MK-CONFIGURE (MK-C) – lightweight, easy to use alternative for GNU Autotools

Aleksey Cheusov
vle@gmx.net

Minsk, Belarus, 2012

About this presentation

- It is a part of official documentation. Latest version is available for download from here **http://mova.org/~cheusov/pub/mk-c/mk-c.pdf**
- part 1: Introduction
- part 2: A number of samples of use
- part 3: More complete list of features, TODO and more.

Concepts behind mk-configure

Design principles and goals

- **I detest code generation** as in Autotools and CMake! **Library approach** is used instead.
- Written in **bmake**, portable version of **NetBSD make(1)**, and UNIX tools. **No heavy dependencies** like python, ruby and perl. As a programming language bmake is not as powerful as RuPyPe, but **bmake+sh is good enough** for this task.
- Basic principles are **similar to traditional bsd.*.mk files**. Actually mk-c contains heavily reworked Mk files from NetBSD.
- **Portability** to all UNIX-like systems.
- KISS. Only about 4000 lines of code.

Concepts behind mk-configure

Design principles and goals

- mk-configure is not only for end-users and packagers but for developers too. So, one of the main goals is to provide a convenient **tool for development**.
- Declarative approach of writing Makefile(s). Build and installation process is controlled with a help of special variables and bmake's include files.
- **Cross-compilation**.
- **Extensibility**. Extensions to mk-configure are implemented using bmake include files and standard UNIX tools, i.e. shell, awk, sed, grep etc. when needed.
- MK-C is **Easy to use**. Only one command is needed to build a project — **mkcmake**. **Only Makefile(s)** are needed for specifying build instructions.

Concepts behind mk-configure

Negative side-effects

- End-users/packagers 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 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, install, uninstall, installdirs, depend etc.

Example 2: Application using non-standard strlcpy(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 strlcpy.c
$
```

Makefile

```
PROG =                  strlcpy_test
SRCS =                  main.c

MKC_SOURCE_FUNCLIBS =   strlcpy
MKC_CHECK_FUNCS3 =      strlcpy:string.h

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

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

Source code

main.c

```
#include <string.h>

#ifndef 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 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 -c main.c
icc -c strlcpy.c
icc -o strlcpy_test main.o strlcpy.o
$ echo _mkc_*
_mkc_cc_type.err _mkc_cc_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

```
PROG =   myapp

MKC_CHECK_FUNCLIBS =     dlopen:dl

.include <mkc.configure.mk>

.if ${HAVE_FUNCLIB.dlopen:U0} ||
    ${HAVE_FUNCLIB.dlopen.dl:U0}
CFLAGS +=         -DPLUGINS_ENABLED=1
.endif

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

Example 3: Application using plugins

How it works on QNX

```
$ mkcmake
checking for compiler type... gcc
checking for function dlopen ( -ldl )... yes
checking for function dlopen... no
checking for program gcc...
      /usr/qnx650/host/qnx6/x86/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 and C++

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 with SunStudio compiler

