

# PMK Tutorial

Damien Couderc  
Xavier Santolaria  
Martin Reindl

February 29, 2004

## 1 Introduction

As make is using a makefile, the pmk binary is using a pmkfile. This file contains a set of commands that perform checks of source dependencies. Following to that, template files are processed to generate a final file after tags substitutions. In addition to this tutorial, we highly recommend to read the pmkfile(5) manpage.

## 2 The settings

The first command you will find in a pmkfile is named SETTINGS. Its function is to define global settings. To begin we are going to use only one template for the makefile. See the following example:

```
SETTINGS{TARGET = ("Makefile.in")}
```

The TARGET option is taking a list as value. The list is delimited by parenthesis. As you can see Makefile.in will be the template name.

## 3 Defining some variables

It can be useful to define some basic variables such as package name and version. For this, we will use the DEFINE command as following:

```
DEFINE{PACKAGE = "hello"VERSION = "0.1"}
```

If the target(s) file(s) contain(s) the @PACKAGE@ and @VERSION@ tags then they will be replaced by the variables values.

## 4 First check

Lets say that we have a hello.c file that needs the string header. For that, we use the CHECK\_HEADER command like following:

```
CHECK_HEADER(header_string){NAME = "string.h"}
```

The NAME option contains the header to check. The string into brackets is the label of the command. If the check succeeds, this label is set as true else it is set as false. We'll see later how to use this label state into conditional command or as dependency with the DEPEND option.

Now let's see what our actual pmkfile contains. First we create the template with the following commands:

```
echo" PACKAGE = @PACKAGE@" > Makefile.inecho" VERSION = @VERSION@" >> Makefil
```

You can now type 'pmk'. The resulting output should look like this:

Processing commands :

- \* Parsing settings Collecting targets : Added 'Makefile.in'. Total 1 target(s) added.
- \* Parsing define defined 'PACKAGE' variable. defined 'VERSION' variable.
- \* Checking header [header\_string] Use C language with CC compiler. Store compiler flags in 'CFLAGS'. Found header 'string.h' : yes.

Process templates : Created '/home/mips/tmp/Makefile'.

And the generated Makefile should look like:

```
PACKAGE=hello VERSION=0.1
```

## 5 Using external libraries

Our hello program also needs the curses library. To check if this library is present we will use the following command:

```
CHECK_LIB(lib_curses) { NAME = "curses" }
```

Which gives as result:

- \* Checking library [lib\_curses] Use C language with CC compiler. Store library flags in 'LIBS'. Found library 'curses' : yes.

We could also check for a specific function in the library:

```
CHECK_LIB(lib_curses) { NAME = "curses" FUNCTION = "getwin" }
```

So you'll get the following:

- \* Checking library [lib\_curses] Use C language with CC compiler. Store library flags in 'LIBS'. Found function 'getwin' in 'curses' : yes.

The linker flags are automatically set in the default LIBS variable. We'll see later how to use an alternate variable.

## 6 Playing with pkg-config support

Let's say that our package has the gtkhello.c source that builds a gtk version of our hello program. That said, we need to check if gtk is present on the system. We have two possibilities to do that: first by using pkg-config or the second one by checking the library.

We are going to learn how to check gtk via pkgconfig support:

```
CHECK_PKG_CONFIG(pc_gtk) { NAME = "gtk+" }
```

If gtk is present you should get the following output when running pmk:

- \* Checking pkg-config module [pc\_gtk] Found package 'gtk+' : yes. Stored compiler flags in 'CFLAGS'. Stored library flags in 'LIBS'.

We could also check for a minimal version of gtk needed by the program.

For example we could look for at least 1.2 version of gtk:

```
CHECK_PKG_CONFIG(pc_gtk) { NAME = "gtk+" VERSION = "1.2" }
```

And the related output:

- \* Checking pkg-config module [pc\_gtk] Found package 'gtk+' : yes. Found version  $i=1.2$  : yes (1.2.10). Stored compiler flags in 'CFLAGS'. Stored library flags in 'LIBS'.

As you can see the compiler and linker flags are automatically stored in the default CFLAGS and LIBS variables. You can provide alternate variables with the CFLAGS and LIBS options like following:

```
CHECK_PKG_CONFIG(pc_gtk) { NAME = "gtk+" VERSION = "1.2"
CFLAGS = "GTK_CFLAGS" LIBS = "GTK_LIBS" }
```

And you'll get:

```
* Checking pkg-config module [pc_gtk] Found package 'gtk+' : yes. Found
version 1.2 : yes (1.2.10). Stored compiler flags in 'GTK_CFLAGS'. Stored
library flags in 'GTK_LIBS'.
```

Notice that the storage variables have changed in the output.

