svlib User Guide
and Programmer's Reference

1 About this document

1.1 Summary

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

1.2 Revision information

Rev	Date	Author	Description
0.0	10 Feb 2014	J Bromley	Initial release for discussion
0.1	23 Feb 2014	J Bromley	Working towards a first release
0.2	02 Mar 2014	J Bromley	All sections present, ready for initial release. Some explanatory text is still missing.
0.3	05 Jun 2014	J Bromley	Fix issues #18, #19 and various other minor errors. Complete most of the text.
0.4	04 Jan 2015	J Bromley	Fix issues #20, #21, #22.
0.5	14 July 2015	J Bromley	Fixes #24, #27, #28, #29, #31

1.3 Copyright information

1.3.1 Limitation of liability

1.3.2 Authorship

This document was prepared at Verilab, Inc (www.verilab.com) by Jonathan Bromley, Paul Marriott and André Winkelmann.

1.3.3 Open source requirements

`svlib` is an open source package, so you are free to examine the source code and modify it in any way you wish. This document is provided in PDF for convenience, and therefore is not strictly open-source. The original editable document is available from the authors on request.

1.4 Contacting the authors

The authors of this package and document can be contacted using the email address svlib@verilab.com. They are always interested to hear of possible corrections or improvements, and aim to respond promptly to any communication.

2 Introduction and Overview

This document describes `svlib`, a utility library for SystemVerilog. It provides functionality that we have found lacking in SystemVerilog for our day-to-day verification work: string processing, operating system interface and many other useful operations.

- Section 3 describes how to integrate `svlib` with your simulator and your own SystemVerilog code.
- Section 4 discusses some underlying principles and conventions of `svlib`, to help you use it with confidence.
- Section 5 documents features of `svlib` that support string manipulations and operations that are not available in the base SystemVerilog language.
- Section 6 introduces the regular expression matching and substitution features.
- Section 7 gives details of the `Pathname` class, which makes it easy to perform common manipulations on a typical filename such as determining the directory, assembling a pathname from its directory components, finding a file's extension, etc. These operations are just specialized string functions; they do not make any access to the file system.
- Section 8 looks at `svlib` facilities for querying the file system. It allows you to find properties of files such as "does this file exist", "on what date was the file last modified", "can I write to this file", "is it a directory" and many others.
- Section 9 offers a collection of operating system query functions. You can easily discover the current (wall-clock) time and date, render a date in various human-readable formats, explore the OS's environment variables, get the contents of a directory as a queue of strings, and read a high-resolution (sub-microsecond) timer.
- Section 10 discusses how errors are handled in `svlib`. By default errors are reported on the simulation console, but you can customize the error handling in various ways and even take full responsibility for handling errors under control of your own SystemVerilog code.
- Section 12 details classes and functions that support reading and writing configuration files in `.ini` or `YAML` format, and explains how to transfer configuration data between your own custom configuration classes and `svlib`'s internal Document Object Model (DOM) representation.
- Section 13 describes a small group of functions allowing running simulations to query the simulation environment. In particular, full details of the simulator command line arguments and options can easily be retrieved.
- Section 14 describes a set of utility functions that make it easier to work with SystemVerilog enumeration types.
- Section 15 describes some convenience features, provided in the form of SystemVerilog macros.
- Finally, section 16 provides a number of ready-to-run examples that show `svlib` facilities in practical use.

3 Compiling and running code that uses *svlib*

The `svlib` codebase is in three distinct parts, all found in the `src/` directory of the distribution:

- macros that are available for user code if you `` `include "svlib_macros.svh" `` appropriately
- SystemVerilog code, forming a single package `svlib_pkg`, that you can compile by asking your tool's analyzer (compiler) to process file `svlib_pkg.sv`
- C code that is called, using the SystemVerilog DPI, by various parts of the library – currently appearing as a single file `dpi/svlib_dpi.c`

To make use of `svlib` it is essential that you compile both `svlib_pkg.sv` and `dpi/svlib_dpi.c`. Users who already have a DPI flow can integrate this compilation (and, optionally, linking to a shared object file) by extending their existing flow in accordance with their tool vendor's guidelines. For new users and for experimenting with `svlib`, however, it is usually simplest to take advantage of the tool vendor's integrated one-step compile/link/run flow. To ease this flow there is a response file `svlib.f` in the `src/` directory.

The placeholder `<svlibRoot>` is used to indicate the directory in which the `svlib` distribution has been placed.

In the following notes for three popular SystemVerilog simulators, all trademarks are acknowledged as the property of their respective owners.

3.1 Command line for Mentor Graphics QuestaSim

Use the `qverilog` one-step flow, as follows:

```
qverilog +incdir+<svlibRoot>/src -f <svlibRoot>/src/svlib.f \
                    <user_options> <user_files>
```

3.2 Command line for Cadence Incisive

Use the `irun` one-step flow, as follows:

```
irun +incdir+<svlibRoot>/src -f <svlibRoot>/src/svlib.f \
                    <user_options> <user_files>
```

3.3 Command line for Synopsys VCS

Use the `vcs` one-step flow, as follows. Note the additional `-LDFLAGS` option that is required to link with an additional C library component that VCS does not link by default. The `-R` option is not mandatory. It simply causes the `simv` executable to start running automatically when compilation and linking is done.

```
vcs -sverilog -R +incdir+<svlibRoot>/src -f <svlibRoot>/src/svlib.f \
                   -LDFLAGS -lrt \
                   <user_options> <user_files>
```

To simplify this, we have created a file `vcs.f` that contains the required `-LDFLAGS` and `-sverilog` options along with the other contents of `svlib.f`, allowing you to use this alternative command line:

```
vcs-R +incdir+<svlibRoot>/src -f <svlibRoot>/src/vcs.f \
                   <user_options> <user_files>
```

4 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 behaviors that are common to the whole library. It is important for users to be aware of these behaviors to avoid unpleasant surprises.

4.1 Overall structure of the library

4.1.1 The package

svlib is organized as a single SystemVerilog package named svlib_pkg. Having compiled it 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 into the scope 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.

4.1.2 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.

4.2 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 4.3 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.

4.3 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.

4.4 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, where 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 10 for details.

4.5 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.

4.6 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.

4.6.1 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`.

4.6.2 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 spelled in `camelCase` (no underscores, all but the first word capitalized). A typical example is the `addNode` function of class `cfgNode`.

4.6.3 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`.

5 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.

5.1 The Str class

5.1.1 Methods that manage a Str object and its contents

```
static function Str Str::create(string s = "");
function void   set   (string s);
function string get   ();
function Str    copy  ();
function int    len   ();
```

As mentioned earlier, users must never invoke the `new` constructor of any `svlib` class. To construct a `Str` object you must call the `Str::create()` method. Optionally, you can pass to this method the initial value of the string.

At any time after creation you can update the contents of a `Str` object by using its `set` method, providing the new string value as argument. `get` returns the object's current contents as a native SystemVerilog string. `len` yields the length of the string.

`copy` creates and returns a completely new `Str` object having the same contents as the invoking object.