```
$ mkcmake
checking for compiler type... sunpro
checking for program CC... /opt/SUNWspro/bin
CC -c -KPIC foo.cc -o foo.os
CC -c -KPIC bar.cc -o bar.os
CC -c -KPIC baz.cc -o baz.os
building shared foobar library (version 1.0)
CC -G -h 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
$
```

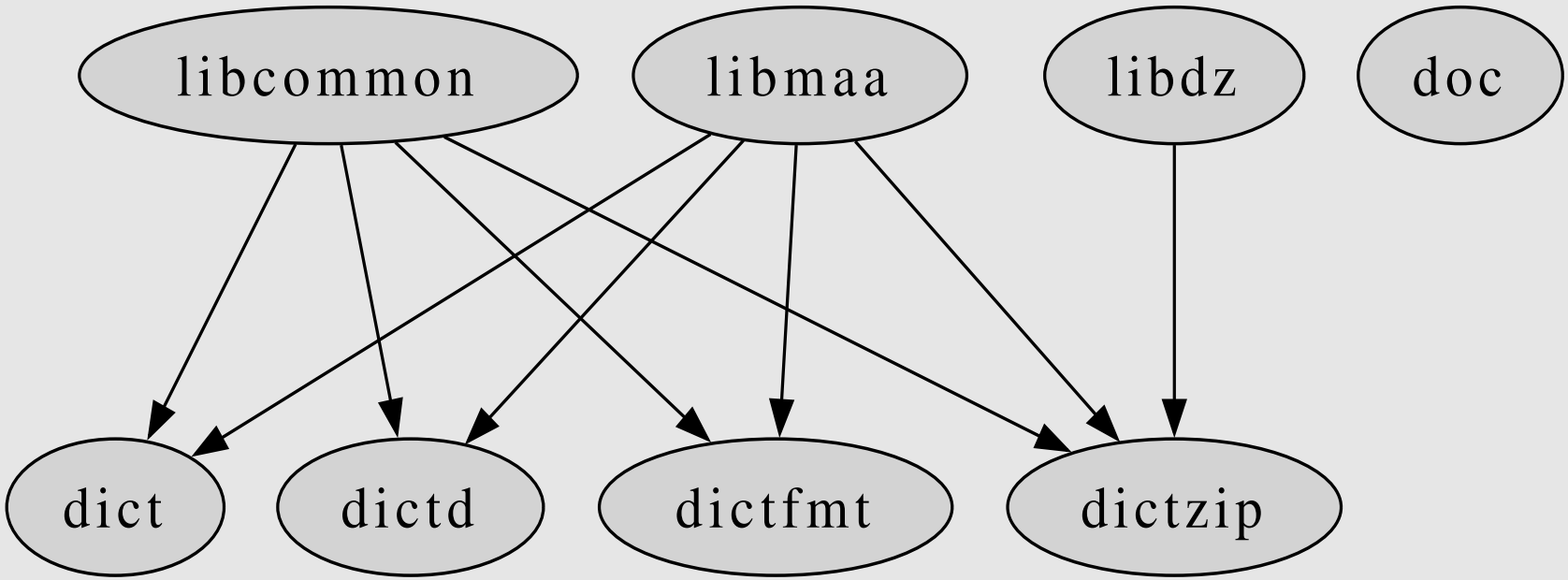
Example 4: Support for shared libraries (Exported symbols)

```
$ cat Makefile
LIB             = foo
INCS            = foo.h
EXPORT_SYMBOLS = foo.sym
SHLIB_MAJOR     = 1
MKSTATICLIB     = no
.include <mkc.lib.mk>
$ mkcmake
awk 'BEGIN {print "{ global:"}  {print $0 ";"}
     END {print "local: *; };"}' foo.sym
     > foo.sym.tmp1 && mv foo.sym.tmp1 foo.sym.tmp
cc    -I.  -c -fPIC -DPIC foo.c -o foo.os
building shared foo library (version 1)
ld -shared -soname libfoo.so.1
   --version-script foo.sym.tmp -o libfoo.so.1  foo.os
$
```

Example 5: Big project consisting of several subprojects

Dependency graph for all subprojects

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.



Example 5: Big project consisting of several subprojects

Files and directories

```
$ ls -l
total 4
drwxr-xr-x 2 cheusov users   1 Jan 26 12:01 dict
drwxr-xr-x 2 cheusov users   1 Jan 26 12:01 dictd
drwxr-xr-x 2 cheusov users   1 Jan 26 12:01 dictfmt
drwxr-xr-x 2 cheusov users   1 Jan 26 12:01 dictzip
drwxr-xr-x 2 cheusov users   1 Jan 26 12:03 doc
drwxr-xr-x 2 cheusov users   1 Jan 26 12:01 libcommon
drwxr-xr-x 2 cheusov users   1 Jan 26 12:01 libdz
drwxr-xr-x 2 cheusov users   1 Jan 26 12:01 libmaa
-rw-r--r-- 1 cheusov users 306 Jan 26 12:03 Makefile
$
```

Example 5: Big project consisting of several subprojects

Source code

Makefile

```
SUBPRJ =   libcommon:dict   # dict depends on libcommon
SUBPRJ +=  libcommon:dictd
SUBPRJ +=  libcommon:dictzip
SUBPRJ +=  libcommon:dictfmt
SUBPRJ +=  libmaa:dict
SUBPRJ +=  libmaa:dictd
SUBPRJ +=  libmaa:dictfmt
SUBPRJ +=  libmaa:dictzip
SUBPRJ +=  libdz:dictzip
SUBPRJ +=  doc

.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 internal library!

