Annex C
Inclusion of Foreign Language Code

This annex describes common guidelines for the inclusion of Foreign Language Code into a SystemVerilog application. This intention of these guidelines is to enable the redistribution of C binaries in shared object form.~~:~~

— ~~enable the redistribution of C binaries in shared object form~~

— ~~ensure the compatibility for all linkage aspects that would appear in the C code.~~

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

In general, Foreign Language Code is provided in the form of object code (compiled for the actual platform) or source code. The capability to include Foreign Language Code in object-code form shall be supported by all simulators as specified here. ~~The capability to include Foreign Language Code in source code form is optional, but shall follow the guidelines in this annex if it is supported by a simulator. The inclusion of object and source code needs to be orthogonal and not depend on each other. Any interferences between both inclusion capabilities shall be avoided.~~

C.1 Overview

This annex defines how to:

— specify the location of the corresponding files within the file system

— specify the files to be loaded (in case of object code) or

— ~~specify the files to be processed (in case of source code)~~

— provide the object code (as a shared library or archive)

— ~~specify any compilation information required for processing source code.~~

Although this annex defines guidelines for a common inclusion methodology, it requires multiple implementations (usually two) of the corresponding facilities. This takes into account that multiple users can have different viewpoints and different requirements on the inclusion of Foreign Language Code.

— A vendor that wants to provide his IP in form of Foreign Language Code often requires a self-contained method for the integration, which still permits an integration by a third party. This use-case is often covered by a bootstrap file approach.

— A project team that specifies a common, standard set of Foreign Language code, might change the code depending on technology, selected cells, back-annotation data, and other items. This-use case is often covered by a set of tool switches, although it might also use the bootstrap file approach.

— An user might want to switch between selections or provide additional code. This-use case is covered by providing a set of tool switches to define the corresponding information, although it might also use the bootstrap file approach.

NOTE—This annex defines a set of switch names to be used for a particular functionality. This is of informative nature; the actual naming of switches is not part of this standard. It might further not be possible to use certain character configurations in all operating systems or shells. Therefore any switch name defined within this document is a recommendation how to name a switch, but not a requirement of the language.

C.2 Location independence

All pathnames specified within this annex are intended to be location-independent, which is accomplished by using the switch `-sv_root`. It can receive a single directory pathname as the value, which is then prepended to any relative pathname that has been specified. In absence of this switch, or when processing relative filenames before any `-sv_root` specification, the current working directory of the user shall be used as the default value.

C.3 Object code inclusion

Compiled object code is required for cases where the compilation and linking of source code is fully handled by the user; thus, the created object code only need be loaded to integrate the Foreign Language Code into a SystemVerilog application. All SystemVerilog applications shall support the integration of Foreign Language Code in object code form. Figure C1— depicts the inclusion of object code and its relations to the various steps involved in this integration process. **Revise this xref w/ Stu; also check/revise variable settings, etc.**

.

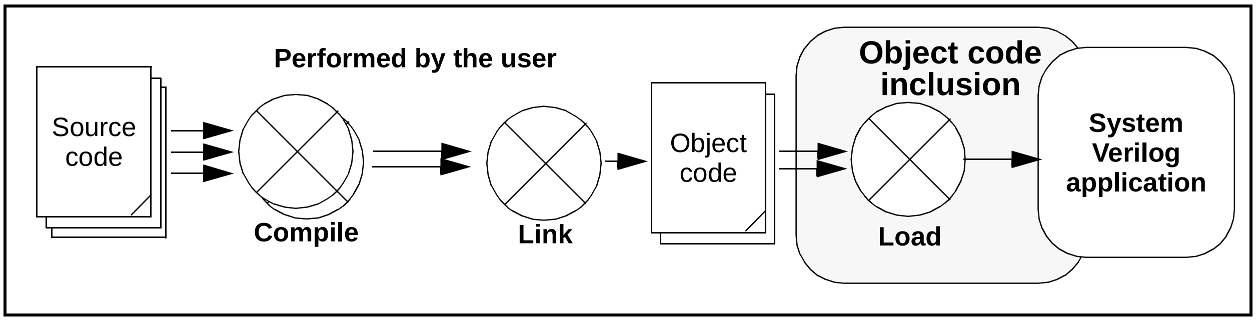


Figure C1— Inclusion of object code into a SystemVerilog application

Compiled object code can be specified by one of the following two methods:

1) by an entry in a bootstrap file; see section C.3.1 for more details on this file and its content. Its location shall be specified with one instance of the switch `-sv_liblist` *`pathname`*. This switch can 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` *`pathname_without_extension`* (i.e., the filename shall be specified without the platform specific extension). The SystemVerilog application is responsible for appending the appropriate extension for the actual platform. This switch can be used multiple times to define multiple libraries holding object code.

