## NAME

version::Internals - Perl extension for Version Objects

## DESCRIPTION

Overloaded version objects for all modern versions of Perl. This documents the internal data representation and underlying code for version.pm. See *version.pod* for daily usage. This document is only useful for users interested in the gory details.

## WHAT IS A VERSION?

For the purposes of this module, a version ''number'' is a sequence of positive integer values separated by one or more decimal points and optionally a single underscore. This corresponds to what Perl itself uses for a version, as well as extending the ''version as number'' that is discussed in the various editions of the Camel book.

There are actually two distinct kinds of version objects:

Decimal versions
: Any version which ''looks like a number'', see ''Decimal Versions''. This also includes versions with a single decimal point and a single embedded underscore, see ''Alpha Versions'', even though these must be quoted to preserve the underscore formatting.

Dotted-Decimal versions
: Also referred to as ''Dotted-Integer'', these contains more than one decimal point and may have an optional embedded underscore, see Dotted-Decimal Versions. This is what is commonly used in most open source software as the ''external'' version (the one used as part of the tag or tarfile name). A leading 'v' character is now required and will warn if it missing.

Both of these methods will produce similar version objects, in that the default stringification will yield the version ''Normal Form'' only if required:

```
$v = version->new(1.002); # 1.002, but compares like 1.2.0
$v = version->new(1.002003); # 1.002003
$v2 = version->new("v1.2.3"); # v1.2.3
```

In specific, version numbers initialized as ''Decimal Versions'' will stringify as they were originally created (i.e. the same string that was passed to `new()`. Version numbers initialized as ''Dotted-Decimal Versions'' will be stringified as ''Normal Form''.

### Decimal Versions

These correspond to historical versions of Perl itself prior to 5.6.0, as well as all other modules which follow the Camel rules for the `$VERSION` scalar. A Decimal version is initialized with what looks like a floating point number. Leading zeros **are** significant and trailing zeros are implied so that a minimum of three places is maintained between subversions. What this means is that any subversion (digits to the right of the decimal place) that contains less than three digits will have trailing zeros added to make up the difference, but only for purposes of comparison with other version objects. For example:

```
# Prints Equivalent to
$v = version->new( 1.2); # 1.2 v1.200.0
$v = version->new( 1.02); # 1.02 v1.20.0
$v = version->new( 1.002); # 1.002 v1.2.0
$v = version->new( 1.0023); # 1.0023 v1.2.300
$v = version->new( 1.00203); # 1.00203 v1.2.30
$v = version->new( 1.002003); # 1.002003 v1.2.3
```

All of the preceding examples are true whether or not the input value is quoted. The important feature is that the input value contains only a single decimal. See also ''Alpha Versions''.

IMPORTANT NOTE: As shown above, if your Decimal version contains more than 3 significant digits after the decimal place, it will be split on each multiple of 3, so 1.0003 is equivalent to v1.0.300, due to the need to remain compatible with Perl's own 5.005_03 == 5.5.30 interpretation. Any trailing zeros are ignored for mathematical comparison purposes.

### Dotted-Decimal Versions

These are the newest form of versions, and correspond to Perl's own version style beginning with 5.6.0. Starting with Perl 5.10.0, and most likely Perl 6, this is likely to be the preferred form. This method normally requires that the input parameter be quoted, although Perl's after 5.8.1 can use v-strings as a special form of quoting, but this is highly discouraged.

Unlike ''Decimal Versions'', Dotted-Decimal Versions have more than a single decimal point, e.g.:

```
# Prints
$v = version->new( "v1.200"); # v1.200.0
$v = version->new("v1.20.0"); # v1.20.0
$v = qv("v1.2.3"); # v1.2.3
$v = qv("1.2.3"); # v1.2.3
$v = qv("1.20"); # v1.20.0
```

In general, Dotted-Decimal Versions permit the greatest amount of freedom to specify a version, whereas Decimal Versions enforce a certain uniformity.

Just like ''Decimal Versions'', Dotted-Decimal Versions can be used as ''Alpha Versions''.