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

libcommon/linkme.mk

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

CPPFLAGS +=        -I${PATH.common}/include
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.

INCS =       maa.h

SHLIB_MAJOR =    1
SHLIB_MINOR =    2
SHLIB_TEENY =    0

# list of exported symbols
EXPORT_SYMBOLS = maa.sym

.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

Source code

libmaa/maa.sym

```
hsh_create
hsh_destroy
hsh_insert
hsh_delete
hsh_retrieve
...
lst_create
lst_destroy
lst_insert
...
set_create
set_destroy
set_add
set_union
...
```

Example 5: Big project consisting of several subprojects

Source code

libdz/Makefile

```
LIB =                   dz
SRCS =                  dz.c

INCS =                  dz.h

MKC_REQUIRE_HEADERS =    zlib.h
MKC_REQUIRE_FUNCLIBS =   deflate:z
EXPORT_SYMBOLS =        dz.sym
SHLIB_MAJOR =           1
SHLIB_MINOR =           0
LDADD =                 -lz

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

libdz/linkme.mk

```
PATH.dz :=          ${.PARSEDIR}
CPPFLAGS +=         -I${PATH.dz}
DPLIBDIRS +=        ${PATH.dz}
```

Example 5: Big project consisting of several subprojects

Source code

dictzip/Makefile

```
PROG =   dictzip
MAN =    dictzip.1

.include "../libcommon/linkme.mk"
.include "../libdz/linkme.mk"
.include "../libmaa/linkme.mk"

LDADD +=        -lcommon -ldz -lmaa

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

Example 5: Big project consisting of several subprojects

How it fails ;-)

```
$ mkcmake configure-dictzip
================================================
configure ===> libcommon
...
configure ===> libmaa
...
================================================
configure ===> libdz
checking for header zlib.h... no
checking for function deflate ( -lz )... no
checking for function deflate... no
ERROR: cannot find header zlib.h
ERROR: cannot find function deflate:z
...
$ echo $?
1
$
```

Example 5: Big project consisting of several subprojects

How it works

```
$ mkcmake dictzip
...
================================================
all ===> libdz
...
checking for header zlib.h... yes
checking for function deflate ( -lz )... yes
...
================================================
all ===> dictzip
...
cc -I../libcommon -I../libdz -I../libmaa -c dictzip.c
cc -L/tmp/hello_dictd/libcommon -L/tmp/hello_dictd/libdz
   -L/tmp/hello_dictd/libmaa  -o dictzip
   dictzip.o -lcommon -lmaa -ldz
$
```

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 configure
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 header lua.h... yes
checking for program cc... /usr/bin/cc
$
```

Example 6: Support for Lua programming language

How it works

```
$ export PREFIX=/usr/pkg
$ mkcmake all
cc -DHAVE_HEADER_LUA_H=1 -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

```
$ mkcmake 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
$
```

Example 7: Custom tests and sizeof of system types

Source code

Makefile

```
MKC_CUSTOM_DIR      = ${.CURDIR}/checks

# m4 supports -P flag (GNU, NetBSD)
M4                 ?= m4 # overridable (gm4)
MKC_REQUIRE_CUSTOM += m4P
MKC_CUSTOM_FN.m4P   = m4P.sh
.export: M4

# __attribute (({con,de}structor))
MKC_REQUIRE_CUSTOM += constructor destructor

# sizeof
MKC_CHECK_SIZEOF    = char short int long void* long-long

LIB                 = mylib
CFLAGS             += -DM4_CMD='"${M4}"'
.include <mkc.lib.mk>
```

Example 7: Custom tests and sizeof of system types

Source code

Files in checks/ subdirectory

```
$ ls checks/
constructor.c
destructor.c
m4P.sh
$
```

Example 7: Custom tests and sizeof of system types

Source code

checks/m4P.sh

```
#!/bin/sh

input (){
    cat <<'EOF'
m4_define(fruit, apple)
fruit
EOF
}

M4=${M4-m4}

if input | ${M4} -P | grep ^apple > /dev/null; then
    echo 1
else
    echo 0
fi
```

Example 7: Custom tests and sizeof of system types

Source code

checks/constructor.c

```
void __attribute ((constructor))
        dummy (void)
{
}
```

checks/destructor.c

```
void __attribute ((destructor))
        dummy (void)
{
}
```

