_installed_file_for_module

```
my $file = MM->_installed_file_for_module($module);
```

Return the first installed .pm $file associated with the $module. The one which will show up when you `use $module`.

$module is something like "strict" or "Test::More".

NAME

ExtUtils::MakeMaker - Create a module Makefile

SYNOPSIS

```
use ExtUtils::MakeMaker;

WriteMakefile(
    NAME              => "Foo::Bar",
    VERSION_FROM      => "lib/Foo/Bar.pm",
);
```

DESCRIPTION

This utility is designed to write a Makefile for an extension module from a Makefile.PL. It is based on the Makefile.SH model provided by Andy Dougherty and the perl5-porters.

It splits the task of generating the Makefile into several subroutines that can be individually overridden. Each subroutine returns the text it wishes to have written to the Makefile.

As there are various Make programs with incompatible syntax, which use operating system shells, again with incompatible syntax, it is important for users of this module to know which flavour of Make a Makefile has been written for so they'll use the correct one and won't have to face the possibly bewildering errors resulting from using the wrong one.

On POSIX systems, that program will likely be GNU Make; on Microsoft Windows, it will be either Microsoft NMake, DMake or GNU Make. See the section on the *MAKE* parameter for details.

ExtUtils::MakeMaker (EUMM) is object oriented. Each directory below the current directory that contains a Makefile.PL is treated as a separate object. This makes it possible to write an unlimited number of Makefiles with a single invocation of WriteMakefile().

All inputs to WriteMakefile are Unicode characters, not just octets. EUMM seeks to handle all of these correctly. It is currently still not possible to portably use Unicode characters in module names, because this requires Perl to handle Unicode filenames, which is not yet the case on Windows.

How To Write A Makefile.PL

See *ExtUtils::MakeMaker::Tutorial*.

The long answer is the rest of the manpage :-)

Default Makefile Behaviour

The generated Makefile enables the user of the extension to invoke

```
perl Makefile.PL # optionally "perl Makefile.PL verbose"
make
make test        # optionally set TEST_VERBOSE=1
make install     # See below
```

The Makefile to be produced may be altered by adding arguments of the form KEY=VALUE. E.g.

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

Other interesting targets in the generated Makefile are

```
make config     # to check if the Makefile is up-to-date
make clean      # delete local temp files (Makefile gets renamed)
make realclean  # delete derived files (including ./blib)
make ci         # check in all the files in the MANIFEST file
make dist       # see below the Distribution Support section
```

make test

MakeMaker checks for the existence of a file named *test.pl* in the current directory, and if it exists it executes the script with the proper set of perl `-I` options.

MakeMaker also checks for any files matching glob("t/*.t"). It will execute all matching files in alphabetical order via the *Test::Harness* module with the `-I` switches set correctly.

You can also organize your tests within subdirectories in the *t/* directory. To do so, use the *test* directive in your *Makefile.PL*. For example, if you had tests in:

```
t/foo
t/foo/bar
```

You could tell make to run tests in both of those directories with the following directives:

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

The first will run all test files in all first-level subdirectories and all subdirectories they contain. The second will run tests in only the *t/foo* and *t/foo/bar*.

If you'd like to see the raw output of your tests, set the `TEST_VERBOSE` variable to true.

```
make test TEST_VERBOSE=1
```

If you want to run particular test files, set the `TEST_FILES` variable. It is possible to use globbing with this mechanism.

```
make test TEST_FILES='t/foobar.t t/dagobah*.t'
```

Windows users who are using `nmake` should note that due to a bug in `nmake`, when specifying `TEST_FILES` you must use back-slashes instead of forward-slashes.

```
nmake test TEST_FILES='t\foobar.t t\dagobah*.t'
```

make testdb

A useful variation of the above is the target `testdb`. It runs the test under the Perl debugger (see *perldebug*). If the file *test.pl* exists in the current directory, it is used for the test.

If you want to debug some other testfile, set the `TEST_FILE` variable thusly:

```
make testdb TEST_FILE=t/mytest.t
```

By default the debugger is called using `-d` option to perl. If you want to specify some other option, set the `TESTDB_SW` variable:

```
make testdb TESTDB_SW=-Dx
```

make install

make alone puts all relevant files into directories that are named by the macros INST_LIB, INST_ARCHLIB, INST_SCRIPT, INST_MAN1DIR and INST_MAN3DIR. All these default to something below ./blib if you are *not* building below the perl source directory. If you *are* building below the perl source, INST_LIB and INST_ARCHLIB default to ../../lib, and INST_SCRIPT is not defined.

The *install* target of the generated Makefile copies the files found below each of the INST_* directories to their INSTALL* counterparts. Which counterparts are chosen depends on the setting of INSTALLDIRS according to the following table:

```
                                INSTALLDIRS set to
                          perl         site          vendor

                   PERLPREFIX      SITEPREFIX          VENDORPREFIX
  INST_ARCHLIB     INSTALLARCHLIB  INSTALLSITEARCH     INSTALLVENDORARCH
  INST_LIB         INSTALLPRIVLIB  INSTALLSITELIB      INSTALLVENDORLIB
  INST_BIN         INSTALLBIN      INSTALLSITEBIN      INSTALLVENDORBIN
  INST_SCRIPT      INSTALLSCRIPT   INSTALLSITESCRIPT   INSTALLVENDORSCRIPT
  INST_MAN1DIR     INSTALLMAN1DIR  INSTALLSITEMAN1DIR  INSTALLVENDORMAN1DIR
  INST_MAN3DIR     INSTALLMAN3DIR  INSTALLSITEMAN3DIR  INSTALLVENDORMAN3DIR
```

The INSTALL... macros in turn default to their %Config ($Config{installprivlib}, $Config{installarchlib}, etc.) counterparts.

You can check the values of these variables on your system with

```
perl '-V:install.*'
```

And to check the sequence in which the library directories are searched by perl, run

```
perl -le 'print join $/, @INC'
```

Sometimes older versions of the module you're installing live in other directories in @INC. Because Perl loads the first version of a module it finds, not the newest, you might accidentally get one of these older versions even after installing a brand new version. To delete *all other versions of the module you're installing* (not simply older ones) set the `UNINST` variable.

```
make install UNINST=1
```

INSTALL_BASE

INSTALL_BASE can be passed into Makefile.PL to change where your module will be installed. INSTALL_BASE is more like what everyone else calls "prefix" than PREFIX is.

To have everything installed in your home directory, do the following.

```
# Unix users, INSTALL_BASE=~ works fine
perl Makefile.PL INSTALL_BASE=/path/to/your/home/dir
```

Like PREFIX, it sets several INSTALL* attributes at once. Unlike PREFIX it is easy to predict where the module will end up. The installation pattern looks like this:

```
INSTALLARCHLIB     INSTALL_BASE/lib/perl5/$Config{archname}
INSTALLPRIVLIB     INSTALL_BASE/lib/perl5
INSTALLBIN         INSTALL_BASE/bin
INSTALLSCRIPT      INSTALL_BASE/bin
INSTALLMAN1DIR     INSTALL_BASE/man/man1
```

```
INSTALLMAN3DIR      INSTALL_BASE/man/man3
```

INSTALL_BASE in MakeMaker and `--install_base` in Module::Build (as of 0.28) install to the same location. If you want MakeMaker and Module::Build to install to the same location simply set INSTALL_BASE and `--install_base` to the same location.

INSTALL_BASE was added in 6.31.

