NAME

version - Perl extension for Version Objects

SYNOPSIS

```
use version;
$version = version->new("12.2.1"); # must be quoted for Perl < 5.8.1
print $version;   # v12.2.1
print $version->numify;  # 12.002001
if ( $version gt "12.2" ) # true

$alphaver = version->new("1.02_03"); # must be quoted!
print $alphaver;  # 1.02_0300
print $alphaver->is_alpha();  # true

$ver = qv("1.2.0");           # v1.2.0

$perlver = version->new(5.005_03); # must not be quoted!
print $perlver;  # 5.005030
```

DESCRIPTION

Overloaded version objects for all modern versions of Perl. This module implements all of the features of version objects which are part of Perl 5.10.0. All previous releases (i.e. before 0.74) are deprecated and should not be used due to incompatible API changes. If you 'use version' in your code, you are strongly urged to set a minimum, e.g.

```
use version 0.74; # to remain compatible with Perl v5.10.0
```

BEST PRACTICES

If you intend for your module to be used by different releases of Perl, and/or for your $VERSION scalar to mean what you think it means, there are a few simple rules to follow:

* Be consistent

  Whichever of the two types of version objects that you choose to employ, you should stick to either *Numeric Versions* or *Extended Versions* and not mix them together. While this is *possible*, it is very confusing to the average user.

  If you intend to use *Extended Versions*, you are strongly encouraged to use the *qv()* operator with a quoted term, e.g.:

  ```
  use version; our $VERSION = qv("1.2.3");
  ```

  on a single line as above.

  At the very least, decide on which of the several ways to initialize your version objects you prefer and stick with it. It is also best to be explicit about what value you intend to assign your version object and to not rely on hidden behavior of the parser.

* Be careful

  If you are using Module::Build or ExtUtils::MakeMaker, so that you can release your module to CPAN, you have to recognize that neither of those programs completely handles version objects natively (yet). If you use version objects with Module::Build, you should add an explicit dependency to the release of version.pm in your Build.PL:

  ```
  my $builder = Module::Build->new(
     ...
     requires => {
  ```

```
            ... ,
            'version'    => 0.50,
    ...,
       },
       ...
    );
```

and it should Just Work(TM). Module::Build will [hopefully soon] include full support for version objects; there are no current plans to patch ExtUtils::MakeMaker to support version objects.

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;
```

Numeric 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 (v1.2.2)...
```

Extended 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* below). 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 numeric version in your code, even for those versions of Perl which support the extended version.

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:

* Numeric Versions

Any initial parameter which "looks like a number", see *Numeric Versions*. This also covers versions with a single decimal point and a single embedded underscore, see *Numeric Alpha*

Versions, even though these must be quoted to preserve the underscore formatting.

* Extended Versions

Any initial parameter which contains more than one decimal point and an optional embedded underscore, see *Extended 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). The use of the exported *qv()* function also produces this kind of version object.

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("1.2.3");   # v1.2.3
```

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

Numeric 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 numeric 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* for how to handle

IMPORTANT NOTE: As shown above, if your numeric 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.

Extended 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 *Numeric Versions*, Extended 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, Extended Versions permit the greatest amount of freedom to specify a version, whereas Numeric Versions enforce a certain uniformity. See also *New Operator* for an additional method of initializing version objects.

Just like *Numeric Versions*, Extended Versions can be used as *Alpha Versions*.

Numeric Alpha Versions

The one time that a numeric version must be quoted is when a alpha form is used with an otherwise numeric version (i.e. a single decimal point). This is commonly used for CPAN releases, where CPAN or CPANPLUS will ignore alpha versions for automatic updating purposes. Since some developers have used only two significant decimal places for their non-alpha releases, the version object will automatically take that into account if the initializer is quoted. For example Module::Example was released to CPAN with the following sequence of $VERSION's:

```
# $VERSION    Stringified
0.01          0.01
0.02          0.02
0.02_01       0.02_01
0.02_02       0.02_02
0.03          0.03
etc.
```

The stringified form of numeric versions will always be the same string that was used to initialize the version object.

Object Methods

Overloading has been used with version objects to provide a natural interface for their use. All mathematical operations are forbidden, since they don't make any sense for base version objects. Consequently, there is no overloaded numification available. If you want to use a version object in a numeric context for some reason, see the *numify* object method.

* New Operator

  Like all OO interfaces, the new() operator is used to initialize version objects. One way to increment versions when programming is to use 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 numeric versions (i.e. 1.10 follows 1.9), so it must be handled as if it were a *Extended 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);          # 1.2.0
$v2 = qv("1.2");        # also 1.2.0
```

As you can see, either a bare number or a quoted string can usually be used interchangably, 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. This is true of subclasses of version as well, see *SUBCLASSING* for details.