Example 7: Custom tests and sizeof of system types

How it works on FreeBSD

```
$ mkcmake
checking for compiler type... gcc
checking for sizeof char... 1
checking for sizeof short... 2
checking for sizeof int... 4
checking for sizeof long... 4
checking for sizeof void*... 4
checking for sizeof long long... 8
checking for custom test m4P... 0 (no)
checking for custom test constructor... 1 (yes)
checking for custom test destructor... 1 (yes)
checking for program cc... /usr/bin/cc
ERROR: custom test m4P failed
*** Error code 1
...
$
```

Example 7: Custom tests and sizeof of system types

How it works on FreeBSD with GNU m4

```
$ M4=gm4 mkcmake
checking for compiler type... gcc
checking for sizeof char... 1
checking for sizeof short... 2
checking for sizeof int... 4
checking for sizeof long... 4
checking for sizeof void*... 4
checking for sizeof long long... 8
checking for custom test m4P... 1 (yes)
checking for custom test constructor... 1 (yes)
checking for custom test destructor... 1 (yes)
checking for program cc... /usr/bin/cc
...
$
```

Example 8: Portable version of AWK from NetBSD

http://mova.org/~cheusov/pub/mk-configure/nbawk/

Makefile (part 1)

```
PROG =          awk
SRCS =          awkgram.y b.c lex.c lib.c main.c parse.c
        proctab.c run.c tran.c
YHEADER =         yes

MKC_COMMON_DEFINES.Linux =          -D_GNU_SOURCE
MKC_COMMON_HEADERS =                  ctype.h stdio.h string.h
MKC_CHECK_FUNCS1 =                  __fpurge:stdio_ext.h fpurge isb
MKC_CHECK_FUNCS3 =                  strlcat
MKC_SOURCE_FUNCLIBS =                  fpurge strlcat

WARNS=        4
WARNERR=      no # do not treat warnings as errors

MKC_REQD=   0.19.0 # mk-configure>=0.19.0 is required
... # to be continued on the next slide
```

Example 8: Portable version of AWK from NetBSD

http://mova.org/~cheusov/pub/mk-configure/nbawk/

Makefile (part 2)

```
... # beginning is on the previous slide
.include <mkc.configure.mk>

.if ${HAVE_FUNC1.isblank:U0}
CPPFLAGS +=         -DHAS_ISBLANK
.endif

.if !${HAVE_FUNC1.fpurge:U1} &&
   !${HAVE_FUNC1.__fpurge.stdio_ext_h:U1}
MKC_ERR_MSG+=         "fpurge(3) cannot be found"
.endif

CPPFLAGS +=     -I.
LDADD +=        -lm

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

Example 8: Portable version of AWK from NetBSD

run.c

```
--- nbawk-20100903/run.c.orig
+++ nbawk-20100903/run.c
@@ -40,6 +40,14 @@
 #include "awk.h"
 #include "awkgram.h"

+#ifndef HAVE_FUNC1_FPURGE
+int fpurge (FILE *stream);
+#endif
+
+#ifndef HAVE_FUNC3_STRLCAT
+size_t strlcat(char *dst, const char *src, size_t size);
+#endif
+
 #define tempfree(x)    if (istemp(x)) tfree(x); else

 void stdinit(void);
```

Example 8: Portable version of AWK from NetBSD

fpurge.c

```
#include <stdio.h>

#if HAVE_FUNC1___FPURGE_STDIO_EXT_H
#include <stdio_ext.h>
#endif

int fpurge(FILE *stream);

int fpurge(FILE *stream)
{
#if HAVE_FUNC1___FPURGE_STDIO_EXT_H
  __fpurge (stream);
  return 0;
#else
#error "cannot find fpurge(3), sorry"
#endif
}
```

Example 8: Portable version of AWK from NetBSD

strlcpy.c

```
If you need this code, you know where to get it from! ;-)
```

Example 8: Portable version of AWK from NetBSD

How it works on Linux

```
$ mkcmake
checking for compiler type... gcc
checking for function fpurge... no
checking for function strlcat... no
checking for func __fpurge ( stdio_ext.h )... yes
checking for func fpurge... no
checking for func isblank... yes
checking for func strlcat... no
checking for program yacc... /usr/bin/yacc
...
cc -Wall -Wstrict-prototypes ...
    -I. -D_GNU_SOURCE -c awkgram.c
...
cc -o awk awkgram.o ... fpurge.o strlcat.o -lm
$ ./awk
usage: ./awk [-F fs] [-v var=value] [-f progfile
    | 'prog'] [file ...]
$
```

Example 9: Cross-compilation

How it works