PREFIX and LIB attribute

PREFIX and LIB can be used to set several INSTALL* attributes in one go. Here's an example for installing into your home directory.

```
# Unix users, PREFIX=~ works fine
perl Makefile.PL PREFIX=/path/to/your/home/dir
```

This will install all files in the module under your home directory, with man pages and libraries going into an appropriate place (usually ~/man and ~/lib). How the exact location is determined is complicated and depends on how your Perl was configured. INSTALL_BASE works more like what other build systems call "prefix" than PREFIX and we recommend you use that instead.

Another way to specify many INSTALL directories with a single parameter is LIB.

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

This will install the module's architecture-independent files into ~/lib, the architecture-dependent files into ~/lib/$archname.

Note, that in both cases the tilde expansion is done by MakeMaker, not by perl by default, nor by make.

Conflicts between parameters LIB, PREFIX and the various INSTALL* arguments are resolved so that:

- setting LIB overrides any setting of INSTALLPRIVLIB, INSTALLARCHLIB, INSTALLSITELIB, INSTALLSITEARCH (and they are not affected by PREFIX);
- without LIB, setting PREFIX replaces the initial `$Config{prefix}` part of those INSTALL* arguments, even if the latter are explicitly set (but are set to still start with `$Config{prefix}`).

If the user has superuser privileges, and is not working on AFS or relatives, then the defaults for INSTALLPRIVLIB, INSTALLARCHLIB, INSTALLSCRIPT, etc. will be appropriate, and this incantation will be the best:

```
perl Makefile.PL;
make;
make test
make install
```

make install by default writes some documentation of what has been done into the file `$(INSTALLARCHLIB)/perllocal.pod`. This feature can be bypassed by calling make pure_install.

AFS users

will have to specify the installation directories as these most probably have changed since perl itself has been installed. They will have to do this by calling

```
perl Makefile.PL INSTALLSITELIB=/afs/here/today \
```

```
        INSTALLSCRIPT=/afs/there/now INSTALLMAN3DIR=/afs/for/manpages
    make
```

Be careful to repeat this procedure every time you recompile an extension, unless you are sure the AFS installation directories are still valid.

Static Linking of a new Perl Binary

An extension that is built with the above steps is ready to use on systems supporting dynamic loading. On systems that do not support dynamic loading, any newly created extension has to be linked together with the available resources. MakeMaker supports the linking process by creating appropriate targets in the Makefile whenever an extension is built. You can invoke the corresponding section of the makefile with

```
make perl
```

That produces a new perl binary in the current directory with all extensions linked in that can be found in INST_ARCHLIB, SITELIBEXP, and PERL_ARCHLIB. To do that, MakeMaker writes a new Makefile, on UNIX, this is called *Makefile.aperl* (may be system dependent). If you want to force the creation of a new perl, it is recommended that you delete this *Makefile.aperl*, so the directories are searched through for linkable libraries again.

The binary can be installed into the directory where perl normally resides on your machine with

```
make inst_perl
```

To produce a perl binary with a different name than `perl`, either say

```
perl Makefile.PL MAP_TARGET=myperl
make myperl
make inst_perl
```

or say

```
perl Makefile.PL
make myperl MAP_TARGET=myperl
make inst_perl MAP_TARGET=myperl
```

In any case you will be prompted with the correct invocation of the `inst_perl` target that installs the new binary into INSTALLBIN.

make inst_perl by default writes some documentation of what has been done into the file `$(INSTALLARCHLIB)/perllocal.pod`. This can be bypassed by calling make pure_inst_perl.

Warning: the inst_perl: target will most probably overwrite your existing perl binary. Use with care!

Sometimes you might want to build a statically linked perl although your system supports dynamic loading. In this case you may explicitly set the linktype with the invocation of the Makefile.PL or make:

```
perl Makefile.PL LINKTYPE=static    # recommended
```

or

```
make LINKTYPE=static                # works on most systems
```

Determination of Perl Library and Installation Locations

MakeMaker needs to know, or to guess, where certain things are located. Especially INST_LIB and INST_ARCHLIB (where to put the files during the make(1) run), PERL_LIB and PERL_ARCHLIB

(where to read existing modules from), and PERL_INC (header files and `libperl*.*`).

Extensions may be built either using the contents of the perl source directory tree or from the installed perl library. The recommended way is to build extensions after you have run 'make install' on perl itself. You can do that in any directory on your hard disk that is not below the perl source tree. The support for extensions below the ext directory of the perl distribution is only good for the standard extensions that come with perl.

If an extension is being built below the `ext/` directory of the perl source then MakeMaker will set PERL_SRC automatically (e.g., `../..`). If PERL_SRC is defined and the extension is recognized as a standard extension, then other variables default to the following:

```
PERL_INC     = PERL_SRC
PERL_LIB     = PERL_SRC/lib
PERL_ARCHLIB = PERL_SRC/lib
INST_LIB     = PERL_LIB
INST_ARCHLIB = PERL_ARCHLIB
```

If an extension is being built away from the perl source then MakeMaker will leave PERL_SRC undefined and default to using the installed copy of the perl library. The other variables default to the following:

```
PERL_INC     = $archlibexp/CORE
PERL_LIB     = $privlibexp
PERL_ARCHLIB = $archlibexp
INST_LIB     = ./blib/lib
INST_ARCHLIB = ./blib/arch
```

If perl has not yet been installed then PERL_SRC can be defined on the command line as shown in the previous section.

Which architecture dependent directory?

If you don't want to keep the defaults for the INSTALL* macros, MakeMaker helps you to minimize the typing needed: the usual relationship between INSTALLPRIVLIB and INSTALLARCHLIB is determined by Configure at perl compilation time. MakeMaker supports the user who sets INSTALLPRIVLIB. If INSTALLPRIVLIB is set, but INSTALLARCHLIB not, then MakeMaker defaults the latter to be the same subdirectory of INSTALLPRIVLIB as Configure decided for the counterparts in %Config, otherwise it defaults to INSTALLPRIVLIB. The same relationship holds for INSTALLSITELIB and INSTALLSITEARCH.

MakeMaker gives you much more freedom than needed to configure internal variables and get different results. It is worth mentioning that make(1) also lets you configure most of the variables that are used in the Makefile. But in the majority of situations this will not be necessary, and should only be done if the author of a package recommends it (or you know what you're doing).

Using Attributes and Parameters

The following attributes may be specified as arguments to WriteMakefile() or as NAME=VALUE pairs on the command line. Attributes that became available with later versions of MakeMaker are indicated.

In order to maintain portability of attributes with older versions of MakeMaker you may want to use *App::EUMM::Upgrade* with your `Makefile.PL`.

ABSTRACT

One line description of the module. Will be included in PPD file.

ABSTRACT_FROM

Name of the file that contains the package description. MakeMaker looks for a line in the POD matching /^($package\s-\s)(.*)/. This is typically the first line in the "=head1 NAME" section. $2 becomes the abstract.

AUTHOR

Array of strings containing name (and email address) of package author(s). Is used in CPAN Meta files (META.yml or META.json) and PPD (Perl Package Description) files for PPM (Perl Package Manager).

BINARY_LOCATION

Used when creating PPD files for binary packages. It can be set to a full or relative path or URL to the binary archive for a particular architecture. For example:

```
perl Makefile.PL BINARY_LOCATION=x86/Agent.tar.gz
```

builds a PPD package that references a binary of the `Agent` package, located in the `x86` directory relative to the PPD itself.

