shell> get_nets -of_objects [get_groups MY_GRP]
```

The tool has commands to create, get, add, remove, and report groups, such as:

- `create_group`—Creates a group in the current block
- `add_to_group`—Adds objects to a group in the current block
- `remove_from_group`—Removes objects from the group in the current block
- `report_group`—Reports groups in the current block
- `remove_groups`—Removes groups from the current block
- `get_groups`—Creates a collection by selecting groups from the current block

For detailed information about working with collections, see *Using Tcl With Synopsys Tools*.

See Also

- Working With Objects

Working With Annotation Shapes

The Fusion Compiler GUI supports the creation of annotations on layout objects. An annotation is a non-maskable shape and stored as text in the design, which might be associated with a specific design object. These shapes allow you to annotate the design elements with additional visual and contextual information in the GUI. You can also specify the type of shape, such as a line, circle, polygon, or a rectangle. For more information on creating annotation shapes using the GUI, see the *Fusion Compiler Graphical User Interface User Guide*

Annotation points are the coordinates that you can use to specify the location of the annotation shapes in a design. You can specify an absolute location or a relative location to an anchor object. If the anchor object moves, the annotation point also moves to maintain its relative placement with the anchor object.

Feedback

Creating Annotation Shapes and Points

You can use the `create_annotation_shape` command to create the annotation shapes, which are saved in the database. The command creates annotation shapes of type circle, line, polygon, or rectangle with an optional text label.

To add a rectangle, use the command as shown in the following example:

```
fc_shell> create_annotation_shape -type rect -color green \
-annotation_points {{200 200} {777 777}}
```

To add a line, use the command as shown in the following example:

```
fc_shell> create_annotation_shape -type line -color red -line_width 2 \
-annotation_points {{123 456} {789 12} {200 200}}
```

Creating a Collection of Annotation Shapes

You can use the `get_annotation_shapes` command to get a collection of the annotation shapes.

To get a collection of the annotation shapes that are matching a specific pattern, use the command as shown in the following example:

```
fc_shell> get_annotation_shapes ANNOTATION_SHAPE*
```

To get a collection of all the annotation shapes in a current block, use the command as shown in the following example:

```
fc_shell> get_annotation_shapes *
```

To filter a collection of the annotation shapes, use the command as shown in the following example:

```
fc_shell> get_annotation_shapes -filter {ungroup_on_remove == last}
```

To get a collection of the annotation shapes containing specific cells, use the command as shown in the following example:

```
fc_shell> get_annotation_shapes -of_objects "U33 U60"
```

Creating a Collection of Annotation Points

You can use the `get_annotation_points` command to get a collection of the annotation points of the design shapes.

To get a collection of the annotation points matching a specific pattern, use the command as shown in the following example:

```
fc_shell> get_annotation_points ANNOTATION_SHAPE*
```

To get a collection of all the annotation points in the current block, use the command as shown in the following example:

```
fc_shell> get_annotation_points *
```

To filter a collection of the annotation points, use the command as shown in the following example:

```
fc_shell> get_annotation_points -filter {ungroup_on_remove == last}
```

To get a collection of the annotation points containing specific cells, use the command as shown in the following example:

```
fc_shell> get_annotation_points -of_objects "U33 U60"
```

Removing Annotation Shapes

You can remove annotation shapes from the current block through the GUI or the shell prompt.

At the shell prompt, use the `remove_annotation_shapes` command to remove one or more annotation shapes.

To remove the annotation shapes matching a specific pattern, use the command as shown in the following example:

```
fc_shell> remove_annotation_shapes ANNOTATION_SHAPE*
```

To remove all the annotation shapes from a design, use the command as shown in the following example:

```
fc_shell> remove_annotation_shapes -all
```

Querying Annotation Shapes

You can search the annotation shapes within a specified region of a design using the `get_objects_by_location` command, as shown in the following example:

```
fc_shell> get_objects_by_location -classes annotation_shape -within
 {{19.4 10.4} {20.7 11.54}}
```

Working With Manufacturing Shapes

A manufacturing shape is a physical object stored as a part of a die block. These shapes include non-functional structures, which lie outside of an active die area and are significant in the multi-die design environment. You can create, query, or remove manufacturing shapes using the Fusion Compiler tool.

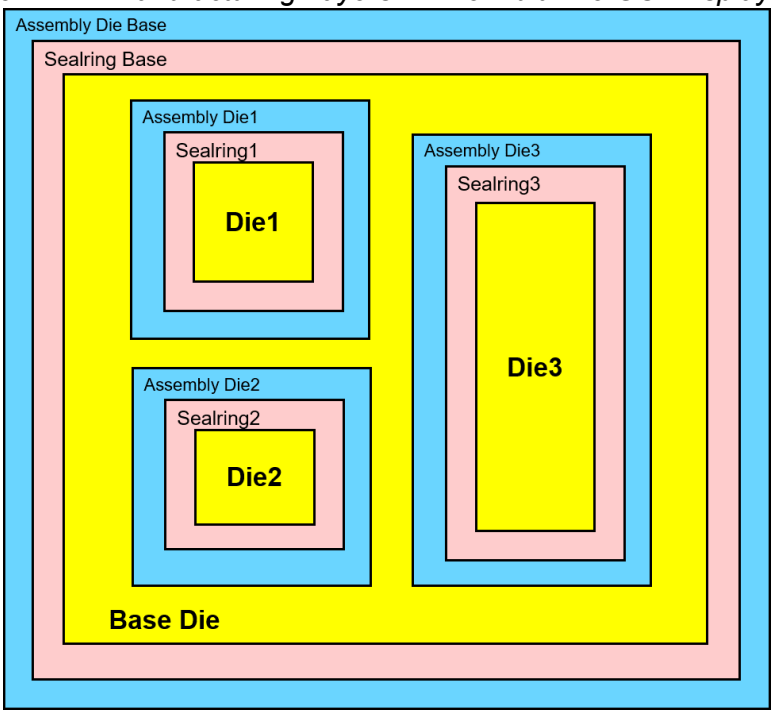
Feedback

The following are manufacturing object types:

- `assembly_die`: is the actual physical die after being cut from a wafer.
- `seal_ring`: is a rectangle larger than a die boundary to protect the die from mechanical damage. The center of a seal ring and the die boundary might not match, but the margins have fixed width.
- `stitching_zone`: is a narrow rectangular zone on a silicon interposer for separating the interposer into two sets of masks.

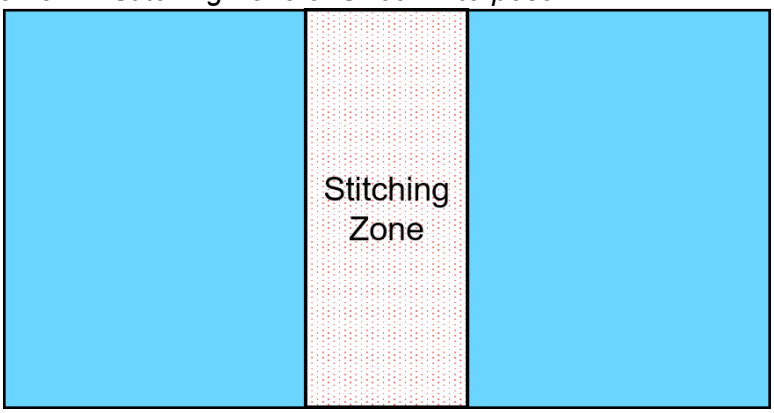
The following figures show the manufacturing shapes for the active dies.

Figure 22 Manufacturing Layers in The Multi-Die GUI Display



Feedback

Figure 23 Stitching Zone of Silicon Interposer



Creating Manufacturing Shapes

To create the manufacturing shapes, use the `create_manufacturing_shape` command. This command creates a new manufacturing shape of the specified type in the current block.

You must specify the `-type` option to define the subtype of the manufacturing shape and the `-boundary` or `-offset` option to specify the dimensions of the shape. The valid values for the `-type` option are `seal_ring`, `assembly_die`, and `stitching_zone`.

To create a specified type of a shape using the `-offset` option, use the command as shown in the following example:

```
fc_shell> create_manufacturing_shape -type seal_ring \
   -name SL1 -offset {10 10 10 10}
```

To create a specified type of a shape using the `-boundary` option, use the command as shown in the following example:

```
fc_shell> create_manufacturing_shape -type assembly_die \
   -boundary {{0 0} {1000 1000}}
```

Querying Manufacturing Shapes

To get information about manufacturing shapes, use the `get_manufacturing_shapes` or `report_manufacturing_shapes` command.

The `get_manufacturing_shapes` command creates a collection of the manufacturing shapes for the current block. You can filter this collection using different command options.

To get the manufacturing shapes for a reference block of the cell instance, use the `get_manufacturing_shapes` command as shown in the following example:

```
fc_shell> get_manufacturing_shapes -of_objects [get_cells DIE1] \
   {SL1 ASSEMBLY_DIE_2}
```

Feedback

To filter a collection for a given expression, use the `get_manufacturing_shapes` command as shown in the following example:

```
fc_shell> get_manufacturing_shapes -filter {type == seal_ring}  \
   {SL1 SL2}
```

The `report_manufacturing_shapes` command prints the name, type, and boundary points of the specified manufacturing shape. If you do not specify any manufacturing shape, the command reports all the shapes in the current block.

To print the name, type, and boundary points of the specified manufacturing shape, use the `report_manufacturing_shapes` command as shown in the following example:

```
fc_shell> report_manufacturing_shapes [get_manufacturing_shapes SL*]
```

Removing Manufacturing Shapes

The `remove_manufacturing_shapes` command removes one or more manufacturing shapes from the current block.

To remove a specified manufacturing shape, use the `remove_manufacturing_shapes` command as shown in the following example:

```
fc_shell> remove_manufacturing_shapes [get_manufacturing_shapes SL1]
```

To remove all the manufacturing shapes from the current block, use the command as shown in the following example:

```
fc_shell> remove_manufacturing_shapes -all
```

Design Hierarchy

By default, the Fusion Compiler tool implements a hierarchical design as physically flat by placing all leaf-level cells, from all hierarchical levels, anywhere in the available chip area.

To implement a lower-level module as a physically distinct block, “commit” that module by using the `commit_block` command. For example,

```
fc_shell> commit_block u0_1
```

This example creates a separate physical hierarchical block for the logical module named u0_1. The cells in module u0_1 are placed only in this physical block, and this block only contains cells from the module u0_1.

See Also

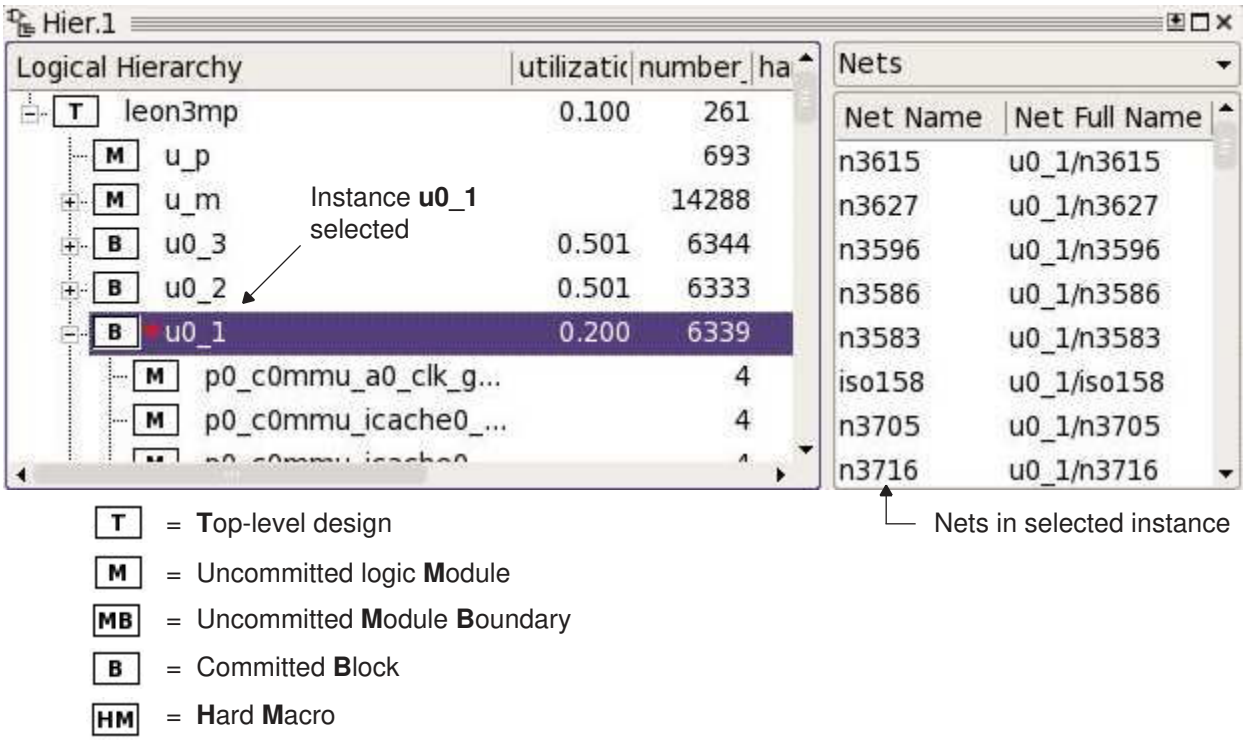
- Navigating the Design Hierarchy
- Hierarchical Query Using get_* Commands

Feedback

Navigating the Design Hierarchy

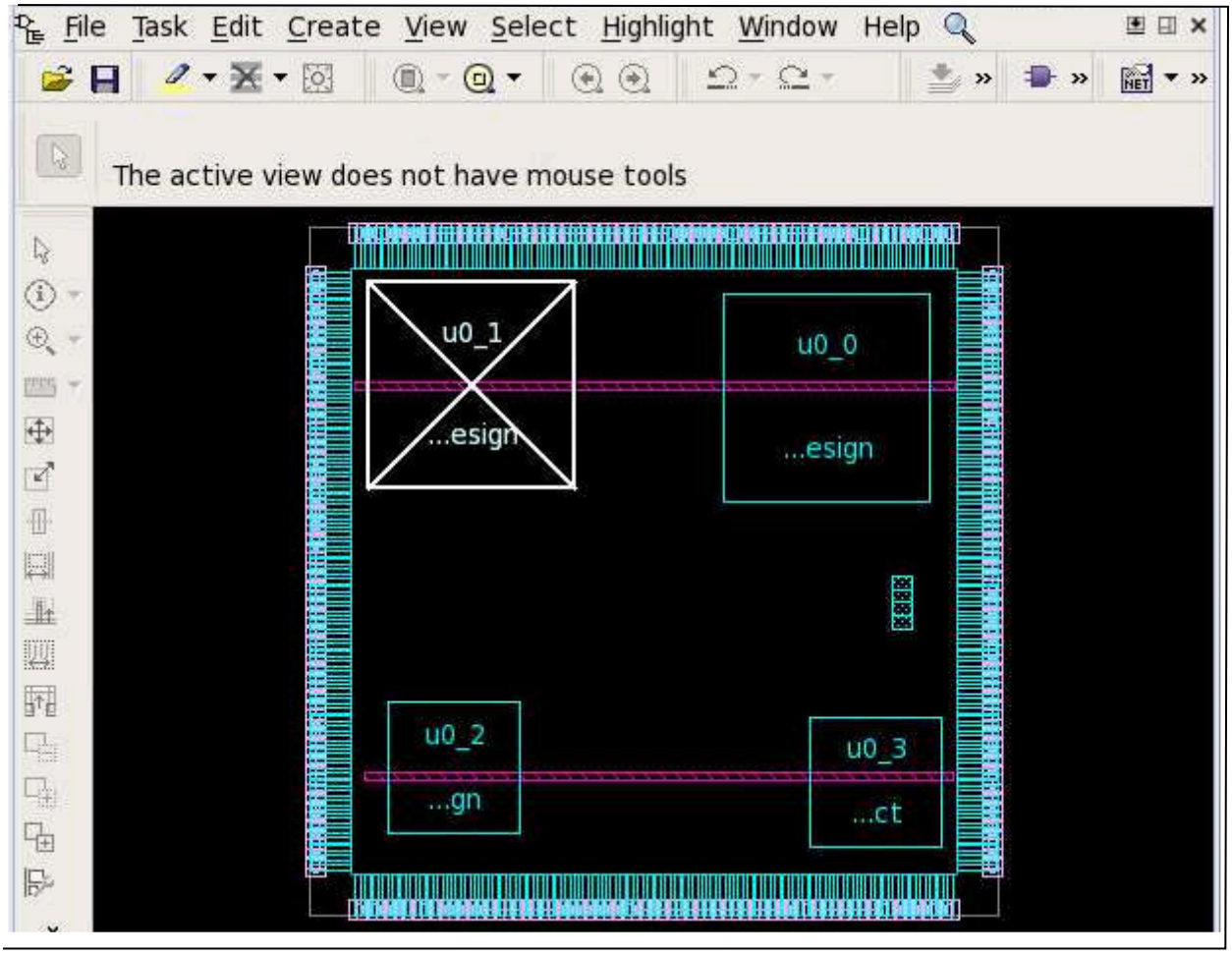
The hierarchy browser in the GUI (View > Hierarchy Browser) lets you visually navigate the logic hierarchy. When you select an instance in the hierarchy browser, the physical block associated with that instance (if any) is highlighted in the physical layout view, as shown in the following figures.

Figure 24 Hierarchy Browser



Feedback

Figure 25 Physical Layout View



The `current_block` and `current_design` commands provide two different ways to set the current block in the Fusion Compiler tool. The `current_block` command sets the current block by specifying the block name, whereas the `current_design` command sets the current block by specifying a hierarchical module name in the logical netlist.

In addition to the current block, there is also a current working instance, which is a hierarchical cell in the netlist of the current block. You can use the `current_instance` command to traverse the logic hierarchy and set the current instance, much like you use the `cd` command in Linux to traverse a file hierarchy.

To report the full hierarchy of the design, use the `report_hierarchy` command. Use the `-physical_context` option to report only the physical (not logical) cells in the hierarchy, and use the `-no_leaf` option to exclude the leaf-level cells and get a more concise report.

Feedback

See Also

- current_block
- current_design
- current_instance
- report_hierarchy
- Hierarchical Query Using get_* Commands

current_block

The `current_block` command takes a block as an argument, such as an object returned by the `get_blocks` command. When the `current_block` command is used with an argument, it sets the current block to the specified block. When used without an argument, it returns the current block.

