NAME

Module::Build - Build and install Perl modules

SYNOPSIS

Standard process for building & installing modules:

```
perl Build.PL
./Build
./Build test
./Build install
```

Or, if you're on a platform (like DOS or Windows) that doesn't require the "./" notation, you can do this:

```
perl Build.PL
Build
Build test
Build install
```

DESCRIPTION

`Module::Build` is a system for building, testing, and installing Perl modules. It is meant to be an alternative to `ExtUtils::MakeMaker` Developers may alter the behavior of the module through subclassing in a much more straightforward way than with `MakeMaker`. It also does not require a `make` on your system - most of the `Module::Build` code is pure-perl and written in a very cross-platform way.

See "MOTIVATIONS" for more comparisons between `ExtUtils::MakeMaker` and `Module::Build`

To install `Module::Build` and any other module that uses `Module::Build` for its installation process, do the following:

```
perl Build.PL # 'Build.PL' script creates the 'Build' script
./Build # Need ./ to ensure we're using this "Build" script
./Build test # and not another one that happens to be in the PATH
./Build install
```

This illustrates initial configuration and the running of three 'actions'. In this case the actions run are 'build' (the default action), 'test', and 'install'. Other actions defined so far include:

```
build manifest
clean manifest_skip
code manpages
config_data pardist
diff ppd
dist ppmdist
distcheck prereq_data
distclean prereq_report
distdir pure_install
distinstall realclean
distmeta retest
distsign skipcheck
disttest test
docs testall
fakeinstall testcover
help testdb
html testpod
install testpodcoverage
installdeps versioninstall
```

You can run the 'help' action for a complete list of actions.

GUIDE TO DOCUMENTATION

The documentation for `Module::Build` is broken up into sections:

General Usage (Module::Build)
: This is the document you are currently reading. It describes basic usage and background information. Its main purpose is to assist the user who wants to learn how to invoke and control `Module::Build` scripts at the command line.

Authoring Reference (Module::Build::Authoring)
: This document describes the structure and organization of `Module::Build` and the relevant concepts needed by authors who are writing *Build.PL* scripts for a distribution or controlling `Module::Build` processes programmatically.

API Reference (Module::Build::API)
: This is a reference to the `Module::Build` API.

Cookbook (Module::Build::Cookbook)
: This document demonstrates how to accomplish many common tasks. It covers general command line usage and authoring of *Build.PL* scripts. Includes working examples.

ACTIONS

There are some general principles at work here. First, each task when building a module is called an ''action''. These actions are listed above; they correspond to the building, testing, installing, packaging, etc., tasks.

Second, arguments are processed in a very systematic way. Arguments are always key=value pairs. They may be specified at `perl Build.PL` time (i.e. `perl Build.PL destdir=/my/secret/place`), in which case their values last for the lifetime of the `Build` script. They may also be specified when executing a particular action (i.e. `Build test verbose=1`), in which case their values last only for the lifetime of that command. Per-action command line parameters take precedence over parameters specified at `perl Build.PL` time.

The build process also relies heavily on the `Config.pm` module. If the user wishes to override any of the values in `Config.pm`, she may specify them like so:

```
perl Build.PL --config cc=gcc --config ld=gcc
```

The following build actions are provided by default.

build
: [version 0.01]

  If you run the `Build` script without any arguments, it runs the `build` action, which in turn runs the `code` and `docs` actions.

  This is analogous to the `MakeMaker` *make all* target.

clean
: [version 0.01]

  This action will clean up any files that the build process may have created, including the `blib/` directory (but not including the `_build/` directory and the `Build` script itself).

code
: [version 0.20]

  This action builds your code base.

  By default it just creates a `blib/` directory and copies any `.pm` and `.pod` files from your `lib/` directory into the `blib/` directory. It also compiles any `.xs` files from `lib/` and places them in `blib/`. Of course, you need a working C compiler (probably the same one that built perl itself) for the compilation to work properly.