5.1.2 Enumeration types

```
typedef enum {NONE, LEFT, RIGHT, BOTH} side_enum;
typedef enum {START, END} origin_enum;
```

These two enumerations are used to specify various optional behaviors of some methods. `side_enum` is used to specify which "side" of a string is to take part in various operations, notably `trim` and `pad`. `origin_enum` is used to specify from which end of a string to count when identifying character positions for the `range` and `replace` operations. `START` specifies the leftmost end of a string, `END` specifies the rightmost end. Further details of these options are described in later sub-sections.

5.1.3 Appending a string to the end of an existing Str object

```
function void append(string s);
```

This function modifies the existing string contents of a `Str` object by appending the specified string to it, using simple string concatenation.

5.1.4 Finding occurrences of a substring

```
function int first (string substr, int ignore=0);
function int last (string substr, int ignore=0);
```

`first()` searches for the first (leftmost) occurrence of the string `substr` within the object`s string contents. It returns the character position, within the original string, of the leftmost character of the sought substring. If the search is unsuccessful (there is no occurrence of `substr` within the original string) then the function returns `-1`. The search is an exact literal match; no wildcard or regular expression matching is performed.

Argument `ignore` specifies where the search is to begin. The default value (`ignore=0`) causes the whole string to be scanned, and the first match returned. If `ignore` is greater than zero, the search will begin at the specified character position. Regardless of the value of `ignore`, the return value after a successful match is the absolute start position of the match within the original string.

`last` behaves in a similar way, but it begins its scan from the rightmost end of the string, and therefore returns the last possible match if the sought substring occurs more than once in the original string. For `last`, the `ignore` argument specifies the number of characters at the rightmost end of the string that are to be ignored – it acts as though the last `ignore` characters were simply not present.

> **NOTE**: The `first` and `last` methods of class `Str` provide a convenient and simple substring search function. Much greater flexibility is available by using regular expression matching as described in section 6, but this flexibility comes at the price of greater setup effort on the part of the programmer, and a small runtime performance cost. In most practical situations these drawbacks are insignificant, and regular expression processing is the preferred choice.

5.1.5 Split and join operations

```
function string sjoin (qs elements);
function qs split (string splitset="", bit keepSplitters=0);
```

> **NOTE**: the type name `qs` is internally defined within `svlib` to mean "queue of strings", but it is **not** available to user code. If you need a type name to represent queue-of-strings you should define your own; it will be fully compatible (*type-equivalent*) to `qs`. Alternatively you can simply declare variables that are queues of strings, and use them as argument and result variables.

The `sjoin` method (unfortunately it cannot be called `join` because that is a SystemVerilog keyword) uses the contents of the `Str` object as a "joiner" to assemble the elements of a queue of strings into a single string. It can be convenient for creating comma-separated lists, for example.

The `split` method takes the existing contents of the `Str` object (which remains unaltered) and breaks it into a queue of strings, using *single character* split-markers ("splitters"). The argument `splitSet` is a string, but it is treated as a set of individual characters; the string contents of the object will be split at every occurrence of any character in the set.

If `splitset` is the empty string (the default) then the result is a queue of single-character strings, with each element being one character of the original string.

If keepSplitters is true (1) and splitset is not the empty string, then the split characters will appear in their appropriate positions as individual members of the result queue. If keepSplitters is false (the default) the split characters wlll not appear in the result.

> **NOTE**: With effect from version 0.5 of svlib, there is a new split method in the Regex class (see section 6). This offers much more flexible functionality than the Str::split method described here, and is to be preferred in most cases.

5.1.6 Extracting substrings and the replace operation

```
function string range (int p, int n, origin_enum origin=START);
function void replace(string rs, int p, int n, origin_enum origin=START);
```

Method range provides a more versatile and consistent version of the SystemVerilog native string's substr operation. It does not suffer from substr's confusing and irregular behavior when one of the boundaries falls outside the string. In section 5.3 you can find details of how to use the p, n and origin arguments to specify a slice of a string in a consistent way. range simply returns the substring thus specified, as a regular SystemVerilog string.

Method replace identifies a substring in exactly the same way, and then replaces that substring with the replacement string rs, modifying the Str object's contents. The replace method is sufficiently flexible that it can do duty for various operations that are sometimes provided as separate functions. In particular:

- part of a string can be deleted (removed) by specifying an empty string for rs
- the append operation (which *is* provided separately, because it is so common) can be rewritten as
  `s.replace(append_string, 0, 0, Str::END);`
- similarly, a prefix can be added to an existing string thus:
  `s.replace(prefix_string, 0, 0, Str::START);`

There is *no* requirement for the replacement string rs to have the same length as the slice of original string that it is replacing.

5.1.7 Adding and removing spaces at the start and end of a string

```
function void trim (side_enum side=BOTH);
function void pad (int width, side_enum side=BOTH);
```

Method trim removes leading and/or trailing white space from a string, modifying the existing contents of the Str object. Argument side specifies which end of the string is to have whitespace trimmed. If side is Str::LEFT, white space is removed from the left (leading) end of the string; RIGHT removes trailing space; BOTH removes whitespace from both ends. Finally, if NONE is specified, the function has no effect.

Whitespace is any of space, tab, newline, carriage-return, and nonbreaking space (ASCII code 160).

If the string consists entirely of whitespace and side is not NONE, the result will be an empty string.

Method pad adds enough leading and/or trailing spaces (always using the space character) to make the resulting string exactly width characters long. If the string is already larger than width, it is not changed (so the result can sometimes be *larger* than width). If side is NONE, the string is unchanged. Otherwise, spaces are added to the specified end of the string as required. If side is BOTH, equal numbers of spaces are added to both sides (with one extra space on the right side if necessary). This method is useful for aligning text for printing in tabular format.

5.1.8 Stripping unwanted characters from a string

```
function void strip (string chars = " \t\n\13\14\15\240\177");
```

Method `strip` removes from the string object all occurrences of any character appearing in `chars`, modifying the existing contents of the `Str` object. The default is to remove all whitespace characters, but you can specify a string containing any characters you wish to remove.

5.1.9 Rendering a string as a SystemVerilog string literal

```
function void quote ();
```

This method updates the object so that it contains the string that would be required to represent the object's original string contents as a SystemVerilog string literal.

Quoting is performed by replacing special characters (backslash, double-quote, control characters etc) with their backslash-escaped equivalents, using the usual escape sequences such as `\"` and `\n`. The more general `\xNN` notation is used where necessary. Finally, the entire string is surrounded by a pair of string quotes (`"`). The result is always a complete, legal SystemVerilog string literal.

This function is intended to be useful for writing SystemVerilog programs that will write out SystemVerilog source code. It may also be useful when writing files in formats such as comma-separated value (CSV).

5.2 Package-level string functions

```
function string str_sjoin(qs elements, string joiner);
function string str_trim(string s, Str::side_enum side=Str::BOTH);
function string str_pad(
          string s, int width, Str::side_enum side=Str::BOTH);
function string str_quote(string s);
function string str_replace(
          string s, string rs,
          int p, int n, Str::origin_enum origin=Str::START);
function string strip (
          string s, string chars = " \t\n\13\14\15\240\177");
```

Sometimes it is inconvenient to create and populate a `Str` object simply in order to perform one or two operations on it. To ease this, svlib provides a few string operations as package-level functions as an alternative to using class methods. In each case the function performs exactly the same actions as the corresponding method of class `Str` (without the `str_` prefix). Internally, each of these functions populates a `Str` object with the argument string `s`, performs the operation, and finally returns the appropriate result. The performance overhead is very small, because the library maintains a pool of `Str` objects in readiness for such operations.

