Utilities Reference Manual

Version 2022.06-SP2, December 2022

SYNOPSYS®

Copyright and Proprietary Information Notice

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at https://www.synopsys.com/company/legal/trademarks-brands.html.
All other product or company names may be trademarks of their respective owners.

Free and Open-Source Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

www.synopsys.com

Feedback

Contents

1. **INTRODUCTION** 5

2. **PyCell Studio Utilities** 6
   - cnpy 6
   - cnpy_nogui 9
   - cnexp 10
   - cnhelp 16
   - cnversion 22
   - cndbg 24
   - pyros 25

3. **OpenAccess Library Tools** 27
   - cngenlib 27
   - cnbundle 34
   - cnrmlib 36
   - cndmfiles 36
   - cnchecklocks 38
   - cnupdatepycell 39
   - cnstrmin 39
   - cnstrmout 40

4. **Santana Technology File Utilities** 42
   - cntechversion 42
   - cntechcheck 44

5. **Conversion Programs** 47
   - cntechconv 47

Contents

cndispconv 48

1

INTRODUCTION

This reference manual documents all the different Synopsys command-line utility programs which are provided by PyCell Studio™. These command-line utility programs provide important parts of the functionality of these products. The documentation of these different command-line utility programs is combined together into a single reference manual to make it easier for the user to find documentation for a particular command-line utility program.

There are four major groups of command-line utility programs which can be used for the following tasks:

- Specialized versions of the standard Python interpreter used with the PyCell Studio Python API (**cnpy**, **cnpy_nogui**, **cnexp** and **cnhelp**), and PyCell Studio utilities (**cnversion**, **cntechversion**, **cnsendcmd**, **cndbg** and **pyros**).
- OpenAccess library creation and management (**cngenlib**, **cnbundle**, **cnrmlib**, **cndmfiles**, **cnchecklocks**, **cnupdatepycell**, **cnstrmin** and **cnstrmout**).
- Santana technology file versioning and checking (**cntechversion** and **cntechcheck**).
- Conversion programs for technology files and display files (**cntechconv** and **cndispconv**).

Some of these command-line utility programs are typically used with a single product, while other utility programs may be used with several products.

2

PyCell Studio Utilities

The PyCell Studio product provides several useful utilities which are used to interact with the PyCell Studio Python API. The **cnpy** and **cnpy_nogui** command-line utilities are special versions of the standard Python interpreter, which are customized to work with the Python API. The **cnexp** utility is another specialized version of the Python interpreter which is customized to allow quick and easy exploration of the Python API. The **cnhelp** utility provides detailed help messages on all classes and methods which are contained in the Python API.

The **cnsendcmd** utility is used to reload PyCell source code without exiting the main application. The **cnversion** utility displays information concerning the installation and the version of PyCell Studio which is installed on the host machine.

cnpy

Description: The **cnpy** utility is a stand-alone version of the standard Python interpreter. This Python interpreter is built upon the standard Python interpreter, with the addition of a number of specific classes and methods used for developing parameterized cells. This version of the Python interpreter ensures that the interface to the OpenAccess library has been pre-initialized, so that the Python API can be directly accessed. In addition, the interface to the Pyros layout viewing tool is also initialized. This allows this Pyros layout viewer to be invoked to display the layout as it is being generated by various commands contained in the Python API.

This **cnpy** Python interpreter program can be invoked as follows:

cnpy [<Python-Interpreter Options>]

[-h, --help]

[-V, --version]

Note that the **cnpy** utility program provides exactly the same command-line options as provided by the standard Python interpreter. This **cnpy** utility supports the current version of the standard Python interpreter, and will pass through any command-line options which are available for the current version of Python. In addition, note that the `-h` (or `--help`) option prints out a help message, which describes the command-line arguments which can be used with this utility program. The `-V` (or `--version`) option prints out version information for this utility program.

This **cnpy** Python interpreter also provides a `PyCell Explorer` module (cni.exp), which can be used to interactively create parameterized cell designs. This `PyCell Explorer` module is very useful for trying out different commands from the Python API, to see how they work. In addition, this `PyCell Explorer` module can be used in conjunction with the Pyros layout viewer, to interactively enter Python API commands to the Python interpreter, and see the resulting layout generated in the Pyros layout viewer.

There are only four commands (Python functions) defined in this `PyCell Explorer` module. Two of these commands are used to either create or open an OpenAccess library, while the other two commands are used to interactively create or inspect a DLO design object within the OpenAccess library. These commands can be described as follows:

createLib(string *libName*, string *libPath* = '', string *techLibName* = '', string *techFilePath*='', *coreDlos*=True) – creates a new OpenAccess library, using the *libName* parameter for the name of this new library. If the *libPath* parameter is specified, then this new library will be created in the specified directory. Otherwise, the library will be created in the current working directory. Note that a technology binding must be created for this OpenAccess library. This is done by means of either the *techLibName* or *techFilePath* parameters. One (and only one) of these parameters must be specified, or else an exception will be raised. The *techLibName* parameter specifies the name of the technology library which should be associated with this OpenAccess library, while the *techFilePath* parameter specifies the path to this technology library. Note that the *libPath* and *techFilePath* parameters can be specified using environment variables and/or tilde home directory notation. If the *coreDlos* parameter is False, then none of the `core` PyCells defined for a technology library will be included in the library. By default, all of these `core` PyCells (Contact, AbutContact, ArrayInstContact, RectC and ViaCut) are included in the library.

openLib(string *libName*, string *libPath* = '') – opens up an existing OpenAccess library, which has the name specified by the *libName* parameter. Note that this OpenAccess library should have been created by an earlier use of the "**createLib**()" function. If the optional *libPath* parameter is specified, then it should specify the file path location for this OpenAccess library; otherwise, the current working directory will be used by default. Note that this *libPath* parameter can be specified using environment variables and/or tilde home directory notation.

explore(string *cellName*, string *viewName* = '', Bool *showGui* = False, Bool *overwrite*=False) – creates a new DLO design object in the open OpenAccess library, and starts a new Python top-level interpreter. This top-level Python interpreter will have imported all of the various modules defined and used by the Python API (including cni.dlo, cni.constants, and cni.aliases modules). The *cellName* parameter must be used to specify the name of the cell being created in the OpenAccess library. If the *viewName* parameter is not specified, then `layout` will be used as a default view name for this DLO design object. If the *showGui* parameter is set to True, then the viewer tool will be invoked. If the *overwrite* parameter is set to False, and a DLO design object already exists in the OpenAccess library, then the user will be asked if it can be deleted. Note that the

OpenAccess library should have already been created or opened through the use of the **createLib**() or **openLib**() commands; otherwise, an exception will be raised.

inspect(string *cellName*, string *viewName* = '', Bool *showGui* = False, Bool *update*=False) – explores an existing DLO design object in the open OpenAccess library, and starts a new Python top-level interpreter. This top-level Python interpreter will have imported all of the various modules defined and used by the Python API (including cni.dlo, cni.constants, and cni.aliases modules). The *cellName* parameter must be used to specify the name of the cell being inspected in the OpenAccess library. If this *cellName* does not exist in the open OpenAccess library, then an exception will be raised. If the *viewName* parameter is not specified, then `layout` will be used as a default view name for this DLO design object. If the *showGui* parameter is set to True, then the viewer tool will be invoked. If the *update* parameter is set to True, then the existing DLO design object in the OpenAccess library can be updated. Note that the OpenAccess library should have already been created or opened through the use of the **createLib**() or **openLib**() commands; otherwise, an exception will be raised.

When this **cnpy** utility program is invoked, then an interactive Python interpreter command-loop environment will be entered. The usual approach is to import the “PyCell Explorer” module (cni.exp), and then set up the environment for exploring the capabilities of the Python API. This interactive command-loop environment should look like the following:

```
$ cnpy

...

Python 2.5.1 (r251:54863, May 1 2007, 11:29:40)

[GCC 3.2.3 20030502 (Red Hat Linux 3.2.3-52)] on linux2

Type "help", "copyright", "credits" or "license" for more information.

>>> import cni.exp

Welcome to

_____ _______ _____ _______ __
/ __ \__ __/ ____/__ / / / / ____/ _______ / /___ _________ ___
/ /_/ / / / / / / / _ \/ / / / __/ | |/_/ __ \/ / __ \/ ___/ _ \ _/
/ _____/ /_/ / /___/ __/ / / / / /_____> </ /_/ / / /_/ / / / / __/ /
/_/ \__, /\_____/\___/_/_/ /_/ /_____/_/|_/ .___/_/\_____/_/ \___/_/
/_____/ /_/

...
```

Type 'openLib(libName)' to open a library and 'explore(dloName)' to interactively explore the generation of a DLO. Use 'createLib(...)' to create new libraries.

Try 'help(openLib)', 'help(createLib)', or 'help(explore)' to find out more about how to use these functions.

>>> openLib('MyPyCellLib')

>>> explore('MyCell', showGui=True)

Start exploring MyPyCellLib/MyPyCell ...

MyPyCell> rect1 = Rect(Layer('metal1'), Box(0,0,1,1))

MyPyCell> rect1.getLayer()

Layer('metal1')

MyPyCell> rect1.getBBox()

Box(0,0,1,1)

MyPyCell>

Examples:

typical usage is to import cni.exp module,

and then use Python API commands

```
import cni.exp
createLib('MyPyCellLib', '/home/user/MyPyCellLib',
   techFilePath = $CNI_ROOT/tech/cni130/santanaTech/Santana.tech)
explore('MyPyCell', showGui=True)
```

now various Python API commands can entered,

and generated layout will be displayed by Pyros

---

cnpy_nogui

Description: The **cnpy_nogui** utility is a stand-alone version of the standard Python interpreter. In addition, this version of the Python interpreter ensures that the interface to the OpenAccess library has been pre-initialized. This is done, so that the Python API can be directly accessed. However, unlike the **cnpy** utility, this **cnpy_nogui** utility does not initialize the interface to the Pyros layout viewing tool. Thus, this **cnpy_nogui** utility should be used whenever there is no need to make use of the Pyros layout viewing tool

to view the layout which is generated by any commands in the Python API. Otherwise, the functionality of this **cnpy_nogui** utility is exactly the same as the **cnpy** utility. This **cnpy_nogui** Python interpreter program can be invoked as follows:

cnpy [<Python-Interpreter Options>]

[-h, --help]

[-V, --version]

Note that this **cnpy_nogui** utility program provides exactly the same command-line options as are already provided by the standard Python interpreter. This **cnpy_nogui** utility supports the current version of the standard Python interpreter, and will pass through any command-line options which are available in the current Python version. In addition, note that the `-h` (or `--help`) option prints out a help message, which describes the command-line arguments which can be used with this utility program.

