SpyGlass® Design Read-In Methodology

Version N-2017.12-SP2, June 2018

SYNOPSYS®

Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

Report an Error

The SpyGlass Technical Publications team welcomes your feedback and suggestions on this publication. Please provide specific feedback and, if possible, attach a snapshot. Send your feedback to *spyglass_support@synopsys.com*.

Contents

Preface

About This Book

The SpyGlass® Design Read-In methodology guide describes the flow for using the Design Read-In methodology.

Contents of This Book

The SpyGlass Design Read-In methodology guide has the following sections:

Section	Description
Reading a Design	How to read-in your design in SpyGlass Tool Suite
Appendix - Single Step Compilation	How to perform single-step precompilation

Typographical Conventions

This document uses the following typographical conventions:

To indicate	Convention Used
Program code	`OUT <= IN;`
Object names	`OUT`
Variables representing objects names	*<sig-name>*
Message	Active low signal name '<sig-name>' must end with _X.
Message location	OUT <= IN;
Reworked example with message removed	OUT_X <= IN;
Important Information	**NOTE:** This rule...

The following table describes the syntax used in this document:

Syntax	Description
[ ] (Square brackets)	An optional entry
{ } (Curly braces)	An entry that can be specified once or multiple times
| (Vertical bar)	A list of choices out of which you can choose one
. . . (Horizontal ellipsis)	Other options that you can specify

Reading a Design

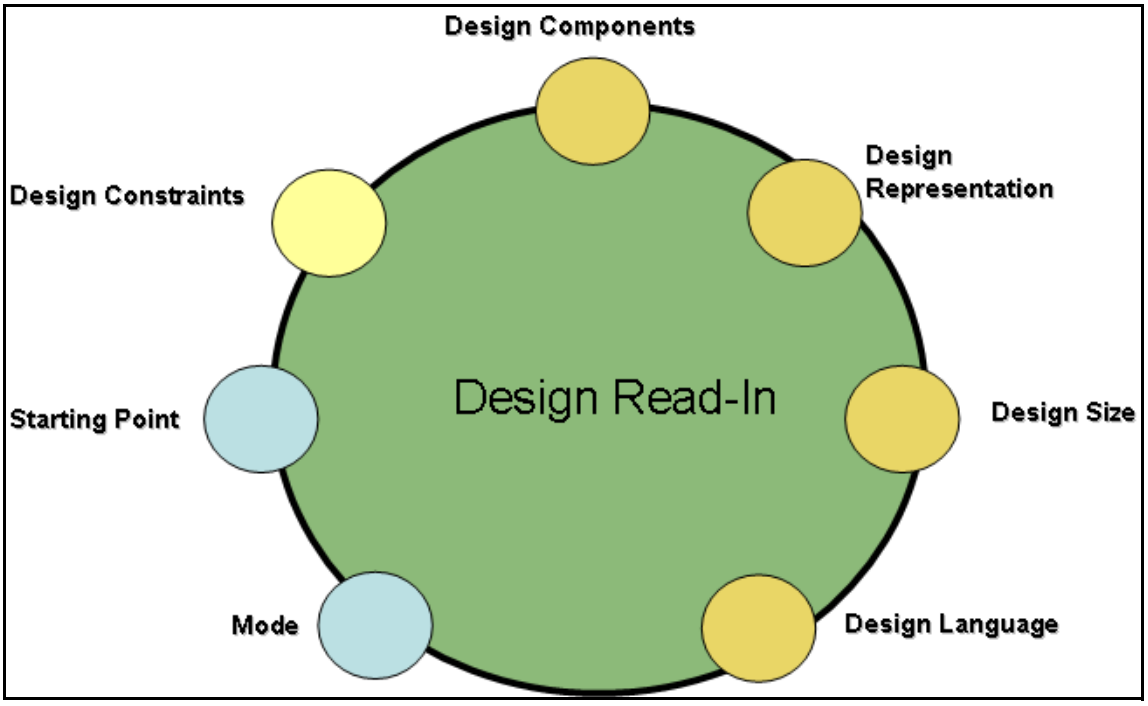
SpyGlass supports various options in terms of analyzing a design. These options are in several dimensions, namely:

- Language: Verilog, VHDL, Mixed
- Language Variations: V2K, VHDL87, VHDL93 etc.
- Completeness of design information: Analysis can continue even in the presence of black boxes etc.
- Library information can be made available in the form of UDPs, synthesizable HDLs, .libs, ILMs etc.

This document provides a step-by-step approach for a user to construct the right commands and/or take the right steps to read in his design into the software. Since, this is a generic document, covering all possible scenarios of designs (black boxes, analog cells, RTL, netlists etc.) you would not need to go through the entire document to read-in your design easily into SpyGlass. Depending on the contents/style of your designs, you might want to skip sections that are not relevant to your current design. The document is based on SpyGlass 5.5.1.

The Design Read Process

Reading a new design into SpyGlass can be a multi-dimensional problem. The following diagram illustrates many of the important aspects of the read-in process:



NOTE: *Starting Point and Mode are decoupled from the intrinsic nature of the design. All other circles are attributes of the design.*

Starting Point and Mode are dependent on design group, design methodology, and culture of the design team or history. Therefore different colors have been used in the diagram.

A design can have many different combinations of the aspects illustrated above. Detailed steps of handling these aspects are described later in this document.

Design Language

Today's SoCs typically use Verilog, VHDL, or a mixture of the two languages. SystemVerilog may proliferate in the coming years. There are variations of these languages where conformance to LRM may or may not exist in some cases. Popular simulator vendors (like MTI, Synopsys/VCS etc) have supported many features outside LRM. On the contrary, there are certain features in LRM that are not supported by some of these simulators. This aspect does have some implications in the design-read-in process.

Support of pragmas in the language also has wide variations. In the process of design-read, these language features need to be considered appropriately (described in later chapters). Synthesizable subsets have in general remained to be a common denominator.

Design Components

In today's SoC designs, the levels of integration and complexity have forced the extensive re-use paradigm. Many diverse design teams and suppliers contribute to a single complex SoC. Thus a design would have many different components, such as IP, PLL, RAM, ROM, DW, cell libraries.

The design-read-in process needs to resolve these components to establish a coherent connectivity representation of the design. In later chapters this document describes detailed steps required to handle these components coherently and correctly for all the policies, rules, and checks performed by the SpyGlass suite of products.

Predefined and Characterized Cell Library Elements

The elements could be as simple as primitive gates or as complex as processor cores like an MPU or GPU. There are multiple variations of the description of these library elements. Some of them might have synthesizable model description. Sometimes only a behavioral model may be available. Sometimes none of this is available (leading to black boxes). Design read in process needs to understand and interpret these elements appropriately for advanced checks. The key idea is to capture the design intent of these elements for the best design analysis.

Typically primitive cell libraries are designed at a particular technology node. Most common format for synthesis library description is the Liberty format. In many cases the cell library simulation models are based on

UDPs. SpyGlass has options to interpret each of these cell primitive formats.

DesignWare® Components

Some vendors like Synopsys provide DesignWare® suite. A logic designer might instantiate some of these components explicitly in a design. DesignWare® components allow the designer to design at higher level of abstraction. Designers use these components to enforce a particular micro-architecture (ex. Ripple adder vs. CS adder).

Special Cells

SoC design may also contain special cells like Analog D/ACs, PLLs, and special I/O cells. These cells may not be part of the standard cell library, or may originate from different suppliers. Design-read-in process needs to handle these components coherently.

RAMS and ROMS

While RAMs and ROMs may fall in the “Special Cell” category, the type of model available for these cells may have a significant impact on run time as well as the results of some of the advanced policies. Based on the available model, deciding how to treat RAMs during analysis is an important decision.

The key idea again is to communicate the connectivity of any design unit ports in the design to SpyGlass.

Design Representation

Users of SpyGlass are encouraged to run SpyGlass tool suite at RTL stage. However SpyGlass is also used by many back-end groups. These groups predominantly deal with structural netlist (usually synthesized and scan inserted). Design read in process needs to ensure that both structural and RTL level design descriptions are handled correctly.

Design Constraints

Constraint files such as the SDC files for Design Compiler help define the design's intent. Just like SDC constraints control how synthesis optimizes a design, SpyGlass has its own constraint files (.sgdc file) to control how SpyGlass analyzes a design. SpyGlass has some rules that help generate SGDC constraints. In addition, SpyGlass may be able to convert SDC into some sgdc constraints. Each advanced policies may require different sgdc constraints.

Design Size

With growing complexity of the designs, the design sizes in general are growing. All the attributes of the chip are growing (die size, gate count, memory count, I/Os, special cells etc). Design read-in needs to ensure that a large design with above illustrated aspects is read-in successfully.

Starting

Design-read-in scripts may be available for other tools such as:

- Simulation
- Logic Synthesis
- Formal Verification (RTL-to-gates or gates-to-gates)
- Static Timing Analysis
- SpyGlass DFT

Many times, scripts for some/all of above are a good place to start for a list of files, libraries, rams, special cells etc. Design read-in needs to incorporate many/all of above starting points in a design environment.

SpyGlass Console needs a project file. The main contents of the project file are the sources needed to describe your design, and, additional options specific to the run/design description.

Put all of your HDL source-files into a file: sources.f

And, put into your Project file:

```
read_file –type sourcelist sources.f
```

Subsequent sections below describe the additional options that need to be included in the project file.