## 7 Using switches

Now we are able to detect gtk we could use a switch to specify if we want the gtk version or not. For this we will use the SWITCHES command and add it between SETTINGS AND DEFINE:

```
SWITCHES { build_gtk_version = FALSE }
```

By default we don't want the gtk version to be build so it's set to FALSE. A switch can be modified by using -e (enable) and -d (disable) options, see pmk(1). For example we could use the following to enable the build of the gtk program:

```
pmk -e build_gtk_version
```

We are going to see the two methods of using switches. First using the DEPEND option:

```
CHECK_PKG_CONFIG(pc_gtk) { NAME = "gtk+" DEPEND = ("build_gtk_version")
VERSION = "1.2" CFLAGS = "GTK_CFLAGS" LIBS = "GTK_LIBS" }
```

As you can see we added 'build\_gtk\_version' in the dependency list of the DEPEND option. A dependency can be a command label or a switch. If all the dependencies in the list are true then the command is executed. Else if the test is required pmk will fail like this:

```
* Checking pkg-config module [pc_gtk] Required 'build_gtk_version' depen-
dency failed.
```

Another method to use switches is the IF conditional command. This will look like the following:

```
IF(build_gtk_version) { CHECK_PKG_CONFIG(pc_gtk) { NAME = "gtk+"
VERSION = "1.2" CFLAGS = "GTK_CFLAGS" LIBS = "GTK_LIBS" } }
```

Of course you can put as many commands as you want into the body. The output looks like the following:

```
┆ Begin of condition [build_gtk_version]
```

```
* Checking pkg-config module [pc_gtk] Found package 'gtk+' : yes. Found
version 1.2 : yes (1.2.10). Stored compiler flags in 'GTK_CFLAGS'. Stored
library flags in 'GTK_LIBS'.
```

```
┆ End of condition [build_gtk_version]
```

It's also possible to negate a value by appending a '!' above the dependency like this:

```
IF(!build_gtk_version) { [commands] }
```

or like this:  
DEPEND = ("!build\_gtk\_version")

## 8 Checking types

As an example we will check for the ISO C99 boolean type: `_Bool`. See the following:

```
CHECK_TYPE(type_bool_isoc) { NAME = "_Bool" }
```

As my compiler is not fully ISO C99 compliant i get an error:

```
* Checking type [type_bool_isoc] Use C language with CC compiler. Found type '_Bool' : no. Error : failed to find type '_Bool'.
```

This test could be useful to detect compilers that are ISO C99 compliant. For those which are not we could include an external definition of this type. That said, `pmk` is exiting with an error by default if a check fails. To avoid this we'll use the `REQUIRED` option:

```
CHECK_TYPE(type_bool_isoc) { REQUIRED = FALSE NAME = "_Bool" }
```

And it gives the following:

```
* Checking type [typen_bool_isoc] Use C language with CC compiler. Found type '_Bool' : no.
```

Now let's see how to use this test. First we must create an header that will contains the tags. Here is `isoc_compat.h.in`:

```
/* ISO C compatibility header */  
/* _Bool type */ @DEF__BOOL@
```

Now add this file in the target list:

```
SETTINGS { TARGET = ("Makefile.in", "isoc_compat.h.in") }
```

The definition tag is easy to build, it's "`DEF__`" followed by the string of the type in uppercase: `_Bool` gives `_BOOL`.

Now see the result:

```
* Checking type [type_bool_isoc] Use C language with CC compiler. Found type '_Bool' : no.
```

Process templates : Created `'/home/mips/tmp/pmk_tutor/Makefile'`. Created `'/home/mips/tmp/pmk_tutor/isoc_compat.h'`.

End of log

And the `isoc_compat.h` file:

```
/* ISO C compatibility header */  
/* _Bool type */ #undef HAVE__BOOL
```

The `@DEF__BOOL@` has been replaced by "`#undef HAVE__BOOL`". Now we can add the code to include the definition of `_Bool` type if it does not exist.

```
/* ISO C compatibility header */  
/* _Bool type */ @DEF__BOOL@  
#ifndef HAVE__BOOL typedef unsigned char _Bool; #endif
```

Which results in the following on my system:

```
/* ISO C compatibility header */  
/* _Bool type */ #undef HAVE__BOOL
```

```
#ifndef HAVE_BOOL typedef unsigned char _Bool; #endif
Mission succeeded !
```