  The `code` action also runs any `.PL` files in your *lib/* directory. Typically these create other files, named the same but without the `.PL` ending. For example, a file *lib/Foo/Bar.pm.PL*

could create the file *lib/Foo/Bar.pm*. The `.PL` files are processed first, so any `.pm` files (or other kinds that we deal with) will get copied correctly.

config_data
[version 0.26]

...

diff [version 0.14]

This action will compare the files about to be installed with their installed counterparts. For .pm and .pod files, a diff will be shown (this currently requires a 'diff' program to be in your PATH). For other files like compiled binary files, we simply report whether they differ.

A `flags` parameter may be passed to the action, which will be passed to the 'diff' program. Consult your 'diff' documentation for the parameters it will accept - a good one is `-u`:

```
./Build diff flags=-u
```

dist
[version 0.02]

This action is helpful for module authors who want to package up their module for source distribution through a medium like CPAN. It will create a tarball of the files listed in *MANIFEST* and compress the tarball using GZIP compression.

By default, this action will use the `Archive::Tar` module. However, you can force it to use binary "tar" and "gzip" executables by supplying an explicit `tar` (and optional `gzip`) parameter:

```
./Build dist --tar C:\path\to\tar.exe --gzip C:\path\to\zip.exe
```

distcheck
[version 0.05]

Reports which files are in the build directory but not in the *MANIFEST* file, and vice versa. (See manifest for details.)

distclean
[version 0.05]

Performs the 'realclean' action and then the 'distcheck' action.

distdir
[version 0.05]

Creates a "distribution directory" named `$dist_name-$dist_version` (if that directory already exists, it will be removed first), then copies all the files listed in the *MANIFEST* file to that directory. This directory is what the distribution tarball is created from.

distinstall
[version 0.37]

Performs the 'distdir' action, then switches into that directory and runs a `perl Build.PL`, followed by the 'build' and 'install' actions in that directory. Use PERL_MB_OPT or *.modulebuildrc* to set options that should be applied during subprocesses

distmeta
[version 0.21]

Creates the *META.yml* file that describes the distribution.

META.yml is a file containing various bits of *metadata* about the distribution. The metadata includes the distribution name, version, abstract, prerequisites, license, and various other data about the distribution. This file is created as *META.yml* in a simplified YAML format.

META.yml file must also be listed in *MANIFEST* - if it's not, a warning will be issued.

The current version of the *META.yml* specification can be found on CPAN as CPAN::Meta::Spec.

distsign
[version 0.16]

Uses `Module::Signature` to create a SIGNATURE file for your distribution, and adds the SIGNATURE file to the distribution's MANIFEST.

disttest
[version 0.05]

Performs the 'distdir' action, then switches into that directory and runs a `perl Build.PL`, followed by the 'build' and 'test' actions in that directory. Use PERL_MB_OPT or *.modulebuildrc* to set options that should be applied during subprocesses

docs
[version 0.20]

This will generate documentation (e.g. Unix man pages and HTML documents) for any installable items under **blib/** that contain POD. If there are no`bindoc` or `libdoc` installation targets defined (as will be the case on systems that don't support Unix manpages) no action is taken for manpages. If there are no `binhtml` or `libhtml` installation targets defined no action is taken for HTML documents.

fakeinstall
[version 0.02]

This is just like the `install` action, but it won't actually do anything, it will just report what it *would* have done if you had actually run the `install` action.

help
[version 0.03]

This action will simply print out a message that is meant to help you use the build process. It will show you a list of available build actions too.

With an optional argument specifying an action name (e.g. `Build help test`), the 'help' action will show you any POD documentation it can find for that action.

html
[version 0.26]

This will generate HTML documentation for any binary or library files under **blib/** that contain POD. The HTML documentation will only be installed if the install paths can be determined from values in `Config.pm`. You can also supply or override install paths on the command line by specifying `install_path` values for the `binhtml` and/or `libhtml` installation targets.