Both methods shall be provided and made available concurrently, to permit any mixture of their usage. Every location can be an absolute pathname or a relative pathname, where the value of the switch `-sv_root` is used to identify an appropriate prefix for relative pathnames (see section C.2 for more details on forming pathnames).

The following conditions also apply.

— The compiled object code itself shall be provided in form of a shared library having the appropriate extension for the actual platform.

NOTE—Shared libraries use, for example, `.so` for Solaris and `.sl` for HP-UX; other operating systems might use different extensions. In any case, the SystemVerilog application needs to identify the appropriate extension.

— The provider of the compiled code is responsible for any external references specified within these objects. Appropriate data needs to be provided to resolve all open dependencies with the correct information.

— The provider of the compiled code shall avoid interferences with other software and ensure the appropriate software version is taken (e.g., in cases where two versions of the same library are referenced). Similar problems can arise when there are dependencies on the expected runtime environment in the compiled object code (e.g., in cases where C++ global objects or static initializers are used).

— The SystemVerilog application need only load object code within a shared library that is referenced by the SystemVerilog code or by registration functions; loading of additional functions included within a shared library can interfere with other parts.

In case of multiple occurrences of the same file (files having the same pathname or which can easily be identified as being identical; e.g., by comparing the inodes of the files to detect cases where links are used to refer the same file), the above order also identifies the precedence of loading; a file located by method 1) shall override files specified by method 2).

All compiled object code need to be loaded in the 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 occurrence. Any library shall only be loaded once.

C.3.1 Bootstrap file

The object code bootstrap file has the following syntax.

1) The first line contains the string `#!SV_LIBRARIES`.

2) An arbitrary amount of entries follow, one entry per line, where every entry holds exactly one library location. Each entry consists only of the *`pathname_without_extension`* of the object code file to be loaded and can be surrounded by an arbitrary number of blanks; at least one blank shall precede the entry in the line. The value *`pathname_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.

C.3.2 Examples

1) If the pathname root has been set by the switch `-sv_root` to `/home/user` and the following object files need to be included:

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

then use either of the methods in Example C-1. Both methods are equivalent.

```
#!SV_LIBRARIES
 myclibs/lib1
 myclibs/lib3
 proj1/clibs/lib4
 proj3/clibs/lib2
```

Bootstrap file method

```
...
-sv_lib myclibs/lib1
-sv_lib myclibs/lib3
-sv_lib proj1/clibs/lib4
-sv_lib proj3/clibs/lib2
...
```

Switch list method

Example C-1Using a simple bootstrap file or a switch list

2) If the current working directory is `/home/user`, using the series of switches shown in Example C-2 (left column) result in loading the following files (right column).

```
-sv_lib svLibrary1
-sv_lib svLibrary2
-sv_root /home/project2/shared_code
-sv_lib svLibrary3
-sv_root /home/project3/code
-sv_lib svLibrary4
```

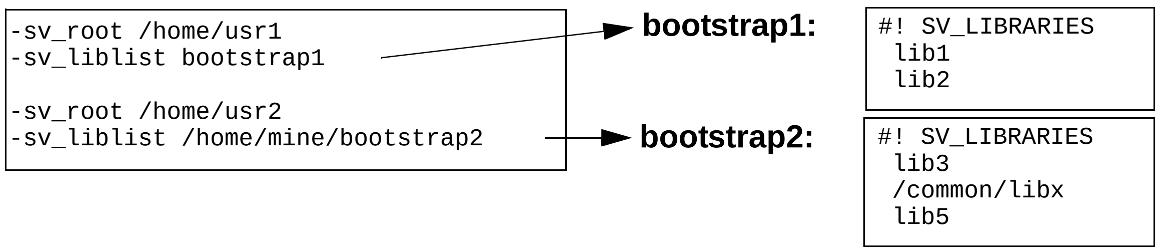
Switches

```
/home/user/svLibrary1.so
/home/user/svLibrary2.so

/home/project2/shared_code/svLibrary3.so

/home/project3/code/svLibrary4.so
```