For adding any option into the project file, use the following command:

```
set_option <option> <value>
```

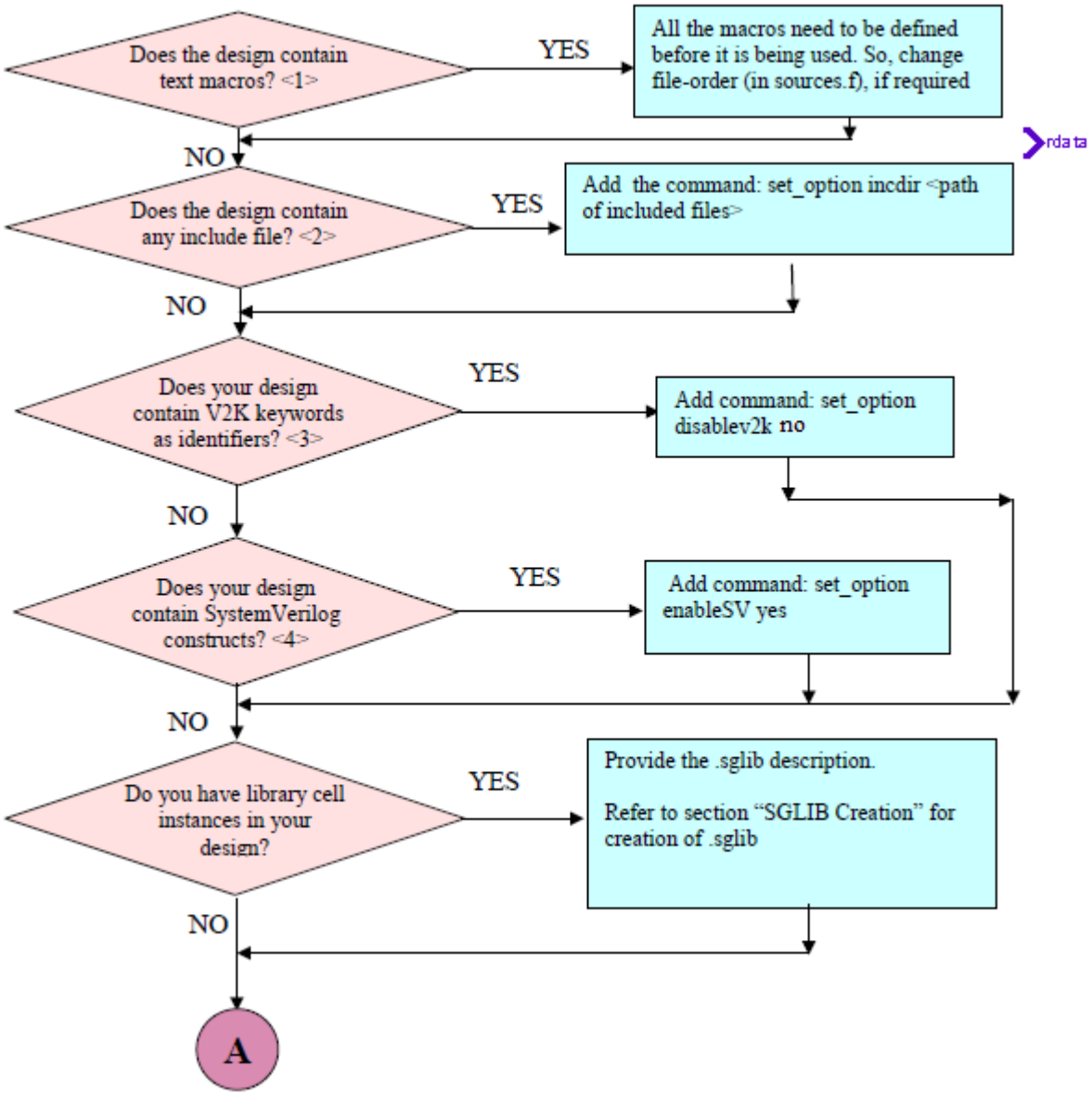
OR

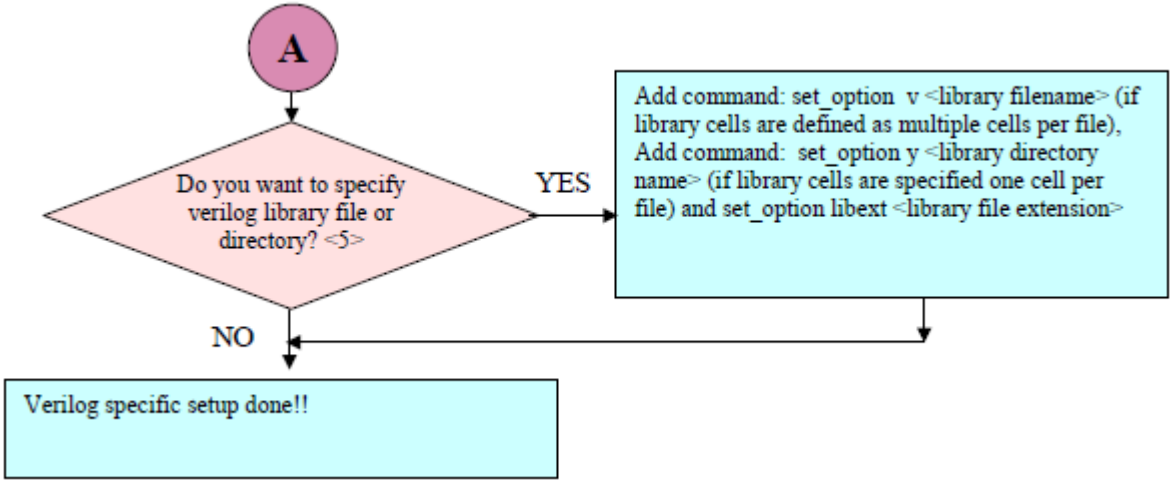
```
set_option <option> {space separated multiple values}
```

Unless otherwise mentioned, all options explained in the rest of the document are to be added in the project file and will follow the above style.

Once the project file is completed, the SpyGlass Console can be invoked through the command: spyglass –project <project file>

Verilog Specific Options





NOTES

1. If a macro is referenced in the design before (or without) being declared, you will get STX_VE_533 as:

   `STX_VE_533 Syntax …. Used macro (…..) has not been defined.`

2. If set_option incdir is not specified correctly, you will get STX_VE_485 as

   `STX_VE_485 Syntax …. Include file ( … ) could not be found or opened in read mode from current working directory ( ….. ) or other include directory paths ( if any )`

   If “included” file name is also included in the sources.f, you might get errors/warnings related to duplicate definitions (depending on what was there in the included file)

3. If the design is traditional Verilog (Verilog95) code, but, uses certain identifiers (which later on became keywords in V2K), the tool would give syntax errors (unless, it is explicitly specified that the design is a Verilog95 design).

4. If the design has SystemVerilog constructs, you will get the STX_VE_479 violation.

   Please add command “set_option enableSV yes” to the project file.

5. If library files/directories are not specified, you will get black boxes corresponding to the instances of for the library cells in the design. If library files are specified as part of sources.f, some rules might be flagged on files/cells where they should not be flagged and you might get multiple-top warning messages as DetectTopDesignUnits.

Design-Read with Precompilation

NOTE: *VHDL/Mixed designs are typically read/developed using precompile concept, hence has been described separately in detail in next section. VHDL by definition supports the concept of Precompilation. However, lately most tools have started supporting the concept of precompilation for Verilog also. Hence, SpyGlass also allows precompilation of Verilog also.*

Precompilation refers to compilation strategy, wherein the design is read-in (compiled) in multiple steps.

A user could (pre)compile a lower level design-unit/VHDL package etc. which is to be used in the higher level design-unit/VHDL package etc.

Now, while, compiling the higher level design-unit/VHDL package, the user need not worry about the source code for the lower-level design-units and simply access the precompiled units.

This allows a user to:

- build up the design in smaller steps for some big designs
- send only the precompiled form to somebody else, while, still not sharing the actual source code
- ensure sanctity of data, by making sure that the lower level design unit/ package is never more updated than a higher level design-unit/package

First, let's understand a few basic concepts involved in precompilation.

There is a concept of Logical name of a library. All library names mentioned in the source code refer to Logical Names.

The command has to map the logical name to a physical path.

So, command: `set_option lib L1 <path>` in the project file indicates that anytime the source code should refer to L1, the corresponding search should be made in <path>.

The logical library WORK is mapped to physical location: ./WORK/ (unless specified otherwise)

Similarly, `set_option work <lib name>` option specifies the name of the logical library, where, the output of the current compilation step needs to be dumped. By default, the logical library WORK is specified as the work library. However, one can modify this specification.

In most situations, the user wants to compile the design in multiple steps. Hence, the user should create a file: libmap.f – which contains the entire

library mapping. This helps the user in following ways:

- A lower level library mapping which has been specified once – need not be specified again and again during higher levels of compilation.
- There is no chance of typos etc. while providing the mapping multiple times – resulting in inconsistency.

NOTE: *Libmap.f file can be incrementally changed for different levels of precompilation.*

Let's say, we want to compile design units specified in files F1, F2 into a logical library L1.

So, the corresponding project file <project>.prj would be:

```
read_file –type sourcelist L1_sources.f
read_file –type sourcelist libmap.f
```

Where, libmap.f contains:

- `-lib L1 < L1_path> ##` for providing logical to physical mapping
- `-work L1 ##` identifies the work library
- and, L1_sources.f contains the design files

…….

F1

F2

……...

This reads the design units described in F1 and F2, and creates the compiled binaries in <pathL1>.

NOTE: *Make sure that the lower level precompile libraries are syntactically clean and error-free before using them at higher level.*

Now, design description in files F3 and F4 make use of these design units (compiled in L1).

So, the project file would be:

```
read_file –type sourcelist L2_sources.f
read_file –type sourcelist libmap.f
```

where, libmap.f contains

```
-lib L1 <pathL1>
-lib L2 <pathL2>
```

```
-work L2
```

and, L2_sources.f contains the design files

……..

F3

F4

…….

This reads the design units described in F3, F4. Wherever needed, appropriate descriptions are picked from <pathL1>. The compiled binaries are dumped in <pathL2> (i.e. logical library WORK)

It is also allowed to have same physical path for multiple Logical libraries, however, same Logical library cannot be mapped to different physical paths.

The SpyGlass VHDL environment comes with the following precompiled logical libraries:

- IEEE
- STD
- SYNOPSYS

These libraries are visible by default to SpyGlass. Hence, there is no need to provide the mapping for the above mentioned 3 libraries.

Libraries compiled on any 32 bit platform are reusable on any other 32 bit platform; similarly, libraries compiled on any 64 bit platform are reusable on any other 64 bit platform.

In general, for Precompiled steps, it's best to go with the Mixed Flow, because it's highly possible that at some stage in the precompilation steps, some interleaving of Verilog with VHDL might take place.

With this basic background, we are ready to do a design read – using precompiled steps.

