NAME

Module::Build::API - API Reference for Module Authors

DESCRIPTION

I list here some of the most important methods in `Module::Build`. Normally you won't need to deal with these methods unless you want to subclass `Module::Build`. But since one of the reasons I created this module in the first place was so that subclassing is possible (and easy), I will certainly write more docs as the interface stabilizes.

CONSTRUCTORS

current()

[version 0.20]

This method returns a reasonable facsimile of the currently-executing `Module::Build` object representing the current build. You can use this object to query its ''*notes()*'' method, inquire about installed modules, and so on. This is a great way to share information between different parts of your build process. For instance, you can ask the user a question during `perl Build.PL`, then use their answer during a regression test:

```
# In Build.PL:
my $color = $build->prompt("What is your favorite color?");
$build->notes(color => $color);

# In t/colortest.t:
use Module::Build;
my $build = Module::Build->current;
my $color = $build->notes('color');
...
```

The way the `current()` method is currently implemented, there may be slight differences between the `$build` object in Build.PL and the one in `t/colortest.t`. It is our goal to minimize these differences in future releases of Module::Build, so please report any anomalies you find.

One important caveat: in its current implementation, `current()` will **NOT** work correctly if you have changed out of the directory that `Module::Build` was invoked from.

new()

[version 0.03]

Creates a new Module::Build object. Arguments to the *new()* method are listed below. Most arguments are optional, but you must provide either the ''module_name'' argument, or ''dist_name'' and one of ''dist_version'' or ''dist_version_from''. In other words, you must provide enough information to determine both a distribution name and version.

add_to_cleanup

[version 0.19]

An array reference of files to be cleaned up when the `clean` action is performed. See also the *add_to_cleanup()* method.

allow_pureperl

[version 0.4005]

A bool indicating the module is still functional without its xs parts. When an XS module is build with --pureperl_only, it will otherwise fail.

auto_configure_requires

[version 0.34]

This parameter determines whether Module::Build will add itself automatically to configure_requires (and build_requires) if Module::Build is not already there. The

required version will be the last 'major' release, as defined by the decimal version truncated to two decimal places (e.g. 0.34, instead of 0.3402). The default value is true.

auto_features
[version 0.26]

This parameter supports the setting of features (see ''feature($name)'') automatically based on a set of prerequisites. For instance, for a module that could optionally use either MySQL or PostgreSQL databases, you might use `auto_features` like this:

```
my $build = Module::Build->new
(
...other stuff here...
auto_features => {
pg_support => {
description => "Interface with Postgres databases",
requires => { 'DBD::Pg' => 23.3,
'DateTime::Format::Pg' => 0 },
},
mysql_support => {
description => "Interface with MySQL databases",
requires => { 'DBD::mysql' => 17.9,
'DateTime::Format::MySQL' => 0 },
},
}
);
```

For each feature named, the required prerequisites will be checked, and if there are no failures, the feature will be enabled (set to `1`). Otherwise the failures will be displayed to the user and the feature will be disabled (set to `0`).

See the documentation for ''requires'' for the details of how requirements can be specified.

autosplit
[version 0.04]

An optional `autosplit` argument specifies a file which should be run through the *AutoSplit::autosplit()* function. If multiple files should be split, the argument may be given as an array of the files to split.

In general I don't consider autosplitting a great idea, because it's not always clear that autosplitting achieves its intended performance benefits. It may even harm performance in environments like mod_perl, where as much as possible of a module's code should be loaded during startup.

build_class
[version 0.28]

The Module::Build class or subclass to use in the build script. Defaults to ''Module::Build'' or the class name passed to or created by a call to ''*subclass()*''. This property is useful if you're writing a custom Module::Build subclass and have a bootstrapping problem — that is, your subclass requires modules that may not be installed when `perl Build.PL` is executed, but you've listed in ''build_requires'' so that they should be available when `./Build` is executed.

build_requires
[version 0.07]

Modules listed in this section are necessary to build and install the given module, but are not necessary for regular usage of it. This is actually an important distinction - it

allows for tighter control over the body of installed modules, and facilitates correct dependency checking on binary/packaged distributions of the module.

See the documentation for "PREREQUISITES" in Module::Build::Authoring for the details of how requirements can be specified.

configure_requires
[version 0.30]

Modules listed in this section must be installed *before* configuring this distribution (i.e. before running the *Build.PL* script). This might be a specific minimum version of `Module::Build` or any other module the *Build.PL* needs in order to do its stuff. Clients like `CPAN.pm` or `CPANPLUS` will be expected to pick `configure_requires` out of the *META.yml* file and install these items before running the `Build.PL`.

Module::Build may automatically add itself to configure_requires. See "auto_configure_requires" for details.

See the documentation for "PREREQUISITES" in Module::Build::Authoring for the details of how requirements can be specified.

test_requires
[version 0.4004]

Modules listed in this section must be installed before testing the distribution.

See the documentation for "PREREQUISITES" in Module::Build::Authoring for the details of how requirements can be specified.

create_packlist
[version 0.28]

If true, this parameter tells Module::Build to create a *.packlist* file during the `install` action, just like `ExtUtils::MakeMaker` does. The file is created in a subdirectory of the `arch` installation location. It is used by some other tools (CPAN, CPANPLUS, etc.) for determining what files are part of an install.

The default value is true. This parameter was introduced in Module::Build version 0.2609; previously no packlists were ever created by Module::Build.

c_source
[version 0.04]

An optional `c_source` argument specifies a directory which contains C source files that the rest of the build may depend on. Any `.c` files in the directory will be compiled to object files. The directory will be added to the search path during the compilation and linking phases of any C or XS files.

[version 0.3604]

A list of directories can be supplied using an anonymous array reference of strings.

conflicts
[version 0.07]

Modules listed in this section conflict in some serious way with the given module. `Module::Build` (or some higher-level tool) will refuse to install the given module if the given module/version is also installed.

See the documentation for "PREREQUISITES" in Module::Build::Authoring for the details of how requirements can be specified.

create_license
[version 0.31]

This parameter tells Module::Build to automatically create a *LICENSE* file at the top level of your distribution, containing the full text of the author's chosen license. This requires `Software::License` on the author's machine, and further requires that the `license` parameter specifies a license that it knows about.

create_makefile_pl
[version 0.19]

This parameter lets you use `Module::Build::Compat` during the `distdir` (or `dist`) action to automatically create a Makefile.PL for compatibility with `ExtUtils::MakeMaker` The parameter's value should be one of the styles named in the Module::Build::Compat documentation.

create_readme
[version 0.22]

This parameter tells Module::Build to automatically create a *README* file at the top level of your distribution. Currently it will simply use `Pod::Text` (or `Pod::Readme` if it's installed) on the file indicated by `dist_version_from` and put the result in the *README* file. This is by no means the only recommended style for writing a *README*, but it seems to be one common one used on the CPAN.