The `-V` (or `--version`) option prints out version information for this utility program.

This **cnpy_nogui** Python interpreter utility also provides the same `PyCell Explorer` module (cni.exp) which is provided by the **cnpy** utility, which can be used to interactively create parameterized cell designs. However, the `explore` command cannot bring up the Pyros layout viewing tool to view the generated layout. This `PyCell Explorer` module can be used to interactively create parameterized cell designs, as well as try out different commands from the Python API. This `PyCell Explorer` module contains two commands to either create or open an OpenAccess library, and two commands to interactively create or inspect a DLO design object within the OpenAccess library.

Examples:

typical usage is to import cni.exp module,

and then use Python API commands

```
import cni.exp
createLib('MyPyCellLib', '/home/user/MyPyCellLib',
   techFilePath = $CNI_ROOT/tech/cni130/santanaTech/Santana.tech)
explore('MyPyCell')
```

now various Python API commands can entered

---

cnexp

Description: The **cnexp** utility is a customized version of the Python interpreter, which has been customized to work with the Python API. All of the classes and methods defined by the Python API are automatically included in this **cnexp** Python interpreter utility. In addition, the `PyCell Explorer` module and the Pyros layout viewing tool are automatically loaded by this **cnexp** utility. Thus, once this **cnexp** utility has been started,

the user can simply enter commands from the Python API and then immediately view the resulting layout which is generated in the Pyros layout viewing tool. This **cnexp** program can be invoked as follows:

cnexp [--libname=LIBNAME]

[--libpath=LIBPATH]

[--cellname=CELLNAME]

[--techfile=TECHFILE]

[--techlib=TECHLIB]

[--no_core_dlos]

[-c, --nogui]

[--opengl]

[--defstyle]

[-h, --help]

[-V, --version]

There are no required arguments for this **cnexp** utility program; there are only a number of options which can be used to customize the environment for the Python interpreter. The `--libname` option can be used to specify the name of the OpenAccess library which should be created to store the design information created by the Python API calls used with this utility. By default, the OpenAccess library name `MyTestingLib` is used. The `--libpath` option can be used to specify the file path location of this OpenAccess library; by default, the directory `/tmp/MyTestingLib` is used. The `--cellname` option specifies the name of the design to be created in the OpenAccess library; by default, the name `MyTestingCell` is used. The `--techfile` option can be used to specify the location of the Santana technology file which should be used to create a technology binding for the OpenAccess library; by default, the $CNI_ROOT/tech/cni130/santanaTech/Santana.tech" technology file is used. In addition, the `--techlib` option can also be used to specify the name of an OpenAccess technology library which should be used to set the technology binding for the OpenAccess library.

Note that if this `--techlib` option is used, then it will override either the default technology file or any technology file specified by use of the --techfile" option.

If the `--no_core_dlos` option is specified, then none of the `core` PyCells which are defined for a Santana technology library will be included in the OpenAccess library. These `core` PyCells are the Contact, AbutContact, ArrayInstContact, RectC and ViaCut PyCells. By default, these `core` PyCells are included in the OpenAccess library, since they are fundamental PyCells used by the Python API and the `Smart Objects`. This `--no_core_dlos` option is only meaningful when the `--tech_file` option is used. The –c (or

--nogui) option can be used to run this utility in command-line mode, where Pyros is not invoked and no Graphical User Interface is available.

The `--opengl` option specifies that OpenGL should be used for rendering, while the related `--defstyle` option specifies that the default style be used for widget rendering. This option is typically used to invoke old X11 rendering which is used by VNC-type tools (such as Citrix). The `-h` (or `--help`) option prints out a help message, which describes the command-line arguments which can be used with this utility program. The `-V` (or `--version`) option prints out version information for this utility program.

Note that if the CNI_DISPLAY_DIR environment variable which is used by the Pyros layout viewing tool is not set, then this **cnexp** utility will automatically set it to the Santana display file corresponding to the technology file setting. By default, this will be the $CNI_ROOT/tech/cni130/santanaDisplay display file setting. If the technology file setting is not a supplied `Santana.tech` technology file, then no default setting will be made for this CNI_DISPLAY_DIR environment variable.

This **cnexp** utility automatically loads the `PyCell Explorer` module (cni.exp) on startup, and is set to explore the specified design, by executing the following two commands:

createLib(LIBNAME, LIBPATH, techFilePath=TECHFILE)

explore(CELLNAME, showGui=True)

In the case that the **cnexp** utility is invoked without any arguments, then all default values will be used when this `PyCell Explorer` module is loaded, as follows:

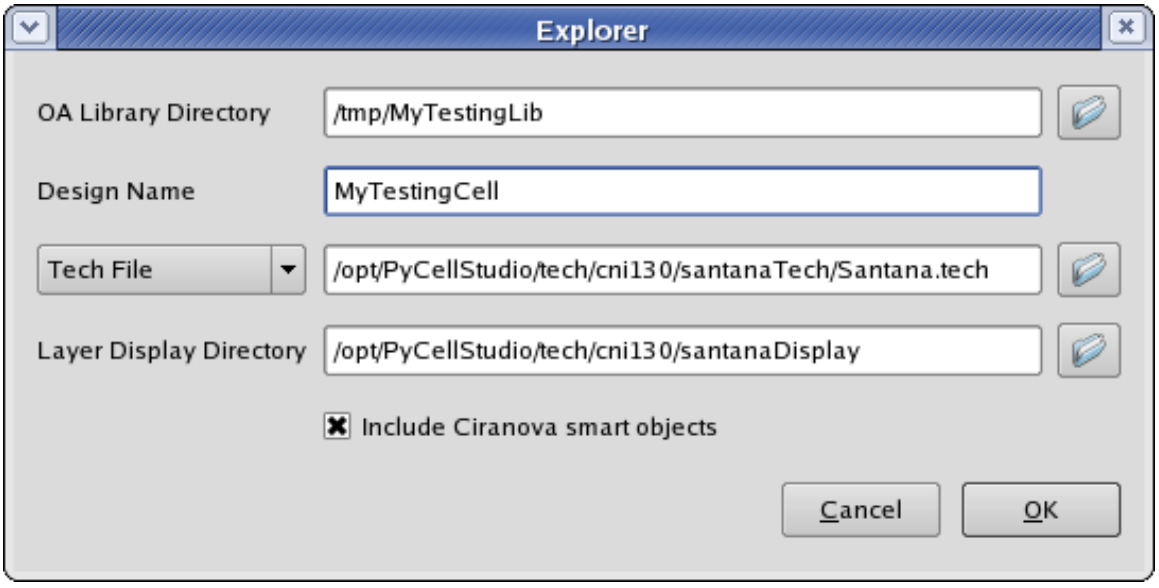
createLib(`MyTestingLib`, `/tmp/MyTestingLib`,

techFilePath=$CNI_ROOT/tech/cni130/santanaTech/Santana.tech)

explore(`MyTestingCell`, showGui=True)

Feedback

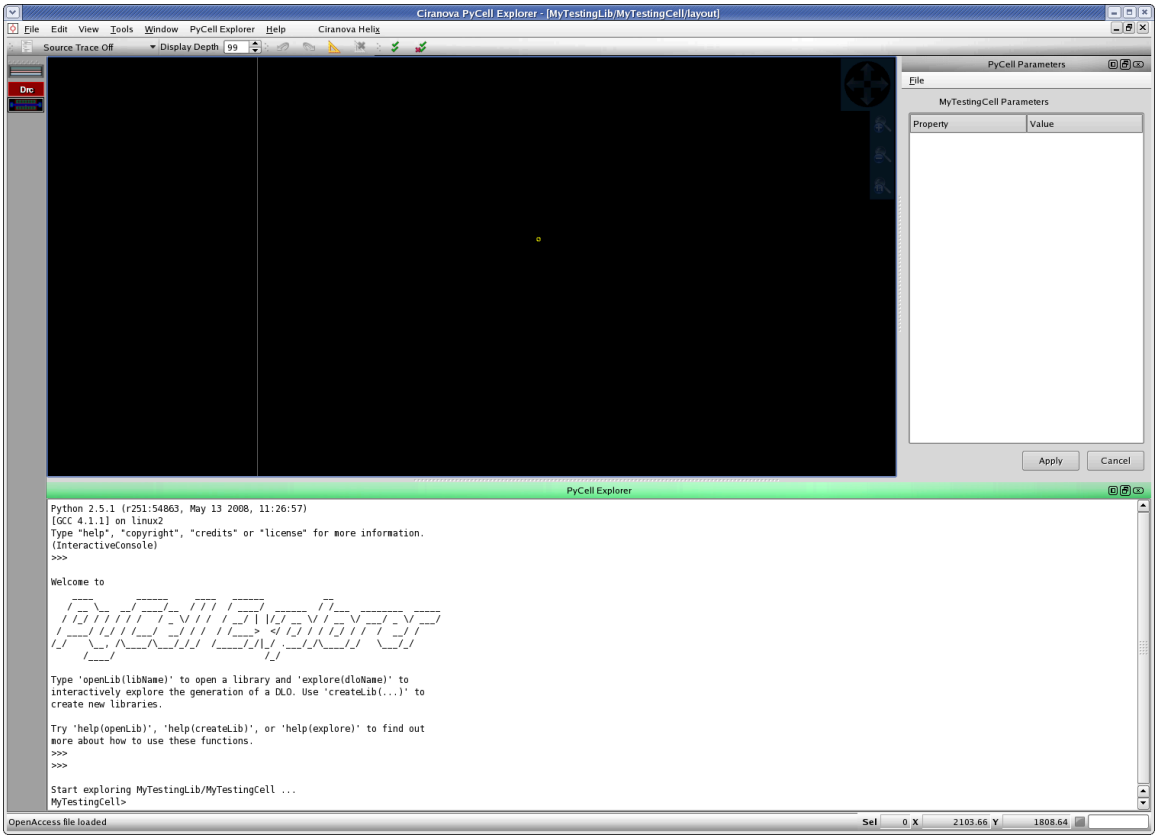
When this **cnpy** utility program is invoked, then the following dialog will be presented to the user, displaying the environment to be used for setting up the PyCell Explorer:



After this dialog is OK-ed, then an interactive Python interpreter command-loop environment will be entered using a Python console docking window, along with a Pyros