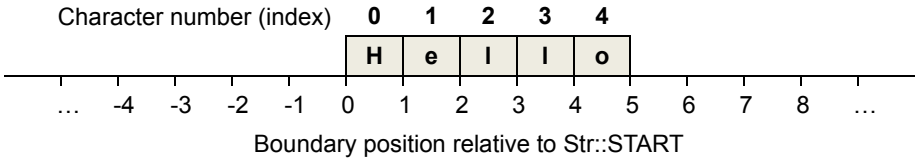
5.3 Specifying a string range

`svlib` uses a single consistent way of specifying substring ranges (slices of a string). It is used explicitly in methods `range` and `replace` of class `Str` (and the corresponding package-level functions `str_range` and `str_replace`), and is also used implicitly elsewhere. It has been designed to avoid some of the difficulties that are presented by the `substr` operation of SystemVerilog's native string type.

5.3.1 Boundary is a position between characters

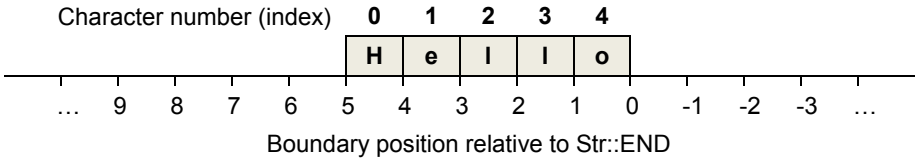
We do *not* specify string ranges in terms of character numbers, because this leads to awkward discontinuities when handling zero-length string slices. The boundary of a string

slice is specified in terms of a position *between characters*. To illustrate this, consider the five-character string `"Hello"`:



Using this approach, we have a consistent way to specify the boundary position of a substring using argument `p`, with the `origin` argument specified as `Str::START` (the default). We also have a straightforward interpretation of boundary positions that are negative, or greater than the string's length.

Alternatively we can specify the boundary relative to the string's `Str::END` (the rightmost position). In this case, the interpretation of different `p` argument values is modified, with `p` counting leftwards from the right-hand (end) character boundary:



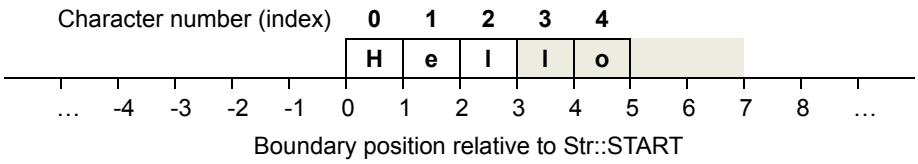
Once again, we have a straightforward interpretation of out-of-range values for `p`. With `origin` specified as `Str::END`, however, we can specify trailing (suffix) parts of the string without needing to worry about exactly how long the string is.

5.3.2 Interpretation of the length argument (n)

Having established a starting (boundary) point for our string range, we now need to consider the length of slice that we wish to take. The interpretation of this argument `n` is not in any way affected by the `origin` value. It specifies how far to move from the `p`-specified boundary in order to find the second boundary of our substring. Positive values of `n` describe movement to the right. Negative values describe movement to the left.

5.3.3 Interpreting the complete range specification

Taken together, the set of three values `origin`, `n` and `p` specifies a range of character positions along the character-boundary number line. For example, if we were to invoke function `str_range(.s("Hello"), .p(3), .n(4), .origin(Str::START))` it would specify the range shown shaded on the diagram below:



`p=3` and `origin=START` takes us to boundary position 3; `n=4` specifies four character positions from the right of the position specified by `p`. However, two of those character positions fall outside the original five-character string, so the result of the `range` operation is the two-character string `"lo"`.

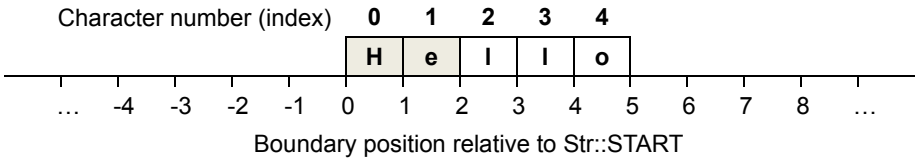
5.3.4 Further examples

Here are a few more examples showing various combinations of `START/END`, positive and negative lengths (`n`), and ranges that may fall outside the string's boundaries.

Example 1:

```
str_range(.s("Hello"), .p(0), .n(2), .origin(Str::START))
```

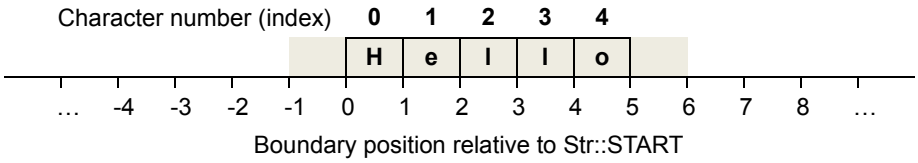
Result: `"He"` (2 characters)



Example 2:

```
str_range(.s("Hello"), .p(-1), .n(7), .origin(Str::START))
```

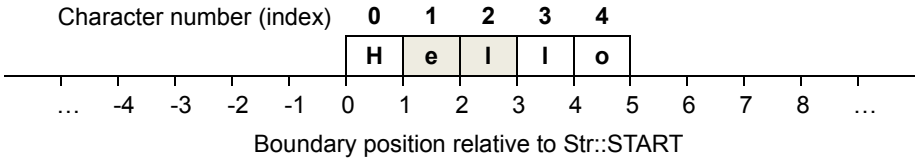
Result: `"Hello"` (5 characters)



Example 3:

```
str_range(.s("Hello"), .p(3), .n(-2), .origin(Str::START))
```

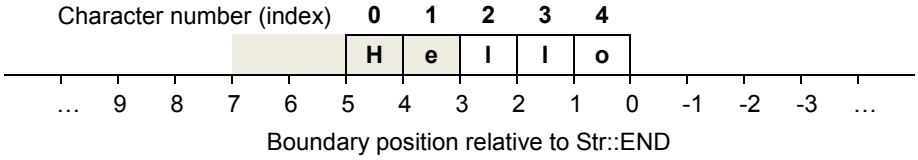
Result: `"el"` (2 characters)



Example 4:

```
str_range(.s("Hello"), .p(7), .n(4), .origin(Str::END))
```

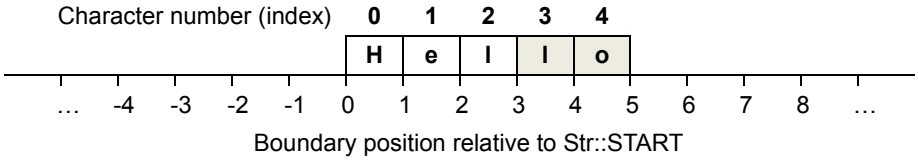
Result: `"He"` (2 characters)



Example 5:

```
str_range(.s("Hello"), .p(5), .n(-2), .origin(Str::START))
```

Result: `"lo"` (2 characters)



6 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 "extended regular expression" dialect of the 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. The regex dialect of svlib is in almost all respects the same as that used by the Unix/Linux command egrep.