If you generate a *README* in this way, it's probably a good idea to create a separate *INSTALL* file if that information isn't in the generated *README*.

dist_abstract
[version 0.20]

This should be a short description of the distribution. This is used when generating metadata for *META.yml* and PPD files. If it is not given then `Module::Build` looks in the POD of the module from which it gets the distribution's version. If it finds a POD section marked "=head1 NAME", then it looks for the first line matching `\s+-\s+(.+)`, and uses the captured text as the abstract.

dist_author
[version 0.20]

This should be something like "John Doe <jdoe@example.com>", or if there are multiple authors, an anonymous array of strings may be specified. This is used when generating metadata for *META.yml* and PPD files. If this is not specified, then `Module::Build` looks at the module from which it gets the distribution's version. If it finds a POD section marked "=head1 AUTHOR", then it uses the contents of this section.

dist_name
[version 0.11]

Specifies the name for this distribution. Most authors won't need to set this directly, they can use `module_name` to set `dist_name` to a reasonable default. However, some agglomerative distributions like `libwww-perl` or `bioperl` have names that don't correspond directly to a module name, so `dist_name` can be set independently.

dist_suffix
[version 0.37]

Specifies an optional suffix to include after the version number in the distribution directory (and tarball) name. The only suffix currently recognized by PAUSE is 'TRIAL', which indicates that the distribution should not be indexed. For example:

```
Foo-Bar-1.23-TRIAL.tar.gz
```

This will automatically do the "right thing" depending on `dist_version` and `release_status`. When `dist_version` does not have an underscore and `release_status` is not 'stable', then `dist_suffix` will default to 'TRIAL'. Otherwise it

will default to the empty string, disabling the suffix.

In general, authors should only set this if they **must** override the default behavior for some particular purpose.

dist_version
[version 0.11]

Specifies a version number for the distribution. See ''module_name'' or ''dist_version_from'' for ways to have this set automatically from a `$VERSION` variable in a module. One way or another, a version number needs to be set.

dist_version_from
[version 0.11]

Specifies a file to look for the distribution version in. Most authors won't need to set this directly, they can use ''module_name'' to set it to a reasonable default.

The version is extracted from the specified file according to the same rules as ExtUtils::MakeMaker and `CPAN.pm`. It involves finding the first line that matches the regular expression

```
/([\$*])(([\w\:\']*)\bVERSION)\b.*\=/
```

eval()-ing that line, then checking the value of the `$VERSION` variable. Quite ugly, really, but all the modules on CPAN depend on this process, so there's no real opportunity to change to something better.

If the target file of ''dist_version_from'' contains more than one package declaration, the version returned will be the one matching the configured ''module_name''.

dynamic_config
[version 0.07]

A boolean flag indicating whether the *Build.PL* file must be executed, or whether this module can be built, tested and installed solely from consulting its metadata file. The main reason to set this to a true value is that your module performs some dynamic configuration as part of its build/install process. If the flag is omitted, the *META.yml* spec says that installation tools should treat it as 1 (true), because this is a safer way to behave.

Currently `Module::Build` doesn't actually do anything with this flag - it's up to higher-level tools like `CPAN.pm` to do something useful with it. It can potentially bring lots of security, packaging, and convenience improvements.

extra_compiler_flags
extra_linker_flags
[version 0.19]

These parameters can contain array references (or strings, in which case they will be split into arrays) to pass through to the compiler and linker phases when compiling/linking C code. For example, to tell the compiler that your code is C++, you might do:

```
my $build = Module::Build->new
(
module_name => 'Foo::Bar',
extra_compiler_flags => ['-x', 'c++'],
);
```

To link your XS code against glib you might write something like:

```
my $build = Module::Build->new
(
module_name => 'Foo::Bar',
dynamic_config => 1,
extra_compiler_flags => scalar `glib-config --cflags`,
extra_linker_flags => scalar `glib-config --libs`,
);
```

extra_manify_args
[version 0.4006]

Any extra arguments to pass to `Pod::Man->new()` when building man pages. One common choice might be `utf8 => 1` to get Unicode support.

get_options
[version 0.26]

You can pass arbitrary command line options to *Build.PL* or *Build*, and they will be stored in the Module::Build object and can be accessed via the "*args()*" method. However, sometimes you want more flexibility out of your argument processing than this allows. In such cases, use the `get_options` parameter to pass in a hash reference of argument specifications, and the list of arguments to *Build.PL* or *Build* will be processed according to those specifications before they're passed on to `Module::Build` own argument processing.

The supported option specification hash keys are:

type
The type of option. The types are those supported by Getopt::Long; consult its documentation for a complete list. Typical types are `=s` for strings, `+` for additive options, and `!` for negatable options. If the type is not specified, it will be considered a boolean, i.e. no argument is taken and a value of 1 will be assigned when the option is encountered.

store
A reference to a scalar in which to store the value passed to the option. If not specified, the value will be stored under the option name in the hash returned by the `args()` method.

default
A default value for the option. If no default value is specified and no option is passed, then the option key will not exist in the hash returned by `args()`.

You can combine references to your own variables or subroutines with unreferenced specifications, for which the result will also be stored in the hash returned by `args()`. For example:

```
my $loud = 0;
my $build = Module::Build->new
(
module_name => 'Foo::Bar',
get_options => {
Loud => { store => \$loud },
Dbd => { type => '=s' },
Quantity => { type => '+' },
}
);

print STDERR "HEY, ARE YOU LISTENING??\n" if $loud;
print "We'll use the ", $build->args('Dbd'), " DBI driver\n";
```

```
print "Are you sure you want that many?\n"
if $build->args('Quantity') > 2;
```

The arguments for such a specification can be called like so:

```
perl Build.PL --Loud --Dbd=DBD::pg --Quantity --Quantity --Quantity
```

WARNING: Any option specifications that conflict with Module::Build's own options (defined by its properties) will throw an exception. Use capitalized option names to avoid unintended conflicts with future Module::Build options.

Consult the Getopt::Long documentation for details on its usage.

include_dirs
[version 0.24]

Specifies any additional directories in which to search for C header files. May be given as a string indicating a single directory, or as a list reference indicating multiple directories.

install_path
[version 0.19]

You can set paths for individual installable elements by using the `install_path` parameter:

```
my $build = Module::Build->new
(
...other stuff here...
install_path => {
lib => '/foo/lib',
arch => '/foo/lib/arch',
}
);
```

installdirs
[version 0.19]

Determines where files are installed within the normal perl hierarchy as determined by *Config.pm*. Valid values are: `core`, `site`, `vendor`. The default is `site`. See "INSTALL PATHS" in Module::Build

license
[version 0.07]

Specifies the licensing terms of your distribution.