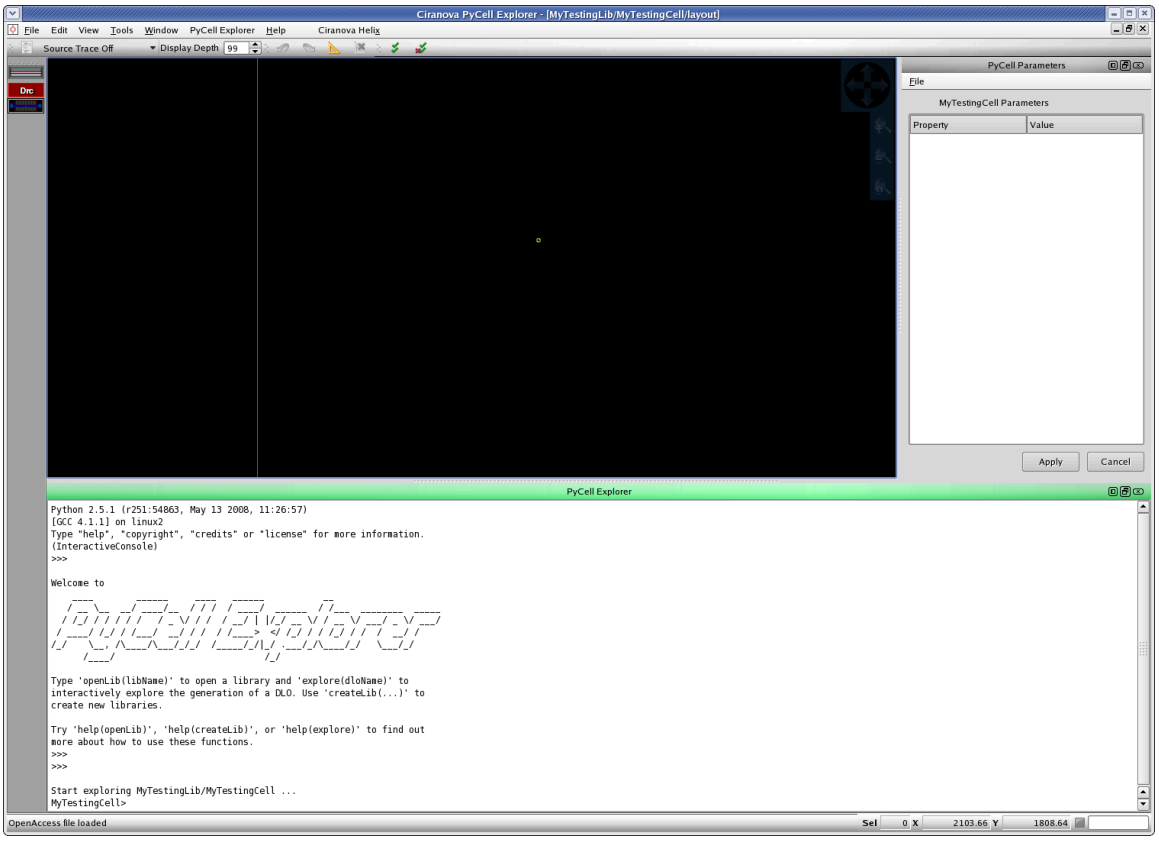
Feedback

layout viewing window to display the generated layout. This interactive command-loop environment should look like the following:



Feedback

Note that we have typed in the Python `Rect()` command to create a rectangle shape on the ‘metal1’ layer. This rectangle object will then be automatically displayed in the Pyros viewing window, shown as follows:



As different Python API commands are interactively entered by the user, then the generated layout will automatically be displayed in the Pyros viewing window. This approach makes it very easy to investigate the capabilities of the many different classes and methods which are contained in the Python API.**Examples:**

Simply invoke cnexp PyCell Explorer tool,

and then enter Python API commands;

generated layout will be displayed by Pyros.

now various Python API commands can entered,

such as the following:

```
rect1 = Rect(Layer('metal1'), Box(0, 0, 1, 1))
rect1.getLayer()
rect1.setLayer(Layer('metal2'))
rect2 = rect1.clone()
```

Feedback

```
rect2.fgPlace(Direction.EAST, rect1)
group1 = self.makeGrouping()
```

cnhelp

Description: The **cnhelp** utility provides help information about the Python classes and methods which are defined in the Python API. More specifically, this help information is provided in the form of standard Python help messages for user-specified class and method names. When this **cnhelp** utility program is invoked, it will enter an interactive environment. As the user enters a particular class name or method name, then help information will be generated for the specified class or method name. The user can exit this interactive command loop at any time, by simply typing control-D.

This **cnhelp** utility program is usually invoked without any command-line arguments. However, command-line arguments are provided to restrict the amount of help information, as well as help and version information, as follows:

cnhelp [-s, --short]

[-h, --help]

[-V, --version]

The `-s` (or `--short`) option is used to reduce the amount of help information which is generated; when this option is used, then no help information concerning inherited classes and methods will be output. Once the basic class inheritance structure for the Python API is understood, then this complete listing of all information concerning inherited classes and methods may be unnecessary. The `-h` (or `--help`) option prints out a help message, which describes the command-line arguments which can be used with this utility program. The `-V` (or `--version`) option prints out version information for this utility program.

Note that if no command-line arguments are specified when this **cnhelp** utility program is invoked, then an interactive command-loop environment will be entered, and the user will be prompted to enter the name of a Python class or method for which help information should be provided. If there is no help information available for the specified name, then an error will be generated.

Note that this **cnhelp** utility program will provide help information for the standard Python classes and methods, as well as help information for the Python API classes and methods which have been built on top of the standard Python language and interpreter. Also note that any method names need to be specified using the name of the class for which the specified method is defined. For example, the `rotate90` method name for the Rect class defined by the Python API needs to be fully specified as `Rect.rotate90`. As another example, the `sort()` method for the standard Python `list` class would need to be fully specified as `list.sort`.

Feedback

If a class name is entered, then a complete description of the class will be provided, including a description of all available methods which are defined for the specified class name. This includes all methods defined for this class, as well as all methods which are inherited by this class from one or more base classes. Each of these methods will be listed, along with a brief description of the parameters and usage for the method.

If a method name is entered, then only a brief description of the parameters and usage for that method will be generated for the help information.

Note that special handling of the input entered at the `Topic?` prompt is provided as a convenience. If no name is entered (just a return), then the full list of available class names will be displayed. If a Python regular expression (enclosed in double quotes) is entered, then a list of class names which satisfy the regular expression will be displayed. For example, `Via` will display all class names containing `Via`, `^Via` will display only those class names starting with `Via`, and `Via$` will display only those class names ending with `Via`. Also note that this regular expression matching ignores case.

When this **cnhelp** utility program is invoked without any command-line arguments, or with only the `-s` arguments, then an interactive command-loop environment will be entered, and the user will be prompted for the name of a Python class or method for which help information should be provided. This interactive command-loop environment should look like the following:

$ cnhelp

...

Use Ctrl-D (i.e. EOF) to exit.

Topic? list.remove

Help on method_descriptor:

remove(...)

L.remove(value) -- remove first occurrence of value

Topic? Instance.flatten

Help on method flatten in module cni.pvt.cnPyDloApi:

flatten(*args, **kwargs) unbound cni.dlo.Instance method

flatten(self) -> Grouping

Flatten Instance to primitive Shapes.

Return Grouping with same name containing Shapes.

Topic?

Feedback

$

Examples:

invoke cnhelp in an interactive command-loop

```
$ cnhelp
...
Use Ctrl-D (i.e. EOF) to exit.
Topic?
```

get help information for basic Python `list` class

```
Topic?
list
Help on class list in module __builtin__:
class list(object)
 |  list() -> new list
 |  list(sequence) -> new list initialized from sequence's items
 |
 |  Methods defined here:
 |
 |  __add__(...)
 |      x.__add__(y) <==> x+y
 |
 |  __contains__(...)
 |      x.__contains__(y) <==> y in x
```

|

get help for the `list` class `sort` and `append` methods

```
Topic?  list.sort
Help on method_descriptor:
sort(...)
    L.sort(cmp=None, key=None, reverse=False) -- stable sort *IN PLACE*;
    cmp(x, y) -> -1, 0, 1
Topic?  list.append
Help on method_descriptor:
append(...)
    L.append(object) -- append object to end
```

get help for Python API `Rect` class

```
Topic?  Rect
Help on class Rect in module cni.dlo:
class Rect(Shape)
 |  Rectangle Shape.
 |
 |  Rect(Layer layer, Box box) -> Rect
 |  Construct a Rect with layer and box.
 |
 |  Method resolution order:
 |      Rect
```

```
 |      Shape
 |      PhysicalComponent
 |      IPhysicalComponent
 |      ManagedObj
 |      __builtin__.object
 |
 |  Methods defined here:
 |
 |  __init__(...)
 |      x.__init__(...) initializes x; see x.__class__.__doc__ for
signature
 |
 |  __repr__(...)
 |
 |  getBottom(...)
 |      getBottom(self) -> Coord
 |
 |  getCoord(...)
 |      getCoord(self, Direction dir) -> Coord
```

get help for `Rect` class `rotate90` and `mirrorX` methods

```
Topic?  Rect.rotate90
Help on method_descriptor:
rotate90(...)
    rotate90(self, Point origin=None) -> self
Topic?  Rect.mirrorX
Help on method_descriptor:
mirrorX(...)
    mirrorX(self, Coord yCoord=0) -> self
```

now exit from the interactive command-loop using control-D

```
Topic?
$
cnsendcmd
```

Description: The **cnsendcmd** utility is used to send commands to a target application which makes use of the PyCell plugin capability, and has enabled `listening` mode to be able to read and process these commands. The pre-defined commands which can be sent by this **cnsendcmd** utility are one of the following: `info`, `reload` and `eval`. These three pre-defined commands also have specific command options which can be specified by this **cnsendcmd** utility. A typical way in which this **cnsendcmd** utility can be used is in a scenario such as the following: The PyCell Explorer tool (invoked by the **cnexp** utility) is being used to view and debug the layout generated by a PyCell in a library. After examining the layout for errors, then the PyCell source code is edited and modified to fix an error in the layout generation. Instead of exiting the PyCell Explorer tool and using the **cngenlib** utility to rebuild the entire PyCell library, it may be easier to simply use this **cnsendcmd** utility to reload the modified PyCell source code for the PyCell, so that this modified version can be tested for correctness within the same PyCell Explorer

tool session. Thus, this **cnsendcmd** utility can be used to provide a more user-friendly environment for PyCell testing and verification.