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 the 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 5.1).

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.

6.1 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 5):

```
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.

6.1.1 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!

```
myRE.setRE("\\\\");
```

6.1.2 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.
- 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; they do not affect the stored match values for a match that has already been performed.

6.1.3 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`.

6.2 Providing a string for testing against the RE

```
function void setStr(Str s);
function void setStrContents(string s);
```

The string to be tested by the regular expression must be supplied as a `Str` object (see section 7). Give your `Regex` object a reference to this `Str` object by calling the `Regex` object's

setStr method. Alternatively, you can bypass the need to construct and populate a Str object by using the Regex object's setStrContents method, which directly populates the Regex's internal Str object with a new string value – constructing a new Str object automatically if needed. As described later, package-level function regex_match provides an alternative way 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 do they disturb stored matches from any earlier match attempts.

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.

6.3 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 6.6 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.

6.4 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 = 0);
function int    getMatchStart (int m = 0);
function int    getMatchLength(int m = 0);
```

With the exception of getMatchCount, these methods extract and return the match specified by argument m. A value of zero (the default) 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.

- getMatchCount returns the number of matches provided by the regular expression. If you call this method on a Regex object that has never been used to do a match, it will return -1. If the most recent match failed, this function returns zero. Otherwise it returns the total number of available matches and submatches (i.e. one larger than the number of capturing groups in the regex). Note that this information is based on the syntax of the regex, and is *not* influenced by what it matched. It always reports the number of *possible* matches, and of course some of those matches may be empty or nonexistent depending on exactly what was matched.

- `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))
```

6.5 Substitution (search-and-replace)

```
function int subst(string substStr, int startPos = 0);
function int substAll(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 sub-matches taken from the regular expression.

Methods `subst` and `substAll` are called on an existing `Regex` object, whose source string, regular expression and options must already have been set up.

`subst` finds and replaces the first match within the `Regex` object's source string, starting from `startPos`. `substAll` finds and replaces every match, again starting from character position `startPos`. The source string is updated to reflect the substitutions, and (as usual) can be retrieved as a `Str` object using the `getStr` method, or as a SystemVerilog string using `getStrContents`.

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.

6.6 Regex-based split operation

```
function qs split(int limit = 0);
```

This operation is intended to supersede the much less flexible `Str::split` method. Its behavior closely follows Perl's `split` function, with only two significant exceptions:

- A regex pattern of `"^"` does ***not*** automatically cause the `NOLINE` matching option to be enabled.
- A regex pattern consisting of a single space character is ***not*** automatically converted to the pattern `"\\s+"`.

The `Regex` object's contents – both its regular expression, which we call *pattern*, and its test string which we call *source* – are used and are unaffected by the operation. Within *source* every match of *pattern* is taken to be a separator that splits *source* into substrings (*fields*) that do not include the separator. A separator can be of any length, including zero (the empty string). If *pattern* matches the empty string, *source* is split into fields each of one character.

If the *source* string begins with a separator match, then there will be a leading empty field, unless the initial separator match is the empty string in which case the leading empty field is removed.

The following example shows a simple use of `split` in which the comma character is used to separate a string into fields:

```
string result[$];
Regex re = Regex::create(",");
re.setStrContents(",first,,second,third");
result = re.split();
```

The result is this 5-element queue. Note the empty fields, which arise because *every* comma is treated as a separator:

```
{"", "first", "", "second", "third"}
```

If *pattern* contains any capturing subexpressions (parenthesised groups) then the captured submatches, in the usual left-to-right order, are added to the result queue immediately after the field that appears to the left of the match. This can be useful to determine which of several possible separators was used between any pair of fields. The following example uses any one of the operators `=`, `+` or `-` as separators, with any amount of whitespace around them, and captures each operator character for further processing:

```
string result[$];
Regex re = Regex::create("\\s*([=+-])\\s*");
re.setStrContents("sum  =  first +  second-third");
result = re.split();
```

The result is this 7-element queue (note that the spaces were *not* captured):

```
{"sum", "=", "first", "+", "second", "-", "third"}
```

The optional `limit` argument determines how much of the string is explored by the split process. If `limit` is greater than zero, it determines the maximum number of fields that will be returned. Only split fields count to this total; captured submatches do not count. If the previous example had done

```
result = re.split(2);
```

then the result queue would have contained the first two fields, along with their trailing separator submatches:

```
{"sum", "=", "first", "+"}
```

If `limit` is negative, the maximum possible number of fields is extracted. If `limit` is zero (the default), the behavior is exactly as for negative `limit` except that empty trailing fields are stripped from the result. Empty trailing fields can occur if the *source* string ends with one or more separator matches.

6.7 Errors in regular expression matching

```
function int    getError();
function string getErrorString();
function void   userErrorHandling(bit suppressAssertions);
```

If the regular expression is valid (syntactically legal) but did not match anything in the input string, then there is no error condition and the `retest` method returns zero to indicate that there was no match. Subsequent calls to `getMatchCount` will return zero.

However, it is sometimes possible for regex matching to give rise to error conditions. The most likely reason is that the regular expression string itself is ill-formed and cannot be used, perhaps because it contains unmatched parentheses or other illegal features. Much more rarely, there may be internal errors in the regular expression processor such as memory overflow. In all these cases, `svlib` has no choice but to yield an error. The matching process

will return zero, just as for an unsuccessful match, but it will also cause an assertion-style error unless you have configured the `Regex` object to suppress errors using the `userErrorHandling` method.

Regardless of the user error handling configuration, the `getError` and `getErrorString` methods can be called to get information about the error (if any) from the most recent match operation. If called before any match operation has been performed, these methods cause the regex to be tested for legality, and return a corresponding result. If the regular expression is valid and matching proceeded without memory overflow or other internal errors, `getError` will return zero. A nonzero result indicates that there was an error, and `getErrorString` can then be called to obtain a more verbose human-readable description of the error.

6.8 Package-level regular expression functions

As with the `Str` class, some methods of `Regex` are also provided as package-level functions that you can call directly, with no need to construct a `Regex` object explicitly.

```
function automatic Regex regex_match(
       string haystack, string needle, int options=0);
function automatic qs regex_split(
       string source, string pattern, int limit=0);
```

`regex_match` provides a simple way to perform a regular expression match. It seeks a match of regular expression `needle`, scanning the source string `haystack`. If there is no match, it returns the result `null`. If there is a match, however, it returns the `Regex` object that was automatically constructed and used for matching, allowing you to perform further matches or to retrieve captured match values. The `options` argument is exactly the same as for the `setOpts` method (see section 0).

`regex_split` packages the `Regex::split` method so that it can easily be used without any need to construct a `Regex` object explicitly. Its `source`, `pattern` and `limit` arguments are handled exactly as described in section 6.6.