With an optional `html_links` argument set to a false value, you can skip the search for other documentation to link to, because that can waste a lot of time if there aren't any links to generate anyway:

```
./Build html --html_links 0
```

install
[version 0.01]

This action will use `ExtUtils::Install` to install the files from `blib/` into the system. See "INSTALL PATHS" for details about how Module::Build determines where to install things, and how to influence this process.

If you want the installation process to look around in `@INC` for other versions of the stuff

you're installing and try to delete it, you can use the `uninst` parameter, which tells ExtUtils::Install to do so:

```
./Build install uninst=1
```

This can be a good idea, as it helps prevent multiple versions of a module from being present on your system, which can be a confusing situation indeed.

installdeps
[version 0.36]

This action will use the `cpan_client` parameter as a command to install missing prerequisites. You will be prompted whether to install optional dependencies.

The `cpan_client` option defaults to 'cpan' but can be set as an option or in *.modulebuildrc*. It must be a shell command that takes a list of modules to install as arguments (e.g. 'cpanp -i' for CPANPLUS). If the program part is a relative path (e.g. 'cpan' or 'cpanp'), it will be located relative to the perl program that executed Build.PL.

```
/opt/perl/5.8.9/bin/perl Build.PL
./Build installdeps --cpan_client 'cpanp -i'
# installs to 5.8.9
```

manifest
[version 0.05]

This is an action intended for use by module authors, not people installing modules. It will bring the *MANIFEST* up to date with the files currently present in the distribution. You may use a *MANIFEST.SKIP* file to exclude certain files or directories from inclusion in the *MANIFEST*. *MANIFEST.SKIP* should contain a bunch of regular expressions, one per line. If a file in the distribution directory matches any of the regular expressions, it won't be included in the *MANIFEST*.

The following is a reasonable *MANIFEST.SKIP* starting point, you can add your own stuff to it:

```
_build
Build$
blib
$
\.bak$
MANIFEST\.SKIP$
CVS
```

See the distcheck and skipcheck actions if you want to find out what the `manifest` action would do, without actually doing anything.

manifest_skip
[version 0.3608]

This is an action intended for use by module authors, not people installing modules. It will generate a boilerplate MANIFEST.SKIP file if one does not already exist.

manpages
[version 0.28]

This will generate man pages for any binary or library files under **blib/** that contain POD. The man pages will only be installed if the install paths can be determined from values in `Config.pm`. You can also supply or override install paths by specifying there values on the command line with the `bindoc` and `libdoc` installation targets.

pardist
[version 0.2806]

Generates a PAR binary distribution for use with PAR or PAR::Dist.

It requires that the PAR::Dist module (version 0.17 and up) is installed on your system.

ppd
[version 0.20]

Build a PPD file for your distribution.

This action takes an optional argument `codebase` which is used in the generated PPD file to specify the (usually relative) URL of the distribution. By default, this value is the distribution name without any path information.

Example:

```
./Build ppd --codebase "MSWin32-x86-multi-thread/Module-Build-0.21.tar.gz"
```

ppmdist
[version 0.23]

Generates a PPM binary distribution and a PPD description file. This action also invokes the `ppd` action, so it can accept the same `codebase` argument described under that action.

This uses the same mechanism as the `dist` action to tar & zip its output, so you can supply `tar` and/or `gzip` parameters to affect the result.

prereq_data
[version 0.32]

This action prints out a Perl data structure of all prerequisites and the versions required. The output can be loaded again using `eval()`. This can be useful for external tools that wish to query a Build script for prerequisites.

prereq_report
[version 0.28]

This action prints out a list of all prerequisites, the versions required, and the versions actually installed. This can be useful for reviewing the configuration of your system prior to a build, or when compiling data to send for a bug report.

pure_install
[version 0.28]

This action is identical to the `install` action. In the future, though, when `install` starts writing to the file *(INSTALLARCHLIB)/perllocal.pod*, `pure_install` won't, and that will be the only difference between them.

realclean
[version 0.01]