The relevant sources are put into sources.f file

The library mapping information is put into libmap.f file

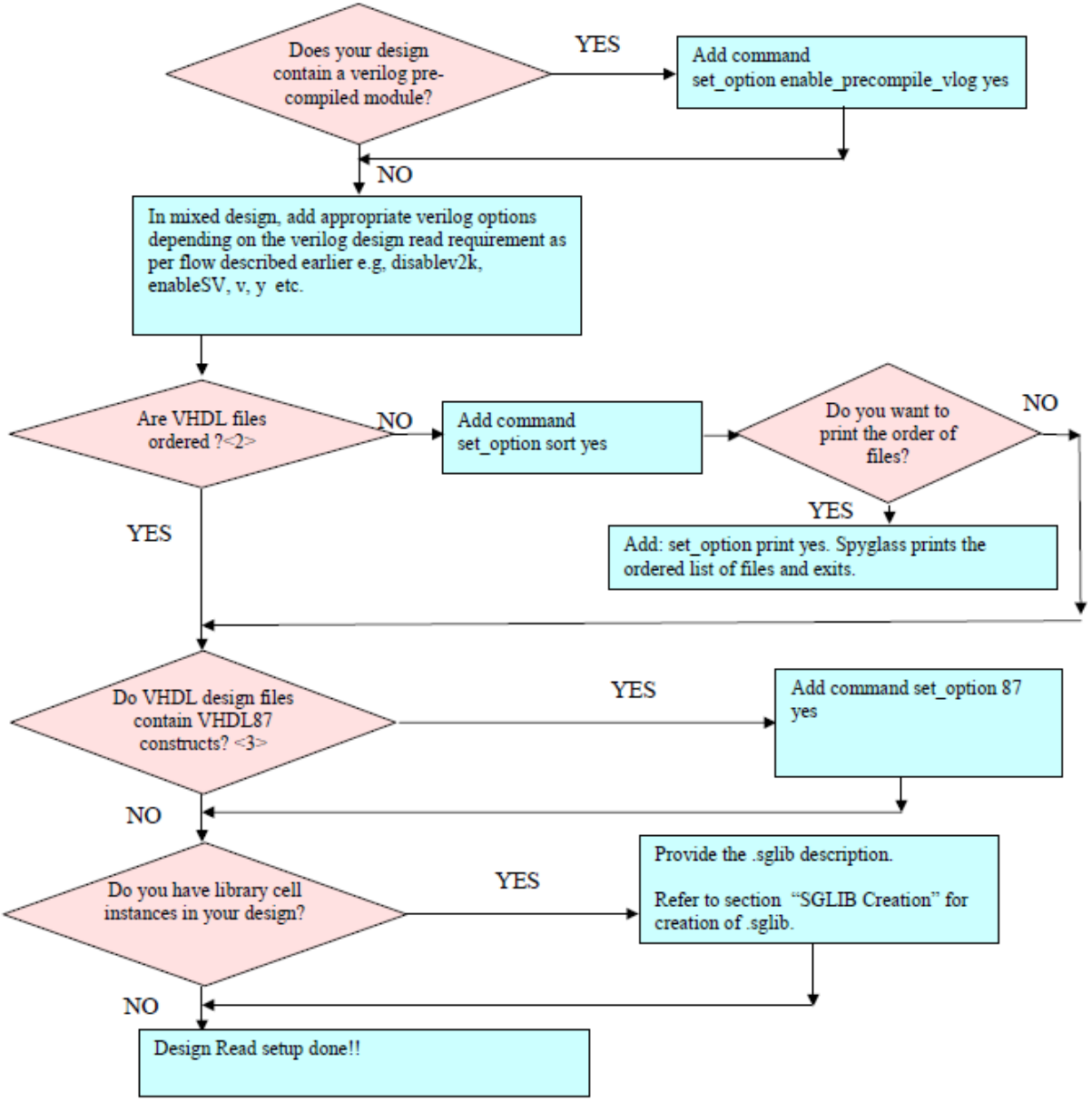
To compile a VHDL/verilog library in the SpyGlass environment, you must tell SpyGlass:

- the name of the working directory into which the library is to be compiled (using the `work` command in the libmap.f file, or, using: set_option work in the project file)
- the physical location of this library (using the `lib` command to set the logical to physical mapping in the libmap.f file, or, using set_option lib <path> in the project file),
- the names of the VHDL files to be compiled in sources.f file (using: read_file –type sourcelist sources.f in the project file)

  If the “work” specification at the compilation stage is not given correctly, then there won’t be any warning or error flagged at this stage. E.g. if something was supposed to be compiled into L1, but work was specified as “L2”, no message would be reported at this stage. However, at a later stage, when a higher level compilation tries to locate the item in L1, it will not be found.

Now, that your sources.f and the libmap.f is complete, you might need to add additional commands in your project file, as per the flow chart below.

Step 1-Precompile the Lowest-level Library


Does your design contain a verilog pre-compiled module?
YES
Add command set_option enable_precompile_vlog yes
NO
In mixed design, add appropriate verilog options depending on the verilog design read requirement as per flow described earlier e.g. disablev2k, enableSV, v, y etc.
Are VHDL files ordered ?<2>
NO
Add command set_option sort yes
Do you want to print the order of files?
NO
YES
Add: set_option print yes. Spyglass prints the ordered list of files and exits.
YES
Do VHDL design files contain VHDL87 constructs? <3>
YES
Add command set_option 87 yes
NO
Do you have library cell instances in your design?
YES
Provide the .sglib description.
Refer to section "SGLIB Creation" for creation of .sglib.
NO
Design Read setup done!!

NOTES

1. If you are not interested in elaboration during precompilation stages, use “set_option noelab yes” command in the project file in order to save elaboration time. However, there is a risk that at the top level an elaboration fails, because of some issues which could have been caught and corrected at the lower level itself. If you are not elaborating the design, it is advisable not to provide sglib. Providing these libraries only increases the overall processing time (because of processing time required for these libraries), even though these libraries will not be used (if elaboration is not being performed).
2. All VHDL files need to be ordered as all the design-entity that is being referenced should have been defined before being referenced.

   If files are not in order, you would encounter various STX_Errors, Warnings – depending on the exact content of the files and the order in which they are specified.

   If in doubt, feel free to add option for sort.
3. Using VHDL87 constructs without the set_option 87 command may result in STX/WRN, such as:

   ```
   WRN_499      WRN_499   LangWarning   … Using 1076-1987 syntax for file declaration
   ```

   If the libraries are needed to be created for 64-bit platform on a 32 bit machine, “set_option dump_all_modes yes” command as precompile libraries are not shared between different platforms.

   This might be needed, because the lower level design units being small enough can be compiled on a 32 bit machine itself. However, the top level design being large enough might need to be run on a 64 bit machine.

Step2-Using Compiled Libraries at a Higher Level

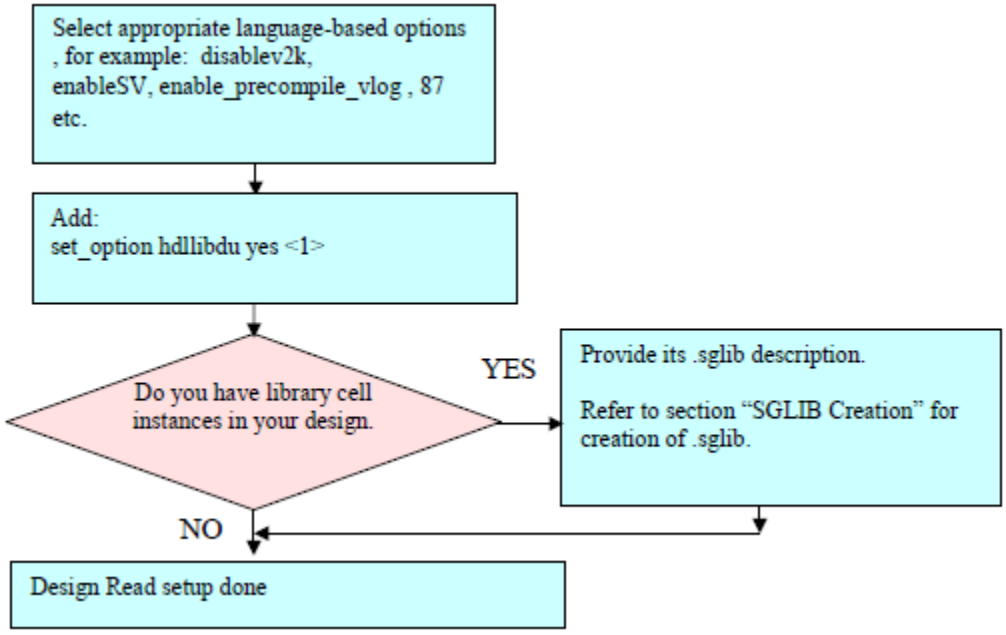
Create your sources.f and the libmap.f as explained above.

Incorrectly specifying precompiled library paths may result in any of the following messages (please refer to *Precompiled Library Mapping* section):

- STX_11: Use of un-declared identifier
- STX_464: Design unit denotes neither a component nor a procedure.

- WRN_384: Design unit does not denote a library or package.
- ErrorAnalyzeBbox: Instance has no definition; black box behavior assumed.

Now, add additional commands in the project file, as per the flowchart below.



NOTE: *The option "hdllibdu" does the rule-checking at the lower level of precompilation also. This option is useful if you are not sure that the lower level precompiled libraries are already clean with respect to rules/checks of your interest.*

Precompiled Library Mapping

As mentioned earlier, if library mapping is not specified properly during compilation stage, the user will not get any error/warning/indication at this stage. However, when the user tries to use this precompiled library at the next stage, he might get errors/warnings etc.

Inability to compile the files in the right place is the most common cause for people having difficulty in Design Read (for VHDL/Mixed involving multiple steps of compilation).

Let’s take an example.

Intent was:

Compile F1.vhd into a logical library L1

Compile F2.vhd. This file has something called:

LIBRARY L1;

The command should be:

```
>%spyglass -project <project file> -designread
```

where, project file contains:

……..

```
read_file –type sourcelist sources.f
read_file –type sourcelist libmap.f
```

……..

and, libmap.f contains

……..

-lib L1 <pathL1>

- work L1

……..

Suppose, during the first stage, the user does not give L1 to the work command, this stage would still go through.

There is no indication in file F1.vhd to say that this needs to be compiled into library L1.

The problem will be faced during the next stage, when the user tries to compile the files in L2_sources.f

Design units will be searched in <pathL1>, while, in the first stage, they were not dumped there (because, the user forgot to specify work L1)

This just adds to the complexity.

So, a quick cheat-sheet kind of mechanism could be:

Map all logical libraries to the same physical path: ./WORK

So, the entry for the above example in the libmap.f file would be modified as:

- lib L1 ./WORK

The only situation where it might not work is:

When there are multiple design-units having different descriptions, but, the same name, contained in different logical libraries; if their physical path becomes same, it's not possible to maintain the distinction between them.

NOTE: *For an example of how to apply precompile and Check Methodology, please see Appendix - Single Step Compilation.*

Various Start Points

One can depend on existing simulation scripts etc. to decide which files to compile into which library.