### Alpha Versions

For module authors using CPAN, the convention has been to note unstable releases with an underscore in the version string. (See CPAN.) version.pm follows this convention and alpha releases will test as being newer than the more recent stable release, and less than the next stable release. Only the last element may be separated by an underscore:

```
# Declaring
use version 0.77; our $VERSION = version->declare("v1.2_3");

# Parsing
$v1 = version->parse("v1.2_3");
$v1 = version->parse("1.002_003");
```

Note that you **must** quote the version when writing an alpha Decimal version. The stringified form of Decimal versions will always be the same string that was used to initialize the version object.

### Regular Expressions for Version Parsing

A formalized definition of the legal forms for version strings is included in the `version::regex` class. Primitives are included for common elements, although they are scoped to the file so they are useful for reference purposes only. There are two publicly accessible scalars that can be used in other code (not exported):

`$version::LAX`

This regexp covers all of the legal forms allowed under the current version string parser. This is not to say that all of these forms are recommended, and some of them can only be used when quoted.

For dotted decimals:

```
v1.2
1.2345.6
v1.23_4
```

The leading 'v' is optional if two or more decimals appear. If only a single decimal is included, then the leading 'v' is required to trigger the dotted-decimal parsing. A leading zero is permitted, though not recommended except when quoted, because of the risk that Perl will treat the number as octal. A trailing underscore plus one or more digits denotes an alpha or development release (and must be quoted to be parsed properly).

For decimal versions:

```
1
1.2345
1.2345_01
```

an integer portion, an optional decimal point, and optionally one or more digits to the right of the decimal are all required. A trailing underscore is permitted and a leading zero is permitted. Just like the lax dotted-decimal version, quoting the values is required for alpha/development forms to be parsed correctly.

`$version::STRICT`

This regexp covers a much more limited set of formats and constitutes the best practices for initializing version objects. Whether you choose to employ decimal or dotted-decimal for is a personal preference however.

v1.234.5

For dotted-decimal versions, a leading ’v’ is required, with three or more sub-versions of no more than three digits. A leading 0 (zero) before the first sub-version (in the above example, ’1’) is also prohibited.

2.3456

For decimal versions, an integer portion (no leading 0), a decimal point, and one or more digits to the right of the decimal are all required.

Both of the provided scalars are already compiled as regular expressions and do not contain either anchors or implicit groupings, so they can be included in your own regular expressions freely. For example, consider the following code:

```
($pkg, $ver) =~ /
^[ \t]*
use [ \t]+($PKGNAME)
(?:[ \t]+($version::STRICT))?
[ \t]*;
/x;
```

This would match a line of the form:

```
use Foo::Bar::Baz v1.2.3; # legal only in Perl 5.8.1+
```

where `$PKGNAME` is another regular expression that defines the legal forms for package names.

## IMPLEMENTATION DETAILS

### Equivalence between Decimal and Dotted-Decimal Versions

When Perl 5.6.0 was released, the decision was made to provide a transformation between the old-style decimal versions and new-style dotted-decimal versions:

```
5.6.0 == 5.006000
5.005_04 == 5.5.40
```

The floating point number is taken and split first on the single decimal place, then each group of three digits to the right of the decimal makes up the next digit, and so on until the number of significant digits is exhausted, **plus** enough trailing zeros to reach the next multiple of three.

This was the method that version.pm adopted as well. Some examples may be helpful:

```
equivalent
decimal zero-padded dotted-decimal
------- ---------- -------------
1.2 1.200 v1.200.0
1.02 1.020 v1.20.0
1.002 1.002 v1.2.0
1.0023 1.002300 v1.2.300
1.00203 1.002030 v1.2.30
1.002003 1.002003 v1.2.3
```

**Quoting Rules**

Because of the nature of the Perl parsing and tokenizing routines, certain initialization values **must** be quoted in order to correctly parse as the intended version, especially when using the `declare` or *"qv()"* methods. While you do not have to quote decimal numbers when creating version objects, it is always safe to quote **all** initial values when using version.pm methods, as this will ensure that what you type is what is used.

Additionally, if you quote your initializer, then the quoted value that goes **in** will be exactly what comes **out** when your `$VERSION` is printed (stringified). If you do not quote your value, Perl's normal numeric handling comes into play and you may not get back what you were expecting.

