



Full credit is given to the above companies including the Operating System (OS) that this PDF file was generated!

Rocky Enterprise Linux 9.2 Manual Pages on command 'ExtUtils::MakeMaker::FAQ.3perl'

\$ man ExtUtils::MakeMaker::FAQ.3perl

ExtUtils::MakeMaker::FAQ(3perl) Perl Programmers Reference Guide ExtUtils::MakeMaker::FAQ(3perl)

NAME

ExtUtils::MakeMaker::FAQ - Frequently Asked Questions About MakeMaker

DESCRIPTION

FAQs, tricks and tips for ExtUtils::MakeMaker.

Module Installation

How do I install a module into my home directory?

If you're not the Perl administrator you probably don't have permission to install a module to its default location. Ways of handling this with a lot less manual effort on your part are perlbrew and local::lib.

Otherwise, you can install it for your own use into your home directory like so:

```
# Non-unix folks, replace ~ with /path/to/your/home/dir
perl Makefile.PL INSTALL_BASE=~
```

This will put modules into ~/lib/perl5, man pages into ~/man and programs into ~/bin.

To ensure your Perl programs can see these newly installed modules, set your "PERL5LIB" environment variable to ~/lib/perl5 or tell each of your programs to look in that directory with the following:

```
use lib "$ENV{HOME}/lib/perl5";
```

or if \$ENV{HOME} isn't set and you don't want to set it for some reason, do it the long way.

```
use lib "/path/to/your/home/dir/lib/perl5";
```

How do I get MakeMaker and Module::Build to install to the same place?

Module::Build, as of 0.28, supports two ways to install to the same location as

MakeMaker.

We highly recommend the `install_base` method, its the simplest and most closely approximates the expected behavior of an installation prefix.

1) Use `INSTALL_BASE / "--install_base"`

MakeMaker (as of 6.31) and `Module::Build` (as of 0.28) both can install to the same locations using the "install_base" concept. See "INSTALL_BASE" in `ExtUtils::MakeMaker` for details. To get MM and MB to install to the same location simply set `INSTALL_BASE` in MM and `--install_base` in MB to the same location.

```
perl Makefile.PL INSTALL_BASE=/whatever
perl Build.PL --install_base /whatever
```

This works most like other language's behavior when you specify a prefix. We recommend this method.

2) Use `PREFIX / "--prefix"`

`Module::Build` 0.28 added support for `--prefix` which works like MakeMaker's `PREFIX`.

```
perl Makefile.PL PREFIX=/whatever
perl Build.PL --prefix /whatever
```

We highly discourage this method. It should only be used if you know what you're doing and specifically need the `PREFIX` behavior. The `PREFIX` algorithm is complicated and focused on matching the system installation.

How do I keep from installing man pages?

Recent versions of MakeMaker will only install man pages on Unix-like operating systems by default. To generate manpages on non-Unix operating systems, make the "manifypods" target.

For an individual module:

```
perl Makefile.PL INSTALLMAN1DIR=none INSTALLMAN3DIR=none
```

If you want to suppress man page installation for all modules you have to reconfigure Perl and tell it 'none' when it asks where to install man pages.

How do I use a module without installing it?

Two ways. One is to build the module normally...

```
perl Makefile.PL
make
make test
```

...and then use `blib` to point Perl at the built but uninstalled module:

```
perl -Mblib script.pl
```

```
perl -Mblib -e '...'
```

The other is to install the module in a temporary location.

```
perl Makefile.PL INSTALL_BASE=~/.tmp
```

```
make
```

```
make test
```

```
make install
```

And then set PERL5LIB to `~/tmp/lib/perl5`. This works well when you have multiple modules to work with. It also ensures that the module goes through its full installation process which may modify it. Again, `local::lib` may assist you here.

How can I organize tests into subdirectories and have them run?

Let's take the following test directory structure:

```
t/foo/sometest.t
```

```
t/bar/othertest.t
```

```
t/bar/baz/anothertest.t
```

Now, inside of the `WriteMakeFile()` function in your `Makefile.PL`, specify where your tests are located with the `test` directive:

```
test => {TESTS => 't/*.t t/**/*.t t/**/*.t'}
```

The first entry in the string will run all tests in the top-level `t/` directory. The second will run all test files located in any subdirectory under `t/`. The third, runs all test files within any subdirectory within any other subdirectory located under `t/`.