## 9 Looking for a binary ?

Easy ! Need to know if strip program is available ? Use the following:

```
CHECK_BINARY(bin_strip) { NAME = "strip" VARIABLE = "STRIP_PATH"
}
```

and add the tag in the Makefile.in template:

```
PACKAGE=@PACKAGE@ VERSION=@VERSION@ STRIP_PATH=@STRIP_PATH@
```

Now run pmk:

```
* Checking binary [bin_strip] Found binary 'strip' : yes.
```

And look at Makefile:

```
PACKAGE=hello VERSION=0.1 STRIP_PATH=/usr/bin/strip
```

Well we got it :)

## 10 Checking a variable

As an example we'll take the case of a program that will run only on 'little endian' byte ordered CPUs. For that matter we'll check the `HW_BYTEORDER` variable set by `pmksetup`:

```
CHECK_VARIABLE(var_endian) { NAME = "HW_BYTEORDER" VALUE
= "LITTLE_ENDIAN" }
```

On big-endian machines the check will fail with an error:

```
* Checking variable [var_endian] Found variable 'HW_BYTEORDER' : yes.
Variable match value 'LITTLE_ENDIAN' : no. Error : variable value does not
match ('LITTLE_ENDIAN' != 'BIG_ENDIAN').
```

## 11 Shared library support

The first step is to detect the compiler used. We do that with the `DETECT` option of the `SETTINGS` command:

```
SETTINGS { TARGET = ("Makefile.in", "isoc.compat.h.in") DETECT =
("CC" ) }
```

In this example we detect the C compiler but we could have detected the C++ compiler by using `DETECT=("CXX")`. You can also detect both compilers with `DETECT=("CC", "CXX")`.

The output should look almost like the following:

```
* Parsing settings Collecting targets : Added 'Makefile.in'. Added 'isoc.compat.h.in'.
Total 2 target(s) added. Detecting compilers : Gathering data for compiler de-
tection. Detecting 'CC' : GNU gcc (version 295). Setting SLCFLAGS to '-fPIC'
Setting SLLDFLAGS to '-shared'
```

In this example the compiler used is `gcc` (version 2.95). As you can see the shared library flags (`-fpic` and `-shared`) are set in the reserved variables.

Now we're going to see how to build the name of the shared library using the BUILD\_SHLIB\_NAME command:

```
BUILD_SHLIB_NAME { NAME = "hello" MAJOR = "2" MINOR = "1"
VERSION_NONE = "LIBNAME" VERSION_MAJ = "LIBNAME_MAJ" VER-
SION_FULL = "LIBNAME_FULL" }
```

This will give the following output:

```
* Building shared library name Shared library support : yes. Setting LIB-
NAME to 'libhello.so' Setting LIBNAME_FULL to 'libhello.so.2.1' Setting LIB-
NAME_MAJ to 'libhello.so.2'
```

To use all this stuff we could use a template of makefile like following:

```
PACKAGE=@PACKAGE@ VERSION=@VERSION@ STRIP_PATH=@STRIP_PATH@
CFLAGS=@CFLAGS@ LIBS=@LIBS@
SLCFLAGS=@SLCFLAGS@ SLLDFLAGS=@SLLDFLAGS@
LIBNAME=@LIBNAME@ LIBNAME_MAJ=@LIBNAME_MAJ@ LIBNAME_FULL=@LIBNAME_F
libhello.o: $(CC) (CFLAGS)(SLCFLAGS) -c libhello.c
$(LIBNAME): libhello.o $(CC) $(SLLDFLAGS) -o $@ libhello.o cp $(LIB-
NAME) $(LIBNAME_MAJ) cp $(LIBNAME) $(LIBNAME_FULL)
```

And after running pmk we should obtain a makefile like this one:

```
PACKAGE=hello VERSION=0.1 STRIP_PATH=/usr/bin/strip
CFLAGS=-O2 -I/usr/include LIBS=-lcurses
SLCFLAGS=-fPIC SLLDFLAGS=-shared
LIBNAME=libhello.so LIBNAME_MAJ=libhello.so.2 LIBNAME_FULL=libhello.so.2.1
libhello.o: $(CC) $(CFLAGS) $(SLCFLAGS) -c libhello.c
$(LIBNAME): libhello.o $(CC) $(SLLDFLAGS) -o $@ libhello.o cp $(LIB-
NAME) $(LIBNAME_MAJ) cp $(LIBNAME) $(LIBNAME_FULL)
TO BE CONTINUED ...
```