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.

1.3 Copyright information

This text is copyright ©Verilab Inc. 2014. It is provided as a companion to the `svlib` utility package. Permission is granted to make unlimited copies of this document, but you must include the whole of Section 1 without alteration.

1.3.1 Limitation of liability

This document, and the accompanying code package are provided "as is" without warranty of any kind. Verilab, Inc accepts no liability for the correct functioning of this package. If you wish to make use of it, you accept full responsibility for the results.

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 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 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.

2.1 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>
```

2.2 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>
```

2.3 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>
```

3 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.

3.1 Overall structure of the library

3.1.1 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 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.

3.1.2 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.

3.2 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 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.

3.3 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.

3.4 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, 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 9 for details.

3.5   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.

3.6   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.

3.6.1   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`.

3.6.2   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`.

3.6.3   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`.

4 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.

4.1 The Str class

4.1.1 Methods that manage a Str object and its contents

```
static function 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.

4.1.2 Methods that manipulate a string in various ways

```
typedef enum {NONE, LEFT, RIGHT, BOTH} side_enum;
typedef enum {START, END} origin_enum;
function void append(string s);
function int first (string substr, int ignore=0);
function int last (string substr, int ignore=0);
function qs split (string splitset="", bit keepSplitters=0);
function string sjoin (qs strings);
function string range (int p, int n, origin_enum origin=START);
function void replace(string rs, int p, int n, origin_enum origin=START);
function void trim (side_enum side=BOTH);
function void pad (int width, side_enum side=BOTH);
```

4.2 Package-level string functions

```
function bit isSpace(byte unsigned ch);
function string str_sjoin(qs elements, string joiner);
function string str_repeat(string s, int n);
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);
```

5 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 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 6).

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.

5.1 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 4):

```
   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.

5.1.1 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!

5.1.2 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.
- 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; they do not affect the stored match values for a match that has already been performed.

5.1.3 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`.

5.2 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 6). 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 do they 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.

5.3 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 5.6 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.

5.4 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);
function int    getMatchStart (int m);
function int    getMatchLength(int m);
```

With the exception of `getMatchCount`, these methods extract and return the match specified by the argument `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.

- `getMatchCount` returns the number of matches provided by the regular expression. 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. If you call this method on a `Regex` object that has never been used to do a match, it will return -1.
- `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))
```

5.5   Substitution (search-and-replace)

```
function int Regex::subst(string substStr, int startPos = 0);
function int Regex::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.

5.6   Errors in regular expression matching

```
function int Regex::getError();
function string Regex::getErrorString();
function void Regex::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 File pathname manipulation

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

```
static function 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();
```

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
```

6.1   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.

6.2   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.

6.3   Important note

The Pathname object and its methods will *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.

7 Querying properties of files

7.1 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;
```

7.2 Query functions

Each of the following 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 9.

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's 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 7.1) 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 8.

8 Querying directories, environment variables, wall-clock time and other operating system properties

8.1 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`.

8.2 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.

8.3 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.

8.4 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.

9 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();
```

10 Document Object Model

```
virtual class svlibCfgBase extends svlibBase;
  pure virtual function cfgObjKind_e kind();
  virtual function string     getName();              return nam
  virtual function string     getLastErrorDetails(); return las
  virtual function cfgError_e getLastError();         return las
  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    getFoundNode(); return foundNode;
  virtual function string     getFoundPath(); return foundPath;
  virtual function cfgNode     getParent();    return parent;
endclass
```

11 Miscellaneous functions

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

11.1 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]`.

11.2 Detailed descriptions of the methods

11.2.1 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 11 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.

11.2.2 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.

11.2.3 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.

11.2.4 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 9.

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

```
typedef enum bit [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.

12 Utility macros

The library provides a small number of utility macros.

12.1 *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)
```

12.2 *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
```