Note that you do not have to use wildcards. You can specify explicitly which subdirectories to run tests in:

```
test => {TESTS => 't/*.t t/foo/*.t t/bar/baz/*.t'}
```

PREFIX vs INSTALL_BASE from Module::Build::Cookbook

The behavior of `PREFIX` is complicated and depends closely on how your Perl is configured. The resulting installation locations will vary from machine to machine and even different installations of Perl on the same machine. Because of this, it's difficult to document where `prefix` will place your modules.

In contrast, `INSTALL_BASE` has predictable, easy to explain installation locations.

Now that `Module::Build` and `MakeMaker` both have `INSTALL_BASE` there is little reason to use `PREFIX` other than to preserve your existing installation locations. If you are starting a fresh Perl installation we encourage you to use `INSTALL_BASE`. If you have

an existing installation installed via PREFIX, consider moving it to an installation structure matching INSTALL_BASE and using that instead.

Generating *.pm files with substitutions eg of \$VERSION

If you want to configure your module files for local conditions, or to automatically insert a version number, you can use EUMM's "PL_FILES" capability, where it will automatically run each *.PL it finds to generate its basename. For instance:

```
# Makefile.PL:
require 'common.pl';

my $version = get_version();

my @pms = qw(Foo.pm);

WriteMakefile(
  NAME => 'Foo',
  VERSION => $version,
  PM => { map { ($_ => "\$(INST_LIB)/$_" ) @pms },
  clean => { FILES => join ' ', @pms },
);

# common.pl:
sub get_version { '0.04' }

sub process { my $v = get_version(); s/__VERSION__/$v/g; }

1;

# Foo.pm.PL:
require 'common.pl';

$_ = join " ", <DATA>;

process();

my $file = shift;

open my $fh, '>', $file or die "$file: $!";

print $fh $_;

__DATA__

package Foo;

our $VERSION = '__VERSION__';

1;
```

You may notice that "PL_FILES" is not specified above, since the default of mapping each .PL file to its basename works well.

If the generated module were architecture-specific, you could replace "\$ (INST_LIB)" above with "\$ (INST_ARCHLIB)", although if you locate modules under lib, that would involve ensuring any "lib/" in front of the module location were removed.

Common errors and problems

"No rule to make target `/usr/lib/perl5/CORE/config.h', needed by `Makefile'"

Just what it says, you're missing that file. MakeMaker uses it to determine if perl has been rebuilt since the Makefile was made. It's a bit of a bug that it halts installation.

Some operating systems don't ship the CORE directory with their base perl install. To solve the problem, you likely need to install a perl development package such as perl-devel (CentOS, Fedora and other Redhat systems) or perl (Ubuntu and other Debian systems).

Philosophy and History

Why not just use <insert other build config tool here>?

Why did MakeMaker reinvent the build configuration wheel? Why not just use autoconf or automake or ppm or Ant or ...

There are many reasons, but the major one is cross-platform compatibility.

Perl is one of the most ported pieces of software ever. It works on operating systems I've never even heard of (see perlport for details). It needs a build tool that can work on all those platforms and with any wacky C compilers and linkers they might have.

No such build tool exists. Even make itself has wildly different dialects. So we have to build our own.

What is Module::Build and how does it relate to MakeMaker?

Module::Build is a project by Ken Williams to supplant MakeMaker. Its primary advantages are:

- ? pure perl. no make, no shell commands
- ? easier to customize
- ? cleaner internals
- ? less cruft

Module::Build was long the official heir apparent to MakeMaker. The rate of both its development and adoption has slowed in recent years, though, and it is unclear what the future holds for it. That said, Module::Build set the stage for something to

become the heir to MakeMaker. MakeMaker's maintainers have long said that it is a dead end and should be kept functioning, while being cautious about extending with new features.

Module Writing

How do I keep my \$VERSION up to date without resetting it manually?

Often you want to manually set the \$VERSION in the main module distribution because this is the version that everybody sees on CPAN and maybe you want to customize it a bit. But for all the other modules in your dist, \$VERSION is really just bookkeeping and all that's important is it goes up every time the module is changed. Doing this by hand is a pain and you often forget.

Probably the easiest way to do this is using perl-reversion in Perl::Version:

```
perl-reversion -bump
```

If your version control system supports revision numbers (git doesn't easily), the simplest way to do it automatically is to use its revision number (you are using version control, right?).