```
fc_shell> current_block design3mp
{mylibA:design3mp.design}

fc_shell> current_block
{mylibA:design3mp.design}
```

current_design

The `current_design` command takes a design object as an argument, such as an object returned by the `get_designs` command. In this context, a *design* is a hierarchical module in the logical netlist. When the `current_design` command is used with an argument, it sets the current block to the block containing the specified design. When used without an argument, it returns the current block, just like the `current_block` command.

```
fc_shell> get_designs
{design3mp mul32_infer0 design3s design3s_2 design3s_2 design3X}

fc_shell> current_design design3X
{mylibB:design3X.design}

fc_shell> current_design
{mylibB:design3X.design}
```

Feedback

current_instance

The current working instance is a hierarchical cell of the current block. The default current instance is the top-level module of the current block. You can use the `current_instance` command to traverse the logic hierarchy and set the current instance much like you use the `cd` command in Linux to traverse a file hierarchy.

```
fc_shell> current_block
{mylibA:design3mp.design}

fc_shell> current_instance
Current instnace is the top-level module of design 'design3mp'

fc_shell> get_cells -filter "is_hierarchical==true"
{u0_0 u0_1 u0_2 u0_3 u_m u_p}

fc_shell> current_instance u0_0
u0_0
fc_shell> current_instance p0_iu0
u0_0/p0_iu0
fc_shell> current_instance .
u0_0/p0_iu0
fc_shell> current_instance ..
u0_0
fc_shell> current_instance ../u0_1/p0_mul0
u0_1/p0_mul0
fc_shell> current_instance
Current instance is the top-level module of design 'design3mp'
```

report_hierarchy

The `report_hierarchy` command reports the hierarchy of a block. By default, it generates a report on the current block showing the full physical and logical reference hierarchy down to the leaf module level.

To get a more concise report, use the `-physical_context` option to report only the physical (not logical) cells in the hierarchy or the `-no_leaf` option to exclude the leaf-level cells, or both. For example,

```
fc_shell> report_hierarchy -physical_context -no_leaf
...
mylibA:design3mp/fp.design
  design3s:design3s/fp.design
     mul32_infer0:mul32_infer0/fp.design
  design3s:design3s_2/fp.design
  design3s:design3s_3/fp.abstract
1
```

Feedback

```
fc_shell> report_hierarchy -physical_context
...
mylibA:design3mp/fp.design
  B4ISH1025_NS                saed32io_wb
  D4I1025_NS                  saed32io_wb
  ISH1025_NS                  saed32io_wb
  ...
  design3s:design3s/fp.design
     AND2X1_RVT               saed32rvt
     AND2X2_RVT               saed32rvt
     AND3X1_RVT               saed32rvt
     ...
     mul32_infer0:mul32_infer0/fp.design
        ...
  design3s:design3s_2/fp.design
     ISOLORX1_HVT             saed32hvt
     LSDNSSX1_HVT             saed32hvt_dlvl
     OAI222X1_RVT             saed32rvt
  design3s:design3s_3/fp.abstract
1
```

```
fc_shell> report_hierarchy
...
mylibA:design3mp/fp.design
  LSUPX1_HVT                      saed32hvt_ulvl
  NBUFFX16_LVT                    saed32lvt
  design3s:design3s/fp.design
     ISOLORAOX1_HVT               saed32hvt
     SNPS_CLOCK_GATE_HIGH_div32_7
     SNPS_CLOCK_GATE_HIGH_div32_8
     ...
     (very long report shows all leaf-level logical and
     physical cells)
     ...
```

By default, multiple occurrences of a block or module are combined into a single report for all instances of the same block unless the occurrences are bound to different views. To expand the report to show the hierarchy of all instances, use the `-hierarchical` option of the command.

For more information, see the man page for the `report_hierarchy` command.

Hierarchical Query Using get_* Commands

The Fusion Compiler tool offers commands to query the cells, nets, pins, and ports in the design database:

```
get_cells
get_nets
```

Feedback

```
get_pins
get_ports
```

Each command creates a collection of objects in the design that meet the criteria specified by the command options. You can use a collection as input to another command, or set a variable to the collection for future use.

The command options let you restrict the scope of the query by object name, hierarchical level, physical context, and object attributes. For example, for the `get_cells` command:

```
get_cells [patterns]
  [-hierarchial]
  [-physical_context]
  [-of_objects objects]
  [-filter expression]
  ...

# get all cells at the current level of hierarchy
fc_shell> get_cells

# get all cells at the current level of hierarchy
fc_shell> get_cells *

# get all cells of BLKA
fc_shell> get_cells BLKA/*

# get cells named R* at all levels of hierarchy
fc_shell> get_cells R* -hierarchical

# get cells named R* in the flat physical context
fc_shell> get_cells R* -physical_context

# get all cells in the flat physical context that own physical pins
# of net n253
fc_shell> get_cells -physical_context -of_objects [get_nets n253]

# get cells named R* at all levels of hierarchy; include only the
# blocks that have a parent (only the cells of subblocks)
fc_shell> get_cells R* -hierarchical \
   -filter defined(parent_block_cell)
```

Query Using Only a Search String

When you query the design for objects using only a search string and no other options, the command searches for objects that have the specified name. The name string can be plain or hierarchical:

```
# get cells named R* at current level
fc_shell> get_cells R*

# get cells named REG1/R* (same as cells named R* in REG1)
fc_shell> get_cells REG1/R*
```

Feedback

The asterisk wildcard character (*) does *not* match with the hierarchy separator character, the forward slash (/). Therefore, a search string without any forward slash characters occurs only at the current level and does not traverse the hierarchy.

Note that the slash character is sometimes used as part of a cell name and not as a hierarchy separator, as demonstrated in the following example.