7 File pathname manipulation

Create a Pathname object, optionally setting it to contain a given file path:

```
static function Pathname Pathname::create(string s = "");
```

Update a Pathname object with a new path value:

```
function void     set           (string path);
function void     append        (string tail);
function void     appendPN      (Pathname tailPN);
```

Extract various components of a pathname, always returning a simple string:

```
function string   get           ();
function string   dirname       (int backsteps=1);
function string   extension     ();
function string   basename      ();
function string   tail          (int backsteps=1);
function string   volume        ();
```

Query whether a pathname is absolute (begins with `/`) or not:

```
function bit      isAbsolute    ();
```

Create a new Pathname object with identical contents to the invoking object:

```
function Pathname copy();
```

7.1 Overview

The Pathname class provides a convenient way to store and manipulate a file name. As the filename is manipulated, its integrity is maintained.

For example, consider the following sequence of activity. First, the path to a certain directory is stored in a Pathname object. Note the doubled path-separator "`//`" in the middle of the name string, and the absence of separator after the final directory name component. Both these features are completely legal, but can easily confuse simple string-based processing of pathnames:

```
Pathname pn = Pathname::create("/user/jb//myDir");
```

This step creates an object of Pathname type, with the given path already saved in it. Next, we append a further path to the current value of `pn`:

```
pn.append("subDir/myFile.txt");
```

If this string were simply appended to the original path name, the result would be nonsense because there was no `/` separator at the end of the original path. However, the Pathname object understands this, and correctly concatenates the directory components – as we can see by asking the Pathname object to provide the new path as a string:

```
$display(pn.get());  // displays /user/jb/myDir/subDir/myFile.txt
```

Note that the doubled separator has been correctly flattened to a single forward-slash, and a separator has been correctly added between the two directory components.

Other manipulations will always return a string, without affecting the stored path value. For example, suppose we wished to find only the trailing filename part of this path, and its extension:

```
$display(pn.tail());  // displays myFile.txt
$display(pn.extension());  // displays .txt
$display(pn.get());   // displays /user/jb/myDir/subDir/myFile.txt
```

Given this behavior, if you wish to use any of these functions to modify a Pathname object you will need to use the `set` method to update your object with the modified value. In the following example, we use the `dirname` method to back up 2 levels of path hierarchy:

```
pn.set( pn.dirname(2) );
$display(pn.get());  // displays /user/jb/myDir
```

7.2 Detailed descriptions of the methods

`set` updates the object with a new pathname. The pathname is normalized (multiple separators are collapsed to a single separator, and any trailing separator is removed).

`get` returns the pathname as a string, in the normalized form.

`tail` returns the last component of the pathname, and `dirname` returns the path with the last component removed. Both methods take an optional `backsteps` argument (default = 1) specifying how many components to back up:

```
$display(pn.tail(3)); // displays myDir/subDir/myFile.txt
```

`extension` returns the part of the pathname after the last period character in the last component, including the period character itself. If there is no period character in the last component, `extension` returns an empty string. `basename` returns the pathname with the `extension`, if any, removed.

`isAbsolute` returns a bit indicating whether the pathname is absolute.

`volume` returns the first component of an absolute pathname or, for a relative pathname, what the first component would be if the pathname were to be made absolute. On Unix and Linux platforms, `volume` invariably returns the single character `/`.

`append` and `appendPN` update the object's pathname by appending a pathname to its tail. They have the same behavior, but `updatePN` takes a second Pathname object as its argument. If the added pathname is absolute, then the original pathname is discarded and the updated pathname becomes the same as the argument.

7.3 Restrictions and special cases

The relative-directory path components `.` and `..` are not treated specially in any way. This design decision allows users to specify relative paths with no risk of the Pathname object corrupting the user's original intent.

Shell special characters such as ~, and environment variables such as `$LD_LIBRARY_PATH`, get no special treatment and are not expanded. They simply remain in the path as component strings.

For the purposes of backstep processing by `dirname` and `tail`, the volume indicator `/` of a relative pathname is treated as a distinct pathname component. For example:

```
pn.set("/short/absolute/path");
$display(pn.dirname(2));  // "/short"
$display(pn.dirname(3));  // "/"
$display(pn.dirname(4));  // ""
```

If the backstep argument supplied to `dirname` or `tail` is negative or zero, `dirname` returns the entire path (exactly like `get`) and `tail` returns the empty string. If the backstep argument is greater than the number of components, `dirname` returns the empty string and `tail` returns the entire path.

7.4 Important note

The Pathname object and its methods *never* make any access to the file system. They merely process strings in a special way that conforms to file naming conventions. You are free to manipulate a completely imaginary, non-existent filename if you wish.

8 Querying properties of files

8.1 Struct definitions for file properties

The following struct definitions are useful for extracting file properties (such as "is this file a directory") from the file mode value returned by the `file_mode` function described later in this section.

```
typedef struct packed {
  bit r;
  bit w;
  bit x;
} sys_fileRWX_s;

typedef struct packed {
  bit          setUID;
  bit          setGID;
  bit          sticky;
  sys_fileRWX_s owner;
  sys_fileRWX_s group;
  sys_fileRWX_s others;
} sys_filePermissions_s;

typedef enum bit [3:0] {
  fTypeFifo    = 4'h1,
  fTypeCharDev = 4'h2,
  fTypeDir     = 4'h4,
  fTypeBlkDev  = 4'h6,
  fTypeFile    = 4'h8,
  fTypeSymLink = 4'hA,
  fTypeSocket  = 4'hC
} sys_fileType_enum;

typedef struct packed {
  sys_fileType_enum     fType;
  sys_filePermissions_s fPermissions;
} sys_fileMode_s;

typedef struct {
  longint       mtime;
  longint       atime;
  longint       ctime;
  longint       size;
  int unsigned  uid;
  int unsigned  gid;
  sys_fileMode_s mode;
} sys_fileStat_s;
```

8.2 Query functions

Each of these package-level functions interrogates a file to find certain properties such as timestamp or file kind. The file is specified by its string pathname. If the file does not exist, a default value (usually zero) is returned, and an error is thrown as described in section 10.

All these functions (except `file_accessible`) also take an argument `asLink`. This argument controls the functions' behavior if the path specifies a symbolic link. If `asLink` is false (the default), then the symbolic link is followed and the query function examines the file that is referenced by that symbolic link. However, if `asLink` is set (true), the query function examines the symbolic link itself.

```
function automatic longint file_mTime(string path, bit asLink=0);
function automatic longint file_aTime(string path, bit asLink=0);
function automatic longint file_cTime(string path, bit asLink=0);
function automatic longint file_size(string path, bit asLink=0);
function automatic sys_fileMode_s file_mode(string path, bit asLink=0);
function automatic bit file_accessible(
                          string path, sys_fileRWX_s mode = 0);
```

`file_mTime/aTime/cTime` query the file's timestamps: modification time, accessed time, and creation time.

`file_size` returns the file's size in bytes.

`file_mode` returns a `sys_fileMode_s` struct (see section 8.1) containing various properties of the file such as its permissions, group and user ownership, and its file type. In particular, it may be useful to query whether a file is or is not a directory:

```
sys_fileMode_s fMode;
fMode = file_mode("maybe/directory");
if (fMode.fType == fTypeDir)
  $display("It's a directory");
```

`file_accessible` allows you to determine whether the current running process is able to access a file in different ways. For example, to determine if you are able to read a file, you need to arrange that the `r` bit of the `sys_fileRWX_s` argument is set:

```
if (file_accessible("maybe/readable", '{r:1, w:0, x:0}))
  $display("OK to read that file");
```

As a special case, if *none* of the access mode bits are set (the default), the function queries whether the file exists:

```
if (file_accessible("maybe/exists"))
  $display("the file exists");
```

To get a list of all files in a directory, see function `sys_fileGlob` described in section **Error! Reference source not found.**.