In CVS, RCS and SVN you use \$Revision\$ (see the documentation of your version control system for details). Every time the file is checked in the \$Revision\$ will be updated, updating your \$VERSION.

SVN uses a simple integer for \$Revision\$ so you can adapt it for your \$VERSION like so:

```
($VERSION) = q$Revision$ =~ /(\d+)/;
```

In CVS and RCS version 1.9 is followed by 1.10. Since CPAN compares version numbers numerically we use a sprintf() to convert 1.9 to 1.009 and 1.10 to 1.010 which compare properly.

```
$VERSION = sprintf "%d.%03d", q$Revision$ =~ /(\d+)\.(\d+)/g;
```

If branches are involved (ie. \$Revision: 1.5.3.4\$) it's a little more complicated.

```
# must be all on one line or MakeMaker will get confused.
```

```
$VERSION = do { my @r = (q$Revision$ =~ /(\d+)/g); sprintf "%d"."%03d" x $#r, @r };
```

In SVN, \$Revision\$ should be the same for every file in the project so they would all have the same \$VERSION. CVS and RCS have a different \$Revision\$ per file so each file will have a different \$VERSION. Distributed version control systems, such as SVK, may have a different \$Revision\$ based on who checks out the file, leading to a different \$VERSION on each machine! Finally, some distributed version control systems, such as

darcs, have no concept of revision number at all.

What's this META.yml thing and how did it get in my MANIFEST?!

META.yml is a module meta-data file pioneered by Module::Build and automatically generated as part of the 'distdir' target (and thus 'dist'). See "Module Meta-Data" in ExtUtils::MakeMaker.

To shut off its generation, pass the "NO_META" flag to "WriteMakefile()".

How do I delete everything not in my MANIFEST?

Some folks are surprised that "make distclean" does not delete everything not listed in their MANIFEST (thus making a clean distribution) but only tells them what they need to delete. This is done because it is considered too dangerous. While developing your module you might write a new file, not add it to the MANIFEST, then run a "distclean" and be sad because your new work was deleted.

If you really want to do this, you can use "ExtUtils::Manifest::manifind()" to read the MANIFEST and File::Find to delete the files. But you have to be careful. Here's a script to do that. Use at your own risk. Have fun blowing holes in your foot.

```
#!/usr/bin/perl -w
use strict;
use File::Spec;
use File::Find;
use ExtUtils::Manifest qw(maniread);
my %manifest = map { ( $_ => 1 ) }
    grep { File::Spec->canonpath($_) }
        keys %{ maniread() };
if( !keys %manifest ) {
    print "No files found in MANIFEST. Stopping.\n";
    exit;
}
find({
    wanted => sub {
        my $path = File::Spec->canonpath($_);
        return unless -f $path;
        return if exists $manifest{ $path };
        print "unlink $path\n";
```

```

        unlink $path;
    },
    no_chdir => 1
},
"."
);

```

Which tar should I use on Windows?

We recommend ptar from Archive::Tar not older than 1.66 with '-C' option.

Which zip should I use on Windows for '[ndg]make zipdist'?

We recommend InfoZIP: <<http://www.info-zip.org/Zip.html>>

XS

How do I prevent "object version X.XX does not match bootstrap parameter Y.YY" errors?

XS code is very sensitive to the module version number and will complain if the version number in your Perl module doesn't match. If you change your module's version # without rerunning Makefile.PL the old version number will remain in the Makefile, causing the XS code to be built with the wrong number.

To avoid this, you can force the Makefile to be rebuilt whenever you change the module containing the version number by adding this to your WriteMakefile() arguments.

```
depend => { '$(FIRST_MAKEFILE)' => '$(VERSION_FROM)' }
```

How do I make two or more XS files coexist in the same directory?

Sometimes you need to have two and more XS files in the same package. There are three ways: "XSMULTI", separate directories, and bootstrapping one XS from another.

XSMULTI Structure your modules so they are all located under lib, such that "Foo::Bar"

is in lib/Foo/Bar.pm and lib/Foo/Bar.xs, etc. Have your top-level

"WriteMakefile" set the variable "XSMULTI" to a true value.

Er, that's it.

Separate directories

Put each XS files into separate directories, each with their own Makefile.PL.

Make sure each of those Makefile.PLs has the correct "CFLAGS", "INC", "LIBS"

etc. You will need to make sure the top-level Makefile.PL refers to each of

these using "DIR".

