0.1 Inclusion of Foreign Language Code

Foreign Language Code - as it is referred by this section - is functionality that is included into SystemVerilog using the DirectC Interface. As a result, all statements of this section do apply only to code included using this interface; code included by using other interfaces (e.g. PLI, VPI) is outside the scope of this section.[1] Due to the nature of the DirectC Interface, most Foreign Language Code will usually be created from C or C++ source code, although nothing precludes the creation of appropriate object code from other languages. This section adheres to this rule, it's content is independent from the actual language used.[2]

In general, Foreign Language Code may be provided in form of object code (compiled for the actual platform) or in form of source code. The capability to include Foreign Language Code in object code form is a mandatory feature that must be supported by all simulators according to the guidelines in this section.[3] The capability to include Foreign Language Code in source code form is optional, but must follow the guidelines in this section, when it is supported by a simulator.[4] The inclusion of object and source code is assumed to be orthogonal and must no be dependent on each other.[5] Any interferences between both inclusion capabilities should be avoided.

This section defines

- how to specify the location of the corresponding files within the file system
- how to specify the files to be loaded (in case of object code) or
- how to specify the files to be processed (in case of source code)
- in which form object code has to be provided (as a shared library or archive)
- how to specify compilation information required for processing source code

It does not define

- how to organize the corresponding files within the system (a certain directory structure)
- how object code files are loaded
- how source code files are processed and finally included in a simulation (besides the requirement that this must be fully transparent to the user)

0.1.1 Inclusion of Object Code

Compiled object code to be loaded can be specified by one of the following three methods:

1. By an entry in a bootstrap file; this file and its content will be described in more detail below. Its location must be specified with one instance of the switch "-sv_liblist <pathname>". This switch may be used multiple times to define the usage of multiple bootstrap files.
2. By specifying the file with one instance of the switch "-sv_lib <path+name_without_extension>"; the filename must be specified without the platform specific extension. The SystemVerilog application is responsible for appending the appropriate extension for the actual platform.
   This switch may be used multiple times to define multiple libraries holding object code.
3. By including the file path and name without the platform specific extension in a list of entries defined within the environment variable SV_LIBRARIES; this variable uses the same notation and syntax as the corresponding variables[6]

1.This should define the scope: DirectC, but not PLI or VPI
2.This should show the independence from C/C++ or other languages (e.g. Java)
3.Object code inclusion is mandatory
4.Source code inclusion is optional, but must follow the rules in this section, if provided
5.Both methods are independent on each other

All these methods must be provided and must be made available concurrently, to permit any mixture of their usage. Every location can be either an absolute pathname or a relative pathname, where the content of the environment variable SV_ROOT is used to identify an appropriate prefix for relative pathnames.

Compiled object code must be provided in form of a shared library or as an archive library having the appropriate extension for the actual platform[1]. When there exists a shared library and an archive library in the same directory, the shared library is used; otherwise the directory search order decides upon the library to be loaded.

In case of multiple occurances of a file with the same name the above order also identifies the precedence of the search; as a result a file located by method 2) will override files specified by method 3). Any library must and will only be loaded once.

All compiled object code must be loaded in specification order similarly to the above scheme; first the content of the bootstrap file is processed starting with the first line, then the set of 'sv_lib' switches is processed in order of their occurance, then the content of the SV_LIBRARIES environment variable is processed from left to right.

The object code bootstrap file has the following syntax:

1. The first line must contain the string: "#!SV_LIBRARIES"
2. It follows an arbitrary amount of entries, one entry per line, where every entry holds exactly one library location. Each entry consists only of the <path+name_without_extension> of the object code file to be loaded and may be surrounded by an arbitrary number of blanks; at least one blank must preceed the entry in the line.
   The value <path+name_without_extension> is equivalent to the value of the switch '-sv_lib'.
3. Any amount of comment lines can be interspersed between the entry lines; a comment line starts with the character '#' after an arbitrary (including zero) amount of blanks and is terminated with a newline.