BUILD_REQUIRES

Available in version 6.5503 and above.

A hash of modules that are needed to build your module but not run it.

This will go into the `build_requires` field of your *META.yml* and the `build` of the `prereqs` field of your *META.json*.

Defaults to `{ "ExtUtils::MakeMaker" => 0 }` if this attribute is not specified.

The format is the same as PREREQ_PM.

C

Ref to array of *.c file names. Initialised from a directory scan and the values portion of the XS attribute hash. This is not currently used by MakeMaker but may be handy in Makefile.PLs.

CCFLAGS

String that will be included in the compiler call command line between the arguments INC and OPTIMIZE.

CONFIG

Arrayref. E.g. [qw(archname manext)] defines ARCHNAME & MANEXT from config.sh. MakeMaker will add to CONFIG the following values anyway: ar cc cccdlflags ccdlflags dlext dlsrc ld lddlflags ldflags libc lib_ext obj_ext ranlib sitelibexp sitearchexp so

CONFIGURE

CODE reference. The subroutine should return a hash reference. The hash may contain further attributes, e.g. {LIBS => ...}, that have to be determined by some evaluation method.

CONFIGURE_REQUIRES

Available in version 6.52 and above.

A hash of modules that are required to run Makefile.PL itself, but not to run your distribution.

This will go into the `configure_requires` field of your *META.yml* and the `configure` of the `prereqs` field of your *META.json*.

Defaults to `{ "ExtUtils::MakeMaker" => 0 }` if this attribute is not specified.

The format is the same as PREREQ_PM.

DEFINE

Something like `"-DHAVE_UNISTD_H"`

DESTDIR

This is the root directory into which the code will be installed. It *prepends itself to the normal prefix*. For example, if your code would normally go into */usr/local/lib/perl* you could set DESTDIR=~/tmp/ and installation would go into *~/tmp/usr/local/lib/perl*.

This is primarily of use for people who repackage Perl modules.

NOTE: Due to the nature of make, it is important that you put the trailing slash on your DESTDIR. *~/tmp/* not *~/tmp*.

DIR

Ref to array of subdirectories containing Makefile.PLs e.g. ['sdbm'] in ext/SDBM_File

DISTNAME

A safe filename for the package.

Defaults to NAME below but with :: replaced with -.

For example, Foo::Bar becomes Foo-Bar.

DISTVNAME

Your name for distributing the package with the version number included. This is used by 'make dist' to name the resulting archive file.

Defaults to DISTNAME-VERSION.

For example, version 1.04 of Foo::Bar becomes Foo-Bar-1.04.

On some OS's where . has special meaning VERSION_SYM may be used in place of VERSION.

DLEXT

Specifies the extension of the module's loadable object. For example:

```
DLEXT => 'unusual_ext', # Default value is $Config{so}
```

NOTE: When using this option to alter the extension of a module's loadable object, it is also necessary that the module's pm file specifies the same change:

```
local $DynaLoader::dl_dlext = 'unusual_ext';
```

DL_FUNCS

Hashref of symbol names for routines to be made available as universal symbols. Each key/value pair consists of the package name and an array of routine names in that package. Used only under AIX, OS/2, VMS and Win32 at present. The routine names supplied will be expanded in the same way as XSUB names are expanded by the XS() macro. Defaults to

```
{"$(NAME)" => ["boot_$(NAME)" ] }
```

e.g.

```
{"RPC" => [qw( boot_rpcb rpcb_gettime getnetconfigent )],
 "NetconfigPtr" => [ 'DESTROY'] }
```

Please see the *ExtUtils::Mksymlists* documentation for more information about the DL_FUNCS, DL_VARS and FUNCLIST attributes.

DL_VARS

Array of symbol names for variables to be made available as universal symbols. Used only under AIX, OS/2, VMS and Win32 at present. Defaults to []. (e.g. [qw(Foo_version Foo_numstreams Foo_tree)])

EXCLUDE_EXT

Array of extension names to exclude when doing a static build. This is ignored if INCLUDE_EXT is present. Consult INCLUDE_EXT for more details. (e.g. [qw(Socket POSIX)])

This attribute may be most useful when specified as a string on the command line: perl Makefile.PL EXCLUDE_EXT='Socket Safe'

EXE_FILES

Ref to array of executable files. The files will be copied to the INST_SCRIPT directory. Make realclean will delete them from there again.

If your executables start with something like #!perl or #!/usr/bin/perl MakeMaker will change this to the path of the perl 'Makefile.PL' was invoked with so the programs will be sure to run properly even if perl is not in /usr/bin/perl.

FIRST_MAKEFILE

The name of the Makefile to be produced. This is used for the second Makefile that will be produced for the MAP_TARGET.

Defaults to 'Makefile' or 'Descrip.MMS' on VMS.

(Note: we couldn't use MAKEFILE because dmake uses this for something else).

FULLPERL

Perl binary able to run this extension, load XS modules, etc...

FULLPERLRUN

Like PERLRUN, except it uses FULLPERL.

FULLPERLRUNINST

Like PERLRUNINST, except it uses FULLPERL.

FUNCLIST

This provides an alternate means to specify function names to be exported from the extension. Its value is a reference to an array of function names to be exported by the extension. These names are passed through unaltered to the linker options file.

H

Ref to array of *.h file names. Similar to C.

IMPORTS

This attribute is used to specify names to be imported into the extension. Takes a hash ref.

It is only used on OS/2 and Win32.

INC

Include file dirs eg: `"-I/usr/5include -I/path/to/inc"`

INCLUDE_EXT

Array of extension names to be included when doing a static build. MakeMaker will normally build with all of the installed extensions when doing a static build, and that is usually the desired behavior. If INCLUDE_EXT is present then MakeMaker will build only with those extensions which are explicitly mentioned. (e.g. [qw(Socket POSIX)])

It is not necessary to mention DynaLoader or the current extension when filling in INCLUDE_EXT. If the INCLUDE_EXT is mentioned but is empty then only DynaLoader and the current extension will be included in the build.

This attribute may be most useful when specified as a string on the command line: perl Makefile.PL INCLUDE_EXT='POSIX Socket Devel::Peek'

INSTALLARCHLIB

Used by 'make install', which copies files from INST_ARCHLIB to this directory if INSTALLDIRS is set to perl.

INSTALLBIN

Directory to install binary files (e.g. tkperl) into if INSTALLDIRS=perl.

INSTALLDIRS

Determines which of the sets of installation directories to choose: perl, site or vendor. Defaults to site.

INSTALLMAN1DIR

INSTALLMAN3DIR

These directories get the man pages at 'make install' time if INSTALLDIRS=perl. Defaults to $Config{installman*dir}.

If set to 'none', no man pages will be installed.

INSTALLPRIVLIB

Used by 'make install', which copies files from INST_LIB to this directory if INSTALLDIRS is set to perl.

Defaults to $Config{installprivlib}.

INSTALLSCRIPT

Used by 'make install' which copies files from INST_SCRIPT to this directory if INSTALLDIRS=perl.

INSTALLSITEARCH

Used by 'make install', which copies files from INST_ARCHLIB to this directory if INSTALLDIRS is set to site (default).

INSTALLSITEBIN

Used by 'make install', which copies files from INST_BIN to this directory if INSTALLDIRS is set to site (default).

INSTALLSITELIB

Used by 'make install', which copies files from INST_LIB to this directory if INSTALLDIRS is set to site (default).