Bootstrapping

Let's assume that we have a package "Cool::Foo", which includes "Cool::Foo"

and "Cool::Bar" modules each having a separate XS file. First we use the following Makefile.PL:

```
use ExtUtils::MakeMaker;

WriteMakefile(
    NAME      => 'Cool::Foo',
    VERSION_FROM => 'Foo.pm',
    OBJECT    => q$(O_FILES)/,
    # ... other attrs ...
);
```

Notice the "OBJECT" attribute. MakeMaker generates the following variables in Makefile:

```
# Handy lists of source code files:

XS_FILES= Bar.xs \
    Foo.xs
C_FILES = Bar.c \
    Foo.c
O_FILES = Bar.o \
    Foo.o
```

Therefore we can use the "O_FILES" variable to tell MakeMaker to use these objects into the shared library.

That's pretty much it. Now write Foo.pm and Foo.xs, Bar.pm and Bar.xs, where Foo.pm bootstraps the shared library and Bar.pm simply loading Foo.pm.

The only issue left is to how to bootstrap Bar.xs. This is done from Foo.xs:

```
MODULE = Cool::Foo PACKAGE = Cool::Foo

BOOT:

# boot the second XS file

boot_Cool__Bar(aTHX_ cv);
```

If you have more than two files, this is the place where you should boot extra XS files from.

The following four files sum up all the details discussed so far.

```
Foo.pm:
-----

package Cool::Foo;
```

```

require DynaLoader;

our @ISA = qw(DynaLoader);

our $VERSION = '0.01';

bootstrap Cool::Foo $VERSION;

1;

Bar.pm:
-----

package Cool::Bar;

use Cool::Foo; # bootstraps Bar.xs

1;

Foo.xs:
-----

#include "EXTERN.h"

#include "perl.h"

#include "XSUB.h"

MODULE = Cool::Foo PACKAGE = Cool::Foo

BOOT:

# boot the second XS file

boot_Cool__Bar(aTHX_ cv);

MODULE = Cool::Foo PACKAGE = Cool::Foo PREFIX = cool_foo_

void

cool_foo_perl_rules()

    CODE:

        fprintf(stderr, "Cool::Foo says: Perl Rules\n");

Bar.xs:
-----

#include "EXTERN.h"

#include "perl.h"

#include "XSUB.h"

MODULE = Cool::Bar PACKAGE = Cool::Bar PREFIX = cool_bar_

void

cool_bar_perl_rules()

    CODE:

```

```
printf(stderr, "Cool::Bar says: Perl Rules\n");
```

And of course a very basic test:

```
t/cool.t:  
-----  
use Test;  
  
BEGIN { plan tests => 1 };  
  
use Cool::Foo;  
use Cool::Bar;  
  
Cool::Foo::perl_rules();  
Cool::Bar::perl_rules();  
  
ok 1;
```

This tip has been brought to you by Nick Ing-Simmons and Stas Bekman.

An alternative way to achieve this can be seen in `Gtk2::CodeGen` and `Glib::CodeGen`.

DESIGN

MakeMaker object hierarchy (simplified)

What most people need to know (superclasses on top.)

```
ExtUtils::MM_Any  
|  
ExtUtils::MM_Unix  
|  
ExtUtils::MM_{Current OS}  
|  
ExtUtils::MakeMaker  
|  
MY
```

The object actually used is of the class `MY` which allows you to override bits of `MakeMaker` inside your `Makefile.PL` by declaring `MY::foo()` methods.

MakeMaker object hierarchy (real)

Here's how it really works:

```
ExtUtils::MM_Any  
|  
ExtUtils::MM_Unix
```


\ | | | / / /

| |
MM_Unix |
| |
MM_Any

NOTE: Each direct MM_Unix subclass is also an MM_Any subclass. This is a temporary hack because MM_Unix overrides some MM_Any methods with Unix specific code. It allows the non-Unix modules to see the original MM_Any implementations.

NOTE: Modules like File::Spec and Exporter have been omitted for clarity.

PATCHING

If you have a question you'd like to see added to the FAQ (whether or not you have the answer) please either:

- ? make a pull request on the MakeMaker github repository
- ? raise a issue on the MakeMaker github repository
- ? file an RT ticket
- ? email makemaker@perl.org

AUTHOR

The denizens of makemaker@perl.org.

SEE ALSO

ExtUtils::MakeMaker

perl v5.34.0

2023-11-23

ExtUtils::MakeMaker::FAQ(3perl)