If you use a mathematic formula that resolves to a floating point number, you are dependent on Perl's conversion routines to yield the version you expect. You are pretty safe by dividing by a power of 10, for example, but other operations are not likely to be what you intend. For example:

```
$VERSION = version->new((qw$Revision: 1.4)[1]/10);
print $VERSION; # yields 0.14
$V2 = version->new(100/9); # Integer overflow in decimal number
print $V2; # yields something like 11.111.111.100
```

Perl 5.8.1 and beyond are able to automatically quote v-strings but that is not possible in earlier versions of Perl. In other words:

```
$version = version->new("v2.5.4"); # legal in all versions of Perl
$newvers = version->new(v2.5.4); # legal only in Perl >= 5.8.1
```

**What about v−strings?**

There are two ways to enter v-strings: a bare number with two or more decimal points, or a bare number with one or more decimal points and a leading 'v' character (also bare). For example:

```
$vs1 = 1.2.3; # encoded as \1\2\3
$vs2 = v1.2; # encoded as \1\2
```

However, the use of bare v-strings to initialize version objects is **strongly** discouraged in all circumstances. Also, bare v-strings are not completely supported in any version of Perl prior to 5.8.1.

If you insist on using bare v-strings with Perl > 5.6.0, be aware of the following limitations:

1) For Perl releases 5.6.0 through 5.8.0, the v-string code merely guesses, based on some characteristics of v-strings. You **must** use a three part version, e.g. 1.2.3 or v1.2.3 in order for this heuristic to be successful.

2) For Perl releases 5.8.1 and later, v-strings have changed in the Perl core to be magical, which means that the version.pm code can automatically determine whether the v-string encoding was used.

3) In all cases, a version created using v-strings will have a stringified form that has a leading 'v' character, for the simple reason that sometimes it is impossible to tell whether one was present initially.

**Version Object Internals**

version.pm provides an overloaded version object that is designed to both encapsulate the author's intended `$VERSION` assignment as well as make it completely natural to use those objects as if they were numbers (e.g. for comparisons). To do this, a version object contains both the original representation as typed by the author, as well as a parsed representation to ease comparisons. Version objects employ overload methods to simplify code that needs to compare, print, etc the objects.

The internal structure of version objects is a blessed hash with several components:

```
bless( {
'original' => 'v1.2.3_4',
'alpha' => 1,
'qv' => 1,
'version' => [
1,
2,
3,
4
]
}, 'version' );
```

original
: A faithful representation of the value used to initialize this version object. The only time this will not be precisely the same characters that exist in the source file is if a short dotted-decimal version like v1.2 was used (in which case it will contain 'v1.2'). This form is **STRONGLY** discouraged, in that it will confuse you and your users.

qv
: A boolean that denotes whether this is a decimal or dotted-decimal version. See "*is_qv()*" in version.

alpha
: A boolean that denotes whether this is an alpha version. NOTE: that the underscore can only appear in the last position. See "*is_alpha()*" in version.

version
: An array of non-negative integers that is used for comparison purposes with other version objects.

**Replacement UNIVERSAL::VERSION**

In addition to the version objects, this modules also replaces the core UNIVERSAL::VERSION function with one that uses version objects for its comparisons. The return from this operator is always the stringified form as a simple scalar (i.e. not an object), but the warning message generated includes either the stringified form or the normal form, depending on how it was called.

For example:

```
package Foo;
$VERSION = 1.2;

package Bar;
$VERSION = "v1.3.5"; # works with all Perl's (since it is quoted)

package main;
use version;

print $Foo::VERSION; # prints 1.2

print $Bar::VERSION; # prints 1.003005

eval "use foo 10";
print $@; # prints "foo version 10 required..."
eval "use foo 1.3.5; # work in Perl 5.6.1 or better
print $@; # prints "foo version 1.3.5 required..."

eval "use bar 1.3.6";
print $@; # prints "bar version 1.3.6 required..."
eval "use bar 1.004"; # note Decimal version
print $@; # prints "bar version 1.004 required..."
```

IMPORTANT NOTE: This may mean that code which searches for a specific string (to determine whether a given module is available) may need to be changed. It is always better to use the built-in comparison implicit in `use` or `require`, rather than manually poking at `class->VERSION` and then doing a comparison yourself.

The replacement UNIVERSAL::VERSION, when used as a function, like this:

```
print $module->VERSION;
```

will also exclusively return the stringified form. See ''Stringification'' for more details.