INSTALLSITEMAN1DIR

INSTALLSITEMAN3DIR

These directories get the man pages at 'make install' time if INSTALLDIRS=site (default). Defaults to $(SITEPREFIX)/man/man$(MAN*EXT).

If set to 'none', no man pages will be installed.

INSTALLSITESCRIPT

Used by 'make install' which copies files from INST_SCRIPT to this directory if INSTALLDIRS is set to site (default).

INSTALLVENDORARCH

Used by 'make install', which copies files from INST_ARCHLIB to this directory if INSTALLDIRS is set to vendor.

INSTALLVENDORBIN

Used by 'make install', which copies files from INST_BIN to this directory if INSTALLDIRS is set to vendor.

INSTALLVENDORLIB

Used by 'make install', which copies files from INST_LIB to this directory if INSTALLDIRS is set to vendor.

INSTALLVENDORMAN1DIR

INSTALLVENDORMAN3DIR

These directories get the man pages at 'make install' time if INSTALLDIRS=vendor. Defaults to $(VENDORPREFIX)/man/man$(MAN*EXT).

If set to 'none', no man pages will be installed.

INSTALLVENDORSCRIPT

Used by 'make install' which copies files from INST_SCRIPT to this directory if INSTALLDIRS is set to vendor.

INST_ARCHLIB

Same as INST_LIB for architecture dependent files.

INST_BIN

Directory to put real binary files during 'make'. These will be copied to INSTALLBIN during 'make install'

INST_LIB

Directory where we put library files of this extension while building it.

INST_MAN1DIR

Directory to hold the man pages at 'make' time

INST_MAN3DIR

Directory to hold the man pages at 'make' time

INST_SCRIPT

Directory where executable files should be installed during 'make'. Defaults to "./blib/script", just to have a dummy location during testing. make install will copy the files in INST_SCRIPT to INSTALLSCRIPT.

LD

Program to be used to link libraries for dynamic loading.

Defaults to $Config{ld}.

LDDLFLAGS

Any special flags that might need to be passed to ld to create a shared library suitable for dynamic loading. It is up to the makefile to use it. (See *"lddlflags" in Config*)

Defaults to $Config{lddlflags}.

LDFROM

Defaults to "$(OBJECT)" and is used in the ld command to specify what files to link/load from (also see dynamic_lib below for how to specify ld flags)

LIB

LIB should only be set at `perl Makefile.PL` time but is allowed as a MakeMaker argument. It has the effect of setting both INSTALLPRIVLIB and INSTALLSITELIB to that value regardless any explicit setting of those arguments (or of PREFIX). INSTALLARCHLIB and INSTALLSITEARCH are set to the corresponding architecture subdirectory.

LIBPERL_A

The filename of the perllibrary that will be used together with this extension. Defaults to libperl.a.

LIBS

An anonymous array of alternative library specifications to be searched for (in order) until at least one library is found. E.g.

```
'LIBS' => ["-lgdbm", "-ldbm -lfoo", "-L/path -ldbm.nfs"]
```

Mind, that any element of the array contains a complete set of arguments for the ld command. So do not specify

```
'LIBS' => ["-ltcl", "-ltk", "-lX11"]
```

See ODBM_File/Makefile.PL for an example, where an array is needed. If you specify a scalar as in

```
'LIBS' => "-ltcl -ltk -lX11"
```

MakeMaker will turn it into an array with one element.

LICENSE

Available in version 6.31 and above.

The licensing terms of your distribution. Generally it's "perl_5" for the same license as Perl itself.

See *CPAN::Meta::Spec* for the list of options.

Defaults to "unknown".

LINKTYPE

'static' or 'dynamic' (default unless usedl=undef in config.sh). Should only be used to force static linking (also see linkext below).

MAGICXS

When this is set to `1`, `OBJECT` will be automagically derived from `O_FILES`.

MAKE

Variant of make you intend to run the generated Makefile with. This parameter lets Makefile.PL know what make quirks to account for when generating the Makefile.

MakeMaker also honors the MAKE environment variable. This parameter takes precedence.

Currently the only significant values are 'dmake' and 'nmake' for Windows users, instructing MakeMaker to generate a Makefile in the flavour of DMake ("Dennis Vadura's Make") or Microsoft NMake respectively.

Defaults to $Config{make}, which may go looking for a Make program in your environment.

How are you supposed to know what flavour of Make a Makefile has been generated for if you didn't specify a value explicitly? Search the generated Makefile for the definition of the MAKE variable, which is used to recursively invoke the Make utility. That will tell you what Make you're supposed to invoke the Makefile with.

MAKEAPERL

Boolean which tells MakeMaker that it should include the rules to make a perl. This is handled automatically as a switch by MakeMaker. The user normally does not need it.

MAKEFILE_OLD

When 'make clean' or similar is run, the $(FIRST_MAKEFILE) will be backed up at this location.

Defaults to $(FIRST_MAKEFILE).old or $(FIRST_MAKEFILE)_old on VMS.

MAN1PODS

Hashref of pod-containing files. MakeMaker will default this to all EXE_FILES files that include POD directives. The files listed here will be converted to man pages and installed as was requested at Configure time.

This hash should map POD files (or scripts containing POD) to the man file names under the `blib/man1/` directory, as in the following example:

```
MAN1PODS            => {
  'doc/command.pod'    => 'blib/man1/command.1',
  'scripts/script.pl'  => 'blib/man1/script.1',
}
```

MAN3PODS

Hashref that assigns to *.pm and *.pod files the files into which the manpages are to be written. MakeMaker parses all *.pod and *.pm files for POD directives. Files that contain POD will be the default keys of the MAN3PODS hashref. These will then be converted to man pages during `make` and will be installed during `make install`.

Example similar to MAN1PODS.

MAP_TARGET

If it is intended that a new perl binary be produced, this variable may hold a name for that binary. Defaults to perl

META_ADD

META_MERGE

Available in version 6.46 and above.

A hashref of items to add to the CPAN Meta file (*META.yml* or *META.json*).

They differ in how they behave if they have the same key as the default metadata. META_ADD will override the default value with its own. META_MERGE will merge its value with the default.

Unless you want to override the defaults, prefer META_MERGE so as to get the advantage of any future defaults.

Where prereqs are concerned, if META_MERGE is used, prerequisites are merged with their counterpart `WriteMakefile()` argument (PREREQ_PM is merged into {prereqs}{runtime}{requires}, BUILD_REQUIRES into `{prereqs}{build}{requires}`, CONFIGURE_REQUIRES into `{prereqs}{configure}{requires}`, and TEST_REQUIRES into `{prereqs}{test}{requires}`). When prereqs are specified with META_ADD, the only prerequisites added to the file come from the metadata, not `WriteMakefile()` arguments.

Note that these configuration options are only used for generating *META.yml* and *META.json* -- they are NOT used for *MYMETA.yml* and *MYMETA.json*. Therefore data in these fields should NOT be used for dynamic (user-side) configuration.

By default CPAN Meta specification `1.4` is used. In order to use CPAN Meta specification `2.0`, indicate with `meta-spec` the version you want to use.

```
META_MERGE        => {

  "meta-spec" => { version => 2 },

  resources => {

    repository => {
        type => 'git',
        url =>
'git://github.com/Perl-Toolchain-Gang/ExtUtils-MakeMaker.git',
        web =>
'https://github.com/Perl-Toolchain-Gang/ExtUtils-MakeMaker',
```

```
        },
      },
    },
```

MIN_PERL_VERSION

Available in version 6.48 and above.

