

# MK-CONFIGURE – lightweight easy to use replacement for GNU Autotools

Aleksey Cheusov  
vle@gmx.net

Minsk, Belarus, 2010

# Concepts behind mk-configure

## Design principles and goals

- ▶ The same way of building projects both for developers and users.
- ▶ The only file describing the project is(are) Makefile(s).
- ▶ The only command required for building is bmake (portable version of NetBSD make).
- ▶ Declarative approach of writing Makefile(s). Build and installation process is controlled with a help of special variables.
- ▶ No code generation. Library approach is used instead.
- ▶ No need to “reinvent” rules for compiling, linking, installing, uninstalling etc. again and again.
- ▶ KISS. Less than 4000 lines of code. No heavy dependencies.

# Concepts behind mk-configure

## Design principles and goals

- ▶ Cross-compilation.
- ▶ Portability to all UNIX-like systems.
- ▶ Modular approach. Extensions to mk-configure are implemented using bmake include files and standard POSIX tools, e.g. shell, awk, sed, grep and so on.

# Concepts behind mk-configure

## Negative side-effects

- ▶ End-users/packagegers have to install bmake and mk-configure to build applications based on mk-configure.

## Example 1: Hello world application

### Source code

#### Makefile

```
PROG=      hello

.include <mkc.prog.mk>
```

#### hello.c

```
#include <stdio.h>

int main (int, char **)
{
    puts ("Hello World!");
    return 0;
}
```

# Example 1: Hello world application

## How it works

```
$ export PREFIX=/usr SYSCONFDIR=/etc  
$ mkcmake  
checking for compiler type... gcc  
checking for program cc... /usr/bin/cc  
cc      -c hello.c  
cc      -o hello hello.o  
$ ./hello  
Hello World!  
$ DESTDIR=/tmp/fakeroot mkcmake installdirs install  
for d in _ /tmp/fakeroot/usr/bin; do  test "$d" = _ ||  
    install -d "$d";  done  
install  -c -s  -o cheusov -g users -m 755  
    hello /tmp/fakeroot/usr/bin/hello  
$
```

Supported targets: all, clean, cleandir (distclean), install, uninstall, installdirs, depend etc.

## Example 2: Application using non-standard strcpy(3)

### Source code

#### files in the directory

```
$ ls -l
total 12
-rw-r--r--  1 cheusov  users  158 May  2 15:04 Makefile
-rw-r--r--  1 cheusov  users  187 May  2 15:05 main.c
-rw-r--r--  1 cheusov  users  332 May  2 15:09 strcpy.c
$
```

#### Makefile

```
PROG=          strcpy_test
SRCS=          main.c

MKC_SOURCE_FUNCLIBS=  strcpy
MKC_CHECK_FUNCS3=    strcpy:string.h

.include <mkc.prog.mk>
```

## Example 2: Application using non-standard strlcpy(3)

### Source code

#### main.c

```
#include <string.h>

#ifdef HAVE_FUNC3_STRLCPY_STRING_H
size_t strlcpy(char *dst, const char *src, size_t siz);
#endif

int main (int argc, char** argv)
{
    /*    Use strlcpy(3) here    */
    return 0;
}
```



## Example 2: Application using non-standard strlcpy(3)

### How it works on Linux

```
$ CC='icc -no-gcc' mkcmake
checking for compiler type... icc
checking for function strlcpy... no
checking for func strlcpy ( string.h )... no
checking for program icc... /opt/intel/cc/10.1.008/bin/icc
icc -no-gcc -c main.c
icc -no-gcc -c strlcpy.c
icc -no-gcc -o strlcpy_test main.o strlcpy.o
$ echo _mkc_*
_mkc_compiler_type.err _mkc_compiler_type.res
_mkc_func3_strlcpy_string_h.c
_mkc_func3_strlcpy_string_h.err
_mkc_func3_strlcpy_string_h.res
_mkc_funclib_strlcpy.c _mkc_funclib_strlcpy.err
_mkc_funclib_strlcpy.res _mkc_prog_cc.err _mkc_prog_cc.res
$
```