As of Module::Build version 0.36_14, you may use a Software::License subclass name (e.g. 'Apache_2_0') instead of one of the keys below.

The legacy list of valid license values include:

apache
The distribution is licensed under the Apache License, Version 2.0 (<http://apache.org/licenses/LICENSE-2.0>).

apache_1_1
The distribution is licensed under the Apache Software License, Version 1.1 (<http://apache.org/licenses/LICENSE-1.1>).

artistic
The distribution is licensed under the Artistic License, as specified by the *Artistic* file in the standard Perl distribution.

artistic_2
: The distribution is licensed under the Artistic 2.0 License (<http://opensource.org/licenses/artistic-license-2.0.php>.)

bsd
: The distribution is licensed under the BSD License (<http://www.opensource.org/licenses/bsd-license.php>).

gpl
: The distribution is licensed under the terms of the GNU General Public License (<http://www.opensource.org/licenses/gpl-license.php>).

lgpl
: The distribution is licensed under the terms of the GNU Lesser General Public License (<http://www.opensource.org/licenses/lgpl-license.php>).

mit
: The distribution is licensed under the MIT License (<http://opensource.org/licenses/mit-license.php>).

mozilla
: The distribution is licensed under the Mozilla Public License. (<http://opensource.org/licenses/mozilla1.0.php> or <http://opensource.org/licenses/mozilla1.1.php>)

open_source
: The distribution is licensed under some other Open Source Initiative-approved license listed at <http://www.opensource.org/licenses/>.

perl
: The distribution may be copied and redistributed under the same terms as Perl itself (this is by far the most common licensing option for modules on CPAN). This is a dual license, in which the user may choose between either the GPL or the Artistic license.

restrictive
: The distribution may not be redistributed without special permission from the author and/or copyright holder.

unrestricted
: The distribution is licensed under a license that is **not** approved by www.opensource.org but that allows distribution without restrictions.

Note that you must still include the terms of your license in your code and documentation - this field only sets the information that is included in distribution metadata to let automated tools figure out your licensing restrictions. Humans still need something to read. If you choose to provide this field, you should make sure that you keep it in sync with your written documentation if you ever change your licensing terms.

You may also use a license type of `unknown` if you don't wish to specify your terms in the metadata.

Also see the `create_license` parameter.

meta_add
: [version 0.28]

A hash of key/value pairs that should be added to the *META.yml* file during the `distmeta` action. Any existing entries with the same names will be overridden.

See the "MODULE METADATA" section for details.

meta_merge
: [version 0.28]

A hash of key/value pairs that should be merged into the *META.yml* file during the

`distmeta` action. Any existing entries with the same names will be overridden.

The only difference between `meta_add` and `meta_merge` is their behavior on hash-valued and array-valued entries: `meta_add` will completely blow away the existing hash or array value, but `meta_merge` will merge the supplied data into the existing hash or array value.

See the "MODULE METADATA" section for details.

module_name
[version 0.03]

The `module_name` is a shortcut for setting default values of `dist_name` and `dist_version_from`, reflecting the fact that the majority of CPAN distributions are centered around one "main" module. For instance, if you set `module_name` to `Foo::Bar` then `dist_name` will default to `Foo-Bar` and `dist_version_from` will default to `lib/Foo/Bar.pm`. `dist_version_from` will in turn be used to set `dist_version`.

Setting `module_name` won't override a `dist_*` parameter you specify explicitly.

needs_compiler
[version 0.36]

The `needs_compiler` parameter indicates whether a compiler is required to build the distribution. The default is false, unless XS files are found or the `c_source` parameter is set, in which case it is true. If true, ExtUtils::CBuilder is automatically added to `build_requires` if needed.

For a distribution where a compiler is *optional*, e.g. a dual XS/pure-Perl distribution, `needs_compiler` should explicitly be set to a false value.

PL_files
[version 0.06]

An optional parameter specifying a set of `.PL` files in your distribution. These will be run as Perl scripts prior to processing the rest of the files in your distribution with the name of the file they're generating as an argument. They are usually used as templates for creating other files dynamically, so that a file like `lib/Foo/Bar.pm.PL` might create the file `lib/Foo/Bar.pm`.

The files are specified with the `.PL` files as hash keys, and the file(s) they generate as hash values, like so:

```
my $build = Module::Build->new
(
module_name => 'Foo::Bar',
...
PL_files => { 'lib/Foo/Bar.pm.PL' => 'lib/Foo/Bar.pm' },
);
```

Note that the path specifications are *always* given in Unix-like format, not in the style of the local system.

If your `.PL` scripts don't create any files, or if they create files with unexpected names, or even if they create multiple files, you can indicate that so that Module::Build can properly handle these created files:

```
PL_files => {
'lib/Foo/Bar.pm.PL' => 'lib/Foo/Bar.pm',
'lib/something.PL' => ['/lib/something', '/lib/else'],
'lib/funny.PL' => [],
}
```

Here's an example of a simple PL file.

```
my $output_file = shift;
open my $fh, ">", $output_file or die "Can't open $output_file: $!";

print $fh <<'END';
#!/usr/bin/perl

print "Hello, world!\n";
END
```

PL files are not installed by default, so its safe to put them in *lib/* and *bin/*.

pm_files
[version 0.19]

An optional parameter specifying the set of `.pm` files in this distribution, specified as a hash reference whose keys are the files' locations in the distributions, and whose values are their logical locations based on their package name, i.e. where they would be found in a ''normal'' Module::Build-style distribution. This parameter is mainly intended to support alternative layouts of files.

For instance, if you have an old-style `MakeMaker` distribution for a module called `Foo::Bar` and a *Bar.pm* file at the top level of the distribution, you could specify your layout in your `Build.PL` like this:

```
my $build = Module::Build->new
(
module_name => 'Foo::Bar',
...
pm_files => { 'Bar.pm' => 'lib/Foo/Bar.pm' },
);
```

Note that the values should include `lib/`, because this is where they would be found in a ''normal'' Module::Build-style distribution.

Note also that the path specifications are *always* given in Unix-like format, not in the style of the local system.

pod_files
[version 0.19]

Just like `pm_files`, but used for specifying the set of `.pod` files in your distribution.

recommends
[version 0.08]

This is just like the ''requires'' argument, except that modules listed in this section aren't essential, just a good idea. We'll just print a friendly warning if one of these modules aren't found, but we'll continue running.

If a module is recommended but not required, all tests should still pass if the module isn't installed. This may mean that some tests may be skipped if recommended dependencies aren't present.

