

## NAME

ExtUtils::MM\_Any - Platform-agnostic MM methods

## SYNOPSIS

FOR INTERNAL USE ONLY!

```
package ExtUtils::MM_SomeOS;

# Temporarily, you have to subclass both. Put MM_Any first.
require ExtUtils::MM_Any;
require ExtUtils::MM_Unix;
@ISA = qw(ExtUtils::MM_Any ExtUtils::Unix);
```

## DESCRIPTION

### FOR INTERNAL USE ONLY!

ExtUtils::MM\_Any is a superclass for the ExtUtils::MM\_\* set of modules. It contains methods which are either inherently cross-platform or are written in a cross-platform manner.

Subclass off of ExtUtils::MM\_Any *and* ExtUtils::MM\_Unix. This is a temporary solution.

### THIS MAY BE TEMPORARY!

## METHODS

Any methods marked *Abstract* must be implemented by subclasses.

### Cross-platform helper methods

These are methods which help writing cross-platform code.

#### os\_flavor Abstract

```
my @os_flavor = $mm->os_flavor;
```

@os\_flavor is the style of operating system this is, usually corresponding to the MM\_\*.pm file we're using.

The first element of @os\_flavor is the major family (ie. Unix, Windows, VMS, OS/2, etc...) and the rest are sub families.

Some examples:

Cygwin98	('Unix', 'Cygwin', 'Cygwin9x')
Windows	('Win32')
Win98	('Win32', 'Win9x')
Linux	('Unix', 'Linux')
MacOS X	('Unix', 'Darwin', 'MacOS', 'MacOS X')
OS/2	('OS/2')

This is used to write code for styles of operating system. See os\_flavor\_is() for use.

#### os\_flavor\_is

```
my $is_this_flavor = $mm->os_flavor_is($this_flavor);
my $is_this_flavor = $mm->os_flavor_is(@one_of_these_flavors);
```

Checks to see if the current operating system is one of the given flavors.

This is useful for code like:

```
if( $mm->os_flavor_is('Unix') ) {  
    $out = `foo 2>&l`;  
}  
else {  
    $out = `foo`;  
}
```

### can\_load\_xs

```
my $can_load_xs = $self->can_load_xs;
```

Returns true if we have the ability to load XS.

This is important because miniperl, used to build XS modules in the core, can not load XS.

### can\_run

```
use ExtUtils::MM;  
my $runnable = MM->can_run($Config{make});
```

If called in a scalar context it will return the full path to the binary you asked for if it was found, or undef if it was not.

If called in a list context, it will return a list of the full paths to instances of the binary where found in PATH, or an empty list if it was not found.

Copied from *IPC::Cmd*, but modified into a method (and removed \$INSTANCES capability).

### can\_redirect\_error

```
$userredirect = MM->can_redirect_error;
```

True if on an OS where qx operator (or backticks) can redirect STDERR onto STDOUT.

### is\_make\_type

```
my $is_dmake = $self->is_make_type('dmake');
```

Returns true if <\$self-make> is the given type; possibilities are:

```
gmake      GNU make  
dmake  
nmake  
bsdmake    BSD pmake-derived
```

### can\_dep\_space

```
my $can_dep_space = $self->can_dep_space;
```

Returns true if make can handle (probably by quoting) dependencies that contain a space. Currently known true for GNU make, false for BSD pmake derivative.

### quote\_dep

```
$text = $mm->quote_dep($text);
```

Method that protects Makefile single-value constants (mainly filenames), so that make will still treat them as single values even if they inconveniently have spaces in. If the make program being used cannot achieve such protection and the given text would need it, throws an exception.

## split\_command

```
my @cmds = $MM->split_command($cmd, @args);
```

Most OS have a maximum command length they can execute at once. Large modules can easily generate commands well past that limit. Its necessary to split long commands up into a series of shorter commands.

`split_command` will return a series of `@cmds` each processing part of the args. Collectively they will process all the arguments. Each individual line in `@cmds` will not be longer than the `$self->max_exec_len` being careful to take into account macro expansion.

`$cmd` should include any switches and repeated initial arguments.

If no `@args` are given, no `@cmds` will be returned.

