NAME

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

DESCRIPTION

FAQs, tricks and tips for `ExtUtils::MakeMaker`.

Module Installation

How do I keep from installing man pages?

Recent versions of MakeMaker will only install man pages on Unix like operating systems.

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
```

...and then set the PERL5LIB environment variable to point at the blib/lib and blib/arch directories.

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

```
perl Makefile.PL PREFIX=~/tmp LIB=~/tmp/lib/perl
```

And then set PERL5LIB to *~/tmp/lib/perl*. 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.

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 is the official heir apparent to MakeMaker and we encourage people to work on M::B rather than spending time adding features to MakeMaker.

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.

Simplest way to do it automatically is to use your version control system's 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) writing it like so:

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

Every time the file is checked in the $Revision$ will be updated, updating your $VERSION.

In CVS 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.

If branches are involved (ie. $Revision: 1.5.3.4$) its 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 };
```

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()`.

XS

How to 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 reruning 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. One way to go is to put them into separate directories, but sometimes this is not the most suitable solution. The following technique allows you to put two (and more) XS files in the same directory.

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

And of course a very basic test:

```
test.pl:
--------
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.

PATCHING

If you have a question you'd like to see added to the FAQ (whether or not you have the answer) please send it to makemaker@perl.org.

AUTHOR

The denizens of makemaker@perl.org.

SEE ALSO

ExtUtils::MakeMaker