Automated tools like CPAN.pm should inform the user when recommended modules

aren't installed, and it should offer to install them if it wants to be helpful.

See the documentation for "PREREQUISITES" in Module::Build::Authoring for the details of how requirements can be specified.

recursive_test_files
[version 0.28]

Normally, `Module::Build` does not search subdirectories when looking for tests to run. When this options is set it will search recursively in all subdirectories of the standard 't' test directory.

release_status
[version 0.37]

The CPAN Meta Spec version 2 adds `release_status` to allow authors to specify how a distribution should be indexed. Consistent with the spec, this parameter can only have one three values: 'stable', 'testing' or 'unstable'.

Unless explicitly set by the author, `release_status` will default to 'stable' unless `dist_version` contains an underscore, in which case it will default to 'testing'.

It is an error to specify a `release_status` of 'stable' when `dist_version` contains an underscore character.

requires
[version 0.07]

An optional `requires` argument specifies any module prerequisites that the current module depends on.

One note: currently `Module::Build` doesn't actually *require* the user to have dependencies installed, it just strongly urges. In the future we may require it. There's also a "recommends" section for things that aren't absolutely required.

Automated tools like CPAN.pm should refuse to install a module if one of its dependencies isn't satisfied, unless a "force" command is given by the user. If the tools are helpful, they should also offer to install the dependencies.

A synonym for `requires` is `prereq`, to help succour people transitioning from `ExtUtils::MakeMaker` The `requires` term is preferred, but the `prereq` term will remain valid in future distributions.

See the documentation for "PREREQUISITES" in Module::Build::Authoring for the details of how requirements can be specified.

script_files
[version 0.18]

An optional parameter specifying a set of files that should be installed as executable Perl scripts when the module is installed. May be given as an array reference of the files, as a hash reference whose keys are the files (and whose values will currently be ignored), as a string giving the name of a directory in which to find scripts, or as a string giving the name of a single script file.

The default is to install any scripts found in a *bin* directory at the top level of the distribution, minus any keys of PL_files.

For backward compatibility, you may use the parameter `scripts` instead of `script_files`. Please consider this usage deprecated, though it will continue to exist for several version releases.

share_dir
[version 0.36]

An optional parameter specifying directories of static data files to be installed as read-only files for use with File::ShareDir. The `share_dir` property supports both distribution-level and module-level share files.

The simplest use of `share_dir` is to set it to a directory name or an arrayref of directory names containing files to be installed in the distribution-level share directory.

```
share_dir => 'share'
```

Alternatively, if `share_dir` is a hashref, it may have `dist` or `module` keys providing full flexibility in defining how share directories should be installed.

```
share_dir => {
dist => [ 'examples', 'more_examples' ],
module => {
Foo::Templates => ['share/html', 'share/text'],
Foo::Config => 'share/config',
}
}
```

If `share_dir` is set, then File::ShareDir will automatically be added to the `requires` hash.

sign
[version 0.16]

If a true value is specified for this parameter, Module::Signature will be used (via the 'distsign' action) to create a SIGNATURE file for your distribution during the 'distdir' action, and to add the SIGNATURE file to the MANIFEST (therefore, don't add it yourself).

The default value is false. In the future, the default may change to true if you have `Module::Signature` installed on your system.

tap_harness_args
[version 0.2808_03]

An optional parameter specifying parameters to be passed to TAP::Harness when running tests. Must be given as a hash reference of parameters; see the TAP::Harness documentation for details. Note that specifying this parameter will implicitly set `use_tap_harness` to a true value. You must therefore be sure to add TAP::Harness as a requirement for your module in ''build_requires''.

test_files
[version 0.23]

An optional parameter specifying a set of files that should be used as `Test::Harness` regression tests to be run during the `test` action. May be given as an array reference of the files, or as a hash reference whose keys are the files (and whose values will currently be ignored). If the argument is given as a single string (not in an array reference), that string will be treated as a `glob()` pattern specifying the files to use.

The default is to look for a *test.pl* script in the top-level directory of the distribution, and any files matching the glob pattern `*.t` in the *t/* subdirectory. If the `recursive_test_files` property is true, then the `t/` directory will be scanned recursively for `*.t` files.

use_tap_harness
[version 0.2808_03]

An optional parameter indicating whether or not to use TAP::Harness for testing rather than Test::Harness. Defaults to false. If set to true, you must therefore be sure to add TAP::Harness as a requirement for your module in "build_requires". Implicitly set to a true value if `tap_harness_args` is specified.

xs_files
[version 0.19]

Just like `pm_files`, but used for specifying the set of `.xs` files in your distribution.

new_from_context(%args)
[version 0.28]

When called from a directory containing a *Build.PL* script (in other words, the base directory of a distribution), this method will run the *Build.PL* and call `resume()` to return the resulting `Module::Build` object to the caller. Any key-value arguments given to `new_from_context()` are essentially like command line arguments given to the *Build.PL* script, so for example you could pass `verbose => 1` to this method to turn on verbosity.

resume()
[version 0.03]

You'll probably never call this method directly, it's only called from the auto-generated `Build` script (and the `new_from_context` method). The `new()` method is only called once, when the user runs `perl Build.PL`. Thereafter, when the user runs`Build test` or another action, the `Module::Build` object is created using the `resume()` method to re-instantiate with the settings given earlier to `new()`.

subclass()
[version 0.06]

This creates a new `Module::Build` subclass on the fly, as described in the "SUBCLASSING" in Module::Build::Authoring section. The caller must provide either a `class` or `code` parameter, or both. The `class` parameter indicates the name to use for the new subclass, and defaults to `MyModuleBuilder`. The `code` parameter specifies Perl code to use as the body of the subclass.

add_property
[version 0.31]

```
package 'My::Build';
use base 'Module::Build';
__PACKAGE__->add_property( 'pedantic' );
__PACKAGE__->add_property( answer => 42 );
__PACKAGE__->add_property(
'epoch',
default => sub { time },
check => sub {
return 1 if /\d+$/;
shift->property_error( "'$_' is not an epoch time" );
return 0;
},
);
```

Adds a property to a Module::Build class. Properties are those attributes of a Module::Build object which can be passed to the constructor and which have accessors to get and set them. All of the core properties, such as `module_name` and `license`, are defined using this class method.

The first argument to `add_property()` is always the name of the property. The second argument can be either a default value for the property, or a list of key/value pairs. The supported keys are:

`default`
: The default value. May optionally be specified as a code reference, in which case the return value from the execution of the code reference will be used. If you need the default to be a code reference, just use a code reference to return it, e.g.:

```
default => sub { sub { ... } },
```

`check`
: A code reference that checks that a value specified for the property is valid. During the execution of the code reference, the new value will be included in the `$_` variable. If the value is correct, the `check` code reference should return true. If the value is not correct, it sends an error message to `property_error()` and returns false.