1. **VerilogXL/VCS users**: If the Design input to the SpyGlass is compatible to tools like VerilogXL/VCS, then the project file can be created as follows

   ```
   read_file –type sourcelist src.f
   set_option y ../mylib
   set_option libext  <extensions>
   set_option define <defines>
   set_option incdir <include directories>
   ```

2. **MTI scripts**: If the Design-Input is from MTI users:

   Translate your modelsim.ini file into libmap.f file as follows:

   The library mapping is specified using the following style, under: [LIBRARY] section

   L1 = ./L1_path -> -lib L1 ./L1_path

   Translate your modelsim script file as follows:

   vmap L2 = L2_path -> Put: -lib L2 ./L2_path into libmap.f file

   vcom -work LIB1 b.vhd c.vhd d.vhd ->

   Put all source files < b.vhd c.vhd d.vhd> into sources.f

   Put: –work LIB1 into libmap.f

TABLE 1 TRANSLATION TABLE:

MTI Options	Equivalent SpyGlass commands in project file
Vlog	set_option enable_precompile_vlog yes
-f <path>	read_file –type sourcelist <path>
-sv	set_option enableSV yes
-work <path>	set_option work WORK set_option lib WORK <path> [in libmap.f file]
+libext+<suffix>	set_option libext <suffix>
+define+<macro[=value]>+	set_option define <macro[=value]>
+incdir+<dir>	set_option incdir <dir>
-v <file>	set_option v <file>
-y <directory>	set_option y <directory>
Vcom	(No Action)
-93	(No Action) as 93 is default in SpyGlass
-87	set_option 87 yes
vmap <llib> <plib>	set_option lib <llib> <plib> [in libmap.f file]

3. **NCsim scripts**: If the Design-Input is from NCSim Users

   Translate each of the following commands from your cds.lib/hdl.var into libmap.f file as follows:

   DEFINE foo <path> -> -lib foo <path> Add to file libmap.f
   Now, translate your NCsim script commands as follows:

   ncvhdl -WORK <lib> ..vhdl files.. ->

   Put all .vhdl files into sources.f

   Put –work <lib> into libmap.f

   ncvlog -WORK <lib> ...verilog files ->

   Put all verilog files into sources.f

   Put –work <lib> into libmap.f

   Put set_option enable_precompile_vlog yes into project file

   Last ncvlog/ncvhdl -> Collect all additional arguments into put into the

project file, through set_option

For NCSim, default is VHDL87 while for SpyGlass, it is VHDL93, hence:

ncvhdl invocation without any language flavor ->

Put `set_option 87 yes` into project file

ncvhdl invocation with -93 flavor -> No extra step, because of “V93"

TABLE 2 TRANSLATION TABLE:

NCsim Options	Equivalent SpyGlass commands in project file
Ncverilog	set_option enable_precompile_vlog yes
+nc64bit or -64BIT	Append -64bit on the command line
-f <file> or –FILE <arg>	read_file –type sourcelist <file> or <arg>
+work+<arg> or WORK <arg>	set_option work <arg> in libmap.f or set_option work in project file
+sv or –SV	set_option enableSV yes
+hdlvar+<arg> or HDLVAR <arg>	set_option work <work_dir> in libmap.f
+define+<macro>	set_option define <macro>
+incdir+<dirs>	set_option incdir <dirs>
+libext+<ext>	set_option libext <ext>
+cdslib+<arg> or –CDSLIB <arg>	parse the file to extract logical/physical lib mappings and add to libmap.f with-lib option.
+nclibdirname+<name>	set_option projectwdir <name>
ncvlog	set_option enable_precompile_vlog yes
-V1995 or –V95	set_option disablev2k yes
-LOGFILE <arg> or +LOGFILE <arg>	The logfile location is determined by projectwdir specification. Cannot be specified independently.
Ncvhdl	set_option 87 yes
-RELAX+<name>	set_option relax_hdl_parsing yes
-V93	“remove” set_option 87 yes // if present
Ncelab	set_option 87 yes; also ensure that the option noelab is not set to yes.

LIB.ENTITY(ARCH)	set_option top ENTITY.ARCH
LIB.ENTITY	set_option top ENTITY

4. **DC scripts**

   The following commands in DC scripts (initial setup files for DC) needs to be translated into a libmap.f file as follows:

   define_design_lib L1 -path ./L1_path -> -lib L1 ./L1_path

   Now, the synthesis script needs to be translated as follows:

   analyze -format vhdl -work L1 a.vhd ->

   Put –work L1 into libmap.f

   Put a.vhd into sources.f

   analyze -format verilog -work L2 b.vhd ->

   Put –work L2 into libmap.f

   Put b.vhd into sources.f

Design-Read with Single-Step Compilation

The primary intent of single-step compilation is to precompile all libraries in a single SpyGlass run, so that there is no longer a need for lengthy scripts, which run multiple times during SpyGlass analysis to individually compile each precompile library.

There are various benefits of single-step compilation, as listed below:

- It eliminates the need for multiple precompilation steps. So, the user can combine all the different precompilation steps in a single SpyGlass run, which would reduce the overall precompilation time.
- There are approaches like “Makefile” based compilation etc. that try to reduce the precompilation time by having only impacted libraries compiled again. The need for such environments would reduce because single-step precompilation would anyway ensure that only impacted libraries are compiled in subsequent runs.
- If there is some common option that needs to be passed in each precompile step, then currently it needs to be added at all SpyGlass invocation. However, with single-step compilation, user needs to add it only once, and it would apply uniformly to all libraries.

For example, let us assume that we have 3 logical libraries L1, L2, and L3, and, the corresponding specification in sources.f is as follows:

```
set_option libhdlfiles L1 "foo1.v foo1.vhd"

set_option libhdlfiles L1 "foo2.v foo2.vhd"
# L1 specified again, this file would be separately compiled.
# Now L1 will have definition of foo1.v foo1.vhd, foo2.v and
# foo2.vhd.

set_option libhdlfiles L2 "foo3.v pack.vhd"

set_option libhdlfiles L3 "ip1.v ip2.v"

set_option libhdlfiles L1 "foo4.vhd"
# L1 specified again, this file would be separately compiled.
# Again, L1 gets appended with foo4.vhd
```

These files would be precompiled in the libraries L1, L1, L2, L3 and L1 respectively. Each `libhdlfiles` command would be taken as separate unit of compilation, and will not be combined with any other

`libhdlfiles` for same or different logical library. If we combine various `libhdlfiles` for same logical library, then it can yield different precompilation results (due to say compiler directives like celldefine, macros etc.) compared to if these are compiled separately. User should create one `libhdlfiles` command each for each individual compilation run.

It normally happens that various libraries have dependency on each other, hence this order of `libhdlfiles` is important to ensure that various libraries are precompiled in the correct order and "`sort`" option is not used with `libhdlfiles` for sorting.

It is assumed that RTL files specified with a given `libhdlfiles` can have backward dependency only on earlier `libhdlfiles` specification, but no forward dependency on subsequent `libhdlfiles` specification. Please note that in case various `libhdlfiles` specification have cyclic dependency, then compilation of one of the libraries would FAIL with STX, as it won't find a design unit which is going to be compiled after it.

If there are any FATAL issues found during precompilation of a given library, run would abort, and any subsequent libraries won't be compiled.

In order to run SpyGlass console:

Put (see: Note 1 below the following flowchart):

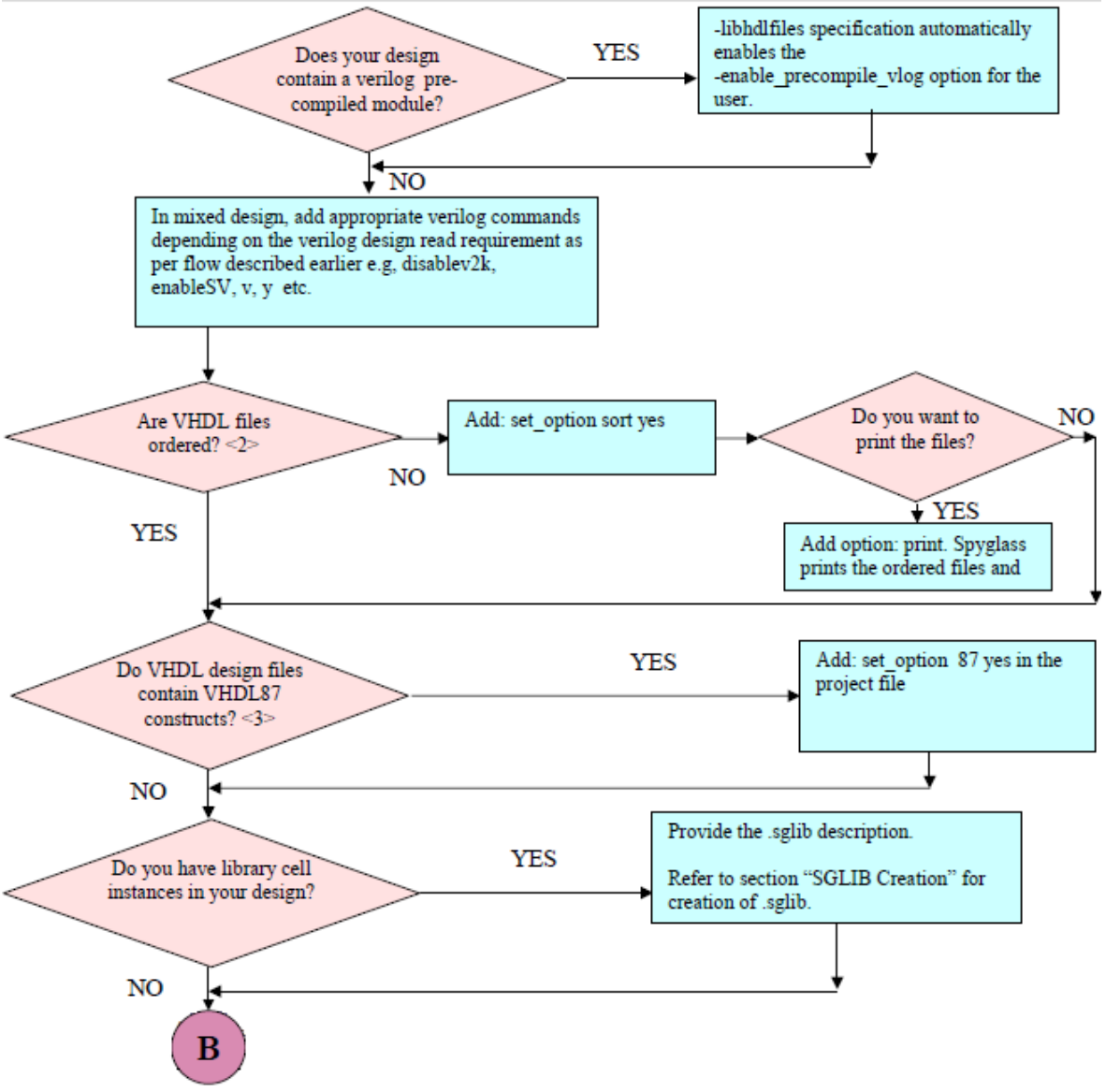
all the design files in sources.f ;

all library mapping into libmap.f;

all libhdlfile mapping (of logical library and the files that need to be compiled in it) into libhdl.f and

create the corresponding project file, using: read_file –type.