9 Operating system queries

All operating system queries are provided as package-level functions. svlib has no class to represent the operating system.

9.1 Wall-clock time and human-readable time formatting

```
function automatic longint sys_dayTime();
function automatic string sys_formatTime(
    input longint epochSeconds,
    input string  format
  );
```

All times returned by the file timestamp query functions are in the usual Unix format of seconds since the beginning of 1970 (the "epoch"). In a similar way, `sys_dayTime` returns the current wall-clock time in that same format.

`sys_formatTime` uses the C library's time formatting features to create human-readable time strings from any seconds-since-1970 epoch time. The `format` string works rather like the format string for `$sformatf` or `$display` (it can include arbitrary plain text along with its `%X` format placeholders) but the format placeholders are very different. They are described in full by the Unix/Linux man page `man 3 strftime`.

9.2 High-resolution timer

```
function automatic longint unsigned sys_nsTime();
```

`sys_nsTime` returns the highest available resolution real-time clock, scaled to nanoseconds. The exact resolution of this clock is not guaranteed, although 256ns resolution appears to be common on x86 Linux systems. It may be useful for performance measurements.

9.3 Directory queries

```
function automatic qs sys_fileGlob(string wildPath);
function automatic string sys_getCwd();
```

`sys_getCwd` returns the full path of the current working directory.

`sys_fileGlob` returns a queue of strings containing all files that match the shell glob pattern `wildPath`, using the normal shell wildcard characters `*` and `?`. It provides a convenient way to determine the files that exist in a certain directory.

9.4 Environment variables

```
function automatic string sys_getEnv(string envVar);
function automatic bit    sys_hasEnv(string envVar);
```

`sys_getEnv` returns the value of the chosen environment variable, or the empty string if the variable does not exist. `sys_hasEnv` returns 1 if the chosen environment variable exists, 0 if it does not. Note that an empty string result from `sys_getEnv` is ambiguous, because it is possible (and, indeed, quite useful) to define an environment variable that has no value.

10 svlib error management

```
function automatic svlibErrorManager error_getManager();
function automatic void error_userHandling(bit user, bit setDefault=0);
function automatic int error_getLast(bit clear = 1);
function automatic string error_text(int err=0);
function automatic string error_details();
function automatic string error_fullMessage();
function automatic qs error_debugReport();
```

Material to be added

11 Document Object Model

```
virtual class svlibCfgBase extends svlibBase;
  pure virtual function cfgObjKind_e kind();
  virtual function      string       getName();
  virtual function      string       getLastErrorDetails();
  virtual function      cfgError_e   getLastError();
  virtual function      string       kindStr();
endclass

virtual class cfgScalar extends svlibCfgBase;
  pure virtual function string  str();
  pure virtual function bit     scan(string s);
endclass

virtual class cfgNode extends svlibCfgBase;
  pure virtual function string  sformat(int indent = 0);
  pure virtual function cfgNode childByName(string idx);
  function              cfgNode lookup(string path);
  virtual function      void    addNode(cfgNode nd);
  virtual function      cfgNode getParent();
endclass
```

11.1 Overview

svlib's Document Object Model (DOM) provides a general-purpose mechanism for building, within SystemVerilog, a tree structure containing arbitrary user-defined data. It is suitable for capturing data stored in common structured file formats such as XML and YAML, and svlib uses the DOM as an intermediate representation of such file formats as part of its configuration file features (see section 12). It can also be used for arbitrary user-defined data.

The DOM currently supports leaf data nodes of string or integral type. It can easily be extended to support other data types if required. Such extension is beyond the scope of this document, and will be documented separately in a *Developers' Guide* document.

11.2 Use in configuration

The DOM is an integral part of svlib's configuration-file mechanism. However, for that usage it is hidden behind a set of convenience macros that allow an object's contents to be read and written from/to a configuration file. It is anticipated that most users will take advantage of these convenience macros, and will have no need to use the DOM directly.

11.3 Other uses of a DOM

Material to be added

12 Configuration files

The current release of svlib supports configuration files in `.ini` format.

[***NOTE**:* At the time of writing, YAML support has not yet been implemented. It will be added in an upcoming release.]

12.1 Overview

In svlib, a configuration file contains a representation of a SystemVerilog object. If that object also contains other object instances (sub-objects) then the configuration file can also represent those sub-objects' contents. Within the file, data items are represented by a name and a value, and in typical file formats those representations must be text strings. File formats such as YAML support hierarchical representations, which readily allow for representation of arrays, associative arrays and sub-objects. The `.ini` file format supports only a very simple kind of hierarchy (sections and named items) and so its ability to represent arrays and sub-objects is extremely limited; on the other hand, it is simple and widely known, and so may be useful in some applications.

To allow for easy support of various file formats, svlib uses the DOM (see section 11) as a format-agnostic intermediate representation of user data. For each supported file format, it is easy to transfer data and variable names between a DOM and the file. svlib can provide completely general serialize/deserialize methods to do these operations without programmer intervention. However, this leaves the problem of how to transfer data between a user's data objects (as used in the user's normal SV code), and the DOM (which is a powerful abstraction for svlib's internal use, but is not convenient for general programming).

For variables (data members) of an object to be represented in the DOM, it is necessary for the names of those variables to be known as text strings. This cannot be done directly in SystemVerilog. Either the user must provide specialized conversion methods for each new object they create, or (*much* more convenient) automation macros can be used. These macros allow the user to indicate which variables of a class are to be included in a DOM, and automatically create appropriate code to support transfer to and from the DOM.

12.2 A simple example

Suppose you have a simple configuration data class, possibly extended from `uvm_sequence_item` or some similar methodology base class:

```
class LocalCfg extends ...;
  int    choice;
  string label;
endclass
```

We can represent an object of this type, with its stored integral and string values, in a svlib DOM. However, copying the data from one form to another is not trivial. By making use of the svlib DOM macros, however, it becomes straightforward:

```
class LocalCfg extends ...;
  int    choice;
  string label;
  `SVLIB_DOM_UTILS_BEGIN(LocalCfg)
   `SVLIB_DOM_FIELD_INT(choice)
   `SVLIB_DOM_FIELD_STRING(label)
  `SVLIB_DOM_UTILS_END
endclass
```

By using these macros, you have created two new methods of your class:

```
function void       fromDOM (cfgNodeMap dom);
function cfgNodeMap toDOM   (string name);
```

These methods are all that is required to transfer the specified contents of your object to or from a svlib DOM (of class `cfgNodeMap`). You don't need to manipulate the DOM object yourself in any way. It is merely an intermediate representation that you can then copy to or from a configuration file. `fromDOM` populates your object's variables from the values stored in the `cfgNodeMap` object `dom`. `toDOM` creates a new `cfgNodeMap` object, and populates it from the specified variables `choice` and `label` in your object; it also annotates the new `cfgNodeMap` object with your specified `name`, so that it can be included in a hierarchy of objects if necessary.