When this method is called, a new property will be installed in the Module::Build class, and an accessor will be built to allow the property to be get or set on the build object.

```
print $build->pedantic, $/;
$build->pedantic(0);
```

If the default value is a hash reference, this generates a special-case accessor method, wherein individual key/value pairs may be set or fetched:

```
print "stuff{foo} is: ", $build->stuff( 'foo' ), $/;
$build->stuff( foo => 'bar' );
print $build->stuff( 'foo' ), $/; # Outputs "bar"
```

Of course, you can still set the entire hash reference at once, as well:

```
$build->stuff( { foo => 'bar', baz => 'yo' } );
```

In either case, if a `check` has been specified for the property, it will be applied to the entire hash. So the check code reference should look something like:

```
check => sub {
return 1 if defined $_ && exists $_->{foo};
shift->property_error(qq{Property "stuff" needs "foo"});
return 0;
},
```

property_error
: [version 0.31]

METHODS

add_build_element($type)
: [version 0.26]

Adds a new type of entry to the build process. Accepts a single string specifying its type-name. There must also be a method defined to process things of that type, e.g. if you add a build element called `'foo'`, then you must also define a method called `process_foo_files()`.

See also ''Adding new file types to the build process'' in Module::Build::Cookbook.

add_to_cleanup(@files)
: [version 0.03]

You may call `$self->add_to_cleanup(@patterns)` to tell `Module::Build` that certain files should be removed when the user performs the `Build clean` action. The arguments to the method are patterns suitable for passing to Perl's `glob()` function, specified in either Unix

format or the current machine's native format. It's usually convenient to use Unix format when you hard-code the filenames (e.g. in *Build.PL*) and the native format when the names are programmatically generated (e.g. in a testing script).

I decided to provide a dynamic method of the `$build` object, rather than just use a static list of files named in the *Build.PL*, because these static lists can get difficult to manage. I usually prefer to keep the responsibility for registering temporary files close to the code that creates them.

args()
[version 0.26]

```
my $args_href = $build->args;
my %args = $build->args;
my $arg_value = $build->args($key);
$build->args($key, $value);
```

This method is the preferred interface for retrieving the arguments passed via command line options to *Build.PL* or *Build*, minus the Module-Build specific options.

When called in a scalar context with no arguments, this method returns a reference to the hash storing all of the arguments; in an array context, it returns the hash itself. When passed a single argument, it returns the value stored in the args hash for that option key. When called with two arguments, the second argument is assigned to the args hash under the key passed as the first argument.

autosplit_file($from, `$to`)
[version 0.28]

Invokes the AutoSplit module on the `$from` file, sending the output to the `lib/auto` directory inside `$to`. `$to` is typically the `blib/` directory.

base_dir()
[version 0.14]

Returns a string containing the root-level directory of this build, i.e. where the `Build.PL` script and the `lib` directory can be found. This is usually the same as the current working directory, because the `Build` script will `chdir()` into this directory as soon as it begins execution.

build_requires()
[version 0.21]

Returns a hash reference indicating the `build_requires` prerequisites that were passed to the `new()` method.

can_action(`$action`)
Returns a reference to the method that defines `$action`, or false otherwise. This is handy for actions defined (or maybe not!) in subclasses.

[version 0.32_xx]

cbuilder()
[version 0.2809]

Returns the internal ExtUtils::CBuilder object that can be used for compiling & linking C code. If no such object is available (e.g. if the system has no compiler installed) an exception will be thrown.

check_installed_status($module, `$version`)
[version 0.11]

This method returns a hash reference indicating whether a version dependency on a certain module is satisfied. The `$module` argument is given as a string like `"Data::Dumper"` or

"perl", and the $version argument can take any of the forms described in ''requires'' above. This allows very fine-grained version checking.

The returned hash reference has the following structure:

```
{
ok => $whether_the_dependency_is_satisfied,
have => $version_already_installed,
need => $version_requested, # Same as incoming $version argument
message => $informative_error_message,
}
```

If no version of $module is currently installed, the have value will be the string "<none>". Otherwise the have value will simply be the version of the installed module. Note that this means that if $module is installed but doesn't define a version number, the have value will be undef - this is why we don't use undef for the case when $module isn't installed at all.

This method may be called either as an object method ($build->check_installed_status($module, $version)) or as a class method (Module::Build->check_installed_status($module, $version)).

check_installed_version($module, $version)
[version 0.05]

Like *check_installed_status()*, but simply returns true or false depending on whether module $module satisfies the dependency $version.

If the check succeeds, the return value is the actual version of $module installed on the system. This allows you to do the following:

```
my $installed = $build->check_installed_version('DBI', '1.15');
if ($installed) {
print "Congratulations, version $installed of DBI is installed.\n";
} else {
die "Sorry, you must install DBI.\n";
}
```

If the check fails, we return false and set $@ to an informative error message.

If $version is any non-true value (notably zero) and any version of $module is installed, we return true. In this case, if $module doesn't define a version, or if its version is zero, we return the special value ''0 but true'', which is numerically zero, but logically true.

In general you might prefer to use check_installed_status if you need detailed information, or this method if you just need a yes/no answer.

compare_versions($v1, $op, $v2)
[version 0.28]

Compares two module versions $v1 and $v2 using the operator $op, which should be one of Perl's numeric operators like != or >= or the like. We do at least a halfway-decent job of handling versions that aren't strictly numeric, like 0.27_02, but exotic stuff will likely cause problems.

In the future, the guts of this method might be replaced with a call out to version.pm.

config($key)
config($key, $value)
config() [deprecated]
[version 0.22]

With a single argument $key, returns the value associated with that key in the Config.pm hash, including any changes the author or user has specified.

With `$key` and `$value` arguments, sets the value for future callers of `config($key)`.

With no arguments, returns a hash reference containing all such key-value pairs. This usage is deprecated, though, because it's a resource hog and violates encapsulation.

config_data($name)
config_data($name => `$value`)
[version 0.26]

With a single argument, returns the value of the configuration variable `$name`. With two arguments, sets the given configuration variable to the given value. The value may be any Perl scalar that's serializable with `Data::Dumper` For instance, if you write a module that can use a MySQL or PostgreSQL back-end, you might create configuration variables called `mysql_connect` and `postgres_connect`, and set each to an array of connection parameters for `DBI->connect()`.

Configuration values set in this way using the Module::Build object will be available for querying during the build/test process and after installation via the generated `...::ConfigData` module, as `...::ConfigData->config($name)`.