```
fc_shell> get_cells I_P
{I_P}

# "*" does not match hierarchy separator, search current level only
fc_shell> get_cells I_P*
{I_P}

# search under cell I_P, one level only
fc_shell> get_cells I_P/*U3
{I_P/U3 I_P/m2/U3}
```

Why did the last search return `I_P/m2/U3`, which appears to be *two* levels below `I_P`? The answer is that the cell is directly under `I_P` and is literally named "`m2/U3`" with the slash character inside the cell name. You can confirm this by querying the cell attributes:

```
fc_shell> get_attribute [get_cells I_P/*U3] -name parent_cell
{I_P I_P}

fc_shell> get_attribute [get_cells I_P/*U3] -name name
U3 m2/U3
```

Query With the -hierarchical Option

When you query the design for objects with the `-hierarchical` option, the command searches for objects throughout the logical hierarchy of the design:

```
fc_shell> get_cells R* -hierarchical
{BLK1/REG43 BLK2/RXF38 BLK2/ACE1/RG95}
```

The command searches for an object of the exact name at each level of hierarchy, like the `find` command in Linux. Therefore, the search string must be a simple name; it *cannot* use hierarchy separator characters:

```
fc_shell> get_cells BLK1/R* -hierarchical
Warning: No cell objects matched 'BLK1/R*' (SEL-004)
Error: Nothing matched for collection (SEL-005)
```

If you include a slash character in the search string while using the `-hierarchical` option, the command searches for objects that literally have the slash character as part of the object name; it does not search a lower level of hierarchy.

Feedback

Query With the -physical_context Option

When you query the design for objects with the `-physical_context` option, the command searches for objects in the flattened physical netlist for the design:

```
fc_shell> get_cells *RLB_27 -physical_context
{I_BLK1/I_REG19/RLB_27}
```

Note that an asterisk wildcard character (`*`) in a search string matches with the forward slash character, interpreting the slash as part of the object name.

In the flattened physical netlist, all cells are considered leaf cells. This is the netlist seen by the router. The full names of the cells can contain slash characters that reflect the hierarchy of the design, but the physical netlist itself is treated as entirely flat and the slash characters are treated as ordinary characters used in object names.

Because the netlist is considered flat, it is not necessary (and not recommended) to use the `-hierarchical` option together with the `-physical_context` option.

Query With the -of_objects Option

When you query the design for objects with the `-of_objects` option, the command searches for objects that are associated with the listed objects:

```
# get cells
fc_shell> get_cells I_P/*
{I_P/U11 I_P/U13 I_P/U161 I_P/U14}

# get pins of cell
fc_shell> get_pins -of_objects [get_cells I_P/U13]
{I_P/U13/A1 I_P/U13/A2 I_P/U13/Y I_P/U13/VDD I_P/U13/VSS}

# get nets connected to cell
fc_shell> get_nets -of_objects [get_cells I_P/U13]
{I_P/n287 I_P/n17 I_P/n26 I_P/VDD I_P/VSS}

# get cells connected to net
fc_shell> get_cells -of_objects [get_nets I_P/n17]
{I_P/U161 I_P/U14 I_P/m2/U13}
```

You cannot specify a search string while also using the `-of_objects` option, but you can filter the results with the `-filter` option.

Feedback

Query With the -filter Option

In a `get_*` command, you can use the `-filter` option to restrict the generated collection to only the objects that meet some filtering condition specified as a Boolean expression.

For example, the following command gets all cells named `B*` but then only allows the cells that have their `ref_name` attribute set to the string `FD2` to be added to the collection:

```
fc_shell> get_cells "B*" -filter "ref_name == FD2"
{B_reg1 B_reg2 B_reg3 B_reg4}
```

The following hierarchy-related attributes can help you find the logical or physical owner of an object:

- `parent_cell` – The *logical* cell that contains the object. For a physical object, this is the cell of the parent block. For a purely logical object, this is different from the cell of the parent block.
- `parent_block` – The *physical* block (libname:block/label.design) that contains the object. The parent block of a block is the same block itself.
- `parent_block_cell` – The first cell that is a block (libname:block/label.design) found while traversing the parent cells up the hierarchy. When the object is associated with a logic hierarchy, the result is different from the parent cell.
- `top_block` – The top-level block (libname:block/label.design) in the current block hierarchy.

For example,

```
# get objects in all levels of hierarchy, both logical and physical
fc_shell> get_cells -hierarchical

# get physical cells of object named "subblock1"
fc_shell> get_cells -physical_context -of_objects subblock1

# get objects from the top level (those that have no parent)
fc_shell> get_cells -hierarchical \
   -filter undefined(parent_block_cell)

# get cells inside all subblocks
fc_shell> get_cells -hierarchical \
   -filter defined(parent_block_cell)
```

Technology Data Access

You can get, report, and modify some technology data using Tcl commands. This is the data specified in a technology file and read into a design library by using the

Feedback

`read_tech_file` command or by using the `-technology` option of the `create_lib` command.

Table 21 shows the technology objects that you can access from the command line and the commands used to query or modify the objects. For all of these objects, you can use the `report_attributes -class object_class` command to query the object's attributes. If an object has settable attributes, you can modify the attributes by using the `set_attribute` command.

Table 21 Technology Objects and Access Commands

Object class	Technology file section	Object access commands

`tech`	`Technology`	`create_tech`
		`get_techs`
		`remove_tech`

`site_def`	`Tile`	`create_site_def`
		`get_site_defs`
		`remove_site_defs`
		`report_site_defs`

`layer`	`Layer`	`create_layer`
		`remove_layers`
		`get_layers`

`purpose`	`LayerDataType`	`create_purpose`
		`remove_purposes`
		`get_purposes`

`via_def`	`ContactCode`	`create_via_def`
		`get_via_defs`
		`remove_via_defs`
		`report_via_defs`

`design_rule`	`DesignRule`	`get_design_rules` `report_design_rules`

`pr_rule`	`PRRule`	`create_pr_rule`
		`get_pr_rules`
		`remove_pr_rules`
		`report_pr_rules`

`density_rule`	`DensityRule`	`create_density_rule`
		`get_density_rules`
		`remove_density_rules`

Feedback

Table 21 Technology Objects and Access Commands (Continued)

Object class	Technology file section	Object access commands
via_rule	ViaRule	create_via_rule get_via_rules remove_via_rules report_via_rules

The tool saves changes to technology data stored in the design library, but it does not save changes to technology data stored in reference libraries, as described in the following table.

Method for storing technology data	Changes saved in the design library?
Technology data stored in cell library referenced by design library	No, available in current session only
Technology data stored in technology library referenced by design library	No, available in current session only
Technology data loaded into design library	Yes

To write out the modified technology data into a technology file, use the `write_tech_file` command. However, the technology file syntax does not support the `symmetry` and `is_default` attributes of site_def objects or the `routing_direction` and `track_offset` attributes of layer objects, so these attributes are not written to the technology file. For more information about preparing the site definition and routing layer technology data, see Completing the Technology Data in the *Library Manager User Guide*.

You can use the `create_routing_rule` command to create nondefault routing rules and the `set_routing_rule` command to apply these rules to specific layers or nets. These nondefault routing rules do not change the default rules defined in the technology file and are not written out by the `write_tech_file` command.

The following session example shows how to query and modify the site definitions in the design library's technology data.

```
fc_shell> list_attributes -application -class site_def
...
Attribute Name      Object     Type       Properties  ...
-------------------------------------------------------
full_name           site_def   string     A,S
height              site_def   distance   A,S
is_default          site_def   boolean    A,S
symmetry            site_def   string     A,S
...
```

Feedback

```
fc_shell> report_site_defs
...
      Width Height Type Is_default Tech   Symmetry Legal_orienta
-----------------------------------------------------------------
unit1 0.15  1.67   core default    saed32 --       R0

fc_shell> report_attributes -application -class site_def \
  [get_site_defs]
...
Design    Object   Type        Attribute Name        Value
-----------------------------------------------------------
design3mp  unit1    string      full_name             unit1
design3mp  unit1    distance    height                1.6720
design3mp  unit1    boolean     is_default            true
design3mp  unit1    string      symmetry
...