Now, modify your project file, based on the flowchart given below


Does your design contain a verilog pre-compiled module?
YES
-libhdlfiles specification automatically enables the -enable_precompile_vlog option for the user.
NO
In mixed design, add appropriate verilog commands depending on the verilog design read requirement as per flow described earlier e.g, disablev2k, enableSV, v, y etc.
Are VHDL files ordered? <2>
NO
Add: set_option sort yes
Do you want to print the files?
NO
YES
Add option: print. Spyglass prints the ordered files and
YES
Do VHDL design files contain VHDL87 constructs? <3>
YES
Add: set_option 87 yes in the project file
NO
Do you have library cell instances in your design?
YES
Provide the .sglib description.
Refer to section "SGLIB Creation" for creation of .sglib.
NO
B

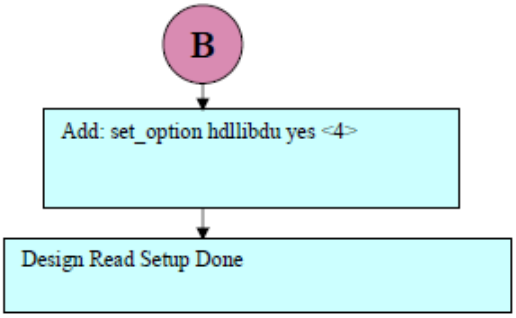


FIGURE 1. Library Compilation In Single Step

1. The complete path of RTL source file that would be precompiled in given logical library needs to be specified. This RTL file can be a Verilog or VHDL file(s) containing any of the valid constructs allowed by the respective language. There can be multiple files specified here, where some could be Verilog and others VHDL. If multiple files are specified then these should be enclosed in single/double quote (for software to identify the end of list).

   <Sources.f> - This file contains the RTL files and the libhdlfiles specification.

   Please note that if a library is found out-of-date due to say its HDL files being updated etc., then it would be re-compiled. Further, all the libraries that are dependent on it, would also be re-compiled even if their HDL files haven't been updated. Also, if any of the design options are changed like define, pragma etc. from one compilation to another, then all libraries would be marked out-of-date and re-compiled. For example,

   ```
   set_option libhdlfiles L1 "a1.v a2.v"
   set_option libhdlfiles L2 "a3.vhd"
   set_option libhdlfiles L3 "a4.v a5.vhd"
   ```

   In this case, if say a3.vhd has been changed, then we will re-compile L2. If L3, which is defined after L2, is using L2, then L3 would also be re-compiled. Further, if any of the design options, such as pragma have been changed, then we will re-compile all libraries L1 to L3 again, just as if it is a fresh run.

2. All VHDL files need to be ordered as all the design-entities that are being referenced should have been defined before being referenced.

   If files are not in order, you would encounter various STX_Errors, Warnings – depending on the exact content of the files and the order in which they are specified.

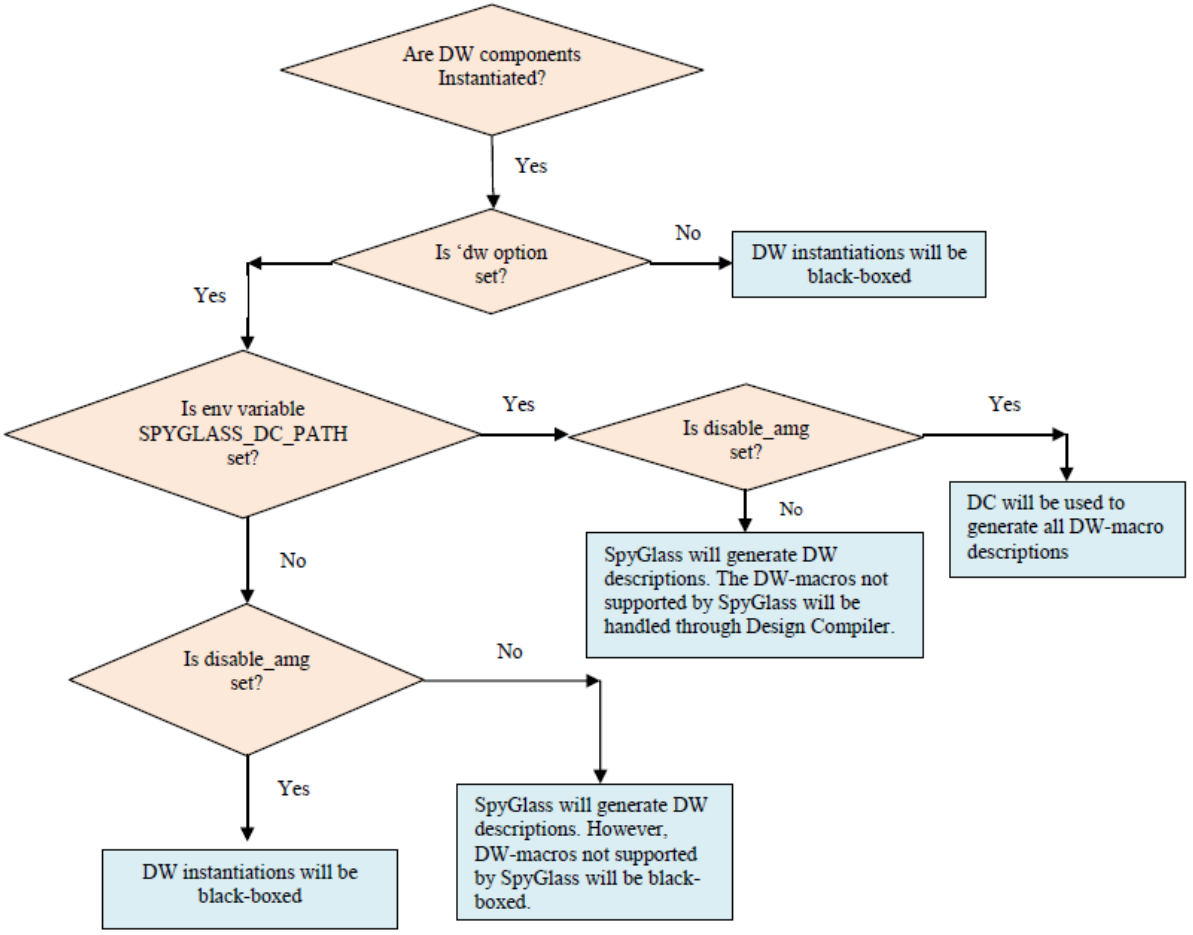
   If in doubt, feel free to add: set_option sort yes.

3. Using VHDL87 constructs without –setting the “87” option may result in STX/WRN, such as:

   ```
   WRN_499      WRN_499   LangWarning   … Using 1076-1987 syntax for file declaration
   ```

4. The option `hdllibdu` does the rule-checking at the lower level of precompilation also. This option is useful if you are not sure that the lower level precompiled libraries are already clean with respect to rules/checks of your interest.

Dealing with DesignWare® Components



Introduction

SpyGlass supports usage of DesignWare (DW) components in the RTL design. (If there are instantiations of modules with name DW_* then it indicates that the design contains DesignWare® components.)

To enable the DW flow, add the option: set_option dw yes.

SpyGlass provides the option of generating the DW module descriptions either through Design Compiler or through the internal SpyGlass module

generator.

DC is picked up from the path set through the environment variable SPYGLASS_DC_PATH. In some cases we find situations where there is a customized environment for invoking Design Compiler (i.e. dc_shell command is actually a user script, which in turn will call up the real dc_shell of Synopsys). Generally, this user script decides the version of the Synopsys tool, queues up the synthesis job in LSF etc. In such a situation, the user need not set the environment variable SPYGLASS_DC_PATH as the wrapper script (dc_shell) automatically sources its path. It needs to be ensured that the wrapper “dc_shell” is visible in the “path” settings.

The DW flow in SpyGlass is currently (since 4.6.0) as follows:

1. Enable DW flow with the `set_option dw yes` command, else all DW instantiations will be treated as black boxes.
2. From 4.6.0, you only need to specify `set_option dw yes` in the project file. The SpyGlass DW module generation capability is enabled by default (only from 4.6.0).
3. SpyGlass makes a list of all the DW instantiations in the design. For the ones which are supported internally (not all DW components are supported internally, but mostly is), SpyGlass generates the DW module descriptions.
4. For the ones that are not supported internally, SpyGlass generates the descriptions using Design Compiler. If Design Compiler license is not available, then these instantiations are treated as Black-Boxes, and a warning is flagged for each of these un-synthesized DW modules.
5. In case user wants all the DW components to be generated through Design Compiler, then he can choose to disable SpyGlass module generation by setting the option ‘disable_amg’.
6. The internally generated descriptions are mapped to the SpyGlass Primitive Library cells whereas the Design Compiler generated descriptions are mapped to GTECH cells by default.

Debug Options

1. For the DW module descriptions that are generated using Design Compiler, the user can generate debug information by specifying the project file command `set_option DEBUG dw`. SpyGlass will dump information on spyglass.out regarding its success/ failure of Design Compiler run.

2. SpyGlass dumps the RTL description of the internally generated module in the area: <$wdir>/spyglass_spysch/dw/SPY_DW_WORK/ .cache_dir_amg/DW01_binenc_0_area.v. So you may want to check if the dumping of the module description is indeed happening in the customer flow

Dealing with Syntax Errors

Check the language variations used (e.g. V2K, SystemVerilog, VHDL 87, VHDL93 etc.) –vs. - language variation specified. Default variants in SpyGlass = VHDL93, Verilog 2001. If the variation being used is different, please specify the appropriate variation.

If the issue is still not resolved, check the exact syntax error being reported, and, try to correct that.

Check if this has to do with presence of Pragmas: translate_off/on; synthesis_off/on (see: Note on Pragma handling within SpyGlass mentioned at the end of this section)

If the issue is still not resolved, add “set_option stop <corresponding_module>”. It is also required to add the option top along with stop in order to avoid multiple top scenarios.