Prior to using this **cnsendcmd** utility, the target application must host the provided PyCell plugin capability. In addition, this target application must also set one or more environment variables which are used by the PyCell plugin when interacting with the **cnsendcmd** utility. These environment variables should be set before the target application is run. These environment variables are the following:

CNI_LISTEN_CMDS: comma-separated list of commands (required)

CNI_LISTEN_PORT: port number to listen to (default is 9981)

CNI_LISTEN_MODE: listening mode for Plugin (default is `Console`)

The CNI_LISTEN_CMDS environment variable should be a comma-separated list of commands from the **cnsendcmd** utility which should be listened to by the PyCell Plugin. This list should be composed of the valid command names `info`, `reload` and `eval`; if `all` is used, then the Plugin will listen for all of these valid commands. Note that this CNI_LISTEN_CMDS environment variable is required to be specified, while the other two environment variables are optional (and will use default values, if not specified). The CNI_LISTEN_PORT is used to specify the port which should be used by the Plugin to listen for commands sent by the **cnsendcmd** utility; by default, port 9981 will be used. The CNI_LISTEN_MODE environment variable is used to set the listening mode for the Plugin; it can be ”Console” or `Quiet` or it can be the file name path for a log file. By default, the `Console` listening mode is used by the PyCell Plugin when listening for commands which are sent by the **cnsendcmd** utility.

It is also necessary to ensure that the PyCell code has first been loaded by the target application; otherwise, an error message will be generated when the **cnsendcmd** utility is used to reload the updated PyCell code. For example, in the case of the PyCell Explorer, it is simply sufficient to change a parameter on a PyCell being viewed, and then using the `Apply` button; this forces the PyCell code to be loaded and re-evaluated.

Note that if any kind of PyCell caching is being used in conjunction with this **cnsendcmd** utility program, then this command may not work properly. In particular, the **cnsendcmd** code reloading capability is meant to be used for development and debugging purposes, and is not meant to be used with PyCell caching.

This **cnsendcmd** utility program can be invoked as follows:

cnsendcmd <command> [<command-options>]

[-s HOST, --host=HOST]

[-p PORT, --port=PORT]

[-h, --help]

[-V, --version]

The <command> argument is the required name of the command which should be executed by the **cnsendcmd** utility. This command should be `info`, `reload` or `eval`. The optional <command-options> contains any arguments which should be specified for the particular command. In the case of the `info` command, there can be no arguments, in which case this command provides information about the target application's PyCell Plugin runtime environment. If the argument is <lib-name>, then information about the specific OpenAccess library is provided. If the argument is `<lib-name> <cell-name> [<view-name>]`, then the Python code package which is associated with the given cell in the OpenAccess library is returned. This Python code package can then be used with the `reload` command, in order to reload the Python code package. For the `reload` command, the required argument is <code-package>, in which case the **cnsendcmd** utility sends a command to the target application to reload the specified code package. For the `eval` command, the required argument is <expression-or-statement> string, in which case the **cnsendcmd** utility sends a command to the target application to evaluate the specified Python expression or statement string. The Python expression or statement is then evaluated by the target application. Note that this `eval` command provides the full power of the Python interpreter, so this command should be carefully used.

In the case of the `info` command, if the target application cannot provide the requested information, then an error message will be generated.

In the case of the `reload` command, if the specified Python code package is unknown or not yet loaded, then an error message will be generated.

In the case of the `eval` command, error messages may be generated, if the specified expression cannot be properly evaluated by the target application.

The `-s` (or `--host`) option is used to specify the target hostname, which is hosting the target listening application which should receive commands from this **cnsendcmd** utility. If this option is not used, then the target hostname defaults to `localhost`.

The `-p` (or `--port`) option is used to specify the port number to use on the target host; this is the port number which should be used to listen for commands which are sent by this **cnsendcmd** utility. If this option is not used, then the port number defaults to 9981.

The `-h` (or `--help`) option prints out a help message, which describes the command-line arguments which can be used with this utility program. The `-V` (or `--version`) option prints out version information for this utility program.

Examples:

These examples assume that `cnexp` (PyCell Explorer)

is the target application and it is being used to

view PyCells in a PyCell library named `MyPyCellLib`

Also ensure that PyCell code is loaded, by forcing

Feedback

```
# PyCell evaluation on PyCell currently being viewed;
# simply change parameter value and hot "Apply" button.
# first obtain information about the listening application
cnsendcmd  info
# now obtain information about PyCell library
cnsendcmd  info MyPyCellLib
# obtain the name of the Python code package for Nmos
cnsendcmd  info MyPyCellLib Nmos
# now use this Python code package to reload the PyCell
# source code for the Nmos cell in this library
cnsendcmd  reload  pkg:MyPyCells
# Change parameter on Nmos PyCell in PyCell Explorer,
# to verify that updated PyCell code has been reloaded.
```

cnversion

Description: The **cnversion** utility displays information concerning the version of PyCell Studio which is currently installed on the host machine. In addition, this **cnversion** utility also performs basic checking of the PyCell Studio installation, and reports any problems which are found. In the cases where potential problems with the installation are detected, then descriptive warning messages will be generated.

There are four major display and checking steps which are performed by this **cnversion** utility, which can be summarized as follows:

- Print out the settings for several useful system environment variables, along with basic version information about the PyCell Studio installation. These environment variables include: CNI_ROOT, CNI_LICENSE_DIR, CNI_LOG_DEFAULT, OA_COMPILER, OA_HOME, OA_PLUGION_PATH and PYTHONHOME. Version information for the PyCell Studio installation and the Python installation are generated. In addition, information about the host computer is also displayed.

The values for the PATH, LD_LIBRARY_PATH and PYTHONPATH environment variables are displayed. Finally, information concerning the PyCell Studio OpenAccess version is displayed.

Feedback

- Check the Python version and installation, and report any problems (or potential problems). This checking includes: 1) check required Python version, 2) check PYTHONPATH environment variable is defined; 3) check PYTHONPATH contains all required entries; 4) check PYTHONPATH entries for correctness.
- Check that the PATH environment variable contains all of the required entries.
- Check that the LD_LIBRARY_PATH environment variable contains all of the required entries. Also check that all required PyCell Studio C++ libraries can be successfully loaded. In addition, check that all shared libraries can be resolved.

This **cnversion** utility program is usually invoked without any command-line arguments. However, command-line arguments are provided to display help and version information, as follows:

cnversion [-h, --help]

[-V, --version]

The `-h` (or `--help`) option prints out a help message, which describes the command-line arguments which can be used with this utility program. The `-V` (or `--version`) option prints out version information for this utility program.

Examples:

invoke cnversion to check PyCell Studio installation

```
$ cnversion
...
CNI_ROOT               = <root_directory>
CNI_LICENSE_DIR        = NOT_DEFINED
CNI_DISPLAY_DIR        = <root_directory>/tech/cni130/santanaDisplay
CNI_LOG_DEFAULT        = /dev/null
OA_COMPILER            = gcc411
OA_HOME                = NOT_DEFINED
OA_PLUGIN_PATH         = NOT_DEFINED
PYTHONHOME             = <root_directory>/plat_linux_gcc411_32/3rd
PyCell Studio version  = 4.3.1-L2
PyCell Studio built    = Jul 22 2009 17:22:02
Python version         = 2.5.1
Installed platform     = plat_linux_gcc411_32
Platform               =
Linux-2.6.9-67.0.15.ELsmp-x86_64-with-redhat-4-Nahant_Update_6
Hostname               = <host_name>.com
PATH:
    <root_directory>/plat_linux_gcc411_32/bin
    <root_directory>/plat_linux_gcc411_32/3rd/bin
    <root_directory>/plat_linux_gcc411_32/3rd/oa/bin/
     linux_rhel30_gcc411_32/opt
    <root_directory>/bin
    /bin
    /usr/bin
```

```
    /usr/local/bin

LD_LIBRARY_PATH:
    <root_directory>/plat_linux_gcc411_32/3rd/oa/lib/
     linux_rhel30_gcc411_32/opt
    <root_directory>/plat_linux_gcc411_32/3rd/lib
    <root_directory>/plat_linux_gcc411_32/lib
    <root_directory>/lib
    /lib64
    /lib
    /usr/lib64
    /usr/lib
    /usr/local/lib64
    /usr/local/lib
    /usr/X11R6/lib64
    /usr/X11R6/lib

PYTHONPATH:
    <root_directory>/plat_linux_gcc411_32/lib
    <root_directory>/plat_linux_gcc411_32/pyext
    <root_directory>/pylib
    <root_directory>/quickstart
PyCell Studio OpenAccess version:
Tool:           oa2strm         22.04.028
Library:        oaDesign        22.04.028       Wed Aug 27 00:10:08 2008
Library:        oaTech          22.04.028       Wed Aug 27 00:10:08 2008
Library:        oaDM            22.04.028       Wed Aug 27 00:10:08 2008
Library:        oaBase          22.04.028       Wed Aug 27 00:10:08 2008
Library:        oaUtil          22.04.028       Wed Aug 27 00:10:08 2008
```

Checking Python setup...

Passed.

Checking PATH...

Passed.

Checking LD_LIBRARY_PATH...

Passed.

Resolving shared library dependencies...

Passed.

---

cndbg

Description: The **cndbg** utility is an interactive debugging environment (IDE) tool, which can be used to debug PyCells. This IDE is integrated with the **pyros** layout viewing

tool, so that it provides an easy-to-use debugging environment. This **cndbg** interactive debugging tool is invoked as follows:

cndbg [--ldf=DESIGNS_FILE]

[--ld=LIB/CELL[/VIEW]]

[--py=PYTHON_FILE]

[--opengl]

[--defstyle]

[-h, --help]

[-V, --version]

Note that there are no required command-line options for this **cndbg** PyCell debugging environment utility. The `--ldf` option specifies that all designs listed in the DESIGNS_FILE file should be opened up and displayed; the syntax used for each line of this design file is `<Lib> <Cell> <View>`. The `--ld` option specifies that the specified design should be opened up and displayed. The syntax for this design name is `Lib/Cell/View`; if the View field is not used, then all views for the design will be displayed. The `--py` option specifies that the PYTHON_FILE Python initialization file should be executed. The `--opengl` option specifies that OpenGL should be used for rendering, while the related `--defstyle` option specifies that the default style be used for widget rendering. This option is typically used to invoke old X11 rendering which is used by VNC-type tools (such as Citrix). The `-h` (or `--help`) option prints out a help message, which describes the command-line arguments which can be used with this utility program. The `-V` (or `--version`) option prints out version information for this utility program.

