

## NAME

XSLoader - Dynamically load C libraries into Perl code

## VERSION

Version 0.17

## SYNOPSIS

```
package YourPackage;
require XSLoader;

XSLoader::load();
```

## DESCRIPTION

This module defines a standard *simplified* interface to the dynamic linking mechanisms available on many platforms. Its primary purpose is to implement cheap automatic dynamic loading of Perl modules.

For a more complicated interface, see *DynaLoader*. Many (most) features of *DynaLoader* are not implemented in *XSLoader*, like for example the `dl_load_flags`, not honored by *XSLoader*.

## Migration from DynaLoader

A typical module using *DynaLoader* starts like this:

```
package YourPackage;
require DynaLoader;

our @ISA = qw( OnePackage OtherPackage DynaLoader );
our $VERSION = '0.01';
bootstrap YourPackage $VERSION;
```

Change this to

```
package YourPackage;
use XSLoader;

our @ISA = qw( OnePackage OtherPackage );
our $VERSION = '0.01';
XSLoader::load 'YourPackage', $VERSION;
```

In other words: replace `require DynaLoader` by `use XSLoader`, remove *DynaLoader* from `@ISA`, change `bootstrap` by `XSLoader::load`. Do not forget to quote the name of your package on the `XSLoader::load` line, and add comma (,) before the arguments (`$VERSION` above).

Of course, if `@ISA` contained only *DynaLoader*, there is no need to have the `@ISA` assignment at all; moreover, if instead of `our` one uses the more backward-compatible

```
use vars qw($VERSION @ISA);
```

one can remove this reference to `@ISA` together with the `@ISA` assignment.

If no `$VERSION` was specified on the `bootstrap` line, the last line becomes

```
XSLoader::load 'YourPackage';
```

If the call to `load` is from `YourPackage`, then that can be further simplified to

```
XSLoader::load();
```

as `load` will use `caller` to determine the package.

### Backward compatible boilerplate

If you want to have your cake and eat it too, you need a more complicated boilerplate.

```
package YourPackage;
use vars qw($VERSION @ISA);

@ISA = qw( OnePackage OtherPackage );
$VERSION = '0.01';
eval {
    require XSLoader;
    XSLoader::load('YourPackage', $VERSION);
    1;
} or do {
    require DynaLoader;
    push @ISA, 'DynaLoader';
    bootstrap YourPackage $VERSION;
};
```

The parentheses about `XSLoader::load()` arguments are needed since we replaced `use XSLoader` by `require`, so the compiler does not know that a function `XSLoader::load()` is present.

This boilerplate uses the low-overhead `XSLoader` if present; if used with an antique Perl which has no `XSLoader`, it falls back to using `DynaLoader`.

### Order of initialization: early load()

*Skip this section if the XSUB functions are supposed to be called from other modules only; read it only if you call your XSUBs from the code in your module, or have a `BOOT:` section in your XS file (see "The `BOOT:` Keyword" in `perlxs`). What is described here is equally applicable to the `DynaLoader` interface.*

A sufficiently complicated module using XS would have both Perl code (defined in `YourPackage.pm`) and XS code (defined in `YourPackage.xs`). If this Perl code makes calls into this XS code, and/or this XS code makes calls to the Perl code, one should be careful with the order of initialization.

The call to `XSLoader::load()` (or `bootstrap()`) calls the module's bootstrap code. For modules build by `xsubpp` (nearly all modules) this has three side effects:

- A sanity check is done to ensure that the versions of the `.pm` and the (compiled) `.xs` parts are compatible. If `$VERSION` was specified, this is used for the check. If not specified, it defaults to `$XS_VERSION // $VERSION` (in the module's namespace)
- the XSUBs are made accessible from Perl
- if a `BOOT:` section was present in the `.xs` file, the code there is called.

Consequently, if the code in the `.pm` file makes calls to these XSUBs, it is convenient to have XSUBs installed before the Perl code is defined; for example, this makes prototypes for XSUBs visible to this Perl code. Alternatively, if the `BOOT:` section makes calls to Perl functions (or uses Perl variables) defined in the `.pm` file, they must be defined prior to the call to `XSLoader::load()` (or `bootstrap()`).

The first situation being much more frequent, it makes sense to rewrite the boilerplate as

```
package YourPackage;
use XSLoader;
use vars qw($VERSION @ISA);

BEGIN {
    @ISA = qw( OnePackage OtherPackage );
    $VERSION = '0.01';

    # Put Perl code used in the BOOT: section here

    XSLoader::load 'YourPackage', $VERSION;
}

# Put Perl code making calls into XSUBS here
```

### The most hairy case

If the interdependence of your `BOOT:` section and Perl code is more complicated than this (e.g., the `BOOT:` section makes calls to Perl functions which make calls to XSUBS with prototypes), get rid of the `BOOT:` section altogether. Replace it with a function `onBOOT()`, and call it like this:

```
package YourPackage;
use XSLoader;
use vars qw($VERSION @ISA);

BEGIN {
    @ISA = qw( OnePackage OtherPackage );
    $VERSION = '0.01';
    XSLoader::load 'YourPackage', $VERSION;
}

# Put Perl code used in onBOOT() function here; calls to XSUBS are
# prototype-checked.

onBOOT;

# Put Perl initialization code assuming that XS is initialized here
```

### DIAGNOSTICS

Can't find '%s' symbol in %s

**(F)** The bootstrap symbol could not be found in the extension module.

Can't load '%s' for module %s: %s

**(F)** The loading or initialisation of the extension module failed. The detailed error follows.

Undefined symbols present after loading %s: %s

**(W)** As the message says, some symbols stay undefined although the extension module was correctly loaded and initialised. The list of undefined symbols follows.

### LIMITATIONS

To reduce the overhead as much as possible, only one possible location is checked to find the extension DLL (this location is where `make install` would put the DLL). If not found, the search for the DLL is transparently delegated to `DynaLoader`, which looks for the DLL along the `@INC` list.

In particular, this is applicable to the structure of `@INC` used for testing not-yet-installed extensions. This means that running uninstalled extensions may have much more overhead than running the same extensions after `make install`.

## KNOWN BUGS

The new simpler way to call `XSLoader::load()` with no arguments at all does not work on Perl 5.8.4 and 5.8.5.

## BUGS

Please report any bugs or feature requests via the `perlbug(1)` utility.

## SEE ALSO

*DynaLoader*

## AUTHORS

Ilya Zakharevich originally extracted `XSLoader` from `DynaLoader`.

CPAN version is currently maintained by Sébastien Aperghis-Tramoni <sebastien@aperghis.net>.

Previous maintainer was Michael G Schwern <schwern@pobox.com>.

## COPYRIGHT & LICENSE

Copyright (C) 1990-2011 by Larry Wall and others.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.