Pairs of arguments will always be preserved in a single command, this is a heuristic for things like `pm_to_blib` and `pod2man` which work on pairs of arguments. This makes things like this safe:

```
$self->split_command($cmd, %pod2man);
```

## make\_type

Returns a suitable string describing the type of makefile being written.

## stashmeta

```
my @recipelines = $MM->stashmeta($text, $file);
```

Generates a set of `@recipelines` which will result in the literal `$text` ending up in literal `$file` when the recipe is executed. Call it once, with all the text you want in `$file`. Make macros will not be expanded, so the locations will be fixed at configure-time, not at build-time.

## echo

```
my @commands = $MM->echo($text);  
my @commands = $MM->echo($text, $file);  
my @commands = $MM->echo($text, $file, \%opts);
```

Generates a set of `@commands` which print the `$text` to a `$file`.

If `$file` is not given, output goes to STDOUT.

If `$opts{append}` is true the `$file` will be appended to rather than overwritten. Default is to overwrite.

If `$opts{allow_variables}` is true, make variables of the form `$(...)` will not be escaped. Other `$` will. Default is to escape all `$`.

Example of use:

```
my $make = join '', map "\t$_\n", $MM->echo($text, $file);
```

## wraplist

```
my $args = $mm->wraplist(@list);
```

Takes an array of items and turns them into a well-formatted list of arguments. In most cases this is simply something like:

```
FOO \  
BAR \  
BAZ
```

**maketext\_filter**

```
my $filter_make_text = $mm->maketext_filter($make_text);
```

The text of the Makefile is run through this method before writing to disk. It allows systems a chance to make portability fixes to the Makefile.

By default it does nothing.

This method is protected and not intended to be called outside of MakeMaker.

**cd Abstract**

```
my $subdir_cmd = $MM->cd($subdir, @cmds);
```

This will generate a make fragment which runs the @cmds in the given \$dir. The rough equivalent to this, except cross platform.

```
cd $subdir && $cmd
```

Currently \$dir can only go down one level. "foo" is fine. "foo/bar" is not. "../foo" is right out.

The resulting \$subdir\_cmd has no leading tab nor trailing newline. This makes it easier to embed in a make string. For example.

```
my $make = sprintf <<'CODE', $subdir_cmd;
foo :
    $(ECHO) what
    %s
    $(ECHO) mouche
CODE
```

**oneliner Abstract**

```
my $oneliner = $MM->oneliner($perl_code);
my $oneliner = $MM->oneliner($perl_code, \@switches);
```

This will generate a perl one-liner safe for the particular platform you're on based on the given \$perl\_code and @switches (a -e is assumed) suitable for using in a make target. It will use the proper shell quoting and escapes.

\$(PERLRUN) will be used as perl.

Any newlines in \$perl\_code will be escaped. Leading and trailing newlines will be stripped. Makes this idiom much easier:

```
my $code = $MM->oneliner(<<'CODE', [...switches...]);
some code here
another line here
CODE
```

Usage might be something like:

```
# an echo emulation
$oneliner = $MM->oneliner('print "Foo\n"');
$make = '$oneliner > somefile';
```

Dollar signs in the \$perl\_code will be protected from make using the `quote_literal` method, unless they are recognised as being a make variable, `$(varname)`, in which case they will be left for

make to expand. Remember to quote make macros else it might be used as a bareword. For example:

```
# Assign the value of the $(VERSION_FROM) make macro to $vf.  
$oneline = $MM->oneline('$vf = "$(VERSION_FROM)"');
```

Its currently very simple and may be expanded sometime in the future to include more flexible code and switches.

### **quote\_literal Abstract**

```
my $safe_text = $MM->quote_literal($text);  
my $safe_text = $MM->quote_literal($text, \%options);
```

This will quote `$text` so it is interpreted literally in the shell.

For example, on Unix this would escape any single-quotes in `$text` and put single-quotes around the whole thing.

If `$options{allow_variables}` is true it will leave `'$(FOO)'` make variables untouched. If false they will be escaped like any other `$`. Defaults to true.

### **escape\_dollarsigns**

```
my $escaped_text = $MM->escape_dollarsigns($text);
```

Escapes stray `$` so they are not interpreted as make variables.

It lets by `$(...)`.