This section only very briefly describes this **cndbg** interactive layout viewing tool utility. For more detailed information, please refer to the separate `Pyros Interactive Layout Viewer User Manual` reference manual, as well as the `PyCell Studio Tutorial` manual.

---

pyros

Description: The **pyros** utility is an interactive layout viewing tool, which can be used to view OpenAccess layout designs. It also provides a number of additional capabilities which are very useful in the development of parameterized cells. This **pyros** interactive layout viewing tool is invoked as follows:

pyros [--commands]

[--commandHelp=CMD]

[--rp=REPLAY_FILE]

[--rps=REPLAY_FILE]

[--ldf=DESIGNS_FILE]

[--ld=LIB/CELL[/VIEW]]

[--py=PYTHON_FILE]

[-h, --help]

[-V, --version]

Note that there are no required command-line options for this **pyros** layout viewing utility. The `--commands` option lists the commands which are currently supported by this layout viewing tool, while the `--commandHelp` option displays the help message for the specified command. Note that this **pyros** layout viewing tool is implemented using commands, so that all user actions can be interpreted as a sequence of one or more commands. Thus, a complete user session with **pyros** can then be stored as a command file. These command files (also called replay files) can then be replayed to recreate the user session. The `--rp` option is used to replay one of these command files. The `--rps` option is used to replay one of these command files in step-mode, where the commands in the replay file are executed one at a time; typing the character 'c' (for continue) at the user's keyboard will step from one command to the next. Note that if it is desired to replay a command file, then it should first be copied from its location in the `.cnitools` directory into another file location; this is required, so that this replay file will not get overwritten. The `--ldf` option specifies that all designs listed in the DESIGNS_FILE file should be opened up and displayed; the syntax used for this file is `<Lib> <Cell> <View>` for each line in this file. The `--ld` option specifies that the specified design should be opened up and displayed. The syntax for this design name is `Lib/Cell/View`; if the View field is not used, then all views for the design will be displayed. The `--py` option specifies that the PYTHON_FILE Python initialization file should be executed. The `-h` (or `--help`) option prints out a help message, which describes the command-line arguments which can be used with this utility program. The `-V` (or `--version`) option prints out version information for this utility program.

This section only very briefly describes this **pyros** interactive layout viewing tool utility. For more detailed information, please refer to the separate `Pyros Interactive Layout Viewer User Manual` reference manual.

3

OpenAccess Library Tools

The PyCell Studio product provides five utilities which are used to manage OpenAccess libraries. The **cngenlib** utility creates an OpenAccess library, which can then be used to store Python parameterized cells (`PyCells`). The **cnbundle** utility can be used to bundle Python code packages with an existing OpenAccess library. The **cnrmlib** utility safely removes an existing OpenAccess library from the file system. The **cndmfiles** utility manages the design management files which may be attached to an existing OpenAccess library. The **cnchecklocks** utility checks for any designs in an OpenAccess library which are currently locked by an active process.

cngenlib

Description: The **cngenlib** utility is used to create a new OpenAccess database library, which can be used to store Python parameterized cells, or to create the technology binding for an existing OpenAccess database library. In addition, this **cngenlib** utility can also be used to update an existing OpenAccess database library with different Python parameterized cell definitions, or to update the technology binding for an existing OpenAccess PyCell library. This **cngenlib** utility program can be invoked as follows:

cngenlib <code-pkg-ID> <oa-lib-name> <oa-lib-path>

```
[-c, --create ]

[-u, --update]

[-v, --view]

[--techfile = TECHFILE]

[--techlib = TECHLIB]

[--force_binary_techfile ]

[--create_techdb=yes|no ]

[--dispfile = DISPFILE]

[--dispdir = DISPDIR]

[--for_pycells=FOR_PYCELLS]
```

Feedback

```
[--viewcellfiles=VIEWCELLFILES]

[--no_core_dlos]

[--srctrace]

[--srctrace_depth=SRCTRACE_DEPTH]

[--bundle=BUNDLE]

[--optimize=OPTIMIZE]

[--libdef_file=LIBRARY_DEFINITION_FILE]

[--verbose]

[--python-2.5.1]

[--python-2.6.2]

[--python-3.8.6]

[ -h, --help ]

[ -V, --version]
```

The <code-pkg-ID> is the required specification of the Python package (or module) which contains the Python parameterized cell source code. A Python code package would be specified using the format `pkg:<python-package-path>`, where the `<python-package-path>` specifies the directory name for the Python source code package. This Python source code package name can either be a single name such as `MyPyCells` or can be a fully qualified name such as `company.analog.devices`. Note that the directory containing the actual Python source code for the parameterized cells must also be made available through the $PYTHONPATH environment variable.

A Python code module would be specified using the format `mod:<python-module-path>`, where the`<python-module-path>` specifies the single Python source code file which contains the Python source code module for the parameterized cells. Note that the Python package and module are very similar; the key difference is that a package is a directory containing several Python source code files, while a module is a single Python source code file. In addition, the Python package directory must contain a file named "__init__.py, which is an initialization file for the Python source code package. This `__init__.py` initialization file should contain a function named `definePcells`, which is used to define the PyCells to be contained in the specified OpenAccess library.

Note that the <python-package-path> should not be the name of any Python code module which is part of the standard Python distribution, or part of the Python API distribution (as documented in the Python API reference manual). For example, the name `code` should not be used as Python source code package name.

In addition to Python packages and modules, this <code-pkg-ID> argument can also be used to specify that the technology binding for the OpenAccess database library be created or updated. This is done by setting this <code-pkg-ID> to the string `tech:only`. In this case, no PyCells will be created, and only the technology binding will be updated.

The <oa-lib-name> argument is the required name of the OpenAccess database library which should be created or updated. The <oa-lib-path> is the required full file path for the location of this OpenAccess library in the file system. The `--create` option is used to indicate that the specified OpenAccess library should be created, while the `--update` option is used to indicate that an existing OpenAccess library should be updated. If the

`--create` option is used, and the specified OpenAccess library already exists under a different file path location, then an error message will be generated. If the `--update` option is used, and the specified OpenAccess library does not exist, then an error message will be generated. Note that these `--create` and `--update` options are mutually exclusive; one and only one of these options can be specified, otherwise an error message will be generated. Note that this <oa-lib-path> argument can be specified using environment variables and/or tilde home directory notation.

The `--update` option can be used to provide a technology library which does not contain a Santana technology file. Thus, PyCell Studio can be used with technology libraries which do not contain Santana technology files. Instead, the existing OpenAccess technology database will be accessed `on the fly` to obtain the technology information which is required to construct the equivalent Santana technology object. In this case, the <oa-lib-name> argument and the <oa-lib-path> argument are required, along with the `tech:only` package id. This updated technology library can then be used to compile libraries of PyCells.

Note that the `--update` option can also be used to conditionally compile PyCells which are not process portable, and should only be compiled for a particular technology. For example, the **cngenlib** command can be used to first compile a library of process portable PyCells for multiple technologies. Then the **cngenlib** command can then be used again with the `--update` option to compile the non process portable PyCells only for a specific technology. Alternatively, the Python API **Lib** class **getTech**() method can be used in the `__init__.py` initialization file to determine the name of the technology which is currently being compiled.

If the `--view` option is used, then the Pyros layout viewer will automatically be invoked to display the PyCells which will be created by this **cngenlib** command. As each PyCell is created, it will be displayed for viewing in this Pyros layout viewing tool. Alternatively, the `--viewcellfiles` option can be used to specify the specific list of PyCells which should be opened up and displayed for viewing by the Pyros layout viewing tool. By default, if the `--view` option is used, then all of the generated PyCells will be displayed for viewing. Note that if the `--view` option is not being used, then any settings for the `--viewcellfiles` option will simply be ignored.

The `--techfile` option is used to specify the file path for the location of a Santana technology file in the file system. This technology file will be used to set the technology binding for the newly created or existing OpenAccess library. Note that this file path option can be specified using environment variables and/or tilde home directory notation. The `--techlib` option is used to specify the name of an OpenAccess technology library which should be used to set the technology binding for the newly created or existing OpenAccess library. Note that these `--techfile` and `--techlib` options are mutually exclusive; one and only one of these options can be specified, otherwise an error message will be generated. Also note that if an OpenAccess library is being created, and neither of these two technology options is specified, then an error message will be generated.

If the `--create_techdb` option is set to `yes`, then the OpenAccess technology database tech.db will be generated; otherwise, this OpenAccess database file will not be generated. Note that this tech.db database file may be created by other EDA tools and already exist; in such cases, this option can be used so that the this tech.db file will not be overwritten. If the `--dispfile` option is used, then the specified path to the Santana display file is used to access the display packet assignments for Layer-Purpose Pairs which will then be written to the resulting tech.db database file. In a similar fashion, if the `--dispdir` option is used, then this directory location will be used to generate display packet assignments for the tech.db database file. Note that this `--dispfile` option can only be used when the `--techfile` option is also specified, while the `--dispdir` option is only meaningful for creating or updating a technology library (so that it cannot be used when the `--techlib` option is specified).

If the `--no_core_dlos` option is specified, then none of the `core` PyCells which are defined for a Santana technology library will be included in the OpenAccess library. These `core` PyCells are the Contact, AbutContact, ArrayInstContact, RectC and the ViaCut PyCells. By default, these `core` PyCells are included in the technology library, since they are fundamental PyCells which are used by the Python API and the `Smart Objects`. In addition, note that if the `--force_binary_techfile` option is used, then the ASCII technology file specified by the `--techfile` option will be used to generate a binary technology file for the OpenAccess library technology binding.

The `--for_pycells` option is used to specify a comma-separated list of cell names for the PyCells which should either be created or updated for this library. By default, all PyCells in the library are either created or updated, when this `--for_pycells` option is not used.

The `--srctrace` option is used to enable Python source code tracing for debugging purposes. In addition, the `--srctrace_depth` option can be used to set the tracing level depth for the Python debugging tool. If this --strctrace_depth” option is not used, then by default, the Python source tracing level depth is set to 4.

The `--bundle` option is used to bundle the Python source code package with the OpenAccess library, so that the Python code is stored with the generated PyCell library, and so does not need to be accessed through the $PYTHONPATH environment

variable, when the compiled library is used later. However, the directory containing the actual Python source code for the parameterized cells must be available through the $PYTHONPATH environment variable when this **cngenlib** utility program is run. The valid values for this option are `source`, `bytecode`, `encrypted` and `encrypted_source`.