Files

Example C-2Using a combination of `-sv_lib` *and* `-sv_root` *switches*

3) Further, using the set of switches and contents of bootstrap files shown in Example C-3:



```
-sv_root /home/usr1
-sv_liblist bootstrap1

-sv_root /home/usr2
-sv_liblist /home/mine/bootstrap2
```

```
#! SV_LIBRARIES
 lib1
 lib2
```

```
#! SV_LIBRARIES
 lib3
 /common/libx
 lib5
```

Example C-3Mixing `-sv_root` *and bootstrap files*

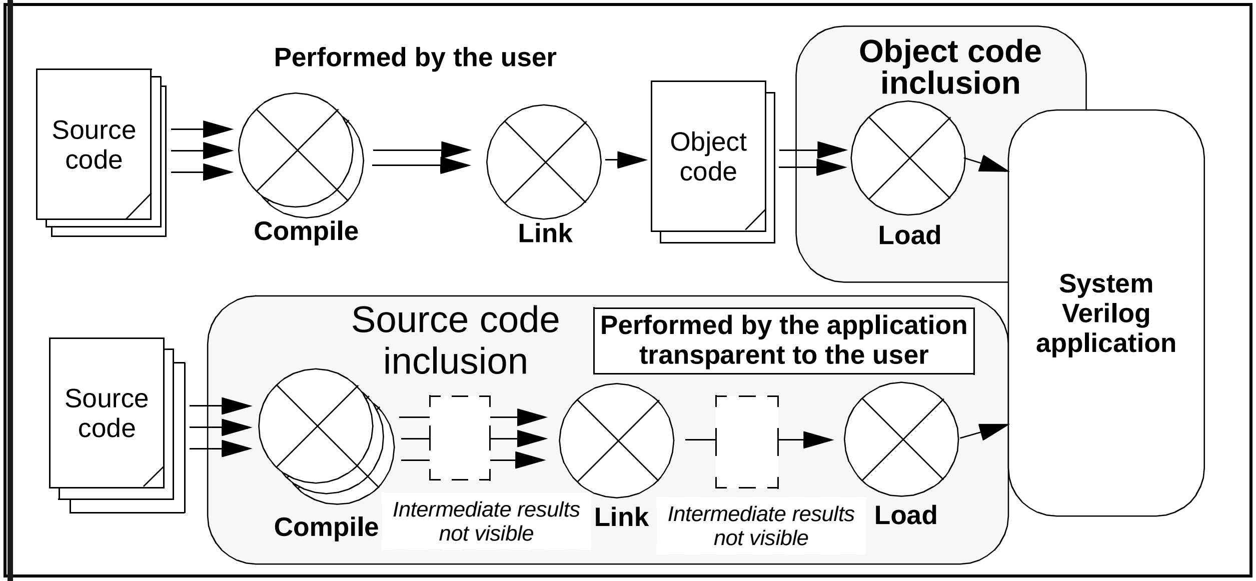
results in loading the following files:

```
/home/usr1/lib1.ext
/home/usr1/lib2.ext
/home/usr2/lib3.ext
/common/libx.ext
/home/usr2/lib5.ext
```

where *`ext`* stands for the actual extension of the corresponding file.

~~C.4 Source code inclusion~~

~~A SystemVerilog application can handle the compilation step and all subsequent activities for the user. As a result, the compilation, linking and loading steps are fully transparent to the user; the only requirement is to specify the source code files for compilation. Source code is an optional feature that can be supported by SystemVerilog applications to support the integration of Foreign Language Code. Figure C2— depicts the inclusion of source code versus object code.~~


Performed by the user
Source code
Compile
Link
Object code
Object code inclusion
Load
System Verilog application
Source code
Source code inclusion
Performed by the application transparent to the user
Compile
Intermediate results not visible
Link
Intermediate results not visible
Load

~~Figure C2—Source code inclusion vs. object code inclusion~~

~~Similarly to the inclusion of object code, source code can be specified by one of the following two methods:~~