Pragma Handling within SpyGlass

For Verilog, SpyGlass supports the translate_off/on pragma. The code within this pragma is simply ignored for compilation, syntax checks etc.

For VHDL, SpyGlass supports the translate_off/on and synthesis_off/on pragmas. The code within these pragmas are compiled, checked for syntax etc.

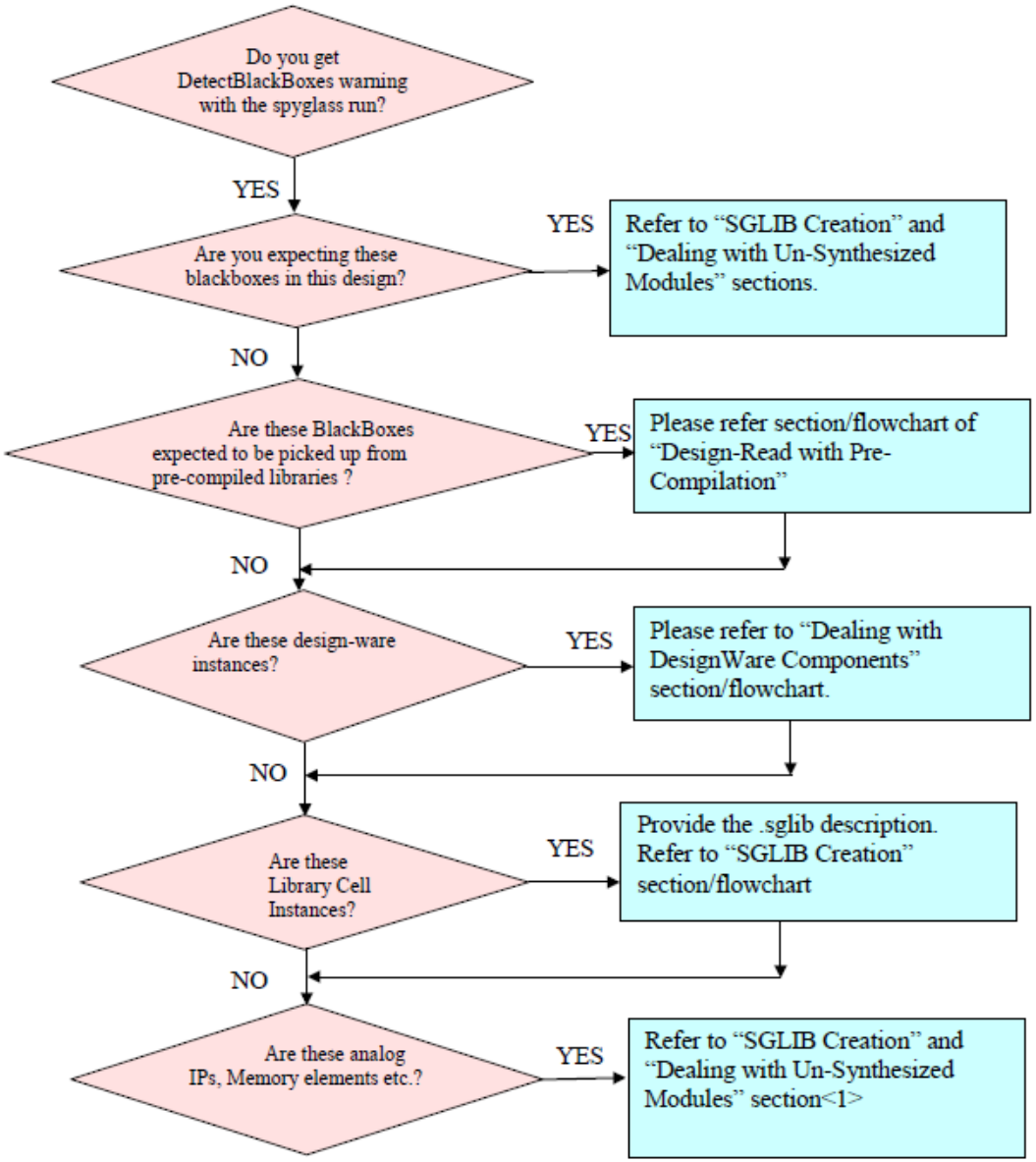
To prevent syntax checks/compilation within the code bounded by translate_off/on, specify the following command:

```
set_option hdlin_translate_off_skip_text yes
```

Similarly, to prevent syntax checks/compilation within the code bounded by synthesis_off/on, specify the following command:

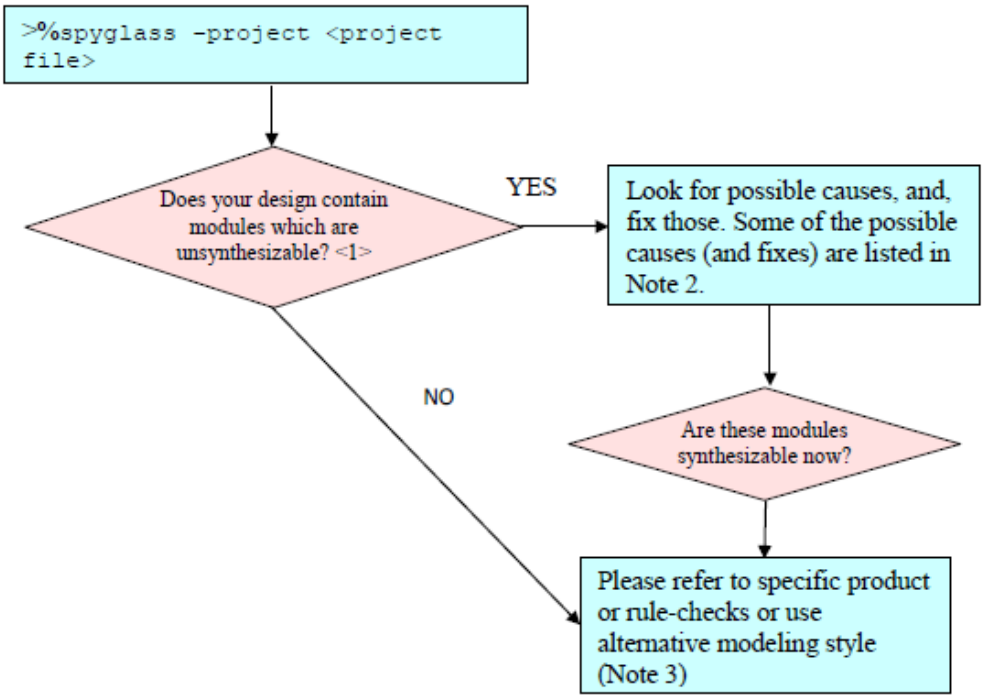
```
set_option hdlin_synthesis_off_skip_text yes
```

Dealing with Black Boxes



NOTE: *You can stop the synthesis of modules like analog IP, PLLs, memory blocks, and behavioral models by using “stop” option. It is also required to add top option along with stop in order to avoid multiple top scenarios.*

Dealing with Unsynthesized Modules



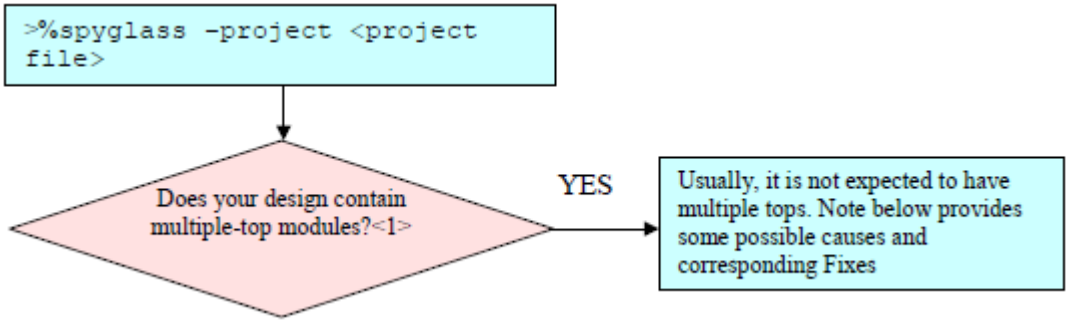
NOTES

1. These would be reported as: “ErrorAnalyzeBbox”.
2. Some of the causes for modules not being synthesized:

   High number of memory-bits: Use set_option handlememory yes. See the section: “DEALING WITH ‘OUT OF MEMORY’ SITUATIONS”.

   Encountered some constructs which are not synthesizable: Correct the module description to get around this issue
3. Depending on the type of analysis, you might want to use some alternative modeling style. Alternative modeling style can be used by different SpyGlass policies. Once SpyGlass Console is invoked, the “setup” stage takes you through a guided flow on creating alternative description for black-boxes, which are not yet resolved so far, using DW, sglib etc.

Dealing with Multiple Top Modules



NOTES

Multiple messages flagged by the multiple messages from the rule DetectTopDesignUnits.

The following could be some of the possible causes for having encountered multiple tops:

- Incomplete elaboration: Check if there are some ELAB errors. Due to these ELAB errors, the hierarchy tree has not been created correctly, causing an underlying module to become independent top. Correct the ELAB error.
- Some library files have been specified without the right option. Thus, unused design units are also becoming independent design units. Either use the right library option for the library, or, specify the `top` project file command.
- Some modules were stopped. So, some underlying module has become independent top. Set the option: top
- Actually, multiple designs have been specified. Set the option: top
- If you really want to do SpyGlass rule checking for each of the tops (e.g. various library cells to be run in one go) – you will need to check the “Allow Multiple Tops” button in Console GUI.