The `source` option means to bundle both the Python source code and the unencrypted Python bytecode files into the library. The `bytecode` option means to bundle only the unencrypted bytecode files, while the `encrypyted` option means to bundle only the encrypted bytecode files. The `encrypted_source` option means to bundle encrypted Python source code files, after they are first compressed to save space. Note that no bytecode files will be bundled with this `encrypted_source` option, unlike the `source` option, in which both Python source and bytecode files are bundled. This `encrypted_source` option is useful to handle possible issues with Python versions, as bytecode files will be version specific. Also note that when the *.py file is not writable, then the copied code directory will be made temporarily writable, so that the resulting *.pyc file is writable.

Note that this `--bundle` option can still be used, even when there are only *.pyc Python compiled bytecode files available. If the `source` option is used, and there are any *.pyc files without associated *.py files, then a warning will be generated, and these *.pyc files will be added to the Python code bundle.

The `--optimize` option is used to control the level of optimization used when the Python bytecode is being bundled in the OpenAccess library. The valid values for this option are 0, 1 or 2, which correspond to no optimization, the Python `-O` optimization level and the Python `-OO` optimization level. Note that the `--view` option cannot be specified when this `--optimize` option is set to an optimization level of 1 or 2; otherwise, an error message will be generated. Also note that this `--optimize` option has no effect, unless the `--bundle` option is also specified.

The `--libdef_file` option is used to specify the library definition file that overwrites the default library definition file and gets loaded by the OpenAccess library.

The `-python-2.5.1` option is used to compile the Python source code package with the OpenAccess library by using Python 2.5.1.

The `-python-2.6.2` option is used to compile the Python source code package with the OpenAccess library by using Python 2.6.2. This is the default option if **cngenlib** does not give any other –python-* options.

The `-python-3.8.6` option is used to compile the Python source code package with the OpenAccess library by using Python 3.8.6.

The `--verbose` option is used to display the internal messages which are generated by the Python code bundling process. This option is only meaningful when the `--bundle` option is also specified.

The `-h` (or `--help`) option prints out a help message, which describes the command-line arguments which can be used with this utility program. The `-V` (or `--version`) option prints out version information for this utility program.

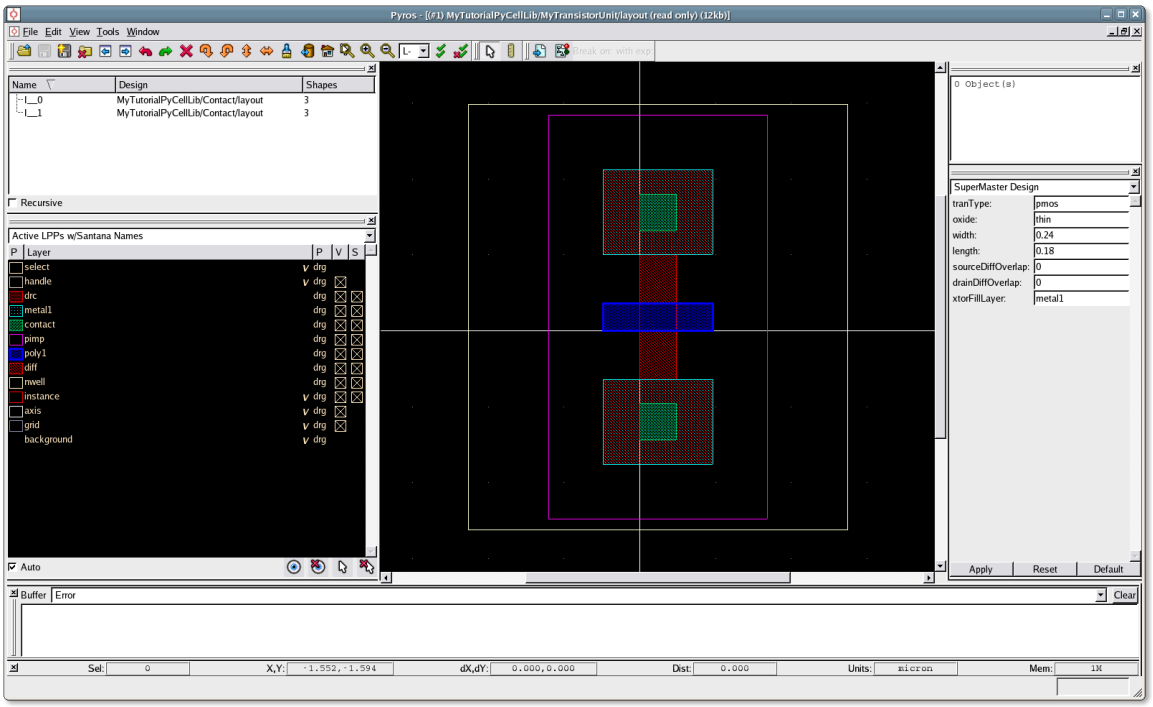
As an example, the following invocation of this **cngenlib** utility will create the OpenAccess library for a package of parameterized cell designs:

cngenlib`--create --view --techfile $CNI_ROOT/tech/cni130/santanaTech/Santana.tech`

`pkg:MyPyCells MyPyCellLib ~/MyPyCellLib`

The `--create` option means that the OpenAccess library should be created; make sure that the directory ~/MyTutorialPyCellLib does not already exist. The `--view` option means that the viewer tool should be brought up to display the parameterized cells after they are created. The `--techfile` option specifies the location of the technology file, while the `pkg:MyTutorialPyCells` option specifies the Python package name for the Python parameterized cell source code. The last two options specify the OpenAccess library name (`MyPyCellLib`), located in the user's home directory.

If this **cngenlib** command were to be successfully run, then the Pyros viewer would display each of the parameterized cells which were compiled into the OpenAccess library, as shown in the following screen shot:



Note that when each parameterized cell is created for the OpenAccess library, its layout will be generated using only default parameter values. Also note that when this **cngenlib**

utility is run, the Python interpreter will be invoked to compile the Python source code. If there are any errors in this Python code, then the **cngenlib** utility will fail and generate one or more error messages from the Python interpreter. These errors must be corrected before the OpenAccess library will be created. In addition, the parameterized cells will only be displayed in the Pyros layout viewing tool if there are no errors.

As a convenience, whenever this **cngenlib** utility is run to create or update a library of parameterized cells, it will automatically generate a timestamp as a property on the OpenAccess library containing these PyCells. The name of this timestamp property is `PyCell_Timestamp`, and its value is a string containing the current time when the **cngenlib** utility was run. Note that this timestamp property can be used to check that an OpenAccess library is a PyCell library, and as well determine when it was last updated.

Examples:

create a new Santana technology library named `MyTechLib`

```
cngenlib --create  --techfile
         $CNI_ROOT/tech/cni180/santanaTech/Santana.tech
         tech:only  MyTechLib /home/user/MyTechLib
```

update existing OpenAccess technology library which

does not contain any Santana technology file

```
cngenlib --update tech:only TechLibOA  /home/user/TechLib
```

create PyCell library using Python code package

```
cngenlib --create --techlib MyTechLib  pkg:MyPyCells
         MyPyCellLib  /home/user/MyPyCellLib
```

create PyCell library using bundled Python code package;

also encrypt this bundled Python source code package

```
cngenlib --create --techlib MyTechLib  --bundle=encrypted
         --verbose  pkg:MyPyCells
         MyPyCellLib  /home/user/MyPyCellLib
```

create PyCell library using Python code module

```
cngenlib --create --techlib MyTechLib  mod:/home/myPyCellCode.py
         MyPyCellLib  /home/user/MyPyCellLib
```

create same PyCell library using Pyros viewer option

```
cngenlib --create --view --techlib MyTechLib
         mod:/home/myPyCellCode.py
         MyPyCellLib  /home/user/MyPyCellLib
```

update existing PyCell library with new PyCell source code;

Feedback

note update is the default, so `--update` is not required.

```
cngenlib  pkg:MyPyCells  MyPyCellLib  /home/user/MyPyCellLib
```

update existing PyCell library for only Nmos/Pmos cells

```
cngenlib --update --for_pycells=Nmos,Pmos pkg:MyCells
         MyPyCellLib /home/user/MyPyCellLib
```

cnbundle

Description: The **cnbundle** utility is used to bundle Python code packages for an existing OpenAccess library. In addition to bundling, this **cnbundle** utility can also be used to unbundle an existing code package which is already bundled for an OpenAccess library, or to simply list all of the bundled code packages for the OpenAccess library. This **cnbundle** utility can only be used with Python code packages. Note that the user must have write permission for the specified OpenAccess library in order to bundle (or unbundle) Python code packages. In the case of listing existing code packages, write permission for the OpenAccess library is not required.

This **cnbundle** utility is often used in conjunction with the **cnsendcmd** utility, which is used to dynamically reload Python PyCell code, while an application is running. This **cnsendcmd** utility requires that the Python code for the application be unbundled, before it can be reloaded. Thus, this **cnbundle** utility can be used (with the --unbundle option) to ensure that the Python code for the application is unbundled.

This **cnbundle** utility program can be invoked as follows:

cnbundle <oa-library-name> [<code-pkg-ID>]

```
[-b BUNDLE, --bundle=BUNDLE]

[-u, --unbundle]

[-l, --list]

[-O OPTIMIZE, --optimize=OPTIMIZE]

[--verbose]

[ -h, --help ]

[ -V, --version]
```

The <oa-library-name> argument is the required name of the OpenAccess library for which Python code packages should be bundled (or unbundled). The <code-pkg-ID> argument is the code package identifier, which should be used with the `--bundle` and

`--unbundle` options. This code package identifier is specified using the format `pkg:<python-package-path>, where the <python-package-path> specifies`

`the full path name for the Python source code package. This Python source code package name should always be a single name for a top-level package, such as "MyPyCells`. This code package identifier is more fully described in the documentation for the **cngenlib** utility.

The `--bundle` option is used to bundle a specified Python source code package with the specified OpenAccess library, so that the Python code is stored with the library. The valid values for this option are `source`, `bytecode`, `encrypted` and `encrypted_source`, as described for the **cngenlib** utility `--bundle` option. The `--unbundle` option is used to unbundle a specified Python code package, so that the Python source code is no longer stored with the OpenAccess library. Note that these `--bundle` and `--unbundle` options are mutually exclusive; one and only one of these options can be specified, otherwise an error message will be generated.

The `--list` option is used to simply list all of the existing Python code packages which are bundled with this OpenAccess library; if there are no bundled code packages for this library, then this is indicated in the output message.

The `--optimize` option is used to control the level of optimization used when the Python bytecode is being bundled in the OpenAccess library. The valid values for this option are 0, 1 or 2, which correspond to no optimization, the Python `-O` optimization level and the Python `-OO` optimization level.

