7.2.18 ExtUtils::MakeMaker - Create a Makefile for a Perl Extension

```
use ExtUtils::MakeMaker;
WriteMakefile( ATTRIBUTE => VALUE, ... );

# which internally is really more like...
%att = (ATTRIBUTE => VALUE, ...);
MM->new(\%att)->flush;
```

When you build an extension to Perl, you need to have an appropriate *Makefile*[9] in the extension's source directory. And while you could conceivably write one by hand, this would be rather tedious. So you'd like a program to write it for you.

> [9] If you don't know what a *Makefile* is, or what the *make*(1) program does with one, you *really* shouldn't be reading this section. We will be assuming that you know what happens when you type a command like `make foo`.

Originally, this was done using a shell script (actually, one for each extension) called *Makefile.SH*, much like the one that writes the *Makefile* for Perl itself. But somewhere along the line, it occurred to the perl5-porters that, by the time you want to compile your extensions, there's already a bare-bones version of the Perl executable called *miniperl*, if not a fully installed *perl*. And for some strange reason, Perl programmers prefer programming in Perl to programming in shell. So they wrote MakeMaker, just so that you can write *Makefile.PL* instead of *Makefile.SH*.

MakeMaker isn't a program; it's a module (or it wouldn't be in this chapter). The module provides the routines you need; you just need to **use** the module, and then call the routines. As with any programming job, there are many degrees of freedom; but your typical *Makefile.PL* is pretty simple. For example, here's *ext/POSIX/Makefile.PL* from the Perl distribution's POSIX extension (which is by no means a trivial extension):

```
use ExtUtils::MakeMaker;
WriteMakefile(
  NAME         => 'POSIX',
```

```
    LIBS       => ["-lm -lposix -lcposix"],
    MAN3PODS   => ' ',   # Pods will be built by installman.
    XSPROTOARG => '-noprototypes',      # XXX remove later?
    VERSION_FROM => 'POSIX.pm',
);
```

Several things are apparent from this example, but the most important is that the WriteMakefile() function uses named parameters. This means that you can pass many potential parameters, but you're only required to pass the ones you want to be different from the default values. (And when we say "many", we mean "many" - there are about 75 of them. See the Attributes section later.)

As the synopsis above indicates, the WriteMakefile() function actually constructs an object. This object has attributes that are set from various sources, including the parameters you pass to the function. It's this object that actually writes your *Makefile*, meshing together the demands of your extension with the demands of the architecture on which the extension is being installed. Like many craftily crafted objects, this MakeMaker object delegates as much of its work as possible to various other subroutines and methods. Many of these may be overridden in your *Makefile.PL* if you need to do some fine tuning. (Generally you don't.)

But let's not lose track of the goal, which is to write a *Makefile* that will know how to do anything to your extension that needs doing. Now as you can imagine, the *Makefile* that MakeMaker writes is quite, er, full-featured. It's easy to get lost in all the details. If you look at the POSIX *Makefile* generated by the bit of code above, you will find a file containing about 122 macros and 77 targets. You will want to go off into a corner and curl up into a little ball, saying, "Never mind, I didn't really want to know."

Well, the fact of the matter is, you really *don't* want to know, nor do you have to. Most of these items take care of themselves - that's what MakeMaker is there for, after all. We'll lay out the various attributes and targets for you, but you can just pick and choose, like in a cafeteria. We'll talk about the *make* targets first, because they're the actions you eventually want to perform, and then work backward to the macros and attributes that feed the targets.

But before we do that, you need to know just a few more architectural features of MakeMaker to make sense of some of the things we'll say. The targets at the end of your *Makefile* depend on the macro definitions that are interpolated into them. Those macro definitions in turn come from any of several places. Depending on how you count, there are about five sources of information for these attributes. Ordered by increasing precedence and (more or less) decreasing permanence, they are:

- Platform-specific values in Perl's Config module, provided by the *Configure* program that was run when Perl was installed on this machine.

- The WriteMakefile() function call arguments in *Makefile.Pl*, supplied by the extension writer. (You saw some of those above.)

- Platform-specific hints in the extension's *hints/* directory, also provided by extension writer. We'll talk about those later.

- Overriding values from the command line for *Makefile.PL* script, supplied by the person who runs the script. These look like KEY=VALUE.

- Overriding values from the command line for *make* itself, supplied by the person who runs the *make*. These also look like KEY=VALUE.