### **escape\_all\_dollarsigns**

```
my $escaped_text = $MM->escape_all_dollarsigns($text);
```

Escapes all `$` so they are not interpreted as make variables.

### **escape\_newlines Abstract**

```
my $escaped_text = $MM->escape_newlines($text);
```

Shell escapes newlines in `$text`.

### **max\_exec\_len Abstract**

```
my $max_exec_len = $MM->max_exec_len;
```

Calculates the maximum command size the OS can exec. Effectively, this is the max size of a shell command line.

### **make**

```
my $make = $MM->make;
```

Returns the make variant we're generating the Makefile for. This attempts to do some normalization on the information from `%Config` or the user.

### **Targets**

These are methods which produce make targets.

**all\_target**

Generate the default target 'all'.

**blibdirs\_target**

```
my $make_frag = $mm->blibdirs_target;
```

Creates the blibdirs target which creates all the directories we use in blib/.

The blibdirs.ts target is deprecated. Depend on blibdirs instead.

**clean (o)**

Defines the clean target.

**clean\_subdirs\_target**

```
my $make_frag = $MM->clean_subdirs_target;
```

Returns the clean\_subdirs target. This is used by the clean target to call clean on any subdirectories which contain Makefiles.

**dir\_target**

```
my $make_frag = $mm->dir_target(@directories);
```

Generates targets to create the specified directories and set its permission to PERM\_DIR.

Because depending on a directory to just ensure it exists doesn't work too well (the modified time changes too often) `dir_target()` creates a `.exists` file in the created directory. It is this you should depend on. For portability purposes you should use the `$(DIRFILESEP)` macro rather than a `/` to separate the directory from the file.

```
yourdirectory$(DIRFILESEP).exists
```

**distdir**

Defines the scratch directory target that will hold the distribution before tar-ing (or shar-ing).

**dist\_test**

Defines a target that produces the distribution in the scratch directory, and runs 'perl Makefile.PL; make ;make test' in that subdirectory.

**xs\_dlsyms\_ext**

Returns file-extension for `xs_make_dlsyms` method's output file, including any "." character.

**xs\_dlsyms\_extra**

Returns any extra text to be prepended to the `$extra` argument of `xs_make_dlsyms`.

**xs\_dlsyms\_iterator**

Iterates over necessary shared objects, calling `xs_make_dlsyms` method for each with appropriate arguments.

**xs\_make\_dlsyms**

```
$self->xs_make_dlsyms(  
    \%attrs, # hashref from %attrs in caller  
    "$self->{BASEEXT}.def", # output file for Makefile target  
    'Makefile.PL', # dependency  
    $self->{NAME}, # shared object's "name"  
    $self->{DLBASE}, # last ::-separated part of name
```

```
$attrs{DL_FUNCS} || $self->{DL_FUNCS} || {}, # various params
$attrs{FUNCLIST} || $self->{FUNCLIST} || [],
$attrs{IMPORTS} || $self->{IMPORTS} || {},
$attrs{DL_VARS} || $self->{DL_VARS} || [],
# optional extra param that will be added as param to Mksymlists
);
```

Utility method that returns Makefile snippet to call Mksymlists.

### **dynamic (o)**

Defines the dynamic target.

### **makemakerdflt\_target**

```
my $make_frag = $mm->makemakerdflt_target
```

Returns a make fragment with the makemakerdflt\_target specified. This target is the first target in the Makefile, is the default target and simply points off to 'all' just in case any make variant gets confused or something gets snuck in before the real 'all' target.

### **manifypods\_target**

```
my $manifypods_target = $self->manifypods_target;
```

Generates the manifypods target. This target generates man pages from all POD files in MAN1PODS and MAN3PODS.

### **metafile\_target**

```
my $target = $mm->metafile_target;
```

Generate the metafile target.

Writes the file META.yml (YAML encoded meta-data) and META.json (JSON encoded meta-data) about the module in the distdir. The format follows Module::Build's as closely as possible.

### **\_fix\_metadata\_before\_conversion**

```
$mm->_fix_metadata_before_conversion( \%metadata );
```

Fixes errors in the metadata before it's handed off to CPAN::Meta for conversion. This hopefully results in something that can be used further on, no guarantee is made though.