No other means shall be provided for identifying the location and filename of compiled object code to be included via the DirectC Interface.

Example:

Assuming the environment variable SV_ROOT has been set to "/home/user" and it is needed to include the following object files

- /home/user/myclibs/lib1.so
- /home/user/myclibs/lib3.so
- /home/user/proj1/clibs/lib4.so
- /home/user/proj3/clibs/lib2.so

6.E.g. LD_LIBRARY_PATH (on Solaris) and LPATH (on HP-UX)

1.Shared libraries use e.g. .so for Solaris, .sl for HP-UX and .dll for Windows, archives use .a on UNIX and .lib on Windows

then this can be accomplished by one of the methods described in the following figure. Each of the four methods is equivalent.

```
#!SV_LIBRARIES
 myclibs/lib1
 myclibs/lib3
 clibs/lib4
 clibs/lib2
```

Example1a: BOOTSTRAP FILE

```
...
-sv_lib myclibs/lib1
-sv_lib myclibs/lib3
-sv_lib clibs/lib4
-sv_lib clibs/lib2
...
```

Example 1b: SWITCH LIST

```
setenv SV_LIBRARIES "myclibs/lib1:myclibs/lib3:clibs/lib4:clibs/lib2"
```

Example 1c: ENVIRONMENT VARIABLE

```
...
-sv_lib myclibs/lib1 -sv_lib clibs/lib2
...

setenv SV_LIBRARIES "myclibs/lib3:clibs/lib4"
```

Example 1d: SWITCH LIST & ENVIRONMENT VARIABLE

Please note that although Example 1d loads the same set object code as the other examples, the load order is different.

0.1.2 Inclusion of Source Code

This subsection describes the inclusion of source code into a SystemVerilog simulation. It assumes the following characteristics:

- The SystemVerilog application permits the inclusion of source code
- The generation of object code from the provided source code is fully transparent to the user[1]

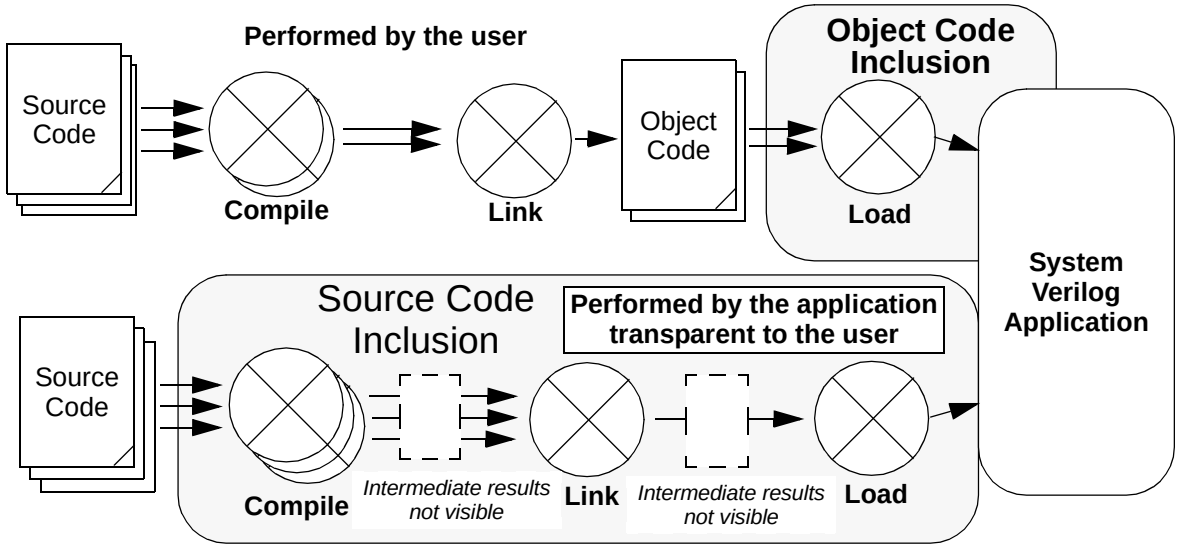

Table 0-1 Object Code Inclusion vs. Source Code Inclusion

1.Otherwise the guidelines of the previous section would apply to the created object code