The first four of these turn into attributes of the object we mentioned, and are eventually written out as macro definitions in your *Makefile*. In most cases, the names of the values are consistent from beginning to end. (Except that the Config database keeps the names in lowercase, as they come from Perl's *config.sh* file. The names are translated to uppercase when they become attributes of the object.) In any case, we'll tend to use the term *attributes* to mean both attributes and the *Makefile* macros derived from them.

The *Makefile.PL* and the *hints* may also provide overriding methods for the object, if merely changing an attribute isn't good enough.

The hints files are expected to be named like their counterparts in *PERL_SRC/hints*, but with a *.pl* filename extension (for example, *next_3_2.pl*), because the file consists of Perl code to be evaluated. Apart from that, the rules governing which hintsfile is chosen are the same as in *Configure*. The hintsfile is **eval**ed within a routine that is a method of our MakeMaker object, so if you want to override or create an attribute, you would say something like:

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

By and large, if your *Makefile* isn't doing what you want, you just trace back the name of the misbehaving attribute to its source, and either change it there or override it downstream.

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

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

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

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

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

If Perl has not yet been installed, then PERL_SRC can be defined as an override on the command line.

7.2.18.1 Targets

Far and away the most commonly used *make* targets are those used by the installer to install the extension. So we aim to make the normal installation very easy:

```
perl Makefile.PL  # generate the Makefile
make              # compile the extension
make test         # test the extension
make install      # install the extension
```

This assumes that the installer has dynamic linking available. If not, a couple of additional commands are also necessary:

```
make perl         # link a new perl statically with this extension
make inst_perl    # install that new perl appropriately
```

Other interesting targets in the generated *Makefile* are:

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

Now we'll talk about some of these commands, and how each of them is related to MakeMaker. So we'll not only be talking about things that happen when you invoke the *make* target, but also about what MakeMaker has to do to generate that *make* target. So brace yourself for some temporal whiplash.

7.2.18.2 Running MakeMaker

This command is the one most closely related to MakeMaker because it's the one in which you actually run MakeMaker. No temporal whiplash here. As we mentioned earlier, some of the default attribute values may be overridden by adding arguments of the form KEY=VALUE. For example:

perl Makefile.PL PREFIX=/tmp/myperl5

To get a more detailed view of what MakeMaker is doing, say:

perl Makefile.PL verbose

7.2.18.3 Making whatever is needed

A *make* command without arguments performs any compilation needed and puts any generated files into staging directories that are named by the attributes INST_LIB, INST_ARCHLIB, INST_EXE, INST_MAN1DIR, and INST_MAN3DIR. These directories default to something below *./blib* if you are *not* building below the Perl source directory. If you *are* building below the Perl source, INST_LIB and INST_ARCHLIB default to *../../lib*, and INST_EXE is not defined.

7.2.18.4 Running tests

The goal of this command is to run any regression tests supplied with the extension, so MakeMaker checks for the existence of a file named *test.pl* in the current directory and, if it exists, adds commands to the test target of the *Makefile* that will execute the script with the proper set of Perl **-I** options (since the files haven't been installed into their final location yet).

MakeMaker also checks for any files matching glob("t/*.t"). It will add commands to the test target that execute all matching files via the Test::Harness module with the **-I** switches set correctly. If you pass TEST_VERBOSE=1, the test target will run the tests verbosely.

7.2.18.5 Installing files

Once the installer has tested the extension, the various generated files need to get put into their final resting places. The install target copies the files found below each of the INST_* directories to their INSTALL* counterparts.

INST_LIB		INSTALLPRIVLIB[10]or INSTALLSITELIB[11]
INST_ARCHLIB		INSTALLARCHLIB[10]or INSTALLSITEARCH[11]
INST_EXE		INSTALLBIN
INST_MAN1DIR		INSTALLMAN1DIR
INST_MAN3DIR		INSTALLMAN3DIR

[10] if INSTALLDIRS set to "perl"

[11] if INSTALLDIRS set to "site"

The INSTALL* attributes in turn default to their %Config counterparts, $Config{installprivlib}, $Config{installarchlib}, and so on.

If you don't set INSTALLARCHLIB or INSTALLSITEARCH, MakeMaker will assume you want them to be subdirectories of INSTALLPRIVLIB and INSTALLSITELIB, respectively. The exact relationship is determined by *Configure*. But you can usually just go with the defaults for all these attributes.