### **\_sort\_pairs**

```
my @pairs = _sort_pairs($sort_sub, \%hash);
```

Sorts the pairs of a hash based on keys ordered according to \$sort\_sub.

### **metafile\_data**

```
my $metadata_hashref = $mm->metafile_data(\%meta_add, \%meta_merge);
```

Returns the data which MakeMaker turns into the META.yml file and the META.json file. It is always in version 2.0 of the format.

Values of %meta\_add will overwrite any existing metadata in those keys. %meta\_merge will be merged with them.

**\_dump\_hash**

```
$yaml = _dump_hash(\%options, %hash);
```

Implements a fake YAML dumper for a hash given as a list of pairs. No quoting/escaping is done. Keys are supposed to be strings. Values are undef, strings, hash refs or array refs of strings.

Supported options are:

```
delta => STR - indentation delta
use_header => BOOL - whether to include a YAML header
indent => STR - a string of spaces
                default: ''

max_key_length => INT - maximum key length used to align
                    keys and values of the same hash
                default: 20
key_sort => CODE - a sort sub
                It may be undef, which means no sorting by keys
                default: sub { lc $a cmp lc $b }

customs => HASH - special options for certain keys
                (whose values are hashes themselves)
                may contain: max_key_length, key_sort, customs
```

**metafile\_file**

```
my $meta_yaml = $mm->metafile_file(@metadata_pairs);
```

Turns the @metadata\_pairs into YAML.

This method does not implement a complete YAML dumper, being limited to dump a hash with values which are strings, undef's or nested hashes and arrays of strings. No quoting/escaping is done.

**distmeta\_target**

```
my $make_frag = $mm->distmeta_target;
```

Generates the distmeta target to add META.yml and META.json to the MANIFEST in the distdir.

**mymeta**

```
my $mymeta = $mm->mymeta;
```

Generate MYMETA information as a hash either from an existing CPAN Meta file (META.json or META.yml) or from internal data.

**write\_mymeta**

```
$self->write_mymeta( $mymeta );
```

Write MYMETA information to MYMETA.json and MYMETA.yml.

**realclean (o)**

Defines the realclean target.

**realclean\_subdirs\_target**

```
my $make_frag = $MM->realclean_subdirs_target;
```



Returns the `realclean_subdirs` target. This is used by the `realclean` target to call `realclean` on any subdirectories which contain Makefiles.

**signature\_target**

```
my $target = $mm->signature_target;
```

Generate the signature target.

Writes the file SIGNATURE with "cpansign -s".

**distsignature\_target**

```
my $make_frag = $mm->distsignature_target;
```

Generates the `distsignature` target to add SIGNATURE to the MANIFEST in the `distdir`.

**special\_targets**

```
my $make_frag = $mm->special_targets
```

Returns a make fragment containing any targets which have special meaning to make. For example, `.SUFFIXES` and `.PHONY`.

**Init methods**

Methods which help initialize the MakeMaker object and macros.

**init\_ABSTRACT**

```
$mm->init_ABSTRACT
```

**init\_INST**

```
$mm->init_INST;
```

Called by `init_main`. Sets up all `INST_*` variables except those related to XS code. Those are handled in `init_xs`.

**init\_INSTALL**

```
$mm->init_INSTALL;
```

Called by `init_main`. Sets up all `INSTALL_*` variables (except `INSTALLDIRS`) and `*PREFIX`.

**init\_INSTALL\_from\_PREFIX**

```
$mm->init_INSTALL_from_PREFIX;
```

**init\_from\_INSTALL\_BASE**

```
$mm->init_from_INSTALL_BASE
```

**init\_VERSION Abstract**

```
$mm->init_VERSION
```

Initialize macros representing versions of MakeMaker and other tools

MAKEMAKER: path to the MakeMaker module.

MM\_VERSION: ExtUtils::MakeMaker Version

MM\_REVISION: ExtUtils::MakeMaker version control revision (for backwards compat)

VERSION: version of your module

VERSION\_MACRO: which macro represents the version (usually 'VERSION')

VERSION\_SYM: like version but safe for use as an RCS revision number

DEFINE\_VERSION: -D line to set the module version when compiling

XS\_VERSION: version in your .xs file. Defaults to \$(VERSION)

XS\_VERSION\_MACRO: which macro represents the XS version.