The minimum required version of Perl for this distribution.

Either the 5.006001 or the 5.6.1 format is acceptable.

MYEXTLIB

If the extension links to a library that it builds, set this to the name of the library (see SDBM_File)

NAME

The package representing the distribution. For example, `Test::More` or `ExtUtils::MakeMaker`. It will be used to derive information about the distribution such as the *DISTNAME*, installation locations within the Perl library and where XS files will be looked for by default (see *XS*).

`NAME` *must* be a valid Perl package name and it *must* have an associated `.pm` file. For example, `Foo::Bar` is a valid `NAME` and there must exist *Foo/Bar.pm*. Any XS code should be in *Bar.xs* unless stated otherwise.

Your distribution **must** have a `NAME`.

NEEDS_LINKING

MakeMaker will figure out if an extension contains linkable code anywhere down the directory tree, and will set this variable accordingly, but you can speed it up a very little bit if you define this boolean variable yourself.

NOECHO

Command so make does not print the literal commands it's running.

By setting it to an empty string you can generate a Makefile that prints all commands. Mainly used in debugging MakeMaker itself.

Defaults to `@`.

NORECURS

Boolean. Attribute to inhibit descending into subdirectories.

NO_META

When true, suppresses the generation and addition to the MANIFEST of the META.yml and META.json module meta-data files during 'make distdir'.

Defaults to false.

NO_MYMETA

When true, suppresses the generation of MYMETA.yml and MYMETA.json module meta-data files during 'perl Makefile.PL'.

Defaults to false.

NO_PACKLIST

When true, suppresses the writing of `packlist` files for installs.

Defaults to false.

NO_PERLLOCAL

When true, suppresses the appending of installations to `perllocal`.

Defaults to false.

NO_VC

In general, any generated Makefile checks for the current version of MakeMaker and the version the Makefile was built under. If NO_VC is set, the version check is neglected. Do not write this into your Makefile.PL, use it interactively instead.

OBJECT

List of object files, defaults to '$(BASEEXT)$(OBJ_EXT)', but can be a long string or an array containing all object files, e.g. "tkpBind.o tkpButton.o tkpCanvas.o" or ["tkpBind.o", "tkpButton.o", "tkpCanvas.o"]

(Where BASEEXT is the last component of NAME, and OBJ_EXT is $Config{obj_ext}.)

OPTIMIZE

Defaults to `-O`. Set it to `-g` to turn debugging on. The flag is passed to subdirectory makes.

PERL

Perl binary for tasks that can be done by miniperl. If it contains spaces or other shell metacharacters, it needs to be quoted in a way that protects them, since this value is intended to be inserted in a shell command line in the Makefile. E.g.:

```
# Perl executable lives in "C:/Program Files/Perl/bin"
# Normally you don't need to set this yourself!
$ perl Makefile.PL PERL='"C:/Program Files/Perl/bin/perl.exe" -w'
```

PERL_CORE

Set only when MakeMaker is building the extensions of the Perl core distribution.

PERLMAINCC

The call to the program that is able to compile perlmain.c. Defaults to $(CC).

PERL_ARCHLIB

Same as for PERL_LIB, but for architecture dependent files.

Used only when MakeMaker is building the extensions of the Perl core distribution (because normally $(PERL_ARCHLIB) is automatically in @INC, and adding it would get in the way of PERL5LIB).

PERL_LIB

Directory containing the Perl library to use.

Used only when MakeMaker is building the extensions of the Perl core distribution (because normally $(PERL_LIB) is automatically in @INC, and adding it would get in the way of PERL5LIB).

PERL_MALLOC_OK

defaults to 0. Should be set to TRUE if the extension can work with the memory allocation routines substituted by the Perl malloc() subsystem. This should be applicable to most extensions with exceptions of those

- with bugs in memory allocations which are caught by Perl's malloc();
- which interact with the memory allocator in other ways than via malloc(), realloc(), free(), calloc(), sbrk() and brk();
- which rely on special alignment which is not provided by Perl's malloc().

NOTE. Neglecting to set this flag in *any one* of the loaded extension nullifies many advantages of

Perl's malloc(), such as better usage of system resources, error detection, memory usage reporting, catchable failure of memory allocations, etc.

PERLPREFIX

Directory under which core modules are to be installed.

Defaults to $Config{installprefixexp}, falling back to $Config{installprefix}, $Config{prefixexp} or $Config{prefix} should $Config{installprefixexp} not exist.

Overridden by PREFIX.

PERLRUN

Use this instead of $(PERL) when you wish to run perl. It will set up extra necessary flags for you.

PERLRUNINST

Use this instead of $(PERL) when you wish to run perl to work with modules. It will add things like -I$(INST_ARCH) and other necessary flags so perl can see the modules you're about to install.

PERL_SRC

Directory containing the Perl source code (use of this should be avoided, it may be undefined)

PERM_DIR

Desired permission for directories. Defaults to 755.

PERM_RW

Desired permission for read/writable files. Defaults to 644.

PERM_RWX

Desired permission for executable files. Defaults to 755.

PL_FILES

MakeMaker can run programs to generate files for you at build time. By default any file named *.PL (except Makefile.PL and Build.PL) in the top level directory will be assumed to be a Perl program and run passing its own basename in as an argument. This basename is actually a build target, and there is an intention, but not a requirement, that the *.PL file make the file passed to to as an argument. For example...

```
perl foo.PL foo
```

This behavior can be overridden by supplying your own set of files to search. PL_FILES accepts a hash ref, the key being the file to run and the value is passed in as the first argument when the PL file is run.

```
PL_FILES => {'bin/foobar.PL' => 'bin/foobar'}

PL_FILES => {'foo.PL' => 'foo.c'}
```

Would run bin/foobar.PL like this:

```
perl bin/foobar.PL bin/foobar
```

If multiple files from one program are desired an array ref can be used.

```
PL_FILES => {'bin/foobar.PL' => [qw(bin/foobar1 bin/foobar2)]}
```

In this case the program will be run multiple times using each target file.

```
perl bin/foobar.PL bin/foobar1
perl bin/foobar.PL bin/foobar2
```

PL files are normally run **after** pm_to_blib and include INST_LIB and INST_ARCH in their `@INC`,

so the just built modules can be accessed... unless the PL file is making a module (or anything else in PM) in which case it is run **before** pm_to_blib and does not include INST_LIB and INST_ARCH in its `@INC`. This apparently odd behavior is there for backwards compatibility (and it's somewhat DWIM). The argument passed to the .PL is set up as a target to build in the Makefile. In other sections such as `postamble` you can specify a dependency on the filename/argument that the .PL is supposed (or will have, now that that is is a dependency) to generate. Note the file to be generated will still be generated and the .PL will still run even without an explicit dependency created by you, since the `all` target still depends on running all eligible to run.PL files.

PM

Hashref of .pm files and *.pl files to be installed. e.g.

```
{'name_of_file.pm' => '$(INST_LIB)/install_as.pm'}
```

By default this will include *.pm and *.pl and the files found in the PMLIBDIRS directories. Defining PM in the Makefile.PL will override PMLIBDIRS.

PMLIBDIRS

Ref to array of subdirectories containing library files. Defaults to ['lib', $(BASEEXT)]. The directories will be scanned and *any* files they contain will be installed in the corresponding location in the library. A libscan() method can be used to alter the behaviour. Defining PM in the Makefile.PL will override PMLIBDIRS.