The PREFIX attribute can be used to redirect all the INSTALL* attributes in one go. Here's the quickest way to install a module in a nonstandard place:

```
perl Makefile.PL PREFIX=~ \
```

The value you specify for PREFIX replaces one or more leading pathname components in all INSTALL* attributes. The prefix to be replaced is determined by the value of $Config{prefix}, which typically has a value like */usr*. (Note that the tilde expansion above is done by MakeMaker, not by *perl* or *make*.)

If the user has superuser privileges and is not working under the Andrew File System (AFS) or relatives, then the defaults for INSTALLPRIVLIB, INSTALLARCHLIB, INSTALLBIN, and so on should be appropriate.

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

If you are using AFS, you must specify the installation directories, since these most probably have changed since Perl itself was installed. Do this by issuing these commands:

```
perl Makefile.PL INSTALLSITELIB=/afs/here/today
   INSTALLBIN=/afs/there/now INSTALLMAN3DIR=/afs/for/manpages
make
```

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

7.2.18.6 Static linking of a new Perl binary

The steps above are sufficient on a system supporting dynamic loading. On systems that do not support dynamic loading, however, the extension has to be linked together statically with everything else you might want in your *perl*

executable. MakeMaker supports the linking process by creating appropriate targets in the *Makefile*. If you say:

```
make perl
```

it will produce a new *perl* binary in the current directory with all extensions linked in that can be found in INST_ARCHLIB, SITELIBEXP, and PERL_ARCHLIB. To do that, MakeMaker writes a new *Makefile*, on UNIX it is called *Makefile.aperl*, but the name may be system-dependent. When you want to force the creation of a new *perl*, we recommend that you delete this *Makefile.aperl* so the directories are searched for linkable libraries again.

The binary can be installed in the directory where Perl normally resides on your machine with:

```
make inst_perl
```

To produce a Perl binary with a different filename than *perl*, either say:

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

or say:

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

In either case, you will be asked to confirm the invocation of the inst_perl target, since this invocation is likely to overwrite your existing Perl binary in INSTALLBIN.

By default *make inst_perl* documents what has been done in the file given by $(INSTALLARCHLIB)/perllocal.pod. This behavior can be bypassed by calling *make pure_inst_perl*.

Sometimes you might want to build a statically linked Perl even though your system supports dynamic loading. In this case you may explicitly set the linktype:

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

7.2.18.7 Attributes you can set

The following attributes can be specified as arguments to WriteMakefile() or as NAME=VALUE pairs on the command line. We give examples below in the form they would appear in your *Makefile.PL*, that is, as though passed as a named parameter to WriteMakefile() (including the comma that comes after it).

C

A reference to an array of *.*c* filenames. It's initialized by doing a directory scan and by derivation from the values of the XS attribute hash. This is not currently used by MakeMaker but may be handy in *Makefile.PL*s.

CONFIG

An array reference containing a list of attributes to fetch from %Config. For example:

CONFIG => [qw(archname manext)],

defines ARCHNAME and MANEXT from *config.sh*. MakeMaker will automatically add the following values to CONFIG:

ar dlext ldflags ranlib
cc dlsrc libc sitelibexp
cccdlflags ld lib_ext sitearchexp
ccdlflags lddlflags obj_ext so

CONFIGURE

A reference to a subroutine returning a hash reference. The hash may contain further attributes, for example, {LIBS => ...}, that have to be determined by some evaluation method. Be careful, because any attributes defined this way will override hints and WriteMakefile() parameters (but not command-line arguments).

DEFINE

An attribute containing additional defines, such as -DHAVE_UNISTD_H.

DIR

A reference to an array of subdirectories containing *Makefile.PL*s. For example, SDBM_FILE has:

DIR => ['sdbm'],

MakeMaker will automatically do recursive MakeMaking if subdirectories contain *Makefile.PL* files. A separate MakeMaker class is generated for each subdirectory, so each MakeMaker object can override methods using the fake MY:: class (see below) without interfering with other MakeMaker objects. You don't even need a *Makefile.PL* in the top level directory if you pass one in via **-M** and **-e**:

perl -MExtUtils::MakeMaker -e 'WriteMakefile()'

DISTNAME

Your name for distributing the package (by *tar* file). This defaults to NAME

below.