XS\_DEFINE\_VERSION: -D line to set the xs version when compiling.

Called by init\_main.

### init\_tools

```
$MM->init_tools();
```

Initializes the simple macro definitions used by tools\_other() and places them in the \$MM object. These use conservative cross platform versions and should be overridden with platform specific versions for performance.

Defines at least these macros.

Macro	Description
NOOP	Do nothing
NOECHO	Tell make not to display the command itself
SHELL	Program used to run shell commands
ECHO	Print text adding a newline on the end
RM_F	Remove a file
RM_RF	Remove a directory
TOUCH	Update a file's timestamp
TEST_F	Test for a file's existence
TEST_S	Test the size of a file
CP	Copy a file
CP_NONEMPTY	Copy a file if it is not empty
MV	Move a file
CHMOD	Change permissions on a file
FALSE	Exit with non-zero
TRUE	Exit with zero
UMASK_NULL	Nullify umask
DEV_NULL	Suppress all command output

### init\_others

```
$MM->init_others();
```

Initializes the macro definitions having to do with compiling and linking used by tools\_other() and places them in the \$MM object.

If there is no description, its the same as the parameter to WriteMakefile() documented in

**tools\_other** ExtUtils::MakeMaker.

```
my $make_frag = $MM->tools_other;
```

Returns a make fragment containing definitions for the macros `init_others()` initializes.

### init\_DIRFILESEP Abstract

```
$MM->init_DIRFILESEP;  
my $dirfilesep = $MM->{DIRFILESEP};
```

Initializes the DIRFILESEP macro which is the separator between the directory and filename in a filepath. ie. / on Unix, \ on Win32 and nothing on VMS.

For example:

```
# instead of $(INST_ARCHAUTODIR)/extralibs.ld  
$(INST_ARCHAUTODIR)$ (DIRFILESEP)extralibs.ld
```

Something of a hack but it prevents a lot of code duplication between MM\_\* variants.

Do not use this as a separator between directories. Some operating systems use different separators between subdirectories as between directories and filenames (for example: VOLUME:[dir1.dir2]file on VMS).

### init\_linker Abstract

```
$mm->init_linker;
```

Initialize macros which have to do with linking.

PERL\_ARCHIVE: path to libperl.a equivalent to be linked to dynamic extensions.

PERL\_ARCHIVE\_AFTER: path to a library which should be put on the linker command line *after* the external libraries to be linked to dynamic extensions. This may be needed if the linker is one-pass, and Perl includes some overrides for C RTL functions, such as malloc().

EXPORT\_LIST: name of a file that is passed to linker to define symbols to be exported.

Some OSes do not need these in which case leave it blank.

### init\_platform

```
$mm->init_platform
```

Initialize any macros which are for platform specific use only.

A typical one is the version number of your OS specific module. (ie. MM\_Unix\_VERSION or MM\_VMS\_VERSION).

### init\_MAKE

```
$mm->init_MAKE
```

Initialize MAKE from either a MAKE environment variable or \$Config{make}.

## Tools

A grab bag of methods to generate specific macros and commands.

**manifypods**

Defines targets and routines to translate the pods into manpages and put them into the INST\_\* directories.

**POD2MAN\_macro**

```
my $pod2man_macro = $self->POD2MAN_macro
```

Returns a definition for the POD2MAN macro. This is a program which emulates the pod2man utility. You can add more switches to the command by simply appending them on the macro.

Typical usage:

```
$(POD2MAN) --section=3 --perm_rw=$(PERM_RW) podfile1 man_page1 ...
```

**test\_via\_harness**

```
my $command = $mm->test_via_harness($perl, $tests);
```

Returns a \$command line which runs the given set of \$tests with Test::Harness and the given \$perl.

Used on the t/\*.t files.

**test\_via\_script**

```
my $command = $mm->test_via_script($perl, $script);
```

Returns a \$command line which just runs a single test without Test::Harness. No checks are done on the results, they're just printed.

Used for test.pl, since they don't always follow Test::Harness formatting.

**tool\_autosplit**

Defines a simple perl call that runs autosplit. May be deprecated by pm\_to\_blib soon.