## Example 2: Application using non-standard strlcpy(3)

### How it works on NetBSD

```
$ mkcmake
```

```
checking for compiler type... gcc
```

```
checking for function strlcpy... yes
```

```
checking for func strlcpy ( string.h )... yes
```

```
checking for program cc... /usr/bin/cc
```

```
cc -DHAVE_FUNC3_STRLCPY_STRING_H=1 -c main.c
```

```
cc -o strlcpy_test main.o
```

```
$
```

## Example 3: Application using plugins

### Source code

#### Makefile

```
MKC_CHECK_FUNCLIBS=      dlopen:dl

PROG=                     myapp

.include <mkc.configure.mk>

.if ${HAVE_FUNCLIB.dlopen} || ${HAVE_FUNCLIB.dlopen.dl}
CFLAGS+= -DPLUGINS_ENABLED=1
.endif

.include <mkc.prog.mk>
```

## Example 3: Application using plugins

### How it works on Linux

```
$ mkmake
```

```
checking for compiler type... gcc
```

```
checking for function dlopen ( -ldl )... yes
```

```
checking for function dlopen... no
```

```
checking for program gcc... /usr/bin/gcc
```

```
gcc -DPLUGINS_ENABLED=1      -c myapp.c
```

```
gcc      -o myapp myapp.o -ldl
```

```
$
```

## Example 3: Application using plugins

### How it works on OpenBSD

```
$ mkcmake
```

```
checking for compiler type... gcc
```

```
checking for function dlopen ( -ldl )... no
```

```
checking for function dlopen... yes
```

```
checking for program cc... /usr/bin/cc
```

```
cc -DPLUGINS_ENABLED=1 -c myapp.c
```

```
cc -o myapp myapp.o
```

```
$
```

## Example 4: Support for shared libraries

### Source code

#### Makefile

```
LIB=                                foobar
SRCS=                              foo.cc bar.cc baz.cc

MKPICLIB?=                          no
MKSTATICLIB?=                       no

SHLIB_MAJOR=                        1
SHLIB_MINOR=                        0

.include <mkc.lib.mk>
```

## Example 4: Support for shared libraries

### How it works on Solaris

```
$ mkcmake
```

```
/opt/SUNWspro/bin/CC      -c -KPIC foo.cc -o foo.os
```

```
/opt/SUNWspro/bin/CC      -c -KPIC bar.cc -o bar.os
```

```
/opt/SUNWspro/bin/CC      -c -KPIC baz.cc -o baz.os
```

```
building shared foobar library (version 1.0)
```

```
/opt/SUNWspro/bin/CC -G -Wl,-h -Wl,libfoobar.so.1
```

```
    -o libfoobar.so.1.0  foo.os bar.os baz.os
```

```
ln -sf libfoobar.so.1.0 libfoobar.so
```

```
ln -sf libfoobar.so.1.0 libfoobar.so.1
```

```
$
```

## Example 4: Support for shared libraries

### How it works on Darwin

```
$ mkcmake
```

```
checking for compiler type... gcc
```

```
checking for program c++... /usr/bin/c++
```

```
c++      -c -fPIC -DPIC foo.cc -o foo.os
```

```
c++      -c -fPIC -DPIC bar.cc -o bar.os
```

```
c++      -c -fPIC -DPIC baz.cc -o baz.os
```

```
building shared foobar library (version 1.0)
```

```
c++ -dynamiclib -install_name
```

```
    /usr/local/lib/libfoobar.1.0.dylib
```

```
    -current_version 2.0 -compatibility_version 2
```

```
    -o libfoobar.1.0.dylib  foo.os bar.os baz.os
```

```
ln -sf libfoobar.1.0.dylib libfoobar.dylib
```

```
ln -sf libfoobar.1.0.dylib libfoobar.1.dylib
```

```
$
```



## Example 5: Big project consisting of several subprojects

### Source code

#### Makefile

```
# This project consists of several subprojects:
# dict, dictd, dictfmt, dictzip, libdz, libmaa
# and libcommon. libcommon contains common code
# for executables and should not be installed.
# SUBPRJ specifies a dependency graph
# for all subprojects.

SUBPRJ=    libcommon:dict    # dict depends on libcommon
SUBPRJ+=   libcommon:dictd
SUBPRJ+=   libcommon:dictzip
SUBPRJ+=   libcommon:dictfmt
SUBPRJ+=   libdz:dictzip
SUBPRJ+=   libmaa:dict
...
.include <mkc.subprj.mk>
```