Similarly the location of source code can be specified by one of the following two methods:

1. By an entry in a bootstrap file; this file and its content will be described in more detail below. Its location must be specified with one instance of the switch “-sv_srclist <pathname>“. This switch may be used multiple times to define the usage of multiple bootstrap files.
2. By specifying the file pathname with one instance of the switch “-sv_src <filepath>” (including the file extension). This switch may be used multiple times to define multiple source code files.

Additionally, directories holding include files needed for the compilation can be specified by:

3. By specifying one instance of the switch “-sv_inc <directorypath>”.
   This switch may be used multiple times to define multiple directories holding source code.
4. By including the include file directory in the list of entries provided as value of the environment variable SV_INCLUDES; this variable uses the same notation and syntax as the corresponding variables

All these methods must be provided and must be made available concurrently, to permit any mixture of their usage. Every location can be either an absolute pathname or a relative pathname, where the content of the environment variable SV_ROOT is used to identify an appropriate prefix for relative pathnames.

Source code must be provided in ASCII format, ’C’ files must use the “.c” extension, C++ or files written in another language must use another extension. The SystemVerilog application can use this guideline to distinguish ’C’ and ’C++’ source code[1], to be able to select the appropriate compiler. This default compiler selection can be overridden by the user by specifying the compiler applications tool invocation path in the environment variables SV_C_COMPILER (ANSI C compiler, used for *.c files) and SV_CPP_COMPILER (compiler for C++, used for files having not *.c as extension); the latter would also be the appropriate place for compiling source code written in another language.[2]

Furthermore the default compilation options set by the SystemVerilog application can be extended or replaced by the user when setting one of the environment variables identified in the following “Environment Variables for Overriding Compilation Options” on page 4

Table 1: Environment Variables for Overriding Compilation Options

Environment Variable	Remarks
SV_C_FLAGS	Replaces the standard C compiler options; resulting call is *<C compiler call> <SV_C_FLAGS> <include directories> -o <output file> <input file>*
SV_C_FLAGS_PREFIX	Appends the value of this environment variable to the standard C compiler options; resulting call is *<C compiler call> <SV_C_FLAGS_PREFIX> <standard options> <include directories> -o <output file> <input file>*
SV_C_FLAGS_SUFFIX	Prepends the value of this environment variable to the standard C compiler options; resulting call is *<C compiler call> <standard options> -o <output file> <input file> <include directories> <SV_C_FLAGS_SUFFIX>*
SV_CPP_FLAGS	Replaces the standard C++ compiler options; resulting call is *<C++ compiler call> <SV_CPP_FLAGS> <include directories> -o <output file> <input file>*

1.This defines the compiler selection criteria
2.This defines a method for the user to override the built-in compiler selection of the vendor

Table 1: Environment Variables for Overriding Compilation Options

Environment Variable	Remarks
SV_CPP_FLAGS_PREFIX	Prepends the value of this environment variable to the standard C++ compiler options; resulting call is *<C++ compiler call> <SV_CPP_FLAGS_PREFIX> <standard options> <include directories> -o <output file> <input file>*
SV_CPP_FLAGS_SUFFIX	Appends the value of this environment variable to the standard C++ compiler options; resulting call is *<C++ compiler call> <standard options> -<include directories> -o <output file> <input file> <SV_CPP_FLAGS_SUFFIX>*

All compiled object code must be compiled in specification order similarly to the above scheme; first the content of the bootstrap file is processed starting with the first line, then the set of 'sv_src' switches is processed in order of their occurance.Similar applies to the inclusion of the include directories in the compilation; the order of specifitation must be preserved.