(Where BASEEXT is the last component of NAME.)

PM_FILTER

A filter program, in the traditional Unix sense (input from stdin, output to stdout) that is passed on each .pm file during the build (in the pm_to_blib() phase). It is empty by default, meaning no filtering is done. You could use:

```
PM_FILTER => 'perl -ne "print unless /^\\#/"',
```

to remove all the leading comments on the fly during the build. In order to be as portable as possible, please consider using a Perl one-liner rather than Unix (or other) utilities, as above. The # is escaped for the Makefile, since what is going to be generated will then be:

```
PM_FILTER = perl -ne "print unless /^\#/"
```

Without the \ before the #, we'd have the start of a Makefile comment, and the macro would be incorrectly defined.

You will almost certainly be better off using the `PL_FILES` system, instead. See above, or the *ExtUtils::MakeMaker::FAQ* entry.

POLLUTE

Release 5.005 grandfathered old global symbol names by providing preprocessor macros for extension source compatibility. As of release 5.6, these preprocessor definitions are not available by default. The POLLUTE flag specifies that the old names should still be defined:

```
perl Makefile.PL POLLUTE=1
```

Please inform the module author if this is necessary to successfully install a module under 5.6 or later.

PPM_INSTALL_EXEC

Name of the executable used to run `PPM_INSTALL_SCRIPT` below. (e.g. perl)

PPM_INSTALL_SCRIPT

Name of the script that gets executed by the Perl Package Manager after the installation of a

PPM_INSTALL_EXEC

Name of the executable used to run `PPM_UNINSTALL_SCRIPT` below. (e.g. perl)

PPM_UNINSTALL_SCRIPT

Name of the script that gets executed by the Perl Package Manager before the removal of a package.

PREFIX

This overrides all the default install locations. Man pages, libraries, scripts, etc... MakeMaker will try to make an educated guess about where to place things under the new PREFIX based on your Config defaults. Failing that, it will fall back to a structure which should be sensible for your platform.

If you specify LIB or any INSTALL* variables they will not be affected by the PREFIX.

PREREQ_FATAL

Bool. If this parameter is true, failing to have the required modules (or the right versions thereof) will be fatal. `perl Makefile.PL` will `die` instead of simply informing the user of the missing dependencies.

It is *extremely* rare to have to use `PREREQ_FATAL`. Its use by module authors is *strongly discouraged* and should never be used lightly.

For dependencies that are required in order to run `Makefile.PL`, see `CONFIGURE_REQUIRES`.

Module installation tools have ways of resolving unmet dependencies but to do that they need a *Makefile*. Using `PREREQ_FATAL` breaks this. That's bad.

Assuming you have good test coverage, your tests should fail with missing dependencies informing the user more strongly that something is wrong. You can write a *t/00compile.t* test which will simply check that your code compiles and stop "make test" prematurely if it doesn't. See *"BAIL_OUT" in Test::More* for more details.

PREREQ_PM

A hash of modules that are needed to run your module. The keys are the module names ie. Test::More, and the minimum version is the value. If the required version number is 0 any version will do. The versions given may be a Perl v-string (see *version*) or a range (see *CPAN::Meta::Requirements*).

This will go into the `requires` field of your *META.yml* and the `runtime` of the `prereqs` field of your *META.json*.

```
    PREREQ_PM => {
        # Require Test::More at least 0.47
        "Test::More" => "0.47",

        # Require any version of Acme::Buffy
        "Acme::Buffy" => 0,
    }
```

PREREQ_PRINT

Bool. If this parameter is true, the prerequisites will be printed to stdout and MakeMaker will exit. The output format is an evalable hash ref.

```
  $PREREQ_PM = {
                 'A::B' => Vers1,
                 'C::D' => Vers2,
                 ...
               };
```

If a distribution defines a minimal required perl version, this is added to the output as an additional line of the form:

```
$MIN_PERL_VERSION = '5.008001';
```

If BUILD_REQUIRES is not empty, it will be dumped as $BUILD_REQUIRES hashref.

PRINT_PREREQ

RedHatism for `PREREQ_PRINT`. The output format is different, though:

```
perl(A::B)>=Vers1 perl(C::D)>=Vers2 ...
```

A minimal required perl version, if present, will look like this:

```
perl(perl)>=5.008001
```

SITEPREFIX

Like PERLPREFIX, but only for the site install locations.

Defaults to $Config{siteprefixexp}. Perls prior to 5.6.0 didn't have an explicit siteprefix in the Config. In those cases $Config{installprefix} will be used.

Overridable by PREFIX

SIGN

When true, perform the generation and addition to the MANIFEST of the SIGNATURE file in the distdir during 'make distdir', via 'cpansign -s'.

Note that you need to install the Module::Signature module to perform this operation.

Defaults to false.

SKIP

Arrayref. E.g. [qw(name1 name2)] skip (do not write) sections of the Makefile. Caution! Do not use the SKIP attribute for the negligible speedup. It may seriously damage the resulting Makefile. Only use it if you really need it.

TEST_REQUIRES

Available in version 6.64 and above.

A hash of modules that are needed to test your module but not run or build it.

This will go into the `build_requires` field of your *META.yml* and the `test` of the `prereqs` field of your *META.json*.

The format is the same as PREREQ_PM.

TYPEMAPS

Ref to array of typemap file names. Use this when the typemaps are in some directory other than the current directory or when they are not named **typemap**. The last typemap in the list takes precedence. A typemap in the current directory has highest precedence, even if it isn't listed in TYPEMAPS. The default system typemap has lowest precedence.

VENDORPREFIX

Like PERLPREFIX, but only for the vendor install locations.

Defaults to $Config{vendorprefixexp}.

Overridable by PREFIX

VERBINST

If true, make install will be verbose

VERSION

Your version number for distributing the package. This defaults to 0.1.

VERSION_FROM

Instead of specifying the VERSION in the Makefile.PL you can let MakeMaker parse a file to determine the version number. The parsing routine requires that the file named by VERSION_FROM contains one single line to compute the version number. The first line in the file that contains something like a $VERSION assignment or `package Name VERSION` will be used. The following lines will be parsed o.k.:

```
    # Good
    package Foo::Bar 1.23                    # 1.23
    $VERSION   = '1.00';                     # 1.00
    *VERSION   = \'1.01';                    # 1.01
    ($VERSION) = q$Revision$ =~ /(\d+)/g;    # The digits in
$Revision$
    $FOO::VERSION = '1.10';                  # 1.10
    *FOO::VERSION = \'1.11';                 # 1.11
```

but these will fail:

```
    # Bad
    my $VERSION         = '1.01';
    local $VERSION      = '1.02';
    local $FOO::VERSION = '1.30';
```

(Putting `my` or `local` on the preceding line will work o.k.)

"Version strings" are incompatible and should not be used.

```
    # Bad
    $VERSION = 1.2.3;
    $VERSION = v1.2.3;
```

version objects are fine. As of MakeMaker 6.35 version.pm will be automatically loaded, but you must declare the dependency on version.pm. For compatibility with older MakeMaker you should load on the same line as $VERSION is declared.

```
    # All on one line
    use version; our $VERSION = qv(1.2.3);
```

The file named in VERSION_FROM is not added as a dependency to Makefile. This is not really correct, but it would be a major pain during development to have to rewrite the Makefile for any smallish change in that file. If you want to make sure that the Makefile contains the correct VERSION macro after any change of the file, you would have to do something like

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

See attribute `depend` below.