This action is just like the `clean` action, but also removes the `_build` directory and the `Build` script. If you run the `realclean` action, you are essentially starting over, so you will have to re-create the `Build` script again.

retest
[version 0.2806]

This is just like the `test` action, but doesn't actually build the distribution first, and doesn't add *blib/* to the load path, and therefore will test against a *previously* installed version of the distribution. This can be used to verify that a certain installed distribution still works, or to see whether newer versions of a distribution still pass the old regression tests, and so on.

skipcheck
[version 0.05]

Reports which files are skipped due to the entries in the *MANIFEST.SKIP* file (See manifest for details)

test
[version 0.01]

This will use `Test::Harness` or `TAP::Harness` to run any regression tests and report their results. Tests can be defined in the standard places: a file called `test.pl` in the top-level directory, or several files ending with `.t` in a `t/` directory.

If you want tests to be 'verbose', i.e. show details of test execution rather than just summary information, pass the argument `verbose=1`.

If you want to run tests under the perl debugger, pass the argument `debugger=1`.

If you want to have Module::Build find test files with different file name extensions, pass the `test_file_exts` argument with an array of extensions, such as `[qw( .t .s .z )]`.

If you want test to be run by `TAP::Harness` rather than `Test::Harness` pass the argument `tap_harness_args` as an array reference of arguments to pass to the TAP::Harness constructor.

In addition, if a file called `visual.pl` exists in the top-level directory, this file will be executed as a Perl script and its output will be shown to the user. This is a good place to put speed tests or other tests that don't use the `Test::Harness` format for output.

To override the choice of tests to run, you may pass a `test_files` argument whose value is a whitespace-separated list of test scripts to run. This is especially useful in development, when you only want to run a single test to see whether you've squashed a certain bug yet:

```
./Build test --test_files t/something_failing.t
```

You may also pass several `test_files` arguments separately:

```
./Build test --test_files t/one.t --test_files t/two.t
```

or use a `glob()`-style pattern:

```
./Build test --test_files 't/01-*.t'
```

testall
[version 0.2807]

[Note: the 'testall' action and the code snippets below are currently in alpha stage, see ``/www.nntp.perl.org/group/perl.module.build/2007/03/msg584.html''`` in ''http:]

Runs the `test` action plus each of the `test$type` actions defined by the keys of the `test_types` parameter.

Currently, you need to define the ACTION_test$type method yourself and enumerate them in the test_types parameter.

```
my $mb = Module::Build->subclass(
code => q(
sub ACTION_testspecial { shift->generic_test(type => 'special'); }
sub ACTION_testauthor { shift->generic_test(type => 'author'); }
)
)->new(
...
test_types => {
special => '.st',
author => ['.at', '.pt' ],
},
...
```

testcover
[version 0.26]

Runs the `test` action using `Devel::Cover` generating a code-coverage report showing which parts of the code were actually exercised during the tests.

To pass options to `Devel::Cover` set the `$DEVEL_COVER_OPTIONS` environment variable:

```
DEVEL_COVER_OPTIONS=-ignore,Build ./Build testcover
```

testdb
[version 0.05]

This is a synonym for the 'test' action with the `debugger=1` argument.

testpod
[version 0.25]

This checks all the files described in the `docs` action and produces `Test::Harness` output. If you are a module author, this is useful to run before creating a new release.

testpodcoverage
[version 0.28]

This checks the pod coverage of the distribution and produces `Test::Harness` output. If you are a module author, this is useful to run before creating a new release.

versioninstall
[version 0.16]

** Note: since `only.pm` is so new, and since we just recently added support for it here too, this feature is to be considered experimental. **

If you have the `only.pm` module installed on your system, you can use this action to install a module into the version-specific library trees. This means that you can have several versions of the same module installed and `use` a specific one like this:

```
use only MyModule => 0.55;
```

To override the default installation libraries in `only::config` specify the `versionlib` parameter when you run the `Build.PL` script:

```
perl Build.PL --versionlib /my/version/place/
```

To override which version the module is installed as, specify the `version` parameter when you run the `Build.PL` script:

```
perl Build.PL --version 0.50
```

See the `only.pm` documentation for more information on version-specific installs.