The *feature()* and `config_data()` methods represent Module::Build's main support for configuration of installed modules. See also "SAVING CONFIGURATION INFORMATION" in Module::Build::Authoring.

conflicts()
[version 0.21]

Returns a hash reference indicating the `conflicts` prerequisites that were passed to the `new()` method.

contains_pod($file) [deprecated]
[version 0.20]

[Deprecated] Please see Module::Metadata instead.

Returns true if the given file appears to contain POD documentation. Currently this checks whether the file has a line beginning with '=pod', '=head', or '=item', but the exact semantics may change in the future.

copy_if_modified(%parameters)
[version 0.19]

Takes the file in the `from` parameter and copies it to the file in the `to` parameter, or the directory in the `to_dir` parameter, if the file has changed since it was last copied (or if it doesn't exist in the new location). By default the entire directory structure of `from` will be copied into `to_dir`; an optional `flatten` parameter will copy into `to_dir` without doing so.

Returns the path to the destination file, or `undef` if nothing needed to be copied.

Any directories that need to be created in order to perform the copying will be automatically created.

The destination file is set to read-only. If the source file has the executable bit set, then the destination file will be made executable.

create_build_script()
[version 0.05]

Creates an executable script called `Build` in the current directory that will be used to execute further user actions. This script is roughly analogous (in function, not in form) to the Makefile created by `ExtUtils::MakeMaker` This method also creates some temporary data in a directory called `_build/`. Both of these will be removed when the `realclean` action is performed.

Among the files created in `_build/` is a *_build/prereqs* file containing the set of prerequisites for this distribution, as a hash of hashes. This file may be `eval()`-ed to obtain the authoritative set of prerequisites, which might be different from the contents of *META.yml* (because *Build.PL* might have set them dynamically). But fancy developers take heed: do not put any fancy custom runtime code in the *_build/prereqs* file, leave it as a static declaration containing only strings and numbers. Similarly, do not alter the structure of the internal `$self->{properties}{requires}` (etc.) data members, because that's where this data comes from.

current_action()
[version 0.28]

Returns the name of the currently-running action, such as ''build'' or ''test''. This action is not necessarily the action that was originally invoked by the user. For example, if the user invoked the ''test'' action, *current_action()* would initially return ''test''. However, action ''test'' depends on action ''code'', so *current_action()* will return ''code'' while that dependency is being executed. Once that action has completed, *current_action()* will again return ''test''.

If you need to know the name of the original action invoked by the user, see ''*invoked_action()*'' below.

depends_on(@actions)
[version 0.28]

Invokes the named action or list of actions in sequence. Using this method is preferred to calling the action explicitly because it performs some internal record-keeping, and it ensures that the same action is not invoked multiple times (note: in future versions of Module::Build it's conceivable that this run-only-once mechanism will be changed to something more intelligent).

Note that the name of this method is something of a misnomer; it should really be called something like `invoke_actions_unless_already_invoked()` or something, but for better or worse (perhaps better!) we were still thinking in `make`-like dependency terms when we created this method.

See also *dispatch()*. The main distinction between the two is that `depends_on()` is meant to call an action from inside another action, whereas `dispatch()` is meant to set the very top action in motion.

dir_contains($first_dir, `$second_dir`)
[version 0.28]

Returns true if the first directory logically contains the second directory. This is just a convenience function because `File::Spec` doesn't really provide an easy way to figure this out (but `Path::Class` does...).

dispatch($action, `%args`)
[version 0.03]

Invokes the build action `$action`. Optionally, a list of options and their values can be passed in. This is equivalent to invoking an action at the command line, passing in a list of options.

Custom options that have not been registered must be passed in as a hash reference in a key named ''args'':

```
$build->dispatch('foo', verbose => 1, args => { my_option => 'value' });
```

This method is intended to be used to programmatically invoke build actions, e.g. by applications controlling Module::Build-based builds rather than by subclasses.

See also *depends_on()*. The main distinction between the two is that `depends_on()` is meant

to call an action from inside another action, whereas `dispatch()` is meant to set the very top action in motion.

dist_dir()
[version 0.28]

Returns the name of the directory that will be created during the `dist` action. The name is derived from the `dist_name` and `dist_version` properties.

dist_name()
[version 0.21]

Returns the name of the current distribution, as passed to the `new()` method in a `dist_name` or modified `module_name` parameter.

dist_version()
[version 0.21]

Returns the version of the current distribution, as determined by the `new()` method from a `dist_version`, `dist_version_from`, or `module_name` parameter.

do_system($cmd, `@args`)
[version 0.21]

This is a fairly simple wrapper around Perl's `system()` built-in command. Given a command and an array of optional arguments, this method will print the command to `STDOUT`, and then execute it using Perl's `system()`. It returns true or false to indicate success or failure (the opposite of how `system()` works, but more intuitive).

Note that if you supply a single argument to `do_system()`, it will/may be processed by the system's shell, and any special characters will do their special things. If you supply multiple arguments, no shell will get involved and the command will be executed directly.

extra_compiler_flags()
extra_compiler_flags(@flags)
[version 0.25]

Set or retrieve the extra compiler flags. Returns an arrayref of flags.

extra_linker_flags()
extra_linker_flags(@flags)
[version 0.25]

Set or retrieve the extra linker flags. Returns an arrayref of flags.

feature($name)
feature($name => `$value`)
[version 0.26]

With a single argument, returns true if the given feature is set. With two arguments, sets the given feature to the given boolean value. In this context, a "feature" is any optional functionality of an installed module. For instance, if you write a module that could optionally support a MySQL or PostgreSQL backend, you might create features called `mysql_support` and `postgres_support`, and set them to true/false depending on whether the user has the proper databases installed and configured.

Features set in this way using the Module::Build object will be available for querying during the build/test process and after installation via the generated `...::ConfigData` module, as `...::ConfigData->feature($name)`.

The `feature()` and `config_data()` methods represent Module::Build's main support for configuration of installed modules. See also "SAVING CONFIGURATION INFORMATION" in Module::Build::Authoring.

fix_shebang_line(@files)
[version 0.??]

Modify any ''shebang'' line in the specified files to use the path to the perl executable being used for the current build. Files are modified in-place. The existing shebang line must have a command that contains `perl`; arguments to the command do not count. In particular, this means that the use of `#!/usr/bin/env perl` will not be changed.

For an explanation of shebang lines, see <http://en.wikipedia.org/wiki/Shebang_%28Unix%29>.

have_c_compiler()
[version 0.21]

Returns true if the current system seems to have a working C compiler. We currently determine this by attempting to compile a simple C source file and reporting whether the attempt was successful.

install_base_relpaths()
install_base_relpaths($type)
install_base_relpaths($type => `$path`)
[version 0.28]

Set or retrieve the relative paths that are appended to `install_base` for any installable element. This is useful if you want to set the relative install path for custom build elements.

