*svlib* User Guide  
and Programmer's Reference

# About this document

## Summary

This document provides a specification and programmer's reference for the SystemVerilog utility library *svlib*.

## Revision information

|  |  |  |  |
| --- | --- | --- | --- |
| Rev | Date | Author | Description |
| 0.0 | 10 Feb 2014 | J Bromley | Initial release for discussion |
|  |  |  |  |

# A few notes about general principles of use

svlib has been designed to be as un-selfish and un-intrusive as possible for use in any SystemVerilog environment. To achieve these goals it was necessary to introduce some underlying behaviours that are common to the whole library. It is important for users to be aware of these behaviours to avoid unpleasant surprises.

## Overall structure of the library

### The package

svlib is organized as a single SystemVerilog package named svlib\_pkg. Having compiled this package using their simulator of choice, users should then import this package into their own code so that the facilities of svlib are readily available.

The package should always be imported at the start of any module or package that needs it, just after the module or package header. Do not be tempted to put your import statement at the outermost scope, outside any module or package - this is very bad practice and should always be avoided.

### Macros

In addition to the package, svlib has a few macros that are useful or necessary when using the package features. To make these macro definitions available, users should do

`include “svlib\_macros.svh”

at the outermost ($unit) scope of their code, outside any module or package. It is safe to include this file as often as you wish, because it is protected by sentinels so that it cannot be compiled twice. Consequently it is a good idea to provide this include at the top of any file that makes use of svlib facilities.

## Classes or package-level functions?

Almost all svlib functionality is provided by classes defined in the package. Users can create instances of these classes (see section below) as required. However, in some situations it is more convenient simply to call a function to do some work for you, rather than going to the trouble of creating an object, populating it with your source data, calling methods on it, and finally extracting your processed data from the object. Many svlib features are available in both forms, so that you can choose whichever is more convenient for you. For more details, see the documentation for each individual feature.

## Constructing svlib objects

Many parts of svlib use SystemVerilog classes. User code must of course create new objects of these class types in order to make use of svlib features. However, in order to avoid unexpected disturbance to random stability and to improve memory management efficiency,

**it is very important that user code should *never* directly call the constructor, new, of any svlib class.** All svlib objects should be created using their built-in static create method, which is documented individually for each class.

This issue is discussed in more detail in the accompanying conference paper [1]. Fortunately, all major SystemVerilog simulators now offer full support for protected constructors. Consequently, all svlib class constructors are declared protected so that it is impossible to call them from user code

## Handling errors

Occasionally, svlib functions may give rise to internal error conditions. This is especially likely if the function calls out to the C library, and there may be issues with memory allocation, file permissions or even the existence of files, and so on. Such errors are invariably passed back for handling within SystemVerilog, but the exact details of how the error is handled are somewhat under the programmer's control. The default behavior is for svlib to throw an assertion-style error, but more subtle control is available. See section ??? for details.

## Internal hidden features of svlib

Some features of svlib are designed to remain hidden from the user. This is done so that the package can maintain consistency of data on the C and SystemVerilog sides of the DPI. However, SystemVerilog does not provide any means to enforce this hiding in the language.

To help users to avoid accidentally breaking this encapsulation, the hidden aspects of svlib are placed in a separate package svlib\_private\_base\_pkg. User code should *never* import this package directly, and should *never* attempt to use any of the data, functions, classes or DPI imports in it.

## Naming conventions

As far as possible, a consistent naming scheme has been used throughout svlib. The naming scheme has been designed with an emphasis on consistency, so that it is easier to remember or guess what a given feature is called. We have also tried to keep names as short as possible for convenience, but sometimes this has not been possible - perhaps because of conflicts with SystemVerilog keywords or other commonly-used packages such as the UVM, or perhaps in order to keep some set of names unique across the whole package.

### Classes

Almost all svlib classes have short names that begin with an uppercase letter and are otherwise all lowercase. For example, the class that represents a regular expression is Regex. There are a few exceptions. In particular, the configuration features have several classes that are named with a prefix cfg, such as cfgNode.

### Class methods

Methods of svlib classes are given names that are as short as possible while striving to remain memorable. Where a name is naturally made up of multiple words, the name is spelt in camelCase (no underscores, all but the first word capitalized). A typical example is the addNode function of class cfgNode.

### Prefixes for package-level functions

Many svlib functions fall naturally into groups. For example, there are several package-level functions concerned with operating system interaction. Those functions all have names beginning with the prefix sys, separated from the main part of the name by an underscore as in sys\_dayTime.

# String manipulation

The SystemVerilog language provides a number of string operations natively. However, experience has shown that the built-in set is not sufficient for many practical string processing tasks, and svlib provides a further set of operations to help meet these requirements.