First, let's see how we can transfer the contents of your object to a DOM and from there to a `.ini` file. We will then look at the inverse operation, reading a `.ini` file into your data object.

[***NOTE**:* The presentation of svlib at DVCon 2014 indicated the use of single-step methods `writeFromDOM` and `readToDOM` to transfer a DOM to or from a file. Those methods are not yet supported in current releases of svlib. Instead, use the idiom indicated here.]

Given the class `LocalCfg` outlined above, with its `SVLIB_DOM` macros, we can stream an object to a `.ini` file thus:

```
cfgNodeMap cfg_DOM;   // A generic DOM object (no need for "new")
cfgFileINI cfg_file = cfgFileINI::create(); // file container
cfgError_enum err;

LocalCfg   cfg = new; // The user configuration object of interest
cfg.choice = 42;
cfg.label = "Local Configuration";

cfg_DOM = cfg.toDOM("cfg");  // Copy user object to DOM representation
err = cfg_file.openW("user_file.ini");  // Open the file for writing
err = cfg_file.serialize(cfg_DOM);      // Write the DOM to the file
err = cfg_file.close();                 // finalize the file
```

At each step you are of course free to examine the error code from each file operation; if it is non-zero, you can report it using `err.name`.

The object has now been written to `user_file.ini`, which should contain the following text:

```
choice=42
label="Local Configuration"
```

Reading a file back into a DOM, and thence to a user object, is almost exactly the reverse operation. Given the same declarations as the previous example, it could be done thus:

```
err = cfg_file.openR("user_file.ini");  // Open the file for reading
err = cfg_file.deserialize(cfg_DOM);    // Read the file into a DOM
err = cfg_file.close();                 // finalize the file
cfg.fromDOM(cfg_DOM);  // Copy DOM representation into user object
```

12.3 Nested objects

One of the most attractive features of the DOM representation is that it can support arbitrarily deep trees of objects. (Unfortunately the `ini` file format supports only one level of nesting; YAML will lift this restriction.) We can illustrate this using a second configuration class, `GlobalCfg`, that contains two `LocalCfg` objects:

```
class GlobalCfg extends ...;
  LocalCfg leftCfg;
  LocalCfg rightCfg;
  string   label;
   `SVLIB_DOM_UTILS_BEGIN(GlobalCfg)
   `SVLIB_DOM_FIELD_OBJECT(leftCfg)
   `SVLIB_DOM_FIELD_OBJECT(rightCfg)
   `SVLIB_DOM_FIELD_STRING(label)
  `SVLIB_DOM_UTILS_END
endclass
```

An object of this type could now be populated from the following `.ini` file. Note how the section names exactly match the variable names for the inner `LocalCfg` objects:

```
label=”global config object”

[rightCfg]
choice=42
label=”Local Configuration”

[leftCfg]
choice=42
label=”Local Configuration”
```

Thanks to the macros, it is now necessary only to read this file into a DOM, and then transfer the DOM into a `GlobalCfg` object. Creation (if necessary) and population of the lower-level objects is fully automated by the `fromDOM` method:

```
GlobalCfg gc = new;
err = cfg_file.openR(“global_file.ini”);  // Open file for reading
err = cfg_file.deserialize(cfg_DOM);      // Read the file into a DOM
err = cfg_file.close();                   // finalize the file
gc.fromDOM(cfg_DOM);  // Copy DOM representation into user objects
```

Similarly, the complete tree of objects can be transferred by a single call to `gc.toDOM()`.

13 Simulation environment queries

```
class Simulator extends svlibBase;
  static function string getToolName();
  static function string getToolVersion();
  static function qs getCmdLine();
endclass
```

The `Simulator` class offers a small set of utilities for querying the simulation environment.

13.1 Tool name and version queries

`Simulator::getToolName()` and `Simulator::getToolVersion()` return vendor-specific strings indicating the current running simulator.

13.2 Simulation command-line query

`Simulator::getCmdLine()` returns a queue of strings, containing all command-line arguments and options that were used to launch the simulator.

Arguments that were stored in "response files" using the `-f` option are flattened. The `-f` option and its associated filename are removed, and all options contained within the response file (including the contents of any nested response files) appear in-line in the list.

If an argument was duplicated on the actual command line, it will appear twice in the list. Apart from the removal of `-f` response file structure, no processing is performed on the argument list.

This query is expected to be especially useful in supporting more sophisticated processing of command line options than is possible with traditional Verilog `$value$plusargs`. In particular, a given plusarg can appear more than once, with different values, and all occurrences can be processed by scanning the list returned by `Simulator::getCmdLine()`. Additionally, pattern matching can be performed on plusargs – it is not necessary to know in advance exactly which plusarg names will be used.

Simulation tool suites from various vendors differ in exactly how they pass user-defined arguments through from compilation to runtime. This is especially important when using single-step compile/elaborate/run flows such as `vcs`, `irun` or `qverilog`. For example, some such flows require you to supply an explicit marker in the argument list, indicating which arguments are to be passed through to the runtime simulator command. Check your tool vendor's documentation for details, and be prepared to experiment within your own chosen tool flow. Generally, `Simulator::getCmdLine()` will reliably reflect the arguments and options supplied to a command that actually runs the simulation tool (`vsim` for Mentor Questa, `ncsim` for Cadence Incisive, and the `simv` executable for Synopsys VCS). Command-line arguments and options supplied to other parts of the tool flow may need special handling to ensure that they are saved in the compiled/elaborated image, and subsequently passed to the running simulation.

14 Utilities for enumeration types

```
class EnumUtils #(type ENUM = int);
  static function ENUM from_name  (string s);
  static function int  pos        (BASE   b);
  static function bit  has_name   (string s);
  static function bit  has_value  (BASE   b);
  static function qe   all_values ();
  static function ENUM match      (BASE b, bit requireUnique = 0);
  static function int  maxNameLength();
endclass
```

The `EnumUtils` class provides various utility methods designed to help with manipulation of enumerated types and their values. The class is parameterized for the enumerated type it will manipulate, and the type parameter `ENUM` *must* be overridden with an actual enumerated type in your code; if you try to use the default `int` parameterization there will be many elaboration-time errors. Ideally the `ENUM` parameter should have no default type; this would lead to much clearer error messages if a user forgets to provide appropriate type parameterization. However, not all simulators support this feature at the time of writing.

In practice it is usually easiest to create a `typedef` to give a simple short name for a type-overridden specialization of `EnumUtils`, as in the following example:

```
// User's enumeration type
typedef enum logic [3:0] {
  RED = 4'b1010, GREEN = 4'b0101, ANY = 4'bxxxx
} shade_enum;
// Typedef for specialization of EnumUtils
typedef EnumUtils#(shade_enum) shadeUtils;
```

The `BASE` type used in some methods of `EnumUtils` is a placeholder for the underlying integral data type of the enum. In our example it is `logic [3:0]`.

14.1 All methods of EnumUtils are static

Because the `EnumUtils` class provides services related to an enumerated *type*, there is no reason for the class ever to be instanced. All its methods are `static` and can be called, with an appropriate class scope prefix, even when no object of the corresponding class type has been created.