DL_FUNCS

A reference to a hash of symbol names for routines to be made available as universal symbols. Each key/value pair consists of the package name and an array of routine names in that package. This attribute is used only under AIX (export lists) and VMS (linker options) at present. The routine names supplied will be expanded in the same way as XSUB names are expanded by the XS attribute.

The default key/value pair looks like this:

```
"$PKG" => ["boot_$PKG"]
```

For a pair of packages named RPC and NetconfigPtr, you might, for example, set it to this:

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

DL_VARS

An array of symbol names for variables to be made available as universal symbols. It's used only under AIX (export lists) and VMS (linker options) at present. Defaults to []. A typical value might look like this:

```
DL_VARS => [ qw( Foo_version Foo_numstreams Foo_tree ) ],
```

EXE_FILES

A reference to an array of executable files. The files will be copied to the INST_EXE directory. A *make realclean* command will delete them from there again.

FIRST_MAKEFILE

The name of the *Makefile* to be produced. Defaults to the contents of MAKEFILE, but can be overridden. This is used for the second *Makefile* that will be produced for the MAP_TARGET.

FULLPERL

A Perl binary able to run this extension.

H

A reference to an array of *.*h* filenames. Similar to C.

INC

Directories containing include files, in **-I** form. For example:

INC => "-I/usr/5include -I/path/to/inc",

INSTALLARCHLIB

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

INSTALLBIN

Used by *make install*, which copies files from INST_EXE to this directory.

INSTALLDIRS

Determines which of the two sets of installation directories to choose: *installprivlib* and *installarchlib* versus *installsitelib* and *installsitearch*. The first pair is chosen with INSTALLDIRS=perl, the second with INSTALLDIRS=site. The default is "site".

INSTALLMAN1DIR

This directory gets the command manpages at *make install* time. It defaults to $Config{installman1dir}.

INSTALLMAN3DIR

This directory gets the library manpages at *make install* time. It defaults to $Config{installman3dir}.

INSTALLPRIVLIB

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

INSTALLSITELIB

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

INSTALLSITEARCH

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

INST_ARCHLIB

Same as INST_LIB, but for architecture-dependent files.

INST_EXE

Directory where executable scripts should be staged during running of *make*. Defaults to ./blib/bin, just to have a dummy location during testing. *make install* will copy the files in INST_EXE to INSTALLBIN.

INST_LIB

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

INST_MAN1DIR

Directory to hold the command manpages at *make* time.

INST_MAN3DIR

Directory to hold the library manpages at *make* time

LDFROM

Defaults to $(OBJECT) and is used in the *ld*(1) command to specify what files to link/load from. (Also see dynamic_lib later for how to specify *ld* flags.)

LIBPERL_A

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

LIBS

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

For example:

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

Note that any element of the array contains a complete set of arguments for the *ld* command. So do not specify:

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

See *NDBM_File/Makefile.PL* for an example where an array is needed. If you specify a scalar as in:

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

MakeMaker will turn it into an array with one element.

LINKTYPE

"static" or "dynamic" (the latter is the default unless usedl=undef in *config.sh*). Should only be used to force static linking. (Also see linkext, later in this chapter).

MAKEAPERL

Boolean that tells MakeMaker to include the rules for making a Perl binary. This is handled automatically as a switch by MakeMaker. The user normally does not need it.

MAKEFILE

The name of the *Makefile* to be produced.

MAN1PODS

A reference to a hash of POD-containing files. MakeMaker will default this to all EXE_FILES files that include POD directives. The files listed here will be converted to manpages and installed as requested at *Configure* time.

MAN3PODS

A reference to a hash of *.pm* and *.pod* files. MakeMaker will default this to all *.pod* and any *.pm* files that include POD directives. The files listed here will be converted to manpages and installed as requested at *Configure* time.

MAP_TARGET

If it is intended that a new Perl binary be produced, this variable holds the name for that binary. Defaults to *perl*.

MYEXTLIB

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

NAME

Perl module name for this extension (for example, DBD::Oracle). This will default to the directory name, but should really be explicitly defined in the *Makefile.PL*.

NEEDS_LINKING

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

NOECHO

Governs *make*'s @ (echoing) feature. By setting NOECHO to an empty string, you can generate a *Makefile* that echos all commands. Mainly used in debugging MakeMaker itself.