1) ~~by an entry in a bootstrap file; see section C.4.3 for more details on this file and its content. Its location shall 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 can be used multiple times to define multiple source code files.~~

~~Any directories holding include files needed for the compilation can be specified as one instance of the switch `-sv_inc` *`directorypath`*. This switch can be used multiple times to define multiple directories holding source code.~~

~~The source code file specifications in a bootstrap file define their include directories associations directly within the file itself; those source code file specifications are not affected by include directory specifications with `-sv_inc`.~~

~~All the above methods shall be provided and made available concurrently, to permit any mixture of their usage. Every location can be an absolute pathname or a relative pathname, where the content of the switch `-sv_root` is used to identify an appropriate prefix for relative pathnames (see section C.2 for more details on forming pathnames).~~

~~A SystemVerilog application shall use the extension of the provided source code files to distinguish between C code (having the extension `.c`) and source code provided in a second language (having any other extension). The default for the second language is C++, but most other languages can be supported as well.~~

~~NOTE—The foreign language interface requires C-linkage, but is itself compiler- and language-independent.~~

~~C.4.1 Invocation~~

~~Dependent on the extension of the corresponding source code files, the SystemVerilog application is responsible for invoking the appropriate compiler with respect to the following scheme:~~

```
~~compiler prefix_flags includes flags filenames suffix_flags~~
```

~~In the above scheme, *compiler* is a placeholder for the invocation of the appropriate compiler, which shall be an ANSI-C compiler in cases where the source code file uses the extension `.c` and a C++ compiler otherwise. Likewise, *includes* is a placeholder for the list of include directories specified for the corresponding source code file. Usually this list contains zero, one, or multiple entries of the form *inc_opt include directory name* delimited by one or multiple blanks; where *inc_opt* denotes the compiler option to specify an include directory.~~

~~NOTE—While this option is named `-I` for many compilers, it can be different for other compilers. The SystemVerilog application shall place no blank between this option name and the include directory. When a user-supplied option requires a blank, this shall be inserted as part of the option name by enclosing the option name in double quotes like a string.~~

~~*filenames* is a placeholder for the specification of the input source code file and the destination file name. Usually this specification will be of the form *src_opt source file dst_opt dest file*; all of these entries are delimited by one or multiple blanks. *src_opt* denotes the compiler option to specify the input (source code) file, while *dst_opt* denotes the compiler option to specify the output (destination) file.~~

~~All three remaining specifications (*flags*, *prefix_flags*, and *suffix_flags*) are used to provide compilation flags; usually only *flags* contains compilation options in a normal compilation step.~~

~~C.4.2 Overriding compilation settings~~

~~All of the above specifications can be overridden by user-specific settings for both compilation modes. Switch specifications override only subsequent source code specifications, as shown in Table C1—.~~

~~Table C1—Switches for overriding compilation settings~~

~~Switch~~	~~Remarks~~
~~`-sv_c_compiler` *value*~~	~~Replaces the `compiler` part in case of a C compilation.~~
~~`-sv_c_inc_opt` *value*~~	~~Replaces the option to include an include directory name `inc_opt` in case of a C compilation.~~
~~`-sv_c_src_opt` *value*~~	~~Replaces the option to specify the source code file name in case of a C compilation.~~
~~`-sv_c_dst_opt` *value*~~	~~Replaces the option to specify the destination file name in case of a C compilation.~~
~~`-sv_c_flags` *value*~~	~~Replaces the `flags` part in case of a C compilation.~~
~~`-sv_c_prefix_flags` *value*~~	~~Replaces the `prefix_flags` part in case of a C compilation.~~
~~`-sv_c_suffix_flags` *value*~~	~~Replaces the `suffix_flags` part in case of a C compilation.~~
~~`-sv_cpp_compiler` *value*~~	~~Replaces the `compiler` part in case of a C++ compilation.~~
~~`-sv_cpp_inc_opt` *value*~~	~~Replaces the option to include an include directory name `inc_opt` in case of a C++ compilation.~~
~~`-sv_cpp_src_opt` *value*~~	~~Replaces the option to specify the source code file name in case of a C++ compilation.~~
~~`-sv_cpp_dst_opt` *value*~~	~~Replaces the option to specify the destination file name in case of a C++ compilation.~~
~~`-sv_cpp_flags` *value*~~	~~Replaces the `flags` part in case of a C++ compilation.~~
~~`-sv_cpp_prefix_flags` *value*~~	~~Replaces the `prefix_flags` part in case of a C++ compilation.~~
~~`-sv_cpp_suffix_flags` *value*~~	~~Replaces the `suffix_flags` part in case of a C++ compilation.~~