VERSION_SYM

A sanitized VERSION with . replaced by _. For places where . has special meaning (some filesystems, RCS labels, etc...)

XS

Hashref of .xs files. MakeMaker will default this. e.g.

```
  {'name_of_file.xs' => 'name_of_file.c'}
```

The .c files will automatically be included in the list of files deleted by a make clean.

XSBUILD

Hashref with options controlling the operation of XSMULTI:

```
  {
    xs => {
        all => {
            # options applying to all .xs files for this distribution
        },
        'lib/Class/Name/File' => { # specifically for this file
            DEFINE => '-Dfunktastic', # defines for only this file
            INC => "-I$funkyliblocation", # include flags for only this
file
            # OBJECT => 'lib/Class/Name/File$(OBJ_EXT)', # default
            LDFROM => "lib/Class/Name/File\$(OBJ_EXT)
$otherfile\$(OBJ_EXT)", # what's linked
        },
    },
  }
```

Note xs is the file-extension. More possibilities may arise in the future. Note that object names are specified without their XS extension.

LDFROM defaults to the same as OBJECT. OBJECT defaults to, for XSMULTI, just the XS filename with the extension replaced with the compiler-specific object-file extension.

The distinction between OBJECT and LDFROM: OBJECT is the make target, so make will try to build it. However, LDFROM is what will actually be linked together to make the shared object or static library (SO/SL), so if you override it, make sure it includes what you want to make the final SO/SL, almost certainly including the XS basename with $(OBJ_EXT) appended.

XSMULTI

When this is set to 1, multiple XS files may be placed under *lib/* next to their corresponding *.pm files (this is essential for compiling with the correct VERSION values). This feature should be considered experimental, and details of it may change.

This feature was inspired by, and small portions of code copied from, *ExtUtils::MakeMaker::BigHelper*. Hopefully this feature will render that module mainly obsolete.

XSOPT

String of options to pass to xsubpp. This might include -C++ or -extern. Do not include typemaps here; the TYPEMAP parameter exists for that purpose.

XSPROTOARG

May be set to -prototypes, -noprototypes or the empty string. The empty string is equivalent to the xsubpp default, or -noprototypes. See the xsubpp documentation for details. MakeMaker defaults to the empty string.

XS_VERSION

Your version number for the .xs file of this package. This defaults to the value of the VERSION attribute.

Additional lowercase attributes

can be used to pass parameters to the methods which implement that part of the Makefile. Parameters are specified as a hash ref but are passed to the method as a hash.

clean

```
    {FILES => "*.xyz foo"}
```

depend

```
{ANY_TARGET => ANY_DEPENDENCY, ...}
```

(ANY_TARGET must not be given a double-colon rule by MakeMaker.)

dist

```
{TARFLAGS => 'cvfF', COMPRESS => 'gzip', SUFFIX => '.gz',
SHAR => 'shar -m', DIST_CP => 'ln', ZIP => '/bin/zip',
ZIPFLAGS => '-rl', DIST_DEFAULT => 'private tardist' }
```

If you specify COMPRESS, then SUFFIX should also be altered, as it is needed to tell make the target file of the compression. Setting DIST_CP to ln can be useful, if you need to preserve the timestamps on your files. DIST_CP can take the values 'cp', which copies the file, 'ln', which links the file, and 'best' which copies symbolic links and links the rest. Default is 'best'.

dynamic_lib

```
{ARMAYBE => 'ar', OTHERLDFLAGS => '...', INST_DYNAMIC_DEP => '...'}
```

linkext

```
{LINKTYPE => 'static', 'dynamic' or ''}
```

NB: Extensions that have nothing but *.pm files had to say

```
{LINKTYPE => ''}
```

with Pre-5.0 MakeMakers. Since version 5.00 of MakeMaker such a line can be deleted safely. MakeMaker recognizes when there's nothing to be linked.

macro

```
{ANY_MACRO => ANY_VALUE, ...}
```

postamble

Anything put here will be passed to MY::postamble() if you have one.

realclean

```
{FILES => '$(INST_ARCHAUTODIR)/*.xyz'}
```

test

Specify the targets for testing.

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

RECURSIVE_TEST_FILES can be used to include all directories recursively under `t` that contain `.t` files. It will be ignored if you provide your own TESTS attribute, defaults to false.

```
{RECURSIVE_TEST_FILES=>1}
```

tool_autosplit

```
{MAXLEN => 8}
```

Overriding MakeMaker Methods

If you cannot achieve the desired Makefile behaviour by specifying attributes you may define private subroutines in the Makefile.PL. Each subroutine returns the text it wishes to have written to the Makefile. To override a section of the Makefile you can either say:

```
sub MY::c_o { "new literal text" }
```

or you can edit the default by saying something like:

```
package MY; # so that "SUPER" works right
sub c_o {
    my $inherited = shift->SUPER::c_o(@_);
    $inherited =~ s/old text/new text/;
    $inherited;
}
```

If you are running experiments with embedding perl as a library into other applications, you might find MakeMaker is not sufficient. You'd better have a look at ExtUtils::Embed which is a collection of utilities for embedding.

If you still need a different solution, try to develop another subroutine that fits your needs and submit the diffs to `makemaker@perl.org`

For a complete description of all MakeMaker methods see *ExtUtils::MM_Unix*.

Here is a simple example of how to add a new target to the generated Makefile:

```
sub MY::postamble {
    return <<'MAKE_FRAG';
$(MYEXTLIB): sdbm/Makefile
        cd sdbm && $(MAKE) all

MAKE_FRAG
}
```

The End Of Cargo Cult Programming

WriteMakefile() now does some basic sanity checks on its parameters to protect against typos and malformatted values. This means some things which happened to work in the past will now throw warnings and possibly produce internal errors.

Some of the most common mistakes:

`MAN3PODS => ' '`

This is commonly used to suppress the creation of man pages. MAN3PODS takes a hash ref not a string, but the above worked by accident in old versions of MakeMaker.

The correct code is `MAN3PODS => { }`.

Hintsfile support

MakeMaker.pm uses the architecture-specific information from Config.pm. In addition it evaluates architecture specific hints files in a `hints/` directory. The hints files are expected to be named like their counterparts in `PERL_SRC/hints`, but with an `.pl` file name extension (eg. `next_3_2.pl`). They are simply `eval`ed by MakeMaker within the WriteMakefile() subroutine, and can be used to execute commands as well as to include special variables. The rules which hintsfile is chosen are the same as in Configure.

The hintsfile is eval()ed immediately after the arguments given to WriteMakefile are stuffed into a hash reference $self but before this reference becomes blessed. So if you want to do the equivalent to override or create an attribute you would say something like

```
$self->{LIBS} = ['-ldbm -lucb -lc'];
```

Distribution Support

For authors of extensions MakeMaker provides several Makefile targets. Most of the support comes from the ExtUtils::Manifest module, where additional documentation can be found.

make distcheck

reports which files are below the build directory but not in the MANIFEST file and vice versa. (See ExtUtils::Manifest::fullcheck() for details)

make skipcheck

reports which files are skipped due to the entries in the `MANIFEST.SKIP` file (See ExtUtils::Manifest::skipcheck() for details)

make distclean

does a realclean first and then the distcheck. Note that this is not needed to build a new distribution as long as you are sure that the MANIFEST file is ok.

make veryclean

does a realclean first and then removes backup files such as `*~`, `*.bak`, `*.old` and `*.orig`

make manifest

rewrites the MANIFEST file, adding all remaining files found (See ExtUtils::Manifest::mkmanifest() for details)

make distdir

Copies all the files that are in the MANIFEST file to a newly created directory with the name `$(DISTNAME)-$(VERSION)`. If that directory exists, it will be removed first.