SGLIB Creation

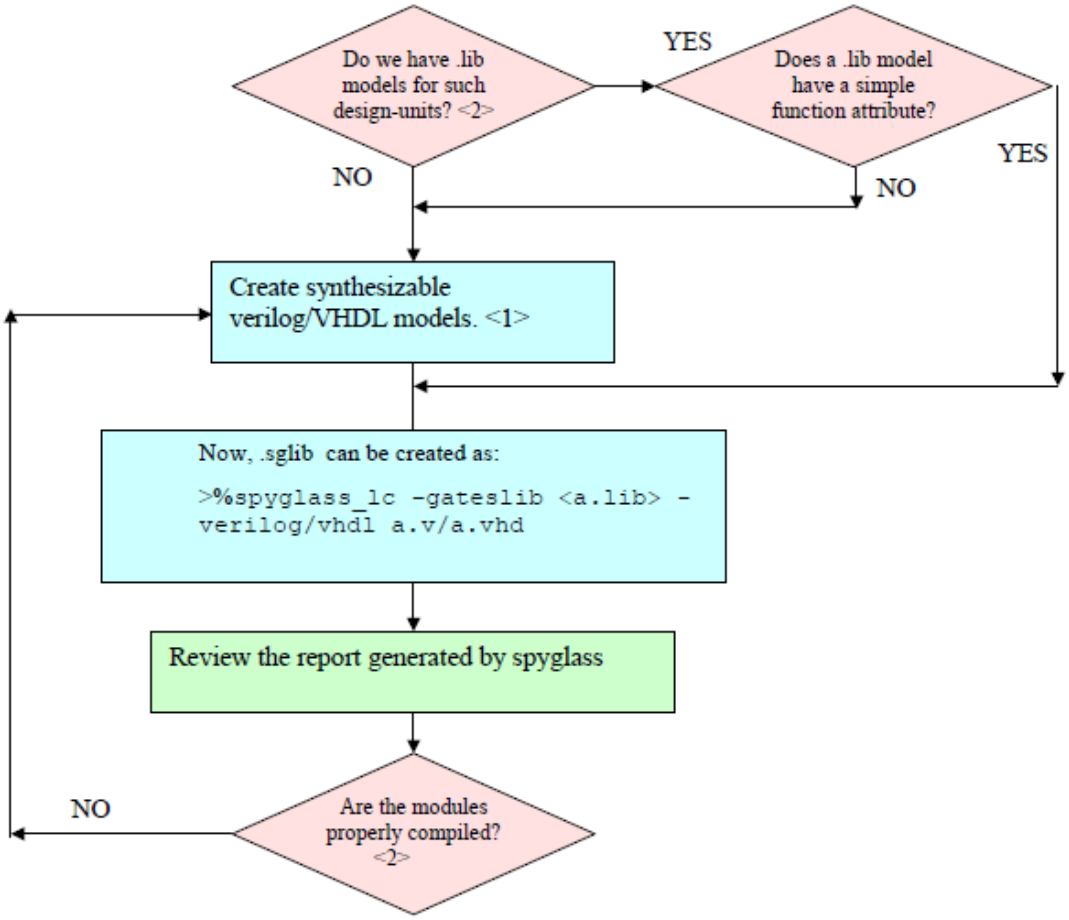


FIGURE 2. For technology cells, Memories, PLLs etc. for which no synthesizable models are available

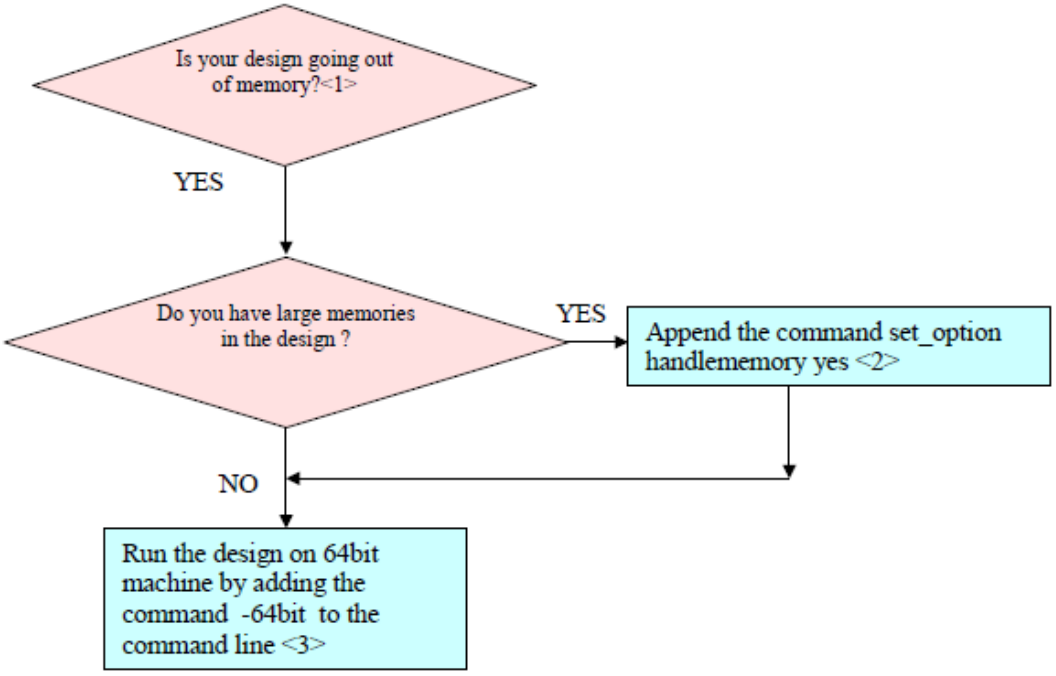
NOTES

1. Creating synthesizable models may not be practical for many models such as analog components or RF components. In most cases where a .lib model does not have a function statement, creating a synthesizable

Verilog or vhdl model will not be practical, and only interfaces can be inferred for these models.

2. If some of the library cells are not synthesizable or sglib file is not created then user can review the message details and .log file. SpyGlass reports the corresponding messages for the issues in sglib creation. If there is some problem in parsing then user might get LIBWARN_*, these all warnings are related to un-parsed library cells.
3. Now, that your .sglib is read, add command: read_file –type sglib <sglib files> to <project>.prj – in order to make it visible for your SpyGlass analysis.

Dealing with Out of Memory Situations



NOTES

1. If the design goes out of memory, SpyGlass exits with a error message at the screen output as:

   ```
   >% ERROR[100]  Memory Allocation Failed. Exiting …
   ```

2. If the handlememory option is set, then the memory-based rule-checks like FIFO checks etc may produce unreliable results. For further details on use of this option, please refer to section “The Memory reduction Feature” in Console User Guide.
3. It should be noted that on a 64 bit machine, since the pointer size is double (that of the pointers on a 32 bit machine), hence, by default memory foot-print on a 64 bit machine would be double that of the foot-print on a 32 bit machine. So, if your design went out of memory on a 4 GB 32-bit machine, when its rerun on a 64 bit m/c, this 64 bit m/c needs to have more than 8 GB of memory in order to proceed further.