**arch\_check**

```
my $arch_ok = $mm->arch_check(
    $INC{"Config.pm"},
    File::Spec->catfile($Config{archlibexp}, "Config.pm")
);
```

A sanity check that what Perl thinks the architecture is and what Config thinks the architecture is are the same. If they're not it will return false and show a diagnostic message.

When building Perl it will always return true, as nothing is installed yet.

The interface is a bit odd because this is the result of a quick refactoring. Don't rely on it.

**File::Spec wrappers**

ExtUtils::MM\_Any is a subclass of File::Spec. The methods noted here override File::Spec.

**catfile**

File::Spec <= 0.83 has a bug where the file part of catfile is not canonicalized. This override fixes that bug.

**Misc**

Methods I can't really figure out where they should go yet.

**find\_tests**

```
my $test = $mm->find_tests;
```

Returns a string suitable for feeding to the shell to return all tests in t/\*.t.

**find\_tests\_recursive**

```
my $tests = $mm->find_tests_recursive;
```

Returns a string suitable for feeding to the shell to return all tests in t/ but recursively. Equivalent to

```
my $tests = $mm->find_tests_recursive_in('t');
```

**find\_tests\_recursive\_in**

```
my $tests = $mm->find_tests_recursive_in($dir);
```

Returns a string suitable for feeding to the shell to return all tests in \$dir recursively.

**extra\_clean\_files**

```
my @files_to_clean = $MM->extra_clean_files;
```

Returns a list of OS specific files to be removed in the clean target in addition to the usual set.

**installvars**

```
my @installvars = $mm->installvars;
```

A list of all the INSTALL\* variables without the INSTALL prefix. Useful for iteration or building related variable sets.

**libscan**

```
my $wanted = $self->libscan($path);
```

Takes a path to a file or dir and returns an empty string if we don't want to include this file in the library. Otherwise it returns the the \$path unchanged.

Mainly used to exclude version control administrative directories from installation.

**platform\_constants**

```
my $make_frag = $mm->platform_constants
```

Returns a make fragment defining all the macros initialized in init\_platform() rather than put them in constants().

**post\_constants (o)**

Returns an empty string per default. Dedicated to overrides from within Makefile.PL after all constants have been defined.

**post\_initialize (o)**

Returns an empty string per default. Used in Makefile.PLs to add some chunk of text to the Makefile after the object is initialized.

**postamble (o)**

Returns an empty string. Can be used in Makefile.PLs to write some text to the Makefile at the end.

## **`_PREREQ_PRINT`**

```
$self->_PREREQ_PRINT;
```

Implements PREREQ\_PRINT.

Refactored out of MakeMaker->new().

## **`_PRINT_PREREQ`**

```
$mm->_PRINT_PREREQ;
```

Implements PRINT\_PREREQ, a slightly different version of PREREQ\_PRINT added by Redhat to, I think, support generating RPMs from Perl modules.

Should not include BUILD\_REQUIRES as RPMs do not include them.

Refactored out of MakeMaker->new().

## **`_perl_header_files`**

```
my $perl_header_files= $self->_perl_header_files;
```

returns a sorted list of header files as found in PERL\_SRC or \$archlibexp/CORE.

Used by perldepend() in MM\_Unix and MM\_VMS via \_perl\_header\_files\_fragment()

## **`_perl_header_files_fragment ($o, $separator)`**

```
my $perl_header_files_fragment= $self->_perl_header_files_fragment("/");
```

return a Makefile fragment which holds the list of perl header files which XS code depends on \$(PERL\_INC), and sets up the dependency for the \$(OBJECT) file.

The \$separator argument defaults to "". MM\_VMS will set it to "" and MM\_UNIX to "/" in perldepend(). This reason child subclasses need to control this is that in VMS the \$(PERL\_INC) directory will already have delimiters in it, but in UNIX \$(PERL\_INC) will need a slash between it and the filename. Hypothetically win32 could use "\\" (but it doesn't need to).

## **AUTHOR**

Michael G Schwern <schwern@pobox.com> and the denizens of makemaker@perl.org with code from ExtUtils::MM\_Unix and ExtUtils::MM\_Win32.