fc_shell> set_attribute -objects [get_site_defs] -name symmetry \
  -value {Y}
{unit}

fc_shell> get_attribute -objects [get_site_defs] -name symmetry
Y

fc_shell> create_site_def -name unit2 -width 0.1520 \
  -height 3.3440 -symmetry Y
{unit2}

fc_shell> get_site_defs
{unit1 unit2}

fc_shell> report_attributes -application -class site_def \
  [get_site_defs unit2]
...
Design    Object   Type        Attribute Name        Value
-----------------------------------------------------------
design3mp  unit1    string      full_name             unit2
design3mp  unit1    distance    height                3.3440
design3mp  unit1    boolean     is_default            false
design3mp  unit1    string      symmetry              Y
...
```

See Also

- Loading the Technology Data

Feedback

Bus and Name Expansion

The tool supports querying of nets, ports, pins, and cells that use bus subscription and name expansion. This feature applies to the `get_cells`, `get_nets`, `get_pins`, and `get_ports` commands, as well as to commands that accept a collection of cells, nets, ports, or pins.

The examples in the following table demonstrate the bus subscripting and name expansion syntax.

Table 22 Bus and Name Expansion Examples

Command	Result	Comments
`get_nets a(0:3)`	`{a0 a1 a2 a3}`	Name expansion: A digit followed immediately by a colon (:) and another digit, in parentheses “()” specifies a name expansion range
`get_nets {a[0:3]}`	`{a[0] a[1] a[2] a[3]}`	Bus subscripting: A digit followed immediately by a colon (:) and another digit, in square braces “[ ]” specifies a bus subscripting range
`get_nets {a[0:1,6:8]}`	`{a[0] a[1] a[6] a[7] a[8]}`	Bus subscripting: Two ranges separated by a comma and enclosed by square braces “[ ]”
`get_nets {a[0:4:2]}`	`{a[0] a[2] a[4]}`	Bus subscripting: From 0 to 4, step by 2, enclosed by square braces “[ ]”
`get_nets {a[0:2][0:2]}`	`{a[0][0] a[0][1] a[0][2] a[1][0] a[1][1] a[1][2] a[2][0] a[2][1] a[2][2]}`	Multidimensional bus subscripting: Two ranges enclosed by curly braces "{ }"
`get_nets a(0:4:2)`	`{a0 a2 a4}`	Name expansion: From 0 to 4, step by 2, enclosed by parentheses “( )”
`get_nets {a(0:2)b[0:4:2,3]}`	`{a0b[0] a0b[2] a0b[4] a0b[3] a1b[0] a1b[2] a1b[4] a1b[3] a2b[0] a2b[2] a2b[4] a2b[3]}`	Combined usage of name expansion and bus subscripting
`get_nets {a(0:1)b[0:2][0:2]}`	`{a0b[0][0] a0b[0][1] a0b[0][2] a0b[1][0] a0b[1][1] a0b[1][2] a0b[2][0] a0b[2][1] a0b[2][2] a1b[0][0] a1b[0][1] a1b[0][2] a1b[1][0] a1b[1][1] a1b[1][2] a1b[2][0] a1b[2][1] a1b[2][2]}`	Combined usage of name expansion and multidimensional bus subscripting

Feedback

For bus subscripting, the default delimiter characters are square braces “[ ]”. To change to one of the other allowed delimiter character sets, “{ }” “( )” or “< >”, set the `design.bus_delimiters` application option:

```
fc_shell> set_app_options -name design.bus_delimiters -value {<>}
design.bus_delimiters <>
```

For cell, net, port, and pin subscripting, the default delimiter characters are parentheses “( )”. To change the default, set the `design.name_expansion_delimiters` application option.

Note:

You cannot combine subscripting with wildcards or with the `-regexp`, `-nocase`, or `-exact` options in the `get_nets`, `get_pins`, and `get_ports` commands.

Reporting Design Information

A block is a container for physical and functional design data. The current block is the default block affected by block-related commands. You can set or report the current block by using the `current_block` command or set the current block for a specified top-level module by using the `current_design` command.

To report information about the design data stored in the current block, use the following commands:

- `report_design`

  By default, this command generates a summary report on the design contents such as the number of cells, hierarchical levels, chip area, and total route length. For details, see Reporting Design Contents.

- `report_unbound`

  This command checks the design for objects that are not linked to a reference library (unbound objects) and reports unbound objects as error or warning messages. For details, see Reporting Unbound Objects.

- `check_physical_constraints`

  This command checks the design for issues related to physical constraints that could lead to problems at later stages of the design flow, and reports these issues as error, warning, and information messages. The checked conditions include basic floorplan data, site rows, move bounds, group bounds, layers, tracks, macros, other cell instances, and placement blockages. For details, see Reporting Physical Constraints.

Feedback

Reporting Design Contents

By default, using the `report_design` command without options generates a summary report on the design contents such as the number of cells, hierarchical levels, chip area, and total route length. For a more detailed report, specify the report type in the command:

```
fc_shell> report_design -library
...
fc_shell> report_design -netlist
...
fc_shell> report_design -netlist -hierarchical
...
fc_shell> report_design -floorplan
...
fc_shell> report_design -routing
...
```

The `report_design` command generates a detailed report as shown in the following example:

```
fc_shell> report_design -netlist
...
--------------------------------------------------------------------------
                            NETLIST INFORMATION
--------------------------------------------------------------------------
CELL INSTANCE INFORMATION
--------------------------------------------------------------------------
Cell Instance Type    Count % of         Area % of siteAreaPerSite
                            total              total
--------------------------------------------------------------------------
TOTAL LEAF CELLS      12973  100  9360583.005  100 unit:36911736
  Standard cells      12704   97    61696.760    0 unit:242763
  Filler cells            0    0        0.000    0
  Diode cells             0    0        0.000    0
  Module cells            4    0  6117000.323   65 unit:24069033
    Soft macro cells      4    0  6117000.323   65 unit:24069033
  Hard macro cells        4    0    49885.923    0 unit:197220
  ...

REFERENCE DESIGN INFORMATION
--------------------------------------------------------------------------
Number of reference designs used:88
--------------------------------------------------------------------------
Name          Type        Count   Width  Height  Area
--------------------------------------------------------------------------
SDFFX1_RVT    lib_cell    1729    5.17   1.67    8.641   ...
NBUFFX32_RVT  lib_cell    1727    6.38   1.67    10.674  ...
AO22X1_RVT    lib_cell    1364    1.52   1.67    2.541   ...
...

NET INFORMATION
```

Feedback

```
-------------------------------------------------------------------
NetType                  Count FloatingNets         Vias Nets/Cells
-------------------------------------------------------------------
Total                    59276        45346       113837       4.569
Signal                   59276        45346       113837       4.569
Power                        0            0            0       0.000
...

NET FANOUT AND PIN COUNT INFORMATION
-------------------------------------------------
Fanout         Netcount     netPinCount     NetCount
-------------------------------------------------
<2                54241     <2                  45346
2                  2452     2                    9123
3                   979     3                    2224
4                   559     4                     979
5                   243     5                     559
...

PORT AND PIN INFORMATION
-------------------------------------------------------------------
Type          Total     Input    Output     Inout      3-st     ...
-------------------------------------------------------------------
Total        103285     86052     18407      1272       484     ...
Macro           600       344       256         0       256     ...
Ports           261       160       228       127         0     ...
...
```

For more information about the `report_design` command, see the man page. Additional usage information is available in the *Fusion Compiler™ User Guide* and the *Fusion Compiler™ Design Planning User Guide*.

Reporting Unbound Objects

You can check for objects that are not linked to a reference library (unbound objects) by using the `report_unbound` command:

```
fc_shell> report_unbound -verbose
Error: Via definition 'VIA23' is missing. 'VIA_S_3 'and `2'
other via(s)which reference this via definition are
unbound. (CHUNB-007)
```

By default, the command reports unbound cell instances, site rows, site arrays, shapes, tracks, layers, pin guides, pin blockages, routing guides, routing corridor shapes, vias, and via definitions:

- For cell instances, vias, site rows, and site arrays

  The command reports the missing reference library and the number of unbound objects for each reference library.

Feedback

- For shapes, tracks, layers, pin guides, pin blockages, routing guides, routing corridor shapes, and via definitions

  The command reports the unbound layers with the related unbound object names.

You can report specific objects only by using options with the `report_unbound` command. For a list of options, see the man page.

By default, the `report_unbound` command checks objects at the top level. To report unbound objects in other levels of the physical hierarchy, specify the `-hierarchical` option.

To link an unbound object to a reference library, use the following guidelines:

- Unbound cell instances

  Check your reference library settings, add a reference library as needed with the `set_ref_libs -add` command, and rebind the block with the `link_block -force -rebind` command.

- Unbound site rows or site arrays

  Check the technology file for a site definition; if there is no site definition, create a site definition by using the `create_site_def` command.

- Unbound vias

  Check the via definitions in the current block or the technology file; if the via definition is missing, create a via definition by using the `create_via_def` command.

You can also use the `check_design` command to report unbound objects, as shown in the following example:

```
fc_shell> check_design -checks unbound
```

The `check_design` report provides the same results as the `report_unbound` report without options.

See Also

- Specifying a Design Library's Reference Libraries
- Reporting Reference Libraries
- Rebinding Reference Libraries of a Design Library
- Reference Library List

Feedback

Reporting Physical Constraints

You can check a design's physical constraints by using the `check_physical_constraints` command. The command checks the design for issues related to physical constraints that could lead to problems at later stages of the design flow, and reports these issues as error, warning, and information messages.

The checked conditions include

- Basic floorplan data
- Site rows
- Move bounds
- Group bounds
- Layers
- Tracks
- Macros and other cell instances
- Placement blockages

To check the physical constraints of the design, run the `check_physical_constraints` command:

```
fc_shell> check_physical_constraints
...
Warning: The spacing of layer 'VTL_N' is greater than the
 difference of the pitch and width of the layer. (DCHK-105)
Warning: the spacing of layer 'PO' is greater than the
 difference of the pitch and width of the layer. (DCHK-105)
Information: The layer 'M8' does not contain any PG shapes. (DCHK-104)
Information: The layer 'M9' does not contain any PG shapes. (DCHK-104)
Warning: The orientation of cell instance 'tapfiller_TAPCELLBWP16P90_0'
 does not match any legal orientations. (DCHK-103)
Warning: The orientation of cell instance 'tapfiller_TAPCELLBWP16P90_1'
 does not match any legal orientations. (DCHK-103)
1
```

For information about the issues the tool identifies and how to fix them, see the man page for the corresponding message ID number.

In addition to generating a report, the `check_physical_constraints` command generates an enhanced messaging system (EMS) database that you can view by using the message browser in the Fusion Compiler GUI. Create the EMS database by running

the `create_ems_database` command before you run the `check_physical_constraints` command:

```
fc_shell> create_ems_database check_hier.ems
fc_shell> check_physical_constraints
fc_shell> save_ems_database
```

In the Fusion Compiler GUI message browser, you can sort, filter, and link the messages to the corresponding man page. For information about viewing the EMS database in the message browser, see the *Fusion Compiler™ User Guide*.

Polygon Manipulation

To create and edit polygon shapes, you can use two kinds of abstract geometric objects:

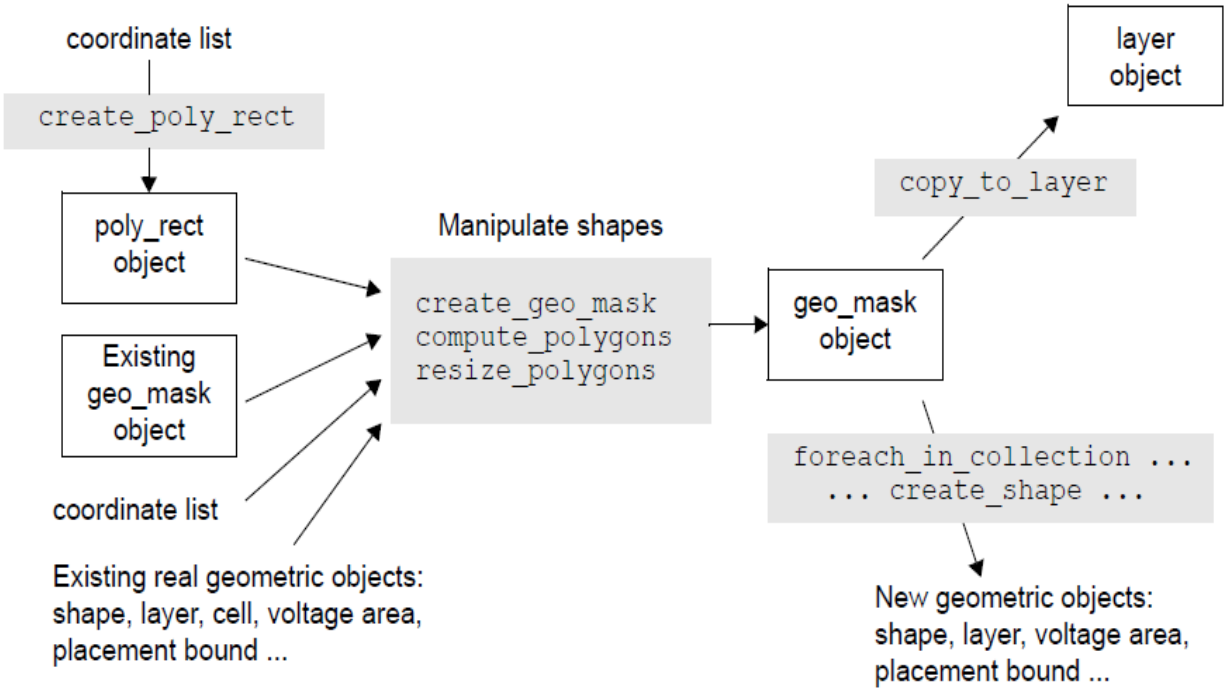
- poly_rect – a single geometric shape that consists of a set of coordinate points
- geo_mask – a collection of poly_rect objects

The poly_rect and geo_mask type objects are abstract and have no direct functional purpose in a block. To create a functional shape such as a mask layer, blockage, or keepout region, you need to convert poly_rect or geo_mask objects into real objects.

The following figure shows the flow to create, edit, and convert abstract shapes to make real shapes.

Feedback

Figure 26 How to Manipulate Geometric Objects



The commands in the following table are available to create, manipulate, and convert polygon shapes.

Table 23 Polygon Manipulation Commands

Command	Description
`compute_area`	Calculates the area of a geometric region
`compute_polygons`	Performs a Boolean operation between two shapes to create a geo_mask object
`copy_to_layer`	Copies a geo_mask object to create a collection of shapes in a mask layer
`create_geo_mask`	Copies geometric objects to create a new geo_mask object
`create_poly_rect`	Creates a collection of poly_rect objects from a list of coordinates and optionally associates a layer with the result
`resize_polygons`	Pushes the edges of specified polygons inward or outward, creating a new geo_mask object
`split_polygons`	Decomposes a geometric region into rectangles or trapezoids, creating a collection of geo_mask or poly_rect objects

Feedback

For details, see

- Creating poly_rect and geo_mask Objects
- Converting geo_mask Objects Into Functional Shapes

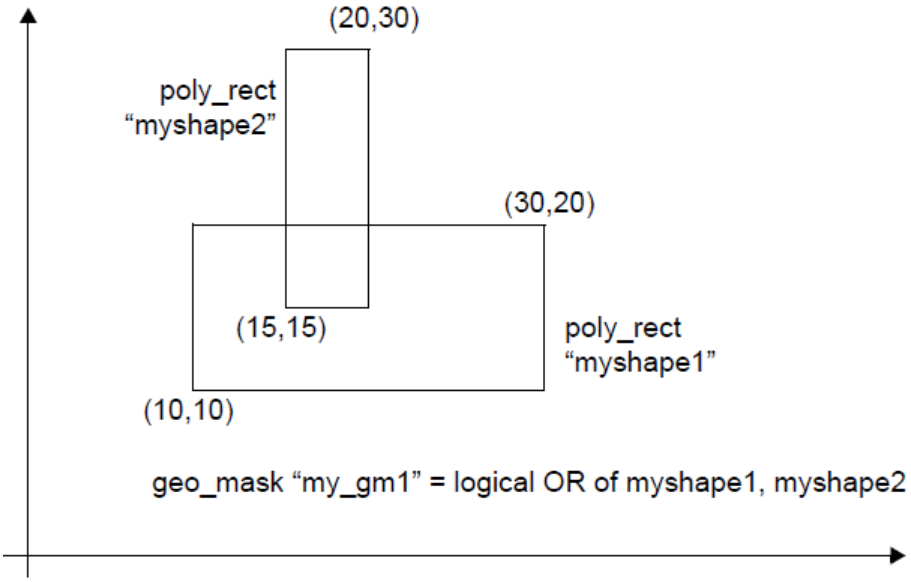
Creating poly_rect and geo_mask Objects

To create a poly_rect object, use the `create_poly_rect` command and specify the vertex coordinates: two points for a rectangle or three or more points for a polygon of any shape.

To create a geo_mask object, you can copy existing geometric shapes with the `create_geo_mask` command or perform operations on existing shapes with the `compute_polygons` or `resize_polygons` command.

In the following example, you create two overlapping rectangular poly_rect objects, perform a logical OR of these shapes to create a new geo_mask object, and copy the geo_mask object to create a rectilinear shape on the M2 layer.

Figure 27 Creating a geo_mask Object From Two poly_rect Objects