The source code bootstrap file has the following syntax:

1. The first line must contain the string: "#!SV_SOURCES"
2. It follows an arbitrary amount of entries, one entry per line, where every entry holds exactly one source code file location. Each entry consists at least of the <path+name_without_extension> of the source code file to be loaded; at least one blank must preceed the entry in the line.
   The value <path+name_without_extension> is equivalent to the value of the switch '-sv_src'.

   Additionally, a list of directory pathnames (separated by blanks) may be specified after the location of the source code file, separated by colon (':')[1]. Each directory entry within this list is equivalent to the value of the switch '-sv_inc'.
3. Any amount of comment lines can be interspersed between the entry lines; a comment line starts with the character '#' after an arbitrary (including zero) amount of blanks and is terminated with a newline.

No other means shall be provided for identifying the location of user specific source code and include files. There is no need to locate or identify the object code created from these sources; this is under the discretion of the application, no further user interaction shall be needed to accomplish this. Also the creation of object code from this source code shall be fully transparent to the user.

Example:

Assuming the need to include the following source files into a simulation:

- /home/user/mycode/model1.c
- /home/user/sysc/model3.sc
- /home/user/proj1/code/model3.cc
- /home/user/proj3/c_code/model4.cpp

Additionally, the include directories

1.The resulting syntax of an entry is:
<source code entry> ::= <blank><pathname>[[<blank>]':'[<blank>]<directories>]
<directories> ::= <pathname>[<blank><pathname>]
<blank> ::= ' '[<blank>]

- /home/user/mycode/includes
- /home/user/common/sysc
- /home/user/proj1/util must be referenced

The first two examples show two possible methods of specifying the source files and include directories for the source code inclusion [assuming SVC_ROOT is set to /home/user]. Please note that within these examples it is not possible to have a finer granularity for assigning include directories to a source code file.

```
#!SV_SOURCES
 mycode/model1.c
 sysc/model3.sc
 proj1/code/model3.cc
 proj3/c_code/model4.cpp


setenv SV_INCLUDES
"mycode/includes:common/sysc:proj1/util"
```

Example 2a: BOOTSTRAP FILE & INCLUDE ENV VAR

```
-sv_src mycode/model1.c
-sv_src sysc/model3.sc
-sv_src proj1/code/model3.cc
-sv_src proj3/c_code/model4.cpp

-sv_inc mycode/includes
-sv_inc /home/user/common/sysc
-sv_inc proj1/util
```

Example 2b: SWITCH ONLY

Please note that any combination of bootstrap file, switches and/or include directories in the SV_INCLUDES environment variable is possible. This is shown in the next example, Example 2c.

```
#!SV_SOURCES
 sysc/model3.sc
 proj3/c_code/model4.cpp

setenv SV_INCLUDES "common/sysc:proj1/util"
```

```
-sv_src mycode/model1.c
-sv_src proj3/c_code/model4.cpp

-sv_inc mycode/includes
```

Example 2c: USING A COMBINATION OF BOOTSTRAP FILE, SWITCHES & INCLUDE ENV VAR

It is further worth to note that the bootstrap file permits a more granular assignment of include directories to source code files, as this is shown in the next example, Example 2d.

```
#!SVC_SOURCES
 mycode/model1.c : mycode/includes proj1/util common/includes
 sysc/model3.sc : common/sysc
 proj1/code/model3.cc : common/includes
 proj3/c_code/model4.cpp : proj1/util common/includes
```

Example 2d: ASSIGNING INCLUDE DIRECTORIES TO SOURCE CODE VIA THE BOOTSTRAP FILE

This example shows the specification of a compilation of

- mycode/model1.c (using a C compiler) with the include directories: mycode/includes, proj1/util, and common/includes
- sysc/model3.sc (using a C++ compiler) with the include directories: common/sysc
- proj1/code/model3.cc (using a C++ compiler) with the include directories: common/includes
- proj3/c_code/model4.cpp (using a C++ compiler) with the include dirs: proj1/util and common/includes