## USAGE DETAILS

### Using modules that use version.pm

As much as possible, the version.pm module remains compatible with all current code. However, if your module is using a module that has defined `$VERSION` using the version class, there are a couple of things to be aware of. For purposes of discussion, we will assume that we have the following module installed:

```
package Example;
use version; $VERSION = qv('1.2.2');
...module code here...
1;
```

Decimal versions always work

Code of the form:

```
use Example 1.002003;
```

will always work correctly. The `use` will perform an automatic `$VERSION` comparison using the floating point number given as the first term after the module name (e.g. above 1.002.003). In this case, the installed module is too old for the requested line, so you would see an error like:

```
Example version 1.002003 (v1.2.3) required--this is only version 1.002002 (v1
```

Dotted-Decimal version work sometimes

With Perl >= 5.6.2, you can also use a line like this:

```
use Example 1.2.3;
```

and it will again work (i.e. give the error message as above), even with releases of Perl which do not normally support v-strings (see ''What about v-strings?'' above). This has to do with that fact that `use` only checks to see if the second term *looks like a number* and passes that to the replacement UNIVERSAL::VERSION This is not true in Perl 5.005_04, however, so you are **strongly encouraged** to always use a Decimal version in your code, even for those versions of Perl which support the Dotted-Decimal version.

### Object Methods

*new()*

Like many OO interfaces, the *new()* method is used to initialize version objects. If two arguments are passed to `new()`, the **second** one will be used as if it were prefixed with ''v''. This is to support historical use of the `qw` operator with the CVS variable `$Revision`, which is automatically incremented by CVS every time the file is committed to the repository.

In order to facilitate this feature, the following code can be employed:

```
$VERSION = version->new(qw$Revision: 2.7 $);
```

and the version object will be created as if the following code were used:

```
$VERSION = version->new("v2.7");
```

In other words, the version will be automatically parsed out of the string, and it will be quoted to preserve the meaning CVS normally carries for versions. The CVS `$Revision$` increments differently from Decimal versions (i.e. 1.10 follows 1.9), so it must be handled as if it were a Dotted-Decimal Version.

A new version object can be created as a copy of an existing version object, either as a class method:

```
$v1 = version->new(12.3);
$v2 = version->new($v1);
```

or as an object method:

```
$v1 = version->new(12.3);
$v2 = $v1->new(12.3);
```

and in each case, `$v1` and `$v2` will be identical. NOTE: if you create a new object using an existing object like this:

```
$v2 = $v1->new();
```

the new object **will not** be a clone of the existing object. In the example case, `$v2` will be an empty object of the same type as `$v1`.

*qv()*

An alternate way to create a new version object is through the exported *qv()* sub. This is not strictly like other q? operators (like qq, qw), in that the only delimiters supported are parentheses (or spaces). It is the best way to initialize a short version without triggering the floating point interpretation. For example:

```
$v1 = qv(1.2); # v1.2.0
$v2 = qv("1.2"); # also v1.2.0
```

As you can see, either a bare number or a quoted string can usually be used interchangeably, except in the case of a trailing zero, which must be quoted to be converted properly. For this reason, it is strongly recommended that all initializers to *qv()* be quoted strings instead of bare numbers.

To prevent the `qv()` function from being exported to the caller's namespace, either use version with a null parameter:

```
use version ();
```

or just require version, like this:

```
require version;
```

Both methods will prevent the *import()* method from firing and exporting the `qv()` sub.

For the subsequent examples, the following three objects will be used:

```
$ver = version->new("1.2.3.4"); # see "Quoting Rules"
$alpha = version->new("1.2.3_4"); # see "Alpha Versions"
$nver = version->new(1.002); # see "Decimal Versions"
```

Normal Form

For any version object which is initialized with multiple decimal places (either quoted or if possible v-string), or initialized using the *qv()* operator, the stringified representation is returned in a normalized or reduced form (no extraneous zeros), and with a leading 'v':

```
print $ver->normal; # prints as v1.2.3.4
print $ver->stringify; # ditto
print $ver; # ditto
print $nver->normal; # prints as v1.2.0
print $nver->stringify; # prints as 1.002,
# see "Stringification"
```