SpyGlass Debugging

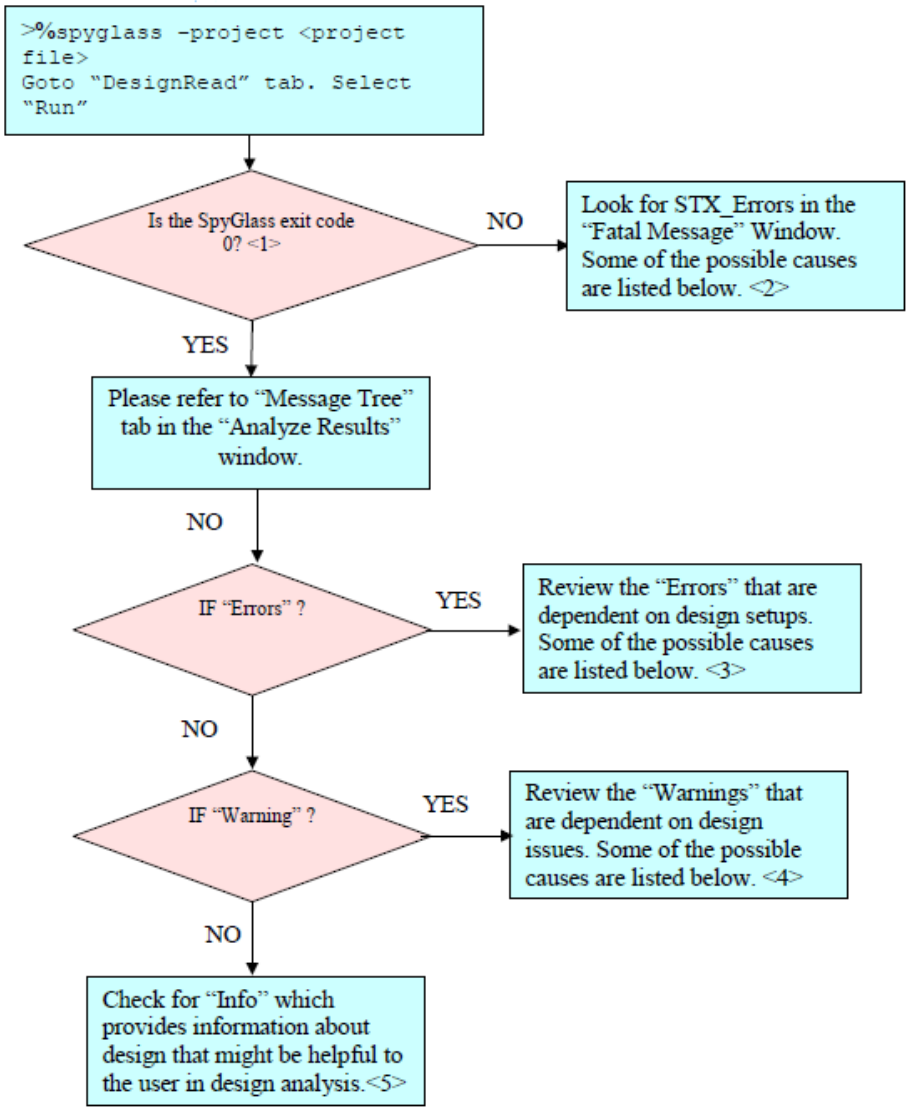


FIGURE 3. Console GUI Flow

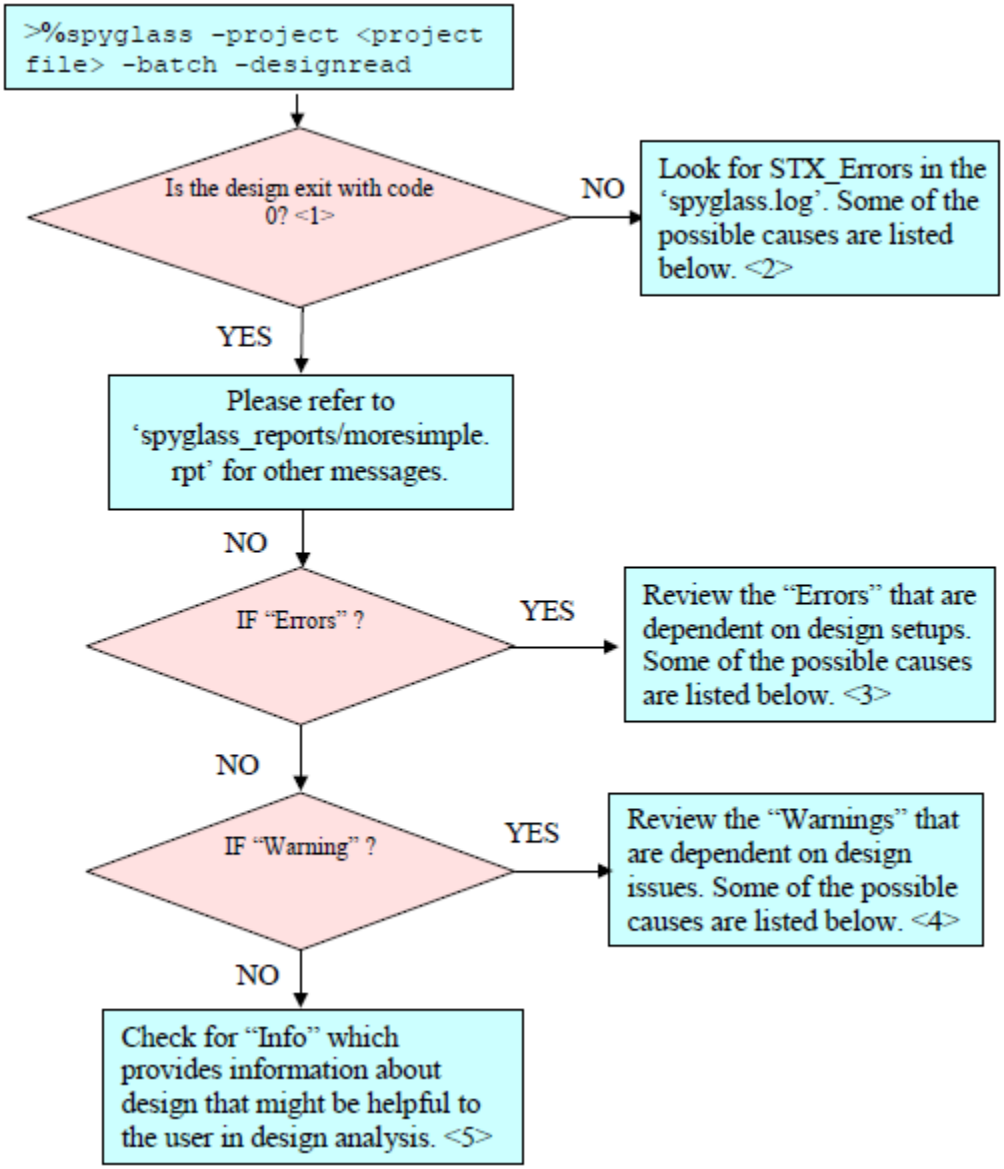


FIGURE 4. Batch mode flow

NOTES

1. SpyGlass Exit Code can be checked in “Session Log” window. Exit Code ‘0’ implies that the SpyGlass set up is clean. For Batch mode flow, look for ‘spyglass.log’ for details.
   There could be many causes for the Fatal Errors due to design set-ups. For Example: If the macros are not defined in SpyGlass, then it results in the following error message:
   a. STX_533 Syntax …. Used macro (…..) has not been defined.
      Another Example: Incorrectly specifying the precompiled library paths results in any of the following messages (Refer to the *Precompiled Library Mapping* section for details):
   b. STX_11 …….. Use of un-declared identifier
   c. STX_464 ……... Design unit denotes neither a component nor a procedure.
2. There could be many causes for the “Errors” in the design. For example, if some modules are not synthesized, you’ll get “SYNTH_*” errors. If there is some black-box existing in the design, you will get “ErrorAnalyzeBBox” errors.
3. Warnings are the design issues and it can be product dependent also. With each run, SpyGlass performs some rule-checks and warning messages are the violation of those rule-checks.
   For Example: In SpyGlass CDC product, Ac_unsync01/Ac_unsync02 message provided the information about “unsynchronized crossing, and in the SpyGlass Lint product, W337 gives information about illegal values being used etc.
4. “Info” messages are the design data that might be useful to the user for the design analysis. It provides with design data like clock, timing, area etc info. This is also product dependent. For example, Clock_Info01 provides data for all the probable clocks in the design.

Design Read

Once your project file and other files that it refers to are ready, you can invoke SpyGlass Console as:

spyglass –project <project file>

In the Console GUI, now, start with Design Read. At this stage, you can decide to include “synthesis” step also (through a check-button provided) – as part of the design read.

This process:

1. Checks whether the design is syntactically correct and complete including checking for missing macros, inconsistent or undefined parameter/generic values, missing include files etc.
2. Reports the information of all the potential top-level design units in the design.
3. Reports all the black box design units. If you get some black boxes warning, refer to section “Dealing with Black Boxes” for more details. If you have not included synthesis as part of the design read step, it’s possible that besides whatever black boxes are reported at this stage, additional items might also become black boxes subsequently, due to some synthesis errors

Further Help

Sometimes, you might encounter a situation, wherein a design which can be read by other tools gives an error in SpyGlass. If you encounter such situations, please contact SpyGlass support.

Save-Restore

SpyGlass provides the feature of Design Save-Restore that allows you to perform different kinds of analysis, without having to re-synthesize the design multiple times. Thus, the feature may significantly improve runtime on repeated SpyGlass analysis runs when the HDL source is unchanged. This feature is particularly more helpful when the designer is handling very big designs, so that the design is synthesized once and can be restored several times for doing the analysis. With the Design Save-Restore feature enabled, SpyGlass saves the synthesized view of the design during the first analysis run. All subsequent analysis runs skip the design parsing step and/or design synthesis step provided the source design has not changed and the analysis options are same as those specified during design saves.

Under Design Save-Restore feature, during “save” you specify the required policies and during “restore”, run any sub-set of these policies. Optionally, you can also specify additional policies to be saved during the design save so that while these policies are not run during the design save but can be run during the design restore. All base policies are always saved together when you specify to save one or more base policies.

A “restore” run can be made using a ‘saved” design database, if:

- you have modified the .sgdc. You can use this to perform various what-if analysis with changing values of case-analysis/test_mode/any other constraint
- you want to modify the rule-sets, as long as these rule-sets are from the same policies, which were used to “save”. You can use this to perform SpyGlass analysis, based on a methodology, which recommends you to run smaller sets of rules, but, in steps.

Sometimes, depending on the selected rules’ characteristics, SpyGlass may not be able to work with saved design view and hence unchanged design’s re-parsing and/or re-synthesis may be required.

This feature is ON by default in console. It has been explained here, so that it is easy for you to understand why the second and subsequent runs appear to be faster.

Where to Look for More Information

Refer to SpyGlass Console User Guide for details.

For further queries, mail to *spyglass_support@synopsys.com*.

Appendix - Single Step Compilation

Let us assume a design, say **MY_TOP**, where SpyGlass is to be run using precompile and Check Methodology:

The design **MY_TOP** is structured with the following design components:

- Cell Library: lib1.lib
- VHDL Design Unit: lib2.vhd
- Verilog Design Unit: lib3.v

Along with precompilation of .lib, both VHDL and Verilog DUs need to be precompiled.

Now, suppose that the user wants to perform SpyGlass CDC checks using SpyGlass on **MY_TOP**. Then the user has to first create a SpyGlass constraints file, say clocks.sgdc, to define all of the design clocks and resets signals. The user can also depend on Console's Setup Wizard to generate the clocks.sgdc file. The Setup Wizard is a step-by-step approach to generate the information.

Similarly, if the user is interested in the constraints (SDC) analysis, then a corresponding SpyGlass constraint file say constraints.sgdc is to be created to include the actual SDC file(s) with requisite design information.

Once the SpyGlass constraint files (.sgdc) are ready, then **MY_TOP** can be analyzed by selecting appropriate templates as needed. For example, select SpyGlass CDC related templates for SpyGlass CDC checking or

constraints related templates for the SDC constraints analysis.

Here is the illustration of actual flow to carry out the above mentioned analysis using SpyGlass based “precompile and Check Methodology”:

STEP1: Precompilation Methodology

```
>% Compile the gateslib lib1.lib into .sglib
>% Compile lib2.vhd into “lib2”
>% spyglass lib3.v into “lib3”
```

The possible sets of options that can be used in the precompilation process are as follows:

- `precompile_lib.map`: This file can be used when to specify all library mapping
- `noelab`: This option is used to save elaboration time. Sometimes user might not be interested in elaboration at the precompiled libraries
- `hdlin_synthesis_off_skip_text`: This switch is used to prevent syntax checks/compilation within the code bounded by synthesis_off/on.
- `enable_precompile_vlog`: This switch is used to enable precompilation of Verilog libraries.

The above mentioned file ‘precompile_lib.map’ would look as follows:

```
set_option lib lib2 /path/to/lib2 // to use VHDL
precompile libraries at the top level
set_option lib lib3 /path/to/lib3 // to use Verilog
precompile library at the top level
```

STEP2: Check Methodology

```
Select the Methodology of interest through Console’s second
tab: Select Methodology and Goals
```

Some of the possible set of top-level commands that can be used during checking at the top level, which are as follows:

- `top_lib.map`: This file can be used to specify all the libraries
- `hdllibdu`: To enable RTLDU rule-checks on precompile libraries

Now, further the above mentioned file ‘top_lib.map’ would look as follows:

```
read_file -type sglib /path/to/lib1/lib1.sglib // to use the precompiled library cell instances at top-level

read_file -type sourcelist precompile_lib.map // where, the library mapping specified earlier can be reused
```

In this example, none of the rule-checking/sanity checking was performed at lower level of precompilation, hence appropriate options are added to carry out the same on the used precompiled libraries at the top level checking.

NOTE: *The options hdlin_synthesis_off_skip_text and hdllibdu, are optional and some users may choose not to use these. Please refer to Atrenta Console User Guide to know details about these commands.*

STEP 1: Single-Step Compilation Methodology

```
Run 1 >% Compile and run SpyGlass on the top level

Remember, to specify top and set hdllibdu
```

The possible set of commands that can be used in the single-step-compilation process, which are as follows:

- `set_option libmaphdlfiles L1 "<verilog_files>"`: This option can be used when all the verilog libraries are put in the logical library L1.
- `set_option libmaphdlfiles L2 "<vhdl_files>"`: This option is used to compile all the vhdl libraries in the logical library L2
- `set_option lib L1 /path/to/L1` to use verilog precompile libraries at the top level.
- `set_option lib L2 /path/to/L2` to use VHDL precompile libraries at the top level.

Run1 – This step will compile all the verilog libraries in L1 and vhdl libraries in L2 and run the top design mymod and perform the rule-checks provided in the template T1.

NOTE: *Precompilation will take place in Run2 if there is any change in the RTL files or precompilation dump is out of date.*