NORECURS

A Boolean that inhibits the automatic descent into subdirectories (see DIR above). For example:

```
NORECURS => 1,
```

OBJECT

A string containing a list of object files, defaulting to $(BASEEXT)$(OBJ_EXT). But it can be a long string containing all object files. For example:

```
OBJECT => "tkpBind.o tkpButton.o tkpCanvas.o",
```

PERL

Perl binary for tasks that can be done by *miniperl*.

PERLMAINCC

The command line that is able to compile *perlmain.c*. Defaults to $(CC).

PERL_ARCHLIB

Same as PERL_LIB for architecture-dependent files.

PERL_LIB

The directory containing the Perl library to use.

PERL_SRC

The directory containing the Perl source code. Use of this should be avoided, since it may be undefined.

PL_FILES

A reference to hash of files to be processed as Perl programs. By default MakeMaker will turn the names of any *.*PL* files it finds (except *Makefile.PL*) into keys, and use the basenames of these files as values. For example:

```
PL_FILES => {'whatever.PL' => 'whatever'},
```

This turns into a Makefile entry resembling:

```
all :: whatever

whatever :: whatever.PL
    $(PERL) -I$(INST_ARCHLIB) -I$(INST_LIB) \
        -I$(PERL_ARCHLIB) -I$(PERL_LIB) whatever.PL
```

You'll note that there's no I/O redirection into *whatever* there. The *.PL files are expected to produce output to the target files themselves.

PM

A reference to a hash of *.pm* files and *.pl* files to be installed. For example:

```
PM => {'name_of_file.pm' => '$(INST_LIBDIR)/install_as.pm'},
```

By default this includes *.pm and *.pl. If a *lib/* subdirectory exists and is not listed in DIR (above) then any *.pm and *.pl files it contains will also be included by default. Defining PM in the *Makefile.PL* will override PMLIBDIRS.

PMLIBDIRS

A reference to an array of subdirectories that contain library files. Defaults to:

```
PMLIBDIRS => [ 'lib', '$(BASEEXT)' ],
```

The directories will be scanned and any files they contain will be installed in the corresponding location in the library. A libscan() method may be used to alter the behavior. Defining PM in the *Makefile.PL* will override PMLIBDIRS.

PREFIX

May be used to set the three INSTALL* attributes in one go (except for probably INSTALLMAN1DIR if it is not below PREFIX according to %Config). They will have PREFIX as a common directory node and will branch from that node into lib/, lib/ARCHNAME or whatever *Configure* decided at the build time of your Perl (unless you override one of them, of course).

PREREQ

A placeholder, not yet implemented. Will eventually be a hash reference: the keys of the hash are names of modules that need to be available to run this extension (for example, Fcntl for SDBM_File); the values of the hash are the desired versions of the modules.

SKIP

An array reference specifying the names of sections of the *Makefile* not to write. For example:

```
SKIP => [qw(name1 name2)],
```

TYPEMAPS

A reference to an array of typemap filenames. (Typemaps are used by the XS preprocessing system.) Use this when the typemaps are in some directory other than the current directory or when they are not named *typemap*. The last typemap in the list takes precedence. A typemap in the current directory has highest precedence, even if it isn't listed in TYPEMAPS. The default system typemap has lowest precedence.

VERSION

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

VERSION_FROM

Instead of specifying the VERSION in the *Makefile.PL*, you can let MakeMaker parse a file to determine the version number. The parsing routine requires that the file named by VERSION_FROM contain one single line to compute the version number. The first line in the file that contains the regular expression:

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

will be evaluated with **eval** and the value of the named variable after the **eval** will be assigned to the VERSION attribute of the MakeMaker object. The following lines will be parsed satisfactorily:

```
$VERSION = '1.00';
( $VERSION ) = '$Revision: 1.64 $ ' =~ /\$Revision:\s+([^\s]+)/;
$FOO::VERSION = '1.10';
```

but these will fail:

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

The file named in VERSION_FROM is added as a dependency to the *Makefile* in order to guarantee that the *Makefile* contains the correct VERSION attribute after a change of the file.

XS

A hash reference of *.xs* files. MakeMaker will default this. For example:

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

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

XSOPT

A string of options to pass to *xsubpp* (the XS preprocessor). This might include -C++ or -extern. Do not include typemaps here; the TYPEMAP parameter exists for that purpose.

XSPROTOARG

May be set to an empty string, which is identical to -prototypes, or -noprototypes. MakeMaker defaults to the empty string.