The `--verbose` option is used to display the internal messages which are generated by the Python code bundling process. This option is only meaningful when the `--bundle` option is specified.

The `-h` (or `--help`) option prints out a help message, which describes the command-line arguments which can be used with this utility program. The `-V` (or `--version`) option prints out version information for this utility program.

Examples:

list all Python code packages bundled with library

```
cnbundle  --list  MyPyCellLib
```

now bundle Python code package with library

```
cnbundle  --bundle=source  MyPyCellLib  pkg:MyPyCells
```

check that this code package is now bundled

```
cnbundle  --list  MyPyCellLib
```

Now unbundle this code package from library

```
cnunbundle  --unbundle  MyPyCellLIb  pkg:MyPyCells
```

check that code package has been unbundled

```
cnbundle --list  MyPyCellLib
```

cnrmlib

Description: The **cnrmlib** utility program is used to safely remove an OpenAccess library directory from the underlying file system. Note that this OpenAcess library directory will only be removed, if the `.oalib` OpenAccess design management file exists in the specified directory location. This hidden `.oalib` file should only be generated when the OpenAccess library is originally created. This approach helps ensure that only OpenAccess library directories will be removed when this **cnrmlib** utility is invoked.

In addition, this **cnrmlib** utility makes a number of other checks, before the directory is safely removed from the file system. This **cnrmlib** utility program can be invoked as follows:

cnrmlib <oa-library-path>

```
[ -h, --help ]

[ -V, --version ]
```

The <oa-library-path> is the required file path for the existing OpenAccess library directory, which should be the directory location for an OpenAcess library. The `-h` (or `--help`) option prints out a help message, which describes the command-line arguments which can be used with this utility program. The `-V` (or `--version`) option prints out version information for this utility program.

This **cnrmlib** utility program will first check that the specified file path exists in the file system, and that it also is the name of a directory in the file system. Checks will also be made to ensure that the specified directory name is not a mount point or the name of a user's home directory. If any of these checks fail, then an appropriate error message will be generated, and the specified directory will not be removed from the file system.

Examples:

remove existing OA library used to store PyCells;

directory location for OA library is specified.

```
cnrmlib  /home/user/MyPyCellLib
```

cndmfiles

Description: The **cndmfiles** utility program is used to attach design management files to an existing OpenAccess library. In addition, any design management files which are currently attached to an OpenAccess library can be detached from the library. As

an additional option, all design management files which are currently attached to an OpenAccess library can be listed.

It is often useful to be able to associate a design management file with the OpenAccess library which will make use of the information which is contained inside this design management file. For example, a file which contains higher-level “meta data” information about parameterized cells can be attached to the OpenAccess parameterized cell library using this **cndmfiles** utility. This `meta data` information could include capabilities of the various parameterized cells in the library, such as information about which parameterized cells in the library can be abutted. Since the amount of such `meta data` may be large, it is more convenient to store this information in a single file and then attach it to the library, rather than to store this information as individual properties. This approach of attaching information as a design management file to the OpenAccess library helps ensure consistency between the parameterized cells and the applications using this OpenAccess library. This **cndmfiles** utility program can be invoked as follows:

cndmfiles <oa-library-name>

```
[-l, --list ]

[-a, --attach <fileName> <filePath> ]

[-d, --detach <fileName> ]

[ -h, --help ]

[ -V, --version ]
```

The <oa-library-name> is the required name of the OpenAccess library to which design management files should be attached (or detached). Note that this OpenAccess library must be defined in the applicable OpenAccess `lib.defs` library definition file; otherwise, an error message will be generated. The `-l` (or `--list`) option prints out a listing of the names of all design management files which are currently attached to the specified OpenAccess library. The `-a` (or `--attach`) option will attach the specified design management file to the specified OpenAccess library. Note that both the <filename> and the actual <filePath> location of the design management file should be specified for this option. The `-d` (or --detach”) option will detach the specified design management file from the OpenAccess library. Note that these `--attach` and `--detach` options are mutually exclusive; only one of these two options can be specified, otherwise an error message will be generated. The `-h` (or `--help`) option prints out a help message, which describes the command-line arguments which can be used with this utility program. The `-V` (or `--version`) option prints out version information for this utility program.

Note that if no command-line options are specified for this **cndmfiles** utility, so that the only argument is the name of the OpenAccess library, then the `--list` option will be set by default, and a listing of all of the design management files currently attached to the specified OpenAccess library will be generated.

Feedback

Examples:

list all design management files in given library

```
cndmfiles  --list  MyPyCellLib
```

attach given file to library, using specified name

```
cndmfiles --attach  MyPyCellLib  myDMFile  myLib/myData.txt
```

detach named design management file from library

```
cndmfiles --detach  MyPyCellLib  myDMFile
```

do not specify any options, will list all attached files

```
cndmfiles  MyPyCellLib
```

---

cnchecklocks

Description: The **cnchecklocks** utility program is used to check and see if any of the designs in an OpenAccess library are currently being locked by any other active process. If any design in the library is currently being locked, then a message providing the name of the locked design will be generated. If no designs are currently being locked, then a message will be output indicating that no designs in the library are locked.

This **cnchecklocks** utility is used to ensure that no other active processes are currently locking any of the designs which are contained in the specified OpenAccess library. This design lock checking should be used before attempting to perform updates or other changes to an existing OpenAccess library. This **cnchecklocks** utility program can be invoked as follows:

cnchecklocks <oa-library-name>

```
[ -h, --help ]

[ -V, --version ]
```

The <oa-library-name> is the name of the OpenAccess library which should be searched for any designs which are currently being locked by any active process. Note that this OpenAccess library must be defined in the applicable OpenAccess `lib.defs` library definition file; otherwise, an error message will be generated. Note that one or more OpenAccess library names can be specified for this <oa-library-name> parameter value. If no library name is specified, then by default, all OpenAccess libraries which are listed in the `lib.defs` file will be checked for designs having locks. The `-h` (or `--help`) option prints out a help message, which describes the command-line arguments which can be used with this utility program. The `-V` (or `--version`) option prints out version information for this utility program.

Feedback

Examples:

check existing OA PyCell library for locked designs

```
cnchecklocks  MyPyCellLib
```

check all OpenAccess libraries in `lib.defs` for locked designs

```
cnchecklocks
```

cnupdatepycell

Description: The **cnupdatepycell** utility program is used to update an existing parameterized cell in an OpenAccess PyCell library. This parameterized cell is updated by simply re-defining the parameterized cell in the library, using the existing PyCell code and technology bindings. This utility is typically used when the parameterized cell determines default parameter values through the use of data stored in external files or data structures (such as CDF attached to the parameterized cell). This utility can then be used to automatically update the default parameter values for the parameterized cell.

This **cnupdatepycell** utility program can be invoked as follows:

cnupdatepycell <oa-library-name> <oa-cell-name> [<oa-view-name>]

```
[ -h, --help ]

[ -V, --version ]
```

The <oa-library-name> is the name of the OpenAccess PyCell library, while <os-cell-name> is the cell name of the PyCell in this library which should be updated. The optional <oa-view-name> is the view name for this PyCell; if this view name is not specified, then the default view name `layout` will be used. Note that the PyCell must exist in the OpenAccess PyCell library; otherwise, an error message will be generated. The `-h` (or `--help`) option prints out a help message, which describes the command-line arguments which can be used with this utility program. The `-V` (or `--version`) option prints out version information for this utility program.

Examples:

update existing Nmos and Pmos cells in PyCell library

```
cnupdatepycell  MyPyCellLib  Nmos
cnupdatepycell  MyPyCellLib  Pmos
```

cnstrmin

Description: The **cnstrmin** utility program is a wrapper script around the existing OpenAccess `strm2oa` utility program. This OpenAccess utility program is used to convert

an existing layout stored in Stream format into the OpenAcess format. This **cnstrmin** utility program simply provides a proper environment for running the OpenAccess `oa2strm` utility within the PyCell Studio environment. This **cnstmin** utility checks that the PyCell Studio environment is properly set up, by checking the PyCell Studio installation and various required environment variable settings. More specifically, this **cnstrmin** utility makes the following checks: 1) CNI_ROOT environment variable is properly set; 2) PyCell Studio is installed; 3) PyCell Plugin is installed; 4) Python is properly installed for PyCell Studio; and 5) the LD_LIBRARY_PATH environment variable is properly set. Once these checks have been successfully made, then the OpenAccess `strm2oa` utility is invoked with the proper environment settings for PyCell Studio, passing the list of arguments which were used when **cnstrmin** was invoked. If any errors with the PyCell Studio installation were detected during this checking process, then error messages will be generated, and the OpenAccess `srtm2oa` utility program will not be invoked.

This **cnstrmin** utility program can be invoked as follows:

cnstrmin -lib LIBRARY -gds GDS <oa-command-options>

```
[ -h, --help ]

[ -V, --version ]
```

The `-lib` option is required and is the name for the OpenAccess library which should be generated to contain the converted Stream layout information. The `-gds` option is the required name for the Stream input file which should be converted to OpenAccess format. The <oa-command-options> is the list of possible options for the OpenAccess `strm2oa` utility program. This possible options can be displayed by using the `-h`

(or `--help`) option for this **cnstrmin** utility program.

Examples:

convert existing Stream GDS layout file to OpenAccess format

```
cnstrmin -lib  MyPyCellLib  -gds  layout.gds
```

cnstrmout

Description: The **cnstrmout** utility program is a wrapper script around the existing OpenAccess `oa2strm` utility program. This OpenAccess utility program is used to convert an existing layout stored in the OpenAcess format into the Stream layout format. This **cnstrmout** utility program simply provides a proper environment for running the OpenAccess `oa2strm` utility within the PyCell Studio environment. This **cnstmout** utility checks that the PyCell Studio environment is properly set up, by checking the PyCell Studio installation and various required environment variable settings. More specifically, this **cnstrmout** utility makes the following checks: 1) CNI_ROOT environment variable is properly set; 2) PyCell Studio is installed; 3) PyCell Plugin is installed; 4) Python is

properly installed for PyCell Studio; and 5) LD_LIBRARY_PATH environment variable is properly set. Once these checks have been successfully made, then the OpenAccess `oa2strm` utility is invoked with the proper environment settings for PyCell Studio, passing the list of arguments which were used when **cnstrmout** was invoked. If any errors with the PyCell Studio installation were detected during this checking process, then error messages will be generated, and the OpenAccess `oa2srtm` utility program will not be invoked.