String operations are, in most cases, available in two different forms, and a programmer is free to choose whichever form is more appropriate to their needs.

The first form is straightforward functions on string values, often (but not always) returning a string result. These functions are defined in the svlib package and consistently have names that begin with the prefix str\_.

The second form is methods of an object of class Str (note the uppercase S). The Str class is a wrapper for a SystemVerilog string, allowing a string to be passed around by reference and making some sequences of operations more convenient.

The obvious drawback to using Str objects rather than simple functions is that an object must be constructed before any operation is performed. This drawback is often outweighed by the efficiency and convenience of being able to offer a Str object, by reference, to many successive operations. As already noted, programmers are free to choose the representation that is most convenient for them. In practice, if you need to do just one operation on a string, the package level functions are likely to be most convenient. If you plan to do many successive operations on the same string, it's usually best to create a Str object and work on that.

# Regular expression processing

svlib supports regular expression matching and substitution within strings.

This document does not describe how to write regular expressions. svlib uses the standard C library's POSIX-compliant regular expression subsystem, and you can find full details of how to write regular expressions in this dialect by consulting the man-page man 7 regex or any of the numerous online regular expression tutorials.

After a regular expression match has succeeded, there are many different things that a user might wish to do with the results. To support this variety of needs, regular expressions in svlib are invariably represented as an object of class Regex. You can call query functions in a Regex object to find out about the matches that were discovered by the last match attempt, and perform substitution operations.

Often, you need to apply the same expression multiple times to a given string – typically because you want to locate not just the first, but every occurrence of a match within the string. To make this more efficient and convenient, regular expression matching works not on a native SystemVerilog string, but on a Str object (see section ).

The basic steps in performing a regular expression match are:

Construct a Str object containing the string that you wish to examine. (In practice it's likely that this object already exists because you have already been working on the string in question.)

Construct a Regex object and set it up to contain your chosen regular expression, together with options such as case sensitivity and end-of-line handling.

Call the test() method of the Regex object to perform the match, returning information about whether the match succeeded (found a match) or failed.

Call other methods of the Regex object to retrieve more detailed results such as matches corresponding to parenthesized groups, or to perform substitution operations.

Convenience functions exist to simplify some of these steps in situations where only standard matching operations are required.

First we describe the more flexible approach in which objects are created explicitly. Later in this section we cover the package-level convenience functions.

## Constructing and configuring a Regex object

static function Regex Regex::create(string re="", int opts=0);

function void setRE(string re);

function void setOpts(int opts);

To perform regular expression matching it is first necessary to construct a Regex object and set it up appropriately. This is done in the usual way by means of the Regex class's static create method (see section ):

Regex myRE = Regex::create();

The newly created object should then be set up with your desired regular expression and options so that searches can later be performed. Setup is accomplished using the setRE and setOpts methods. Alternatively, it is possible to pass in the regular expression string and options values as arguments of the create method.

### Setting the regular expression

The regular expression that you wish to use is of course a string itself. By calling the setRE method of an existing Regex object you can set up the object's regular expression string.

Suppose, for example, that you wish to search for a string of any three uppercase letters. Having created the Regex object as shown above, you would then supply the regular expression string thus:

myRE.setRE("[A-Z]{3}");

Some care is required when specifying the regular expression string. Many regular expressions require backslash-escapes to indicate special characters, or to remove special regex meaning from characters such as $ or square brackets. Because the expression is typically specified as a string literal, you must be aware that SystemVerilog quoted strings also use backslash as an escape character. This usually means that you need to specify each backslash twice. For example, a regex matching one or more dollar-signs is \$+ but to write that as a SystemVerilog string literal you must specify "\\$+" as in the following example:

myRE.setRE("\\$+");

Within a quoted string literal, SystemVerilog uses the double backslash to denote a single backslash character. There is one especially unpleasant case of this backslash escape problem. Suppose you wish to write a regex that matches a single backslash character. The regular expression you need is \\ (two backslashes) - but to express that as a SystemVerilog string literal requires *four* consecutive backslashes!

### Configuring matching options

Before you use a regular expression, you can configure some optional features of its operation. The current version of svlib supports POSIX-standard extended regular expressions, which have only two such optional features: case insensitivity, and end-of-line matching. By default, matching is case-sensitive and end-of-line within a string is treated like any other character. By calling a Regex object's setOpts method you can configure these two options, passing an integer value that is the logical OR of a series of bit flags. The available flags are:

* NOCASE for case-insensitive matching; zero for case-sensitive matching
* NOLINE to enable special treatment of end-of-line; zero for no special treatment

If NOCASE is enabled, then matching makes no distinction between upper and lower-case letters, either in the regular expression itself or in the string being tested.