## Example 5: Big project consisting of several subprojects

### Source code

#### **libcommon/Makefile**

```
# Internal static library that implements functions
# common for dict, dictd, dictfmt
# and dictzip applications

LIB=                common
SRCS=               str.c iswalnum.c # and others

MKINSTALL=         no # Do not install it!

.include <mkc.lib.mk>
```

#### **libcommon/linkme.mk**

```
PATH.common:=      ${.PARSEDIR}

CPPFLAGS+=         -I${PATH.common}
DPLIBDIRS+=        ${PATH.common}
```

## Example 5: Big project consisting of several subprojects

### Source code

#### libmaa/Makefile

```
LIB=          maa
SRCS=         set.c prime.c log.c # etc.

SHLIB_MAJOR=  1
SHLIB_MINOR=  2
SHLIB_TEENY=  0

.include <mkc.lib.mk>
```

#### libmaa/linkme.mk

```
PATH.maa:=    ${.PARSEDIR}

CPPFLAGS+=    -I${PATH.maa}
DPLIBDIRS+=   ${PATH.maa}
```

## Example 5: Big project consisting of several subprojects

### How it works

```
$ mkcmake all-dictd
```

```
=====
all ==> libcommon
```

```
...
```

```
=====
all ==> libmaa
```

```
...
```

```
=====
all ==> dictd
```

```
...
```

```
checking for program cc... /usr/bin/cc
```

```
cc -I../libcommon -I../libmaa -c dictd.c
```

```
cc -L/tmp/hello_dictd/libcommon -L/tmp/hello_dictd/libmaa  
-o dictd dictd.o -lcommon -lmaa
```

```
$
```

## Example 6: Support for Lua programming language

### Source code

#### Makefile

```
SCRIPTS=  foobar      # scripts written in Lua
LUA_LMODULES=  foo bar # modules written in Lua
LUA_CMODULE=  baz      # Lua module written in C

.include <mkc.lib.mk>
```

## Example 6: Support for Lua programming language

### How it works

```
$ mkcmake errorcheck
```

```
checking for program pkg-config...
```

```
    /usr/pkg/bin/pkg-config
```

```
checking for [pkg-config] lua... 1 (yes)
```

```
checking for [pkg-config] lua --cflags...
```

```
    -I/usr/pkg/include
```

```
checking for [pkg-config] lua --libs...
```

```
    -Wl,-R/usr/pkg/lib -L/usr/pkg/lib -llua -lm
```

```
checking for [pkg-config] lua --variable=INSTALL_LMOD...
```

```
    /usr/pkg/share/lua/5.1
```

```
checking for [pkg-config] lua --variable=INSTALL_CMOD...
```

```
    /usr/pkg/lib/lua/5.1
```

```
checking for compiler type... gcc
```

```
checking for program cc... /usr/bin/cc
```

```
$
```

## Example 6: Support for Lua programming language

### How it works

```
$ export PREFIX=/usr/pkg
$ mkcmake all
cc      -I/usr/pkg/include -I/usr/pkg/include -c
      -fPIC -DPIC baz.c -o baz.os
building shared baz library (version 1.0)
cc -shared -Wl,-soname -Wl,libbaz.so.1 -o baz.so  baz.os
      -Wl,-R/usr/pkg/lib -L/usr/pkg/lib -llua -lm
$
```

## Example 6: Support for Lua programming language

### How it works

```
$ mkmake installdirs install DESTDIR=/tmp/fakeroot
```

```
...
```

```
$ find /tmp/fakeroot -type f
```

```
/tmp/fakeroot/usr/pkg/bin/foobar
```

```
/tmp/fakeroot/usr/pkg/lib/lua/5.1/baz.so
```

```
/tmp/fakeroot/usr/pkg/share/lua/5.1/foo.lua
```

```
/tmp/fakeroot/usr/pkg/share/lua/5.1/bar.lua
```

```
$
```