This **cnstrmout** utility program can be invoked as follows:

cnstrmout -gds GDS -lib LIBRARY <oa-command-options>

```
[ -h, --help ]

[ -V, --version ]
```

The `-gds` option is required and is the name for the Stream input file which should be converted to OpenAccess format. The `-lib` option is required and is the name for the OpenAccess library which should contain the layout information which should be converted from the Stream format. The <oa-command-options> is the list of possible options for the OpenAccess `oa2strm` utility program. These possible options can be displayed by using the `-h` (or `--help`) option for this **cnstrmout** utility program.

Examples:

convert existing OpenAccess layout format to Stream format file

```
cnstrmout -gds  layout.gds  -lib  MyPyCellLib
```

4

Santana Technology File Utilities

The PyCell Studio product provides two utilities which are used to work with the Santana technology file. The **cntechversion** utility can be used to create specific versions of the Santana technology file information, for use by PyCell developers and users. An instance of a PyCell can be created which is bound to a specified version of the Santana technology information, allowing the generated PyCell layout to be `frozen`. The **cntechcheck** utility can be used to perform additional checking on the contents of the Santana technology file, such as consistency checking between different sections of the Santana technology file. This checking can help find potentially subtle errors in the Santana technology file, which could otherwise be difficult to detect.

cntechversion

Description: The **cntechversion** utility program is used to create a version of the Santana technology information for an OpenAccess Santana technology library.

This versioning mechanism allows PyCell Studio users to bind instances of PyCells to a specified version of the Santana technology information. Thus, developers can `freeze` layout which is generated by PyCells at specific points during the circuit design process. The versioning for a PyCell is handled by means of a special pre-defined version parameter for the PyCell; this special parameter is named `cnTechVersion`, and should be set to the desired version string for the Santana technology library.

This **cntechversion** utility is used to manage the versioning process for an existing Santana technology library. This utility can be used to create a version of the Santana technology information, to remove an existing version, to update an existing version, or to obtain all version information about the Santana technology library. In addition, this utility allows the developer to specify the versioning name string which should be used when a specific technology version is created, as well as to provide comments about a specific technology version. Note that these comments will be displayed when versioning information about the Santana technology library is requested.

This **cntechversion** utility program can be invoked as follows:

cntechversion [<tech-library-name>]

```
[ -c, --create ]

[ -r, --remove ]
```

Feedback

```
[ -e, --rename ]

[ -u, --update ]

[ -l, --list ]

[ -n, --name <tech_version> ]

[ -d, --description <description> ]

[ -m, --new_name <tech_version> ]

[ -h, --help ]

[ -V, --version ]
```

The <tech-library-name> is the name of the OpenAccess Santana technology library which is being used for versioning. Note that this OpenAccess library must be defined in the applicable OpenAccess `lib.defs` library definition file; otherwise, an error message will be generated. The `-c` (or `--create`) option will create a version of the Santana technology information which is identical to the current state of the <tech-lib-name> technology library. The `-r` (or `--remove`) option will remove an existing technology version, using the version name which is specified by the `-n` (or `--name`) option.

The `-e` (or --rename") option will rename an existing technology version, using the new version name specified by the `-m` (or `--new_name`) option. The `-u` (or `--update`) option will update an existing technology version, using the description specified by the `-d` (or `--description`) option. If the technology version which is specified by the `-n` (or `--name`) option does not exist, then an error message will be generated.

The `-l` (or `--list`) option lists all existing technology versions, showing the description and time of creation, along with any update information. Note that if only the <tech-library-name> parameter value is provided, then the listing of versioning information will be performed by default. The `-h` (or `--help`) option prints out a help message, which describes the command-line arguments which can be used with this utility program. The `-V` (or `--version`) option prints out version information for this utility program.

Example:

check existing technology library for versioning information

```
cntechversion  cni130  --list
```

create new version using existing technology information

```
cntechversion  cni130  --create  --name = 1.0
```

check that new version was created

```
cntechversion  cni130  --list
```

```
# can also check for versioning information using default
cntechversion  cni130

# now update this version with additional description
cntechversion  cni130 --update  --description = "initial version"

# check that versioning information was updated
cntechversion  cni130  --list
```

cntechcheck

Description: The **cntechcheck** utility program is used to analyze the Santana technology file information for possible errors or inconsistencies, which could otherwise be difficult to detect. This **cntechcheck** utility can be used to check for possible inconsistencies between different sections of the Santana technology file. For example, it can be used to check that the bottom layer and top layer which is used in the `viaLayer` section of the technology file have the proper mask layer numbers in the `maskNumbers` section. As a further example, this **cntechcheck** utility can also be used to check that layer names used in the `layerMaterials` section of the technology file is actually a mask layer, as defined in the `maskNumbers` section.

This **cntechcheck** utility can also be used to check for consistency between the Santana technology file and the OpenAccess technology database, if it exists within the same technology library. For example, checks are made to ensure that user units and database units information and grid values are consistent between the Santana technology file and the OpenAccess technology database.

The following is a short summary of the different items which are currently being checked and analyzed by this **cntechcheck** utility:

- Check consistency of the database units per user unit value between the Santana technology file and any existing OpenAccess technology database.
- Check that layer names used in `layerMaterials` section of Santana technology file are actually mask layers (as defined in the `maskNumbers` section).
- Check that layer names used in `viaLayers` section of the Santana technology file have the proper mask number layers (as defined in the `maskNumbers` section).

Feedback

- Check that two different mask numbers are not assigned to the same layer name in the `maskNumbers` section of the Santana technology file.
- Check that the mask numbers for the layers used in the `viaLayers` section of the Santana technology file are consistent. That is, the mask number for the upper layer is greater than the mask number for the via layer, and the mask number for the via layer is greater than the mask number for the bottom layer.

Note that it is expected that additional checking will be added to this **cntechcheck** utility on a continual basis.

This **cntechcheck** utility program can be invoked as follows:

cntechcheck <input-tech>

[-l, --library]

[-f, --techfile]

[-v, --cnTechVersion <tech-version>]

[-o, --output <file-path>]

[-h, --help]

[-V, --version]

The required <input-tech> is the name of the Santana technology file which is being checked and analyzed for possible errors or inconsistencies. The `-l` (or `--library`) option specifies that <input-tech> name is the name of the technology library. Note

that this `-l` option is the default option for this **cnchecktech** utility program. The `-f` (or `--techfile`) option is used to specify that a separate Santana technology file which is not located in any technology library should be checked. If this option is used, then the <input-tech> option should specify the file path to this Santana technology file. The `-v` (or `--cnTechVersion`) option is used to specify the version of the Santana technology file contained inside the technology library. Note that this option can only be used in conjunction with the `--library` option. The `-o` (or `--output <file-path>`) option can be used to specify that error or warning messages generated by this utility should be output to a separate log file. By default, all output generated by this utility is written to the standard output. The `-h` (or `--help`) option prints out a help message, which describes the command-line arguments which can be used with this utility program.

The `-V` (or `--version`) option prints out version information for this utility program.

Example:

check existing technology library; --library is default option

```
cnchecktech  cni130 --library
cntechcheck  cni130
```

Feedback

check separate Santana technology file

```
cntechcheck  $CNI_ROOT/tech/cni130/src/Santana.tech  --techfile
```

can also use versioning information for checking

```
cntechcheck  cni130  --library --cnTechVersion  1.2
```

redirect output to separate logging file

```
cntechcheck  cni130  --output /home/techcheck.out
```

5

Conversion Programs

The PyCell Studio product provides different conversion programs as part of the standard product offering. These conversion programs can be used to convert existing technology files (or libraries) and display files into the Santana specific formats which are used by the PyCell Studio product. These conversion utilities are provided to make it as easy as possible for the design engineer to adopt and deploy the PyCell Studio product in their existing design flow.

There are two different conversion programs which can be used to convert existing technology files and technology display files into the corresponding Santana file formats:

- The `cntechconv` program converts an existing Cadence DF II technology file into the Santana technology file format. In addition, this conversion program can also be used to convert any OpenAccess technology library into the Santana technology file format, or to convert layer names in an existing Santana technology file.
- The `cndispconv` program converts an existing Cadence DF II technology file and associated Display Resource File (*.drf) into the Santana technology display file format, which is used by the graphical programs.

This section briefly describes these two PyCell Studio conversion utilities. For more detailed information, please refer to the relevant sections of the “Santana Conversion Utilities” reference manual.

cntechconv

Description: The **cntechconv** conversion utility converts an existing Cadence DF II technology file into the Santana technology file format, and is invoked as follows:

cntechconv <tech-file-name> [-o <output-file-name>]

```
[ -m <mapping-file-name> ]

[ -M <mapping-file-name> ]

[ -f <tech-file-format> ]

[ -h ]

[ -V ]
```

The <tech-file-name> is the required file path for the existing Cadence DF II technology file (or OpenAccess library or Santana technology file); all of the other command-line options are optional. The `-o` option specifies the file path for the output Santana technology file which will be generated; by default, this file is named `<tech-file-name>.converted.Santana.tech`. The `-m` and `-M` options specify the file path for any mapping file which should be used during this conversion process; by default, no mapping of layers (or purposes) will take place. Any unmapped layer or purpose names will be ignored if the `-m` option is used, while all unmapped layers and purposes will be used as is with their original names, if the `-M` option is used. Only one of these `-m` and `-M` command-line options can be used. The `-f` option specifies the format used by the input technology file; this can either be `cds` for the Cadence DF II technology file, `oa` for the OpenAccess library, or `cni` for the Santana technology file. Note that the `cni` option value should only be used when converting layer names in an existing Santana technology file. By default, the value `cds` for the Cadence DF II technology files is used for the value of the `-f` option. The `-h` option prints out a help message, while the `-V` option prints out version information.

cndispconv

Description: The **cndispconv** conversion utility converts the display information contained in an existing Cadence DF II technology file and Cadence DF II Display Resource file into the equivalent Santana file formats. This **cndispconv** conversion program can be invoked as follows:

cndispconv --cdstech = <cds-tech-file-name>

--cdsdisplay = <cds-display-file-name>

[--santanatech = <santana-tech-file-name>]

[-h]

[-V]

The <cds-tech-file-name> is the required file path for the existing Cadence DF II technology file, which will be converted to the Santana technology display file format. The <cds-display-file-name> is the required file path for the Cadence DF II Display Resource file (*.drf file), which contains the detailed display information which is referenced by the display information contained in the Cadence DF II technology file. The optional `--santanatech` command-line option specifies the file path for the optional Santana technology file which can be used as an optional part of the conversion process. The `-h` option prints out a help message, while the `-V` option prints out version information.