If NOLINE is enabled, end-of-line characters are treated specially in the following ways:

* The match-any-character wildcard '.' (period) will not match an end-of-line character.
* The anchors ^ (start) and $ (end) will match not only the beginning and end of the whole string being tested, but also the beginning and end of any physical line in the string. ^ will match the anchor point just after any end-of-line, and $ will match the anchor point just before any end-of-line.

The options flags may be changed at any time, and will affect any match operations performed subsequently.

### Setting and configuring the regular expression at creation

As you can see from the create function prototype, creation and setup can be performed together by supplying the regular expression string and the options value as optional arguments to create.

## Providing a string for testing against the RE

function void Regex::setStr(Str s);

function void Regex::setStrContents(string s);

The string to be tested by the regular expression must be supplied as a Str object (see section ). Give your Regex object a reference to this Str object by calling the Regex object's setStr or setStrContents method. As described later, there are alternative ways to achieve this that may be more convenient in some situations.

You can call these methods at any time. They do not affect the regular expression string or the options that have been configured for the Regex object, nor does it disturb the stored matches from any earlier match attempts. However, the next match attempt will of course apply to the new string.

If you do not already have a Str object containing your test string, it is convenient to construct the Str object implicitly by passing a regular SystemVerilog string to the setStrContents method:

myRE.setStrContents(“the string you wish to test”);

Of course, any string expression is appropriate as the argument - it does not have to be a literal.

## Testing the string against the RE

function int test(Str s, int startPos=0);

function int retest(int startPos);

To discover whether a regular expression matches a string, call the test or retest methods. The difference is that test allows you to pass in a new Str object containing the string you wish to test, avoiding the need to invoke setStr or setStrContents. By contrast, retest performs a further match test on the existing Str object held by the Regex. In both cases, the startPos argument specifies the starting point of the match.

The result value is 1 if the match succeeded, zero if there was no match or if there was some error. Error handling is described in section ??? below.

Matching always ignores the first startPos characters of the string. This allows multiple matches to be found by repeatedly calling retest with progressively increasing startPos values until it returns zero indicating that there are no further matches.

## Retrieving matches and sub-matches

After using a Regex object to perform a successful match, you can call methods of the object to get information about the various matches and sub-matches that were found by the match attempt.

function int getMatchCount();

function string getMatchString(int m);

function int getMatchStart (int m);

function int getMatchLength(int m);

These methods extract and return the match specified by the value m. A value of zero indicates the whole regular expression match. Values between 1 and 9 correspond to strings that matched sub-expression groups in the regular expression, numbered in left-to-right order of their opening left parenthesis in the usual way.

* getMatchString returns the matching string itself (a slice of the original string).
* getMatchStart returns the left-most character position of the match.
* getMatchLength returns the number of characters in the match.

If you call any of these functions on a Regex object whose most recent match was unsuccessful, or if you supply a value of m that is larger than the number of sub-matches in the original regular expression, then there will be no error, but:

* getMatchString returns an empty string.
* getMatchStart returns -1.
* getMatchLength returns zero.

getMatchString(m) is always exactly equivalent to calling the range method on the Str object containing the string that was searched:

range(getMatchStart(m), getMatchLength(m))

## Substitution (search-and-replace)

function int Regex::subst(Str s, string substStr, int startPos = 0);

function int Regex::substAll(Str s, string substStr, int startPos = 0);

The Regex class supports substitution, in which the part of a string that matched your regular expression is replaced with some other string. As usual in regular expression search-and-replace, the replacement string can contain matches and submatches taken from the regular expression.

Methods subst and substAll are called on an existing Regex object, whose regular expression and options must already have been set up. The existing string to be tested (if any) that was held by the Regex object is thrown away, and the new Str object s is used to provide the source string.

subst finds and replaces the first match within s, starting from startPos. substAll finds and replaces every match, again starting from character position startPos.

The replacement string substStr can be a simple string value. However, it can also contain placeholders that will be replaced with match values taken from the corresponding regular expression match. These placeholders, often indicated by markers such as \1 in common regex dialects, are indicated in svlib using a dollar sign followed by a single digit. $0 refers to the whole match that was found by the regular expression (you can also write $\_ or $& if you prefer; they mean exactly the same thing). Sub-matches (up to a maximum of nine) are indicated by $1..$9. A dollar sign followed by any other character (including $ itself) is replaced with the character after the $ - so, for example, $R would be replaced by a single letter R, and $$ by a single dollar sign.