XS_VERSION

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

7.2.18.8 Additional lowercase attributes

There are additional lowercase attributes that you can use to pass parameters to the methods that spit out particular portions of the *Makefile*. These attributes are not normally required.

clean

Extra files to clean.

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

depend

Extra dependencies.

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

dist

Options for distribution (see "Distribution Support" below).

```
dist => {
  TARFLAGS => 'cvfF',
  COMPRESS => 'gzip',
  SUFFIX => 'gz',
  SHAR => 'shar -m',
  DIST_CP => 'ln',
},
```

If you specify COMPRESS, then SUFFIX should also be altered, since it is

needed in order to specify for *make* the target file of the compression. Setting DIST_CP to "ln" can be useful if you need to preserve the timestamps on your files. DIST_CP can take the values "cp" (copy the file), "ln" (link the file), or "best" (copy symbolic links and link the rest). Default is "best".

dynamic_lib

Options for dynamic library support.

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

installpm

Some installation options having to do with AutoSplit.

```
{SPLITLIB => '$(INST_LIB)' (default) or '$(INST_ARCHLIB)'}
```

linkext

Linking style.

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

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

```
linkext => {LINKTYPE => ""},
```

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

macro

Extra macros to define.

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

realclean

Extra files to really clean.

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

7.2.18.9 Useful Makefile macros

Here are some useful macros that you probably shouldn't redefine because they're derivative.

FULLEXT

Pathname for extension directory (for example, *DBD/Oracle*).

BASEEXT

Basename part of FULLEXT. May be just equal to FULLEXT.

ROOTEXT

Directory part of FULLEXT with leading slash (for example, */DBD*)

INST_LIBDIR

$(INST_LIB)$(ROOTEXT)

INST_AUTODIR

$(INST_LIB)/auto/$(FULLEXT)

INST_ARCHAUTODIR

$(INST_ARCHLIB)/auto/$(FULLEXT)

7.2.18.10 Overriding MakeMaker methods

If you cannot achieve the desired *Makefile* behavior by specifying attributes, you may define private subroutines in the *Makefile.PL*. Each subroutine returns the text it wishes to have written to the *Makefile*. To override a section of the *Makefile* you can use one of two styles. You can just return a new value:

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

or you can edit the default by saying something like:

```
sub MY::c_o {
   my $self = shift;
   local *c_o;
   $_=$self->MM::c_o;
   s/old text/new text/;
   $_;
}
```

Both methods above are available for backward compatibility with older *Makefile.PL*s.

If you still need a different solution, try to develop another subroutine that better fits your needs and then submit the diffs to either perl5-porters@nicoh.com or comp.lang.perl.modules as appropriate.

7.2.18.11 Distribution support

Module Distribution support

For authors of extensions, MakeMaker provides several *Makefile* targets. Most of the support comes from the ExtUtils::Manifest module, where additional documentation can be found. Note that a *MANIFEST* file is basically just a list of filenames to be shipped with the kit to build the extension.

make distcheck

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

make skipcheck

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

make distclean

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

make manifest

> Rewrites the *MANIFEST* file, adding all remaining files found. (See ExtUtils::Manifest::mkmanifest() for details.)

make distdir

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

make disttest

> Makes *distdir* first, and runs *perl Makefile.PL*, *make*, and *make test* in that directory.

make tardist

> First does a command $(PREOP), which defaults to a null command. Does a *make distdir* next and runs *tar*(1) on that directory into a tarfile. Then deletes the *distdir*. Finishes with a command $(POSTOP), which defaults to a null command.

make dist

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

make uutardist

Runs a *tardist* first and *uuencode*s the tarfile.

make shdist

First does a command $(PREOP), which defaults to a null command. Does a *distdir* next and runs *shar* on that directory into a sharfile. Then deletes the *distdir*. Finishes with a command $(POSTOP), which defaults to a null command. Note: for *shdist* to work properly, a *shar* program that can handle directories is mandatory.

make ci

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

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

Parameter	**Default**
CI	('ci -u')
COMPRESS	('compress')
POSTOP	('@ :')
PREOP	('@ :')
RCS_LABEL	('rcs -q -Nv$(VERSION_SYM):')
SHAR	('shar')
SUFFIX	('Z')
TAR	('tar')
TARFLAGS	('cvf')

An example:

```
WriteMakefile( 'dist' => { COMPRESS=>"gzip", SUFFIX=>"gz" })
```

7.2.17 ExtUtils::Liblist - Determine Libraries to Use and How to Use Them

---