With no argument, it returns a reference to a hash containing all elements and their respective values. This hash should not be modified directly; use the multiple argument below form to change values.

The single argument form returns the value associated with the element `$type`.

The multiple argument form allows you to set the paths for element types. `$value` must be a relative path using Unix-like paths. (A series of directories separated by slashes, e.g. `foo/bar`.) The return value is a localized path based on `$value`.

Assigning the value `undef` to an element causes it to be removed.

install_destination($type)
[version 0.28]

Returns the directory in which items of type `$type` (e.g. `lib`, `arch`, `bin`, or anything else returned by the ''*install_types()*'' method) will be installed during the `install` action. Any settings for `install_path`, `install_base`, and `prefix` are taken into account when determining the return value.

install_path()
install_path($type)
install_path($type => `$path`)
[version 0.28]

Set or retrieve paths for specific installable elements. This is useful when you want to examine any explicit install paths specified by the user on the command line, or if you want to set the install path for a specific installable element based on another attribute like `install_base()`.

With no argument, it returns a reference to a hash containing all elements and their respective values. This hash should not be modified directly; use the multiple argument below form to change values.

The single argument form returns the value associated with the element `$type`.

The multiple argument form allows you to set the paths for element types. The supplied

`$path` should be an absolute path to install elements of `$type`. The return value is `$path`.

Assigning the value `undef` to an element causes it to be removed.

install_types()
[version 0.28]

Returns a list of installable types that this build knows about. These types each correspond to the name of a directory in *blib/*, and the list usually includes items such as `lib`, `arch`, `bin`, `script`, `libdoc`, `bindoc`, and if HTML documentation is to be built, `libhtml` and `binhtml`. Other user-defined types may also exist.

invoked_action()
[version 0.28]

This is the name of the original action invoked by the user. This value is set when the user invokes *Build.PL*, the *Build* script, or programmatically through the *dispatch()* method. It does not change as sub-actions are executed as dependencies are evaluated.

To get the name of the currently executing dependency, see "*current_action()*" above.

notes()
notes($key)
notes($key => `$value`)
[version 0.20]

The `notes()` value allows you to store your own persistent information about the build, and to share that information among different entities involved in the build. See the example in the `current()` method.

The `notes()` method is essentially a glorified hash access. With no arguments, `notes()` returns the entire hash of notes. With one argument, `notes($key)` returns the value associated with the given key. With two arguments, `notes($key, $value)` sets the value associated with the given key to `$value` and returns the new value.

The lifetime of the `notes` data is for "a build" - that is, the `notes` hash is created when `perl Build.PL` is run (or when the `new()` method is run, if the Module::Build Perl API is being used instead of called from a shell), and lasts until `perl Build.PL` is run again or the `clean` action is run.

orig_dir()
[version 0.28]

Returns a string containing the working directory that was in effect before the *Build* script *chdir()*-ed into the `base_dir`. This might be useful for writing wrapper tools that might need to *chdir()* back out.

os_type()
[version 0.04]

If you're subclassing Module::Build and some code needs to alter its behavior based on the current platform, you may only need to know whether you're running on Windows, Unix, MacOS, VMS, etc., and not the fine-grained value of Perl's `$O` variable. The `os_type()` method will return a string like `Windows`, `Unix`, `MacOS`, `VMS`, or whatever is appropriate. If you're running on an unknown platform, it will return `undef` - there shouldn't be many unknown platforms though.

is_vmsish()
is_windowsish()
is_unixish()
Convenience functions that return a boolean value indicating whether this platform behaves respectively like VMS, Windows, or Unix. For arbitrary reasons other platforms don't get their own such functions, at least not yet.

prefix_relpaths()
prefix_relpaths($installdirs)
prefix_relpaths($installdirs, `$type`)
prefix_relpaths($installdirs, `$type` => `$path`)
[version 0.28]

Set or retrieve the relative paths that are appended to `prefix` for any installable element. This is useful if you want to set the relative install path for custom build elements.

With no argument, it returns a reference to a hash containing all elements and their respective values as defined by the current `installdirs` setting.

With a single argument, it returns a reference to a hash containing all elements and their respective values as defined by `$installdirs`.

The hash returned by the above calls should not be modified directly; use the three-argument below form to change values.

The two argument form returns the value associated with the element `$type`.

The multiple argument form allows you to set the paths for element types. `$value` must be a relative path using Unix-like paths. (A series of directories separated by slashes, e.g. `foo/bar`.) The return value is a localized path based on `$value`.

Assigning the value `undef` to an element causes it to be removed.

get_metadata()
[version 0.36]

This method returns a hash reference of metadata that can be used to create a YAML datastream. It is provided for authors to override or customize the fields of *META.yml*. E.g.

```
package My::Builder;
use base 'Module::Build';

sub get_metadata {
my $self, @args = @_;
my $data = $self->SUPER::get_metadata(@args);
$data->{custom_field} = 'foo';
return $data;
}
```

Valid arguments include:

- `fatal` — indicates whether missing required metadata fields should be a fatal error or not. For META creation, it generally should, but for MYMETA creation for end-users, it should not be fatal.
- `auto` — indicates whether any necessary configure_requires should be automatically added. This is used in META creation.

This method is a wrapper around the old prepare_metadata API now that we no longer use YAML::Node to hold metadata.

prepare_metadata() [deprecated]
[version 0.36]

[Deprecated] As of 0.36, authors should use `get_metadata` instead. This method is preserved for backwards compatibility only.

It takes three positional arguments: a hashref (to which metadata will be added), an optional arrayref (to which metadata keys will be added in order if the arrayref exists), and a hashref of arguments (as provided to get_metadata). The latter argument is new as of 0.36. Earlier

versions are always fatal on errors.

Prior to version 0.36, this method took a YAML::Node as an argument to hold assembled metadata.

prereq_failures()
[version 0.11]

Returns a data structure containing information about any failed prerequisites (of any of the types described above), or `undef` if all prerequisites are met.

The data structure returned is a hash reference. The top level keys are the type of prerequisite failed, one of ''requires'', ''build_requires'', ''conflicts'', or ''recommends''. The associated values are hash references whose keys are the names of required (or conflicting) modules. The associated values of those are hash references indicating some information about the failure. For example:

```
{
have => '0.42',
need => '0.59',
message => 'Version 0.42 is installed, but we need version 0.59',
}
```

or

```
{
have => '<none>',
need => '0.59',
message => 'Prerequisite Foo isn't installed',
}
```

This hash has the same structure as the hash returned by the `check_installed_status()` method, except that in the case of ''conflicts'' dependencies we change the ''need'' key to ''conflicts'' and construct a proper message.