Because of the need to use the class's name as a scope prefix, it is usually easiest to create a `typedef` to give a simple short name for each type-overridden specialization of `EnumUtils`, as in the following example:

```
// User's enumeration type
typedef enum logic [3:0] {
  RED = 4'b1010, GREEN = 4'b0101, DF = 4'b11x1
} shade_enum;
// Typedef for specialization of EnumUtils
typedef EnumUtils#(shade_enum) shadeUtils;
```

The `BASE` type used in some methods of `EnumUtils` is a placeholder for the underlying integral data type of the enum. In our example it is `logic [3:0]`.

14.2 Detailed descriptions of the methods

14.2.1 Lookup based on string name

Although SystemVerilog allows you to determine the string name of the label corresponding to a value of enumerated type, using its `.name` method, there is no built-in means to go the other way: given a string, find the enumeration value that it matches. `svlib` provides support for this requirement.

`hasName` allows you to determine whether a given string name is one of the set of available enumeration labels of a given type. For example, given the type declaration `shade_enum` in section 12 above:

- `shadeUtils::hasName("rubbish")` returns zero, because the label is not one of the names of type `shade_enum`
- `shadeUtils::hasName("RED")` returns 1, because the name RED exists in the type `shade_enum`
- `shadeUtils::fromName("RED")` returns the value RED (4'b1010)
- `shadeUtils::fromName("rubbish")` returns the value 4'bxxxx, the default (initialization) value of a member of the enum. Any attempt such as this to find the value from a non-existent name will yield the enum's default value. This may or may not be a legal value of the enum. It will be `'x` if the enum is 4-state and `'0` if the enum is 2-state.

14.2.2 Lookup based on underlying value

Although it is always possible in SystemVerilog to cast an integral value to an enumeration, such casting can sometimes yield a non-existent enumeration value. Function `hasValue` easily allows you to check whether a given integral value is a member of the value-set of an enumeration type, yielding 1 if the specified value exists in the enumeration, and zero if it does not exist.

The values of an enumerated type are not always specified in ascending numerical order. Occasionally, a value's position (rank) in the specified order may be of interest. To discover the rank order of an enumeration value within the type, use method `pos` on either the enumeration or its underlying numeric value. For example, in our type `shade_enum`, value `GREEN` (4'b0101) is the second value in the list, and therefore has rank 1. Consequently:

- `shadeUtils::pos(GREEN)` returns 1
- `shadeUtils::pos(4'b0101)` returns 1
- `shadeUtils::pos(4'b0011)` returns -1, because the value 4'b0011 does not appear in our enumerated type.

Ranks are numbered from 0 to N-1 where N is the number of enumeration values in the type. The `pos` method returns -1 when its argument value does not appear in the enumerated type.

14.2.3 Getting a list of all values in the enumerated type

The `allValues` method returns a queue of values of the enumerated type, containing all the enumerated values in their declaration order. The placeholder type name `qe` stands for "queue of enumerations" so, given the declarations already supplied, the following code fragment would be legal:

```
shade_enum se[$];
se = shadeUtils::allValues();
foreach (se[i]) $display("se[%0d] = %s", i, se[i].name);
```

The results would be

```
se[0] = RED
se[1] = GREEN
se[2] = DF
```

Note that the placeholder type `qe` is *not* visible to user code; you must instead create your own type that is a queue of the required enumeration type.

14.2.4 Wildcard matching of a value to an enumeration

The ability to put X or Z values in enumerations can occasionally be useful to indicate that certain bit positions in the enumeration are don't care. However, the usefulness of this trick is limited because it's awkward to find whether a given integral value matches one of your enums. The `match` function can help with this task. Given any integral value, it determines which enumeration matches it, using `==?` wildcard matching between your integral value (on the left) and the enumeration values (on the right). Sometimes a value can match more than one enumeration. If this occurs, `match` will return the first match in the list – unless you specify unique matching by setting the `requireUnique` argument to 1.

If no match is found, or if `requireUnique` has been specified and there is more than one match:

- `match` returns the argument value `b`, cast to the enum type
- the function throws an error, which (as usual) can be handled by the user if per-process user error handling has been specified as described in section 10.

For example, suppose we have opcodes for a simple microprocessor, defined by this enum:

```
typedef enum logic [7:0] {
  ADD = 8'b00_xxx_xxx, // 2-bit opcode, two 3-bit register addresses
  SUB = 8'b01_xxx_xxx, // similar
  ...
  SHIFT = 8'b11_xxx_00x, // 3-bit reg adrs, 1-bit shift direction
  ...
} opcode_enum;
```

Now we observe the processor's instruction register, and we wish to decide which opcode is present. A simple match operation won't work, because of the varying bit positions of the wildcard Xs. However, the `match` method of `EnumUtils` will do the job, effectively scanning through all possible enum values and comparing them with the instruction value using `==?` comparison:

```
bit [7:0] instruction;
opcode_enum opcode;
// pick value out of the instruction register somehow
instruction = ...;
// find which opcode it represents
opcode = EnumUtils#(opcode_enum)::match(instruction);
```

Note that in this example we have directly used the appropriate specialization of `EnumUtils`, without creating any named typedef. Either method is appropriate; you are free to choose whichever is clearer or more convenient in your code.

15 Utility macros

The library provides a small number of utility macros.

15.1 foreach_enum

It is sometimes useful to iterate over the set of values in an enumerated type. Although this is not difficult to do in SystemVerilog, the required code pattern is somewhat clumsy and obscures the intent of the code. To ease this problem, `svlib` offers a macro that provides iteration over an enumeration in a similar style to a foreach-loop.

```
typedef enum {FIRST = 1, SECOND = 5, THIRD = 3} ordinal_enum;
...
`foreach_enum(ordinal_enum, value)
  $display("%s has integral value %0d", value.name, value);
```

The variable (`value` in this example) is declared, locally, as a variable of the enum type, just as the iteration variable in a foreach-loop is declared locally. You can choose any name you wish for this variable, so long as it is a legal SystemVerilog identifier. The output from this example will be:

```
FIRST has integral value 1
SECOND has integral value 5
THIRD has integral value 3
```

Like any other loop construct, `foreach_enum` can be used with begin..end. Additionally, you can provide a third argument which is a locally declared variable of `int` type; this variable simply counts from 0 to `num`-1 (where `num` is the number of named values in the enumeration type):

```
`foreach_enum(ordinal_enum, value, j) begin
  if (j==0)
    $display("Skipping the first value");
  else
    $display("Value at position %0d is %s(%0d)",
                              j, value.name, value);
end
```

The output from that example is:

```
Skipping the first value
Value at position 1 is SECOND(5)
Value at position 2 is THIRD(3)
```

15.2 foreach_line

The `foreach_line` macro is simply syntactic sugar for looping over every line in a plain-text file. The file is assumed to have been already opened for reading, and the macro does not close the file when it's done. The macro acts like a `foreach` loop, and locally declares two new variables: one `string` to hold each line in turn, and one `int` to hold the line number within the file. Each line is presented *with* its trailing end-of-line character, if any (you can easily remove that character with the `str_trim` function).

```
int fileID;
fileID = $fopen("some_text_file", "r");
if (fileID == 0) ... // failed to open the file - error
// Now the file is open, we can scan the lines in it
`foreach_line(fileID, line, lineNum) begin
    $display("Line %2d: \"%s\"", lineNum, line);
end
```

16 Worked examples

Material to be added