OPTIONS

Command Line Options

The following options can be used during any invocation of `Build.PL` or the Build script, during any action. For information on other options specific to an action, see the documentation for the respective action.

NOTE: There is some preliminary support for options to use the more familiar long option style. Most options can be preceded with the `--` long option prefix, and the underscores changed to dashes (e.g. `--use-rcfile`). Additionally, the argument to boolean options is optional, and boolean options can be negated by prefixing them with `no` or `no-` (e.g. `--noverbose` or `--no-verbose`).

quiet
: Suppress informative messages on output.

verbose
: Display extra information about the Build on output. `verbose` will turn off `quiet`

cpan_client
: Sets the `cpan_client` command for use with the `installdeps` action. See `installdeps` for more details.

use_rcfile
: Load the *~/.modulebuildrc* option file. This option can be set to false to prevent the custom resource file from being loaded.

allow_mb_mismatch
: Suppresses the check upon startup that the version of Module::Build we're now running under is the same version that was initially invoked when building the distribution (i.e. when the `Build.PL` script was first run). As of 0.3601, a mismatch results in a warning instead of a fatal error, so this option effectively just suppresses the warning.

debug
: Prints Module::Build debugging information to STDOUT, such as a trace of executed build actions.

Default Options File (*.modulebuildrc*)

[version 0.28]

When Module::Build starts up, it will look first for a file, *ENV{HOME}/.modulebuildrc*. If it's not found there, it will look in the *.modulebuildrc* file in the directories referred to by the environment variables `HOMEDRIVE` + `HOMEDIR`, `USERPROFILE`, `APPDATA`, `WINDIR`, `SYS$LOGIN`. If the file exists, the options specified there will be used as defaults, as if they were typed on the command line. The defaults can be overridden by specifying new values on the command line.

The action name must come at the beginning of the line, followed by any amount of whitespace and then the options. Options are given the same as they would be on the command line. They can be separated by any amount of whitespace, including newlines, as long there is whitespace at the beginning of each continued line. Anything following a hash mark (`#`) is considered a comment, and is stripped before parsing. If more than one line begins with the same action name, those lines are merged into one set of options.

Besides the regular actions, there are two special pseudo-actions: the key `*` (asterisk) denotes any global options that should be applied to all actions, and the key 'Build_PL' specifies options to be applied when you invoke `perl Build.PL`.

```
* verbose=1 # global options
diff flags=-u
install --install_base /home/ken
--install_path html=/home/ken/docs/html
installdeps --cpan_client 'cpanp -i'
```

If you wish to locate your resource file in a different location, you can set the environment variable `MODULEBUILDRC` to the complete absolute path of the file containing your options.

Environment variables

MODULEBUILDRC
[version 0.28]

Specifies an alternate location for a default options file as described above.

PERL_MB_OPT
[version 0.36]

Command line options that are applied to Build.PL or any Build action. The string is split as the shell would (e.g. whitespace) and the result is prepended to any actual command-line arguments.

INSTALL PATHS

[version 0.19]

When you invoke Module::Build's `build` action, it needs to figure out where to install things. The nutshell version of how this works is that default installation locations are determined from *Config.pm*, and they may be overridden by using the `install_path` parameter. An `install_base` parameter lets you specify an alternative installation root like */home/foo*, and a `destdir` lets you specify a temporary installation directory like */tmp/install* in case you want to create bundled-up installable packages.