```
fc_shell> current_block
{libA:block1.design}
fc_shell> set myshape1 [create_poly_rect \
   -boundary {{10 10} {30 20}}]
{{{10 10} {30 20}}}
fc_shell> set myshape2 [create_poly_rect \
   -boundary {{15 15} {20 30}}]
{{{15 15} {20 40}}}
```

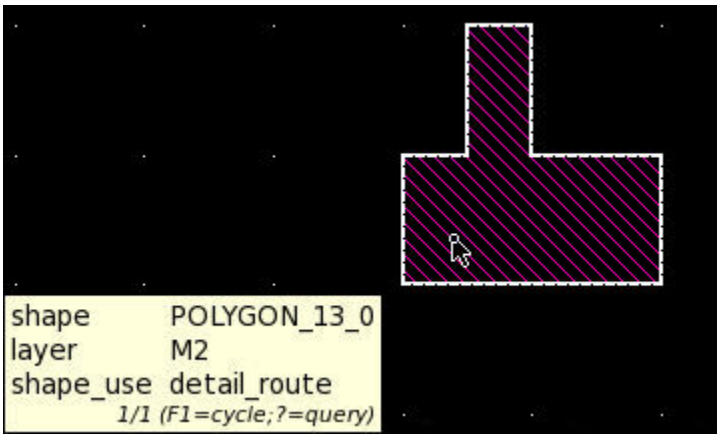
Feedback

```
fc_shell> set my_gm1 [compute_polygons \
  -objects1 $myshape1 -objects2 $myshape2 -operation OR]
...
fc_shell> report_attributes -application -class geo_mask $my_gm1
...
Design        Object          Type        Attribute Name    Value
-------------------------------------------------------------------------
block1        geo_mask        boolean     is_empty          false
block1        geo_mask        string      object_class      geo_mask
block1        geo_mask        collection  poly_rects        {20 40} ...
block1        geo_mask        int         shape_count       1
1

fc_shell> copy_to_layer -layer M2 -geo_masks $my_gm1
{POLYGON_13_0}
fc_shell> save_block
...
```

These commands create a rectilinear shape on the M2 layer, which you can display in the GUI as shown in the following figure.

Figure 28 geo_mask Object Copied to M2 Layer Shape



Alternatively, you can use the `compute_polygons` command directly with coordinate lists:

```
fc_shell> set my_gm1 [compute_polygons \
  -objects1 {{10 10} {30 20}} -objects2 {{15 15} {20 30}} \
  -operation OR]
...
fc_shell> copy_to_layer -layer M2 -geo_masks $my_gm1
{POLYGON_13_0}
```

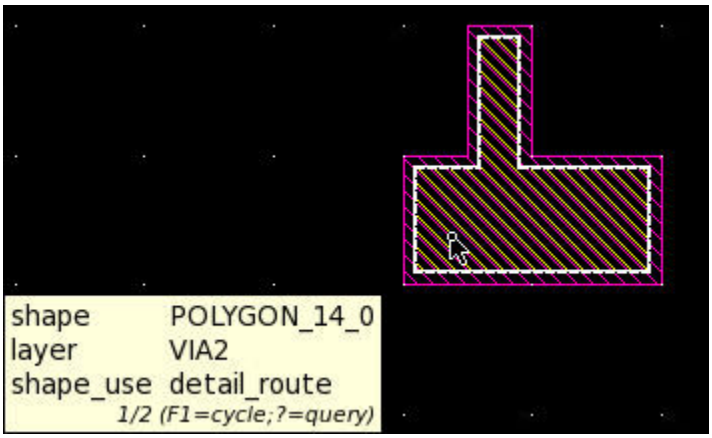
Another polygon manipulation command is the `resize_polygons` command, which pushes the edges of specified polygons inward or outward, creating a new geo_mask object. For example, the following script takes the existing shapes on the M2 layer, pushes

Feedback

the edges inward by 1.0 unit to make a new geo_mask object, and then converts the geo_mask object into a new shape on the VIA2 layer.

```
set sh2 [get_shapes -of_objects M2]
set gm2 [resize_polygons -objects $sh2 -size -1]
copy_to_layer -layer VIA2 -geo_masks $gm2
```

Figure 29 M3 Layer Shape Created by Resizing



See Also

- Polygon Manipulation
- Converting geo_mask Objects Into Functional Shapes

Converting geo_mask Objects Into Functional Shapes

The poly_rect and geo_mask type objects are abstract and have no direct functional purpose in a block. To create a functional shape such as a mask layer, blockage, or keepout region, you need to convert poly_rect or geo_mask objects into real objects by using a command such as `copy_to_layer`, `create_placement_blockage`, or `create_shape`.

The following example creates a geo_mask object and copies it to a shape on the M2 layer.

```
fc_shell> set pr2 [create_poly_rect -boundary {{10 10} {20 30}}]
...
fc_shell> set gm2 [create_geo_mask -objects $pr2]
...
fc_shell> copy_to_layer -layer M2 -geo_masks $gm2
{RECT_13_0}
```

Feedback

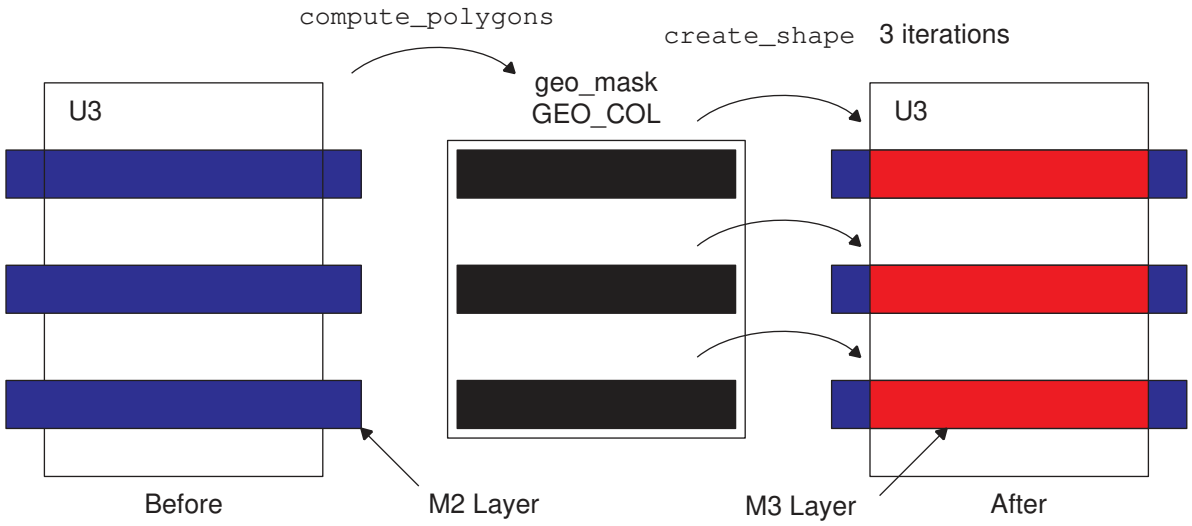
A geo_mask object is a collection of poly_rect objects. If the geo_mask object contains multiple distinct shapes, you can iterate through the collection by using the `foreach_in_collection` command, and convert each poly_rect object to a real object.

For example, the following script creates a geo_mask object named GEO_COL, which contains a set of shapes generated by the overlap of cell U3 with all shapes on layer M2. The script then iterates through the GEO_COL collection and converts each shape to a new shape on the M3 layer, as shown in Figure 30.