Additionally, it will create META.yml and META.json module meta-data file in the distdir and add this to the distdir's MANIFEST. You can shut this behavior off with the NO_META flag.

make disttest

Makes a distdir first, and runs a `perl Makefile.PL`, a make, and a make test in that directory.

make tardist

First does a distdir. Then a command $(PREOP) which defaults to a null command, followed by $(TO_UNIX), which defaults to a null command under UNIX, and will convert files in distribution directory to UNIX format otherwise. Next it runs `tar` on that directory into a tarfile and deletes the directory. Finishes with a command $(POSTOP) which defaults to a null command.

make dist

Defaults to $(DIST_DEFAULT) which in turn defaults to tardist.

make uutardist

Runs a tardist first and uuencodes the tarfile.

make shdist

First does a distdir. Then a command $(PREOP) which defaults to a null command. Next it runs `shar` on that directory into a sharfile and deletes the intermediate directory again. Finishes with a command $(POSTOP) which defaults to a null command. Note: For shdist to work properly a `shar` program that can handle directories is mandatory.

make zipdist

First does a distdir. Then a command $(PREOP) which defaults to a null command. Runs `$(ZIP) $(ZIPFLAGS)` on that directory into a zipfile. Then deletes that directory. Finishes

with a command $(POSTOP) which defaults to a null command.

make ci

Does a $(CI) and a $(RCS_LABEL) on all files in the MANIFEST file.

Customization of the dist targets can be done by specifying a hash reference to the dist attribute of the WriteMakefile call. The following parameters are recognized:

```
CI           ('ci -u')
COMPRESS     ('gzip --best')
POSTOP       ('@ :')
PREOP        ('@ :')
TO_UNIX      (depends on the system)
RCS_LABEL    ('rcs -q -Nv$(VERSION_SYM):')
SHAR         ('shar')
SUFFIX       ('.gz')
TAR          ('tar')
TARFLAGS     ('cvf')
ZIP          ('zip')
ZIPFLAGS     ('-r')
```

An example:

```
WriteMakefile(
    ...other options...
    dist => {
        COMPRESS => "bzip2",
        SUFFIX   => ".bz2"
    }
);
```

Module Meta-Data (META and MYMETA)

Long plaguing users of MakeMaker based modules has been the problem of getting basic information about the module out of the sources *without* running the *Makefile.PL* and doing a bunch of messy heuristics on the resulting *Makefile*. Over the years, it has become standard to keep this information in one or more CPAN Meta files distributed with each distribution.

The original format of CPAN Meta files was *YAML* and the corresponding file was called *META.yml*. In 2010, version 2 of the *CPAN::Meta::Spec* was released, which mandates JSON format for the metadata in order to overcome certain compatibility issues between YAML serializers and to avoid breaking older clients unable to handle a new version of the spec. The *CPAN::Meta* library is now standard for accessing old and new-style Meta files.

If *CPAN::Meta* is installed, MakeMaker will automatically generate *META.json* and *META.yml* files for you and add them to your *MANIFEST* as part of the 'distdir' target (and thus the 'dist' target). This is intended to seamlessly and rapidly populate CPAN with module meta-data. If you wish to shut this feature off, set the `NO_META` `WriteMakefile()` flag to true.

At the 2008 QA Hackathon in Oslo, Perl module toolchain maintainers agreed to use the CPAN Meta format to communicate post-configuration requirements between toolchain components. These files, *MYMETA.json* and *MYMETA.yml*, are generated when *Makefile.PL* generates a *Makefile* (if *CPAN::Meta* is installed). Clients like *CPAN* or *CPANPLUS* will read these files to see what prerequisites must be fulfilled before building or testing the distribution. If you wish to shut this feature off, set the `NO_MYMETA` `WriteMakeFile()` flag to true.

Disabling an extension

If some events detected in *Makefile.PL* imply that there is no way to create the Module, but this is a normal state of things, then you can create a *Makefile* which does nothing, but succeeds on all the "usual" build targets. To do so, use

```
use ExtUtils::MakeMaker qw(WriteEmptyMakefile);
WriteEmptyMakefile();
```

instead of WriteMakefile().

This may be useful if other modules expect this module to be *built* OK, as opposed to *work* OK (say, this system-dependent module builds in a subdirectory of some other distribution, or is listed as a dependency in a CPAN::Bundle, but the functionality is supported by different means on the current architecture).

Other Handy Functions

prompt

```
my $value = prompt($message);
my $value = prompt($message, $default);
```

The `prompt()` function provides an easy way to request user input used to write a makefile. It displays the $message as a prompt for input. If a $default is provided it will be used as a default. The function returns the $value selected by the user.

If `prompt()` detects that it is not running interactively and there is nothing on STDIN or if the PERL_MM_USE_DEFAULT environment variable is set to true, the $default will be used without prompting. This prevents automated processes from blocking on user input.

If no $default is provided an empty string will be used instead.

Supported versions of Perl

Please note that while this module works on Perl 5.6, it is no longer being routinely tested on 5.6 - the earliest Perl version being routinely tested, and expressly supported, is 5.8.1. However, patches to repair any breakage on 5.6 are still being accepted.

ENVIRONMENT

PERL_MM_OPT

Command line options used by `MakeMaker->new()`, and thus by `WriteMakefile()`. The string is split as the shell would, and the result is processed before any actual command line arguments are processed.

```
  PERL_MM_OPT='CCFLAGS="-Wl,-rpath -Wl,/foo/bar/lib" LIBS="-lwibble
-lwobble"'
```

PERL_MM_USE_DEFAULT

If set to a true value then MakeMaker's prompt function will always return the default without waiting for user input.

PERL_CORE

Same as the PERL_CORE parameter. The parameter overrides this.

SEE ALSO

Module::Build is a pure-Perl alternative to MakeMaker which does not rely on make or any other external utility. It is easier to extend to suit your needs.

Module::Install is a wrapper around MakeMaker which adds features not normally available.

ExtUtils::ModuleMaker and *Module::Starter* are both modules to help you setup your distribution.

CPAN::Meta and *CPAN::Meta::Spec* explain CPAN Meta files in detail.

File::ShareDir::Install makes it easy to install static, sometimes also referred to as 'shared' files. *File::ShareDir* helps accessing the shared files after installation.

Dist::Zilla makes it easy for the module author to create MakeMaker-based distributions with lots of bells and whistles.

AUTHORS

Andy Dougherty `doughera@lafayette.edu`, Andreas König `andreas.koenig@mind.de`, Tim Bunce `timb@cpan.org`. VMS support by Charles Bailey `bailey@newman.upenn.edu`. OS/2 support by Ilya Zakharevich `ilya@math.ohio-state.edu`.

Currently maintained by Michael G Schwern `schwern@pobox.com`

Send patches and ideas to `makemaker@perl.org`.

Send bug reports via http://rt.cpan.org/. Please send your generated Makefile along with your report.

For more up-to-date information, see *https://metacpan.org/release/ExtUtils-MakeMaker*.

Repository available at *https://github.com/Perl-Toolchain-Gang/ExtUtils-MakeMaker*.

LICENSE