# Features

## 1. Automatic detection of system configuration (**mkc.configure.mk**)

- ▶ header presence (MKC\_{CHECK,REQUIRE}\_HEADERS)
- ▶ function declaration (MKC\_{CHECK,REQUIRE}\_FUNCS[n])
- ▶ type declaration (MKC\_{CHECK,REQUIRE}\_TYPES)
- ▶ structure member (MKC\_{CHECK,REQUIRE}\_MEMBERS)
- ▶ variable declaration (MKC\_{CHECK,REQUIRE}\_VARS)
- ▶ define declaration (MKC\_{CHECK,REQUIRE}\_DEFINES)
- ▶ type size (MKC\_CHECK\_SIZEOF)
- ▶ function implementation in the library  
(MKC\_{CHECK,REQUIRE}\_FUNCLIBS and  
MKC\_SOURCE\_FUNCLIBS)
- ▶ checks for programs (MKC\_{CHECK,REQUIRE}\_PROGS)
- ▶ user's custom checks (MKC\_{CHECK,REQUIRE}\_CUSTOM)
- ▶ built-in checks (MKC\_CHECK\_BUILTINS), e.g. endianness,  
prog\_flex, prog\_bison, prog\_gawk or prog\_gm4)

# Features

2. Building, installing, uninstalling, cleaning etc. Supported targets: all, installdirs, install, uninstall, clean, cleandir (distclean) and others.
3. Building standalone programs (**mkc.prog.mk**), static and shared libraries (**mkc.lib.mk**) using C, C++, Objective C, Pascal and Fortran compilers. Shared libraries support is provided for numerous OSes: NetBSD, FreeBSD, OpenBSD, DragonFlyBSD, MirOS BSD, Linux, Solaris, Darwin (MacOS-X), Tru64, QNX, HP-UX, Cygwin (no support for shared object files yet) and compilers: GCC, Intel C/C++ compilers, Portable C compiler aka pcc, DEC C/C++ compiler, HP C/C++ compiler, Oracle SunPro and others.
4. Handling of man pages.

# Features

5. Building info pages from texinfo sources.
6. Handling of scripts as well as plain text files, i.e. installing or uninstalling.
7. Cross-building. mk-configure itself doesn't run produced executables, so you can freely use cross-tools (compiler, linker etc.). Also you can override any variable initialized by mk-configure.
8. Support for pkg-config (**mkc\_imp.pkg-config.mk**).
9. Support for Lua programming language (**mkc\_imp.lua.mk**).
10. Support for yacc and lex.
11. Support for projects with multiple subprojects (**mkc.subprj.mk** and **mkc.subdir.mk**).

# Features

12. Parts of mk-configure functionality is accesible via individual programs, e.g. **mkc\_install**, **mkc\_check\_compiler**, **mkc\_check\_header**, **mkc\_check\_funclib**, **mkc\_check\_decl**, **mkc\_check\_prog**, **mkc\_check\_sizeof** and **mkc\_check\_custom**.

# MK-CONFIGURE in real world

## Packages in UNIX-like system and distributions

NetBSD make (bmake) is packaged in the following OSes:

- ▶ FreeBSD, NetBSD
- ▶ Gentoo Linux, Fedora Linux, AltLinux
- ▶ Debian/Ubuntu  
deb <http://movs.org/~cheusov/pub/debian lenny main>  
deb-src <http://movs.org/~cheusov/pub/debian lenny main>

mk-configure is packaged in the following OSes

- ▶ FreeBSD, NetBSD
- ▶ AltLinux
- ▶ Debian/Ubuntu  
deb <http://movs.org/~cheusov/pub/debian lenny main>  
deb-src <http://movs.org/~cheusov/pub/debian lenny main>

# MK-CONFIGURE in real world

## Real life samples of use

- ▶ Lightweight modular malloc Debugger.  
**<http://sf.net/projects/lmdbg>**
- ▶ Any project based on traditional **bsd.{prog,lib,subdir}.mk** can trivially be converted to use mk-configure.
  - ▶ **<http://sourceforge.net/projects/runawk>**  
Modules Framework for AWK programming language
  - ▶ **<http://sourceforge.net/projects/paexec>**  
Parallel Executer
  - ▶ **<http://mova.org/~cheusov/pub/distbb/>**  
Distributed fault tolerant bulk build tool for pkgsrc  
**[http://mova.org/~cheusov/pub/pkg\\_online/](http://mova.org/~cheusov/pub/pkg_online/)**  
Client/server package search system for pkgsrc
  - ▶ NetBSD, FreeBSD and OpenBSD! ;-)
  - ▶ ...

# MK-CONFIGURE in real world

My own opensource software projects using mk-configure (romb), Mk files (box) and others (oval)