~~All provided source code need to be compiled in the specification order of the previous 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 occurrence. This also applies to the inclusion of any include directories in the compilation; the order of specification needs to be preserved.~~

~~C.4.3 Bootstrap file~~

~~The source code bootstrap file has the following syntax.~~

1) ~~The first line contains the string `#!SV_SOURCES`.~~

2) ~~An arbitrary amount of entries follow, one entry per line, where every entry holds exactly one source code file location. Each entry consists only of the *pathname* of the source code file to be loaded; at least one blank shall precede the entry in the line. The value *pathname* is equivalent to the value of the switch `-sv_src`.~~

   ~~Additionally, a list of directory pathnames (separated by blanks) can be specified after the source code file, separated by colon (`:`). 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.~~

~~There is no need to locate or identify the compiled object code created from these sources; this is under the discretion of the application. The resulting syntax of an entry (point 2) is:~~

```
source code entry ::= blank pathname [[ blank ] : [ blank ] directories ]
directories ::= pathname [ blank pathname ]
blank ::= ’’ [ blank ]
```

~~C.4.4 Examples~~

1) ~~If the following source files are to be included in 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
```

~~then the following include directories also need to be referenced.~~

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

~~Example C-4 shows one way of specifying the source files and include directories for the source code inclusion (assuming `-sv_root` specifies `/home/user`).~~

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

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

~~*Example C-4Using a switch list*~~

~~2) The bootstrap file also permits a more granular assignment of include directories to source code files, as shown in Example C-5.~~

```
~~#!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 C-5Assigning include directories to source code via the bootstrap file*~~

~~This example shows the compilation of:~~

~~`/home/user/mycode/model1.c` (using a C compiler) with the include directories: `mycode/includes`, `proj1/util`, and `common/includes`~~

~~`/home/user/sysc/model3.sc` (using a C++ compiler) with the include directories: `common/sysc`~~

~~`/home/user/proj1/code/model3.cc` (using a C++ compiler) with the include directories: `common/includes`~~

~~`/home/user/proj3/c_code/model4.cpp` (using a C++ compiler) with the include directories: `proj1/util` and `common/includes`.~~

~~3) Finally, a highly customized compilation with user-specific options can be obtained by specifying the appropriate switch settings:~~

~~*-sv_root /home/user*~~
~~*-sv_inc incl_dir*~~

```
~~-sv_src model_list/model1.c~~
~~-sv_inc common_inc~~
~~-sv_cpp_compiler /usr/bin/g++~~
~~-sv_cpp_prefix_flags "-O3"~~
~~-sv_src model_list/model2.cpp~~
~~-sv_src model_list/model3.c~~
~~-sv_c_compiler /usr/ccs/acc~~
~~-sv_c_prefix_flags "-g -DDEBUG"~~
~~-sv_cpp_prefix_flags "-g -DDEBUG"~~
~~-sv_root /home/projects/common~~
~~-sv_inc shared_includes~~
~~-sv_src model4.c~~
~~-sv_src model5.cpp~~
```

~~then the following compilations are performed:~~

~~`/home/user/model_list/model1.c` (using the default C compiler) with the include directory: `/home/user/incl_dir`~~

~~`/home/user/model_list/model2.cpp` (using the C++ compiler `/usr/bin/g++` and the compiler flags `-O3`) with the include directory: `/home/usr/common_inc`~~

~~`/home/user/model_list/model3.c` (using the default C compiler) with the include directory: `/home/usr/common_inc`~~

~~`/home/projects/common/model4.c` (using the default C compiler with the compiler flags `-g -DDEBUG`) with the include directory: */home/projects/common/shared_includes*~~

~~`/home/projects/common/model5.cpp` (using the C++ compiler `/usr/bin/g++` and the compiler flags `-g -DDEBUG`) with the include directory: */home/projects/common/shared_includes*.~~