Natively, Module::Build provides default installation locations for the following types of installable items:

lib Usually pure-Perl module files ending in *.pm*.

arch
"Architecture-dependent" module files, usually produced by compiling XS, Inline, or similar code.

script
Programs written in pure Perl. In order to improve reuse, try to make these as small as possible - put the code into modules whenever possible.

bin "Architecture-dependent" executable programs, i.e. compiled C code or something. Pretty rare to see this in a perl distribution, but it happens.

bindoc
Documentation for the stuff in `script` and `bin`. Usually generated from the POD in those files. Under Unix, these are manual pages belonging to the 'man1' category.

libdoc
Documentation for the stuff in `lib` and `arch`. This is usually generated from the POD in *.pm* files. Under Unix, these are manual pages belonging to the 'man3' category.

binhtml
This is the same as `bindoc` above, but applies to HTML documents.

libhtml
This is the same as `libdoc` above, but applies to HTML documents.

Four other parameters let you control various aspects of how installation paths are determined:

installdirs
The default destinations for these installable things come from entries in your system's `Config.pm`. You can select from three different sets of default locations by setting the `installdirs` parameter as follows:

```
'installdirs' set to:
core site vendor

uses the following defaults from Config.pm:

lib => installprivlib installsitelib installvendorlib
arch => installarchlib installsitearch installvendorarch
script => installscript installsitescript installvendorscript
bin => installbin installsitebin installvendorbin
bindoc => installman1dir installsiteman1dir installvendorman1dir
libdoc => installman3dir installsiteman3dir installvendorman3dir
binhtml => installhtml1dir installsitehtml1dir installvendorhtml1dir [*]
libhtml => installhtml3dir installsitehtml3dir installvendorhtml3dir [*]

* Under some OS (eg. MSWin32) the destination for HTML documents is
determined by the C<Config.pm> entry C<installhtmldir>.
```

The default value of `installdirs` is ''site''. If you're creating vendor distributions of module packages, you may want to do something like this:

```
perl Build.PL --installdirs vendor
```

or

```
./Build install --installdirs vendor
```

If you're installing an updated version of a module that was included with perl itself (i.e. a ''core module''), then you may set `installdirs` to ''core'' to overwrite the module in its present location.

(Note that the 'script' line is different from `MakeMaker` - unfortunately there's no such thing as ''installsitescript'' or ''installvendorscript'' entry in `Config.pm`, so we use the ''installsitebin'' and ''installvendorbin'' entries to at least get the general location right. In the future, if `Config.pm` adds some more appropriate entries, we'll start using those.)

install_path

Once the defaults have been set, you can override them.

On the command line, that would look like this:

```
perl Build.PL --install_path lib=/foo/lib --install_path arch=/foo/lib/arch
```

or this:

```
./Build install --install_path lib=/foo/lib --install_path arch=/foo/lib/arch
```

install_base

You can also set the whole bunch of installation paths by supplying the `install_base` parameter to point to a directory on your system. For instance, if you set `install_base` to ''/home/ken'' on a Linux system, you'll install as follows:

```
lib => /home/ken/lib/perl5
arch => /home/ken/lib/perl5/i386-linux
script => /home/ken/bin
bin => /home/ken/bin
bindoc => /home/ken/man/man1
libdoc => /home/ken/man/man3
binhtml => /home/ken/html
libhtml => /home/ken/html
```

Note that this is *different* from how `MakeMaker`'s `PREFIX` parameter works. `install_base` just gives you a default layout under the directory you specify, which may have little to do

with the `installdirs=site` layout.

The exact layout under the directory you specify may vary by system - we try to do the ''sensible'' thing on each platform.

destdir
: If you want to install everything into a temporary directory first (for instance, if you want to create a directory tree that a package manager like `rpm` or `dpkg` could create a package from), you can use the `destdir` parameter:

```
perl Build.PL --destdir /tmp/foo
```

or

```
./Build install --destdir /tmp/foo
```

This will effectively install to ''/tmp/foo/$sitelib'', ''/tmp/foo/$sitearch'', and the like, except that it will use `File::Spec` to make the pathnames work correctly on whatever platform you're installing on.

prefix
: Provided for compatibility with `ExtUtils::MakeMaker` PREFIX argument. `prefix` should be used when you want Module::Build to install your modules, documentation, and scripts in the same place as `ExtUtils::MakeMaker` PREFIX mechanism.