Examples:

```
# Check a required dependency on Foo::Bar
if ( $build->prereq_failures->{requires}{Foo::Bar} ) { ...

# Check whether there were any failures
if ( $build->prereq_failures ) { ...

# Show messages for all failures
my $failures = $build->prereq_failures;
while (my ($type, $list) = each %$failures) {
while (my ($name, $hash) = each %$list) {
print "Failure for $name: $hash->{message}\n";
}
}
```

prereq_data()
[version 0.32]

Returns a reference to a hash describing all prerequisites. The keys of the hash will be the various prerequisite types ('requires', 'build_requires', 'test_requires', 'configure_requires', 'recommends', or 'conflicts') and the values will be references to hashes of module names and version numbers. Only prerequisites types that are defined will be included. The `prereq_data` action is just a thin wrapper around the `prereq_data()` method and dumps the hash as a string that can be loaded using `eval()`.

prereq_report()
[version 0.28]

Returns a human-readable (table-form) string showing 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. The `prereq_report` action is just a thin wrapper around the `prereq_report()` method.

prompt($message, `$default`)
[version 0.12]

Asks the user a question and returns their response as a string. The first argument specifies the message to display to the user (for example, `"Where do you keep your money?"`). The second argument, which is optional, specifies a default answer (for example, `"wallet"`). The user will be asked the question once.

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.

To prevent automated processes from blocking, the user must either set PERL_MM_USE_DEFAULT or attach something to STDIN (this can be a pipe/file containing a scripted set of answers or /dev/null.)

If no `$default` is provided an empty string will be used instead. In non-interactive mode, the absence of `$default` is an error (though explicitly passing `undef()` as the default is valid as of 0.27.)

This method may be called as a class or object method.

recommends()
[version 0.21]

Returns a hash reference indicating the `recommends` prerequisites that were passed to the `new()` method.

requires()
[version 0.21]

Returns a hash reference indicating the `requires` prerequisites that were passed to the `new()` method.

rscan_dir($dir, `$pattern`)
[version 0.28]

Uses `File::Find` to traverse the directory `$dir`, returning a reference to an array of entries matching `$pattern`. `$pattern` may either be a regular expression (using `qr//` or just a plain string), or a reference to a subroutine that will return true for wanted entries. If `$pattern` is not given, all entries will be returned.

Examples:

```
# All the *.pm files in lib/
$m->rscan_dir('lib', qr/\.pm$/)

# All the files in blib/ that aren't *.html files
$m->rscan_dir('blib', sub {-f $_ and not /\.html$/});

# All the files in t/
$m->rscan_dir('t');
```

runtime_params()
runtime_params($key)
[version 0.28]

The `runtime_params()` method stores the values passed on the command line for valid properties (that is, any command line options for which `valid_property()` returns a true value). The value on the command line may override the default value for a property, as well as any value specified in a call to `new()`. This allows you to programmatically tell if `perl Build.PL` or any execution of `./Build` had command line options specified that override valid properties.

The `runtime_params()` method is essentially a glorified read-only hash. With no arguments, `runtime_params()` returns the entire hash of properties specified on the command line. With one argument, `runtime_params($key)` returns the value associated with the given key.

The lifetime of the `runtime_params` data is for ''a build'' - that is, the `runtime_params` hash is created when `perl Build.PL` is run (or when the `new()` method is called, if the Module::Build Perl API is being used instead of called from a shell), and lasts until `perl Build.PL` is run again or the `clean` action is run.

script_files()
[version 0.18]

Returns a hash reference whose keys are the perl script files to be installed, if any. This corresponds to the `script_files` parameter to the `new()` method. With an optional argument, this parameter may be set dynamically.

For backward compatibility, the `scripts()` method does exactly the same thing as `script_files()`. `scripts()` is deprecated, but it will stay around for several versions to give people time to transition.

up_to_date($source_file, `$derived_file`)
up_to_date(@source_files, @derived_files)
[version 0.20]

This method can be used to compare a set of source files to a set of derived files. If any of the source files are newer than any of the derived files, it returns false. Additionally, if any of the derived files do not exist, it returns false. Otherwise it returns true.

The arguments may be either a scalar or an array reference of file names.

y_n($message, `$default`)
[version 0.12]

Asks the user a yes/no question using `prompt()` and returns true or false accordingly. The user will be asked the question repeatedly until they give an answer that looks like ''yes'' or ''no''.

The first argument specifies the message to display to the user (for example, `"Shall I invest your money for you?"`), and the second argument specifies the default answer (for example, `"y"`).

Note that the default is specified as a string like `"y"` or `"n"`, and the return value is a Perl boolean value like 1 or 0. I thought about this for a while and this seemed like the most useful way to do it.

This method may be called as a class or object method.

Autogenerated Accessors
In addition to the aforementioned methods, there are also some get/set accessor methods for the following properties:

PL_files()
allow_mb_mismatch()
allow_pureperl()
auto_configure_requires()
autosplit()
base_dir()
bindoc_dirs()
blib()
build_bat()
build_class()
build_elements()
build_requires()
build_script()
bundle_inc()
bundle_inc_preload()
c_source()
config_dir()
configure_requires()
conflicts()
cpan_client()
create_license()
create_makefile_pl()
create_packlist()
create_readme()
debug()
debugger()
destdir()
dynamic_config()
extra_manify_args()
get_options()
html_css()
include_dirs()
install_base()
installdirs()
libdoc_dirs()
license()
magic_number()
mb_version()
meta_add()
meta_merge()
metafile()
metafile2()
module_name()
mymetafile()
mymetafile2()
needs_compiler()
orig_dir()
perl()
pm_files()
pod_files()
pollute()
prefix()

prereq_action_types()
program_name()
pureperl_only()
quiet()
recommends()
recurse_into()
recursive_test_files()
requires()
scripts()
sign()
tap_harness_args()
test_file_exts()
test_requires()
use_rcfile()
use_tap_harness()
verbose()
xs_files()

MODULE METADATA

If you would like to add other useful metadata, `Module::Build` supports this with the `meta_add` and `meta_merge` arguments to "*new()*". The authoritative list of supported metadata can be found at CPAN::Meta::Spec but for convenience - here are a few of the more useful ones:

keywords
: For describing the distribution using keyword (or "tags") in order to make CPAN.org indexing and search more efficient and useful.

resources
: A list of additional resources available for users of the distribution. This can include links to a homepage on the web, a bug tracker, the repository location, and even a subscription page for the distribution mailing list.

AUTHOR

Ken Williams <kwilliams@cpan.org>

COPYRIGHT

SEE ALSO

perl(1), Module::Build(3), Module::Build::Authoring(3), Module::Build::Cookbook(3), ExtUtils::MakeMaker(3)

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