```
set SM [get_cells U3]
set GEO_COL [compute_polygons -operation AND \
  -objects1 $SM -objects2 [get_layers M2] ]

foreach_in_collection PR \
  [get_attribute $GEO_COL poly_rects] {
     set PRPL [get_attribute $PR point_list]
     create_shape -shape_type polygon -layer M3 -boundary $PRPL
}
```

Figure 30 Iterating Through a geo_mask Collection to Make New Shapes



See Also

- Polygon Manipulation
- Creating poly_rect and geo_mask Objects

Feedback

Undoing and Redoing Changes to the Design

You can undo (reverse) or redo (reapply) many of the operations performed in the tool.

To undo the most recent change, use the `undo` command:

```
fc_shell> create_block BLK1
...
fc_shell> create_block BLK2
...
fc_shell> list_blocks
Lib lib_A /path/libs/lib_A current
  *>  0 BLK1.design May-01-20:28
  *>  0 BLK2.design May-01-20:28 current
2
fc_shell> undo
1
fc_shell> list_blocks
Lib lib_A /path/libs/lib_A current
  *>  0 BLK1.design May-01-20:28 current
1
fc_shell> undo
1
fc_shell> list_blocks
Lib lib_A /path/libs/lib_A current
0
```

To redo the most recent operation reversed by an `undo` command, use the `redo` command:

```
fc_shell> redo
1
fc_shell> list_blocks
Lib lib_A /path/libs/lib_A current
  *>  0 BLK1.design May-01-20:28
1
fc_shell> redo
1
fc_shell> list_blocks
Lib lib_A /path/libs/lib_A current
  *>  0 BLK1.design May-01-20:28 current
  *>  0 BLK2.design May-01-20:28
2
```

You can only undo commands that change the design database, such as `create_block`. You cannot undo commands that merely change the tool context, such as `current_design`. Also, some types of design changes cannot be reversed.

To determine whether you can undo the effects of a particular command:

```
fc_shell> get_undo_info -command create_block
Information: The command 'create_block' is undoable. (UNDO-014)
1
fc_shell> get_undo_info -command current_lib
Information: The command 'current_lib' is independent of the undo system.
(UNDO-013)
2
```

For more information, see

- Undoing or Redoing Multiple Changes
- Disabling or Limiting the Undo Feature

Undoing or Redoing Multiple Changes

To undo or redo multiple changes in a single command, use the `-levels` option:

```
# Reverse the 3 most recent changes
fc_shell> undo -levels 3
3

# Reapply the 2 most recent reversed changes
fc_shell> redo -levels 2
2
```

To redo all recent changes and restore the design to the most recent available state:

```
fc_shell> redo -all
5
```

The number returned by this command (5 in this example) indicates the number of changes redone.

The tool maintains an internal *undo list* to keep track of the most recent design changes that can be reversed. If you plan to make multiple changes that you might want to undo later, you can set an *undo marker* in the list, and later undo all of changes back to the point at which you set the marker:

```
fc_shell> create_block block1
...
fc_shell> create_undo_marker my_marker
...
fc_shell> create_block block2
...
fc_shell> create_block block3
...
fc_shell> undo -marker my_marker
2
```

Feedback

```
fc_shell> list_blocks
Lib lib_B /path/libs/lib_B current
  *>  0 block1.design May-02-21:08 current
1
```

Conversely, if you plan to undo multiple recent changes that you might want to redo later, you can set an undo marker, and later redo all of the changes forward to the point at which you set the marker:

```
fc_shell> create_block block1
...
fc_shell> create_block block2
...
fc_shell> create_block block3
...
fc_shell> create_undo_marker mrkA
...
fc_shell> undo
...
fc_shell> undo
...
fc_shell> list_blocks
Lib lib_A /path/libs/lib_A current
  *>  0 block1.design May-02-21:18 current
1
fc_shell> redo -marker mrkA
2
fc_shell> list_blocks
Lib lib_A /path/libs/lib_A current
  *>  0 block1.design May-02-21:18 current
  *>  0 block2.design May-02-21:18
  *>  0 block3.design May-02-21:18
3
```

Note:

If you run a command that is not reversible, the tool clears the undo list and you can no longer reverse previous operations.

To get information about the current undo list:

```
fc_shell> get_undo_info
is_enabled true is_undoable true undo_cmd_name "create_block"
is_redoable false redo_cmd_name "" depth 4 max_depth 100
memory 104747 max_memory 1000000000
```

Feedback

Disabling or Limiting the Undo Feature

The undo feature uses runtime and memory to maintain the design history. If you do not need this feature, you can disable it:

```
fc_shell> set_app_options -name shell.undo.enabled -value false
shell.undo.enabled false
```

To disable the undo feature for one or more commands or to execute a block of commands as a single “undoable” unit, use the `eval_with_undo` command.

By default, the tool stores no more than 100 changes and uses no more than 1GB of memory to maintain the history of changes for the undo command. You can optionally decrease these limits to reduce runtime and memory usage:

```
fc_shell> set_app_options -name shell.undo.max_levels -value 50
shell.undo.max_levels 50
fc_shell> set_app_options -name shell.undo.max_memory \
   -value 500000000
shell.undo.max_memory 500000000
```

Conversely, you can increase these limits to handle a larger undo history.

Schema Versions

The NDM database infrastructure is periodically updated to support new database features. Each new version of the database is called a “schema” and each schema has a version number. When you save a design library with the `save_lib` command or save a cell library with the `commit_workspace` command, the library information is stored under the tool’s current schema.

To get a report of tool and schema version numbers and tool compatibility, use the `report_versions` command:

```
fc_shell> report_versions
J-2014.06
  ICC2 schema revision: 1.0
...

L-2016.03
   ICC2 schema revision: 1.165
   Compatibility:
        - ICV K-2015.12-SP2
        - PrimeRail K-2015.12-SP2

   Includes Support for:
        - Conn view for rail analysis
...
```

Feedback

To determine the schema number of a library, open the library and use the `get_attribute` command to report the library's `read_from_schema_version` attribute:

```
fc_shell> get_attribute -objects [current_lib] \
  -name read_from_schema_version
1.165
fc_shell> get_attribute -objects [get_libs tech32hvt] \
  -name read_from_schema_version
1.063
```

A schema number such as "1.165" consists of a major version number before the decimal point ("1") and a minor version number after the decimal point ("165").

NDM libraries are backward-compatible, but not directly forward-compatible. In other words, you can read in an older library with a newer tool, which updates the library from the older schema to the current one. However, you cannot always directly read a newer library with an older tool.

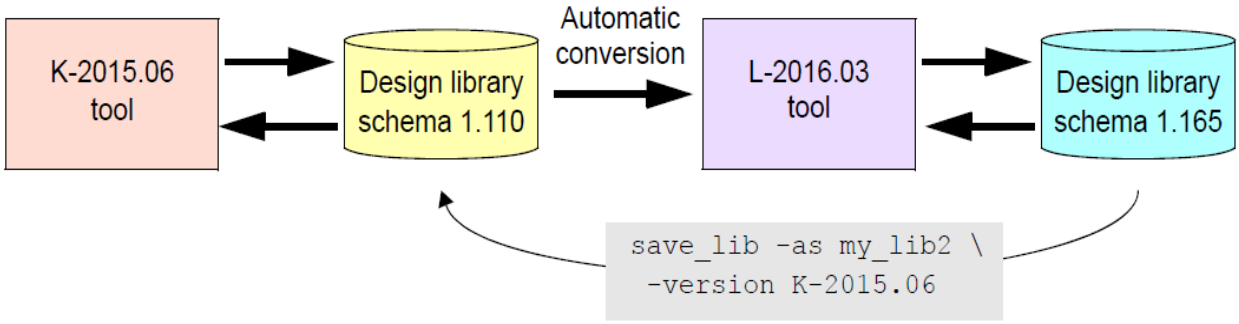
Service pack releases of the tools (such as L-2016.03-SP2) are generally both forward and backward compatible with each other and with the baseline release (such as L-2016.03).

When you save a design library, the tool saves it under the current schema by default. To make a newer library compatible with an older tool, open the library in the newer tool and then save it under a new name by using the `save_lib` command with the `-version` option:

```
fc_shell> current_lib
{my_lib}
fc_shell> save_lib -as my_lib2 -version K-2015.06
...
```

Then the new library can be opened by the older tool, as shown in the following figure.

Figure 31 NDM Database Schema Compatibility



If your newer design library has many blocks but you only need a few blocks in the older format, to reduce the conversion time and save disk space, copy the few needed blocks

Feedback

into a separate new design library, and then save the smaller design library in the older format.