0.0.1 Inclusion of Object Code

The inclusion of compiled object code is required for cases where the compilation and linking of source code is fully handled by the user; thus only loading of the created object code is needed to integrate the foreign language code into a SystemVerilog application.

It must be supported by all SystemVerilog applications as a minimum requirement to support the integration of Foreign Language Code. Figure 1, “Inclusion of Object Code into a SystemVerilog Application”, depicts the inclusion of object code and its relations

.

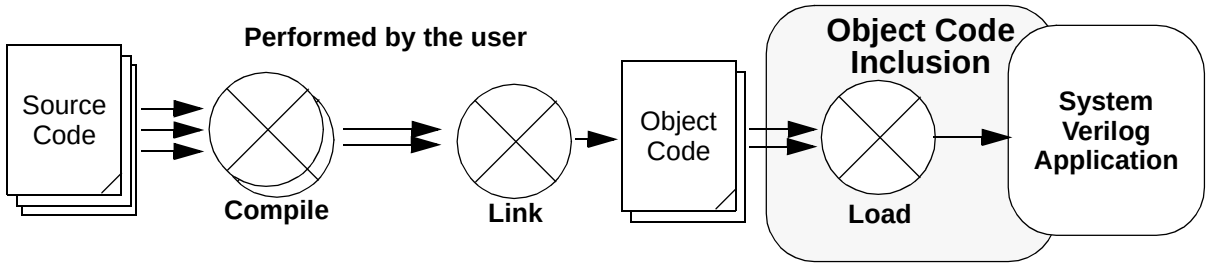

Figure 0-1 Inclusion of Object Code into a SystemVerilog Application

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.

Both 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 value of the switch -sv_root is used to identify an appropriate prefix for relative pathnames.[1]

Compiled object code must be provided in form of a shared library having the appropriate extension for the actual platform[2]. The provider of the compiled code is responsible for any external references specified within these objects. Appropriate data must be provided to resolve all open dependencies with the correct information.[3] The SystemVerilog application should 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 should be avoided, because they might interfere with other parts.

1.Refer to the corresponding rules in the introduction for more details on forming pathnames.

2.Shared libraries use e.g. .so for Solaris, .sl for HP-UX, other operating systems might use different extensions. In any case, it is the task of the SystemVerilog application to identify the appropriate extension.

3.Special care must be taken to avoid interferences with other software and to ensure the appropriate software version is taken (e.g. in cases where two versions of the same library are referenced). Similar problems might arise when there are dependencies in the compiled object code on the expected runtime environment (e.g. in cases where C++ global objects or static initializers are used).

In case of multiple occurances of the same file[1] the above order also identifies the precedence of loading; as a result a file located by method 1) will override files specified by method 2). 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. Any library must and will only be loaded once.

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.

Examples:

1) Assuming the pathname root has been set by the switch '-sv_root' 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

then this can be accomplished by one of the methods in the following figure. Both methods are 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

2) Assuming the current working directory is "/home/user" the following series of switches (left column) will 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
```

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

/home/project2/shared_code/svLibrary3.so

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

Example 2: '-sv_lib'/'-sv_root' switches and the resulting file names

1.This refers to files having the same pathname or can be easily 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.

3) Further, given the following set of switches and contents of bootstrap files:

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

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

bootstrap1:

```
#! SV_LIBRARIES
 lib1
 lib2
```

bootstrap2:

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

Example 3: Mixing -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.