In order to preserve the meaning of the processed version, the normalized representation will always contain at least three sub terms. In other words, the following is guaranteed to always be true:

```
my $newver = version->new($ver->stringify);
if ($newver eq $ver ) # always true
{...}
```

Numification

Although all mathematical operations on version objects are forbidden by default, it is possible to retrieve a number which corresponds to the version object through the use of the `$obj`->numify method. For formatting purposes, when displaying a number which corresponds a version object, all sub versions are assumed to have three decimal places. So for example:

```
print $ver->numify; # prints 1.002003004
print $nver->numify; # prints 1.002
```

Unlike the stringification operator, there is never any need to append trailing zeros to preserve the correct version value.

Stringification

The default stringification for version objects returns exactly the same string as was used to create it, whether you used `new()` or `qv()`, with one exception. The sole exception is if the object was created using `qv()` and the initializer did not have two decimal places or a leading 'v' (both optional), then the stringified form will have a leading 'v' prepended, in order to support round-trip processing.

For example:

```
Initialized as Stringifies to
============== ==============
version->new("1.2") 1.2
version->new("v1.2") v1.2
qv("1.2.3") 1.2.3
qv("v1.3.5") v1.3.5
qv("1.2") v1.2 ### exceptional case
```

See also UNIVERSAL::VERSION as this also returns the stringified form when used as a class method.

IMPORTANT NOTE: There is one exceptional cases shown in the above table where the "initializer" is not stringwise equivalent to the stringified representation. If you use the `qv`() operator on a version without a leading 'v' **and** with only a single decimal place, the stringified output will have a leading 'v', to preserve the sense. See the "*qv()*" operator for more details.

IMPORTANT NOTE 2: Attempting to bypass the normal stringification rules by manually applying *numify()* and *normal()* will sometimes yield surprising results:

```
print version->new(version->new("v1.0")->numify)->normal; # v1.0.0
```

The reason for this is that the *numify()* operator will turn "v1.0" into the equivalent string "1.000000". Forcing the outer version object to *normal()* form will display the mathematically equivalent "v1.0.0".

As the example in "*new()*" shows, you can always create a copy of an existing version object with the same value by the very compact:

```
$v2 = $v1->new($v1);
```

and be assured that both `$v1` and `$v2` will be completely equivalent, down to the same internal representation as well as stringification.

Comparison operators

Both `cmp` and `<=>` operators perform the same comparison between terms (upgrading to a version object automatically). Perl automatically generates all of the other comparison operators based on those two. In addition to the obvious equalities listed below, appending a single trailing 0 term does not change the value of a version for comparison purposes. In other words "v1.2" and "1.2.0" will compare as identical.

For example, the following relations hold:

```
As Number As String Truth Value
------------ --------------- -----------
$ver > 1.0 $ver gt "1.0" true
$ver < 2.5 $ver lt true
$ver != 1.3 $ver ne "1.3" true
$ver == 1.2 $ver eq "1.2" false
$ver == 1.2.3.4 $ver eq "1.2.3.4" see discussion below
```

It is probably best to chose either the Decimal notation or the string notation and stick with it, to reduce confusion. Perl6 version objects **may** only support Decimal comparisons. See also ‘‘Quoting Rules’’.

WARNING: Comparing version with unequal numbers of decimal points (whether explicitly or implicitly initialized), may yield unexpected results at first glance. For example, the following inequalities hold:

```
version->new(0.96) > version->new(0.95); # 0.960.0 > 0.950.0
version->new("0.96.1") < version->new(0.95); # 0.096.1 < 0.950.0
```

For this reason, it is best to use either exclusively ‘‘Decimal Versions’’ or ‘‘Dotted-Decimal Versions’’ with multiple decimal points.

Logical Operators

If you need to test whether a version object has been initialized, you can simply test it directly:

```
$vobj = version->new($something);
if ( $vobj ) # true only if $something was non-blank
```

You can also test whether a version object is an alpha version, for example to prevent the use of some feature not present in the main release:

```
$vobj = version->new("1.2_3"); # MUST QUOTE
...later...
if ( $vobj->is_alpha ) # True
```

## AUTHOR

John Peacock <jpeacock@cpan.org>

## SEE ALSO

perl.