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

```
$ver   = version->new("1.2.3.4"); # see "Quoting" below
$alpha = version->new("1.2.3_4"); # see "Alpha versions" below
$nver  = version->new(1.002);     # see "Numeric Versions" above
```

* 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 numeric notation or the string notation and stick with it, to reduce confusion. Perl6 version objects **may** only support numeric comparisons. See also *Quoting*.

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 *Numeric Versions* or *Extended 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
```

Quoting

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 *qv()* operator. In all cases, a floating point number passed to version->new() will be identically converted whether or not the value itself is quoted. This is not true for *qv()*, however, when trailing zeros would be stripped on an unquoted input, which would result in a very different version object.

In addition, in order to be compatible with earlier Perl version styles, any use of versions of the form 5.006001 will be translated as v5.6.1. In other words, a version with a single decimal point will be parsed as implicitly having three digits between subversions, but only for internal comparison purposes.

The complicating factor is that in bare numbers (i.e. unquoted), the underscore is a legal numeric character and is automatically stripped by the Perl tokenizer before the version code is called. However, if a number containing one or more decimals and an underscore is quoted, i.e. not bare, that is considered a *Alpha Version* and the underscore is significant.

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 will be 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?

Beginning with Perl 5.6.0, an alternate method to code arbitrary strings of bytes was introduced, called v-strings. They were intended to be an easy way to enter, for example, Unicode strings (which contain two bytes per character). Some programs have used them to encode printer control characters (e.g. CRLF). They were also intended to be used for $VERSION, but their use as such has been problematic from the start.

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 (especially the leading 'v' style), since the meaning will change depending on which Perl you are running. It is better to directly use *Extended Versions* to ensure the proper interpretation.

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.

Types of Versions Objects

There are two types of Version Objects:

* Ordinary versions

  These are the versions that normal modules will use. Can contain as many subversions as required. In particular, those using RCS/CVS can use the following:

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

  and the current RCS Revision for that file will be inserted automatically. If the file has been moved to a branch, the Revision will have three or more elements; otherwise, it will have only two. This allows you to automatically increment your module version by using the Revision number from the primary file in a distribution, see *"VERSION_FROM" in ExtUtils::MakeMaker*.

* Alpha Versions

  For module authors using CPAN, the convention has been to note unstable releases with an underscore in the version string, see *CPAN*. Alpha releases will test as being newer than the more recent stable release, and less than the next stable release. For example:

```
$alphaver = version->new("12.03_01"); # must be quoted
```

obeys the relationship

```
12.03 < $alphaver < 12.04
```

Alpha versions with a single decimal point will be treated exactly as if they were *Numeric Versions*, for parsing and output purposes. The underscore will be output when an alpha version is stringified, in the same place as it was when input.

Alpha versions with more than a single decimal point will be treated exactly as if they were *Extended Versions*, and will display without any trailing (or leading) zeros, in the *Version Normal* form. For example,

```
$newver = version->new("12.3.1_1");
print $newver; # v12.3.1_1
```

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, 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 = "1.3.5"; # 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 numeric 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.

SUBCLASSING

This module is specifically designed and tested to be easily subclassed. In practice, you only need to override the methods you want to change, but you have to take some care when overriding new() (since that is where all of the parsing takes place). For example, this is a perfect acceptable derived class:

```
package myversion;
use base version;
sub new {
    my($self,$n)=@_;
    my $obj;
    # perform any special input handling here
    $obj = $self->SUPER::new($n);
    # and/or add additional hash elements here
    return $obj;
}
```

See also *version::AlphaBeta* on CPAN for an alternate representation of version strings.

NOTE: Although the *qv* operator is not a true class method, but rather a function exported into the caller's namespace, a subclass of version will inherit an import() function which will perform the correct magic on behalf of the subclass.

EXPORT

qv - Extended Version initialization operator

AUTHOR

John Peacock <jpeacock@cpan.org>

SEE ALSO

perl.