The following are equivalent.

```
perl Build.PL --prefix /tmp/foo
perl Makefile.PL PREFIX=/tmp/foo
```

Because of the complex nature of the prefixification logic, the behavior of PREFIX in `MakeMaker` has changed subtly over time. Module::Build's --prefix logic is equivalent to the PREFIX logic found in `ExtUtils::MakeMaker` 6.30.

The maintainers of `MakeMaker` do understand the troubles with the PREFIX mechanism, and added INSTALL_BASE support in version 6.31 of `MakeMaker`, which was released in 2006.

If you don't need to retain compatibility with old versions (pre-6.31) of `ExtUtils::MakeMaker` or are starting a fresh Perl installation we recommend you use `install_base` instead (and `INSTALL_BASE` in `ExtUtils::MakeMaker` See ''Installing in the same location as ExtUtils::MakeMaker'' in Module::Build::Cookbook for further information.

MOTIVATIONS

There are several reasons I wanted to start over, and not just fix what I didn't like about `MakeMaker`:

- I don't like the core idea of `MakeMaker`, namely that `make` should be involved in the build process. Here are my reasons:
  - + When a person is installing a Perl module, what can you assume about their environment? Can you assume they have `make`? No, but you can assume they have some version of Perl.
  - + When a person is writing a Perl module for intended distribution, can you assume that they know how to build a Makefile, so they can customize their build process? No, but you can assume they know Perl, and could customize that way.

  For years, these things have been a barrier to people getting the build/install process to do what they want.
- There are several architectural decisions in `MakeMaker` that make it very difficult to customize its behavior. For instance, when using `MakeMaker` you do `use ExtUtils::MakeMaker` but the object created in `WriteMakefile()` is actually blessed into a package name that's created on the fly, so you can't simply subclass `ExtUtils::MakeMaker`

There is a workaround `MY` package that lets you override certain `MakeMaker` methods, but only certain explicitly preselected (by `MakeMaker`) methods can be overridden. Also, the method of customization is very crude: you have to modify a string containing the Makefile text for the particular target. Since these strings aren't documented, and *can't* be documented (they take on different values depending on the platform, version of perl, version of `MakeMaker`, etc.), you have no guarantee that your modifications will work on someone else's machine or after an upgrade of `MakeMaker` or perl.

- It is risky to make major changes to `MakeMaker`, since it does so many things, is so important, and generally works. `Module::Build` is an entirely separate package so that I can work on it all I want, without worrying about backward compatibility with `MakeMaker`.
- Finally, Perl is said to be a language for system administration. Could it really be the case that Perl isn't up to the task of building and installing software? Even if that software is a bunch of `.pm` files that just need to be copied from one place to another? My sense was that we could design a system to accomplish this in a flexible, extensible, and friendly manner. Or die trying.

TO DO

The current method of relying on time stamps to determine whether a derived file is out of date isn't likely to scale well, since it requires tracing all dependencies backward, it runs into problems on NFS, and it's just generally flimsy. It would be better to use an MD5 signature or the like, if available. See `cons` for an example.

```
- append to perllocal.pod
- add a 'plugin' functionality
```

AUTHOR

Ken Williams <kwilliams@cpan.org>

Development questions, bug reports, and patches should be sent to the Module-Build mailing list at <module-build@perl.org>.

Bug reports are also welcome at <http://rt.cpan.org/NoAuth/Bugs.html?Dist=Module-Build>.

The latest development version is available from the Git repository at <https://github.com/Perl-Toolchain-Gang/Module-Build>

COPYRIGHT

SEE ALSO

perl(1), Module::Build::Cookbook, Module::Build::Authoring, Module::Build::API, ExtUtils::MakeMaker

META.yml Specification: CPAN::Meta::Spec

<http://www.dsmit.com/cons/>

<http://search.cpan.org/dist/PerlBuildSystem/>