```
$ export SYSROOT=/tmp/destdir.sparc64
$ export TOOLCHAIN_PREFIX=sparc64--netbsd-
$ export TOOLCHAIN_DIR=/tmp/tooldir.sparc64/bin
$ uname -srm
NetBSD 5.99.56 amd64
$ mkcmake
checking for compiler type... gcc
/tmp/tooldir.sparc64/bin/sparc64--netbsd-gcc
   --sysroot=/tmp/destdir.sparc64 -c hello.c -o hello.o
/tmp/tooldir.sparc64/bin/sparc64--netbsd-gcc
   --sysroot=/tmp/destdir.sparc64 -o hello hello.o
$ file hello
hello: ELF 64-bit MSB executable, SPARC V9, relaxed
   memory ordering, (SYSV), dynamically linked (uses
   shared libs), for NetBSD 5.99.56, not stripped
$
```

Features

1. Automatic detection of OS features (**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**, **install**, **uninstall**, **clean**, **cleandir**, **installdirs**, **depend** and others.
3. Building **standalone programs** (**mkc.prog.mk**), **static, shared and dynamically loaded libraries** (**mkc.lib.mk**) using **C**, **C++** and **Objective C** compilers. Shared libraries support is provided for numerous OSes: **NetBSD**, **FreeBSD**, **OpenBSD**, **DragonFlyBSD**, **MirOS BSD**, **Linux**, **Solaris**, **Darwin** (MacOS-X), **Interix**, **Tru64**, **QNX**, **HP-UX**, **Cygwin** (no support for shared libraries and DLLs yet) and compilers: **GCC**, **Intel C/C++** compilers, **Portable C compiler** AKA **pcc**, **DEC C/C++ compiler**, **HP C/C++ compiler**, **Oracle SunStudio** and others.

Features

4. Handling **man** pages, **info** manuals and **POD** documents.
5. Handling **scripts** as well as **plain text files**, i.e. installing, uninstalling and handling **.in files** (replacing, for example, **@bindir@**, **@sysconfdir@**, **@version@** fragments with real values).
6. **Cross-compilation**. mk-configure itself doesn't run target system executables, so you can freely use cross-tools (compiler, linker etc.). You can also override/set any variable initialized by mk-configure.
7. Support for **pkg-config**.
8. Support for **Lua** programming language.
9. Support for **yacc** and **lex**.

Features

11. Support for projects with multiple subprojects (**mkc.subprj.mk** and **mkc.subdir.mk**).
12. Special targets bin_tar, bin_targz, bin_tarbz2, bin_zip, bin_deb creates .tar, .tar.gz, .tar.bz2, .zip archives and .deb package (on Debian Linux).
13. 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

Packagers are welcome! ;-)

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

- FreeBSD, pkgsrc (NetBSD, DragonFlyBSD...) (**devel/bmake**)
- Gentoo Linux, Fedora Linux, AltLinux
- Debian/Ubuntu
  deb http://mova.org/~cheusov/pub/debian lenny main
  deb-src http://mova.org/~cheusov/pub/debian lenny main

mk-configure is packaged in the following OSes

- FreeBSD, pkgsrc (NetBSD, DragonFlyBSD...)
  (**devel/mk-configure**)
- Debian/Ubuntu
  deb http://mova.org/~cheusov/pub/debian lenny main
  deb-src http://mova.org/~cheusov/pub/debian lenny main

MK-CONFIGURE in real world

Real life samples of use

- Lightweight modular malloc Debugger.
  http://sourceforge.net/projects/lmdbg/
  http://pkgsrc.se/wip/lmdbg/
- NetBSD version of AWK programming language, ported to other Operating Systems.
  http://mova.org/~cheusov/pub/mk-configure/nbawk/
- Modules support for AWK programming language
  http://sourceforge.net/projects/runawk/
  http://pkgsrc.se/lang/runawk/
- Tool for distributing tasks over network or CPUs
  http://sourceforge.net/projects/paexec/
  http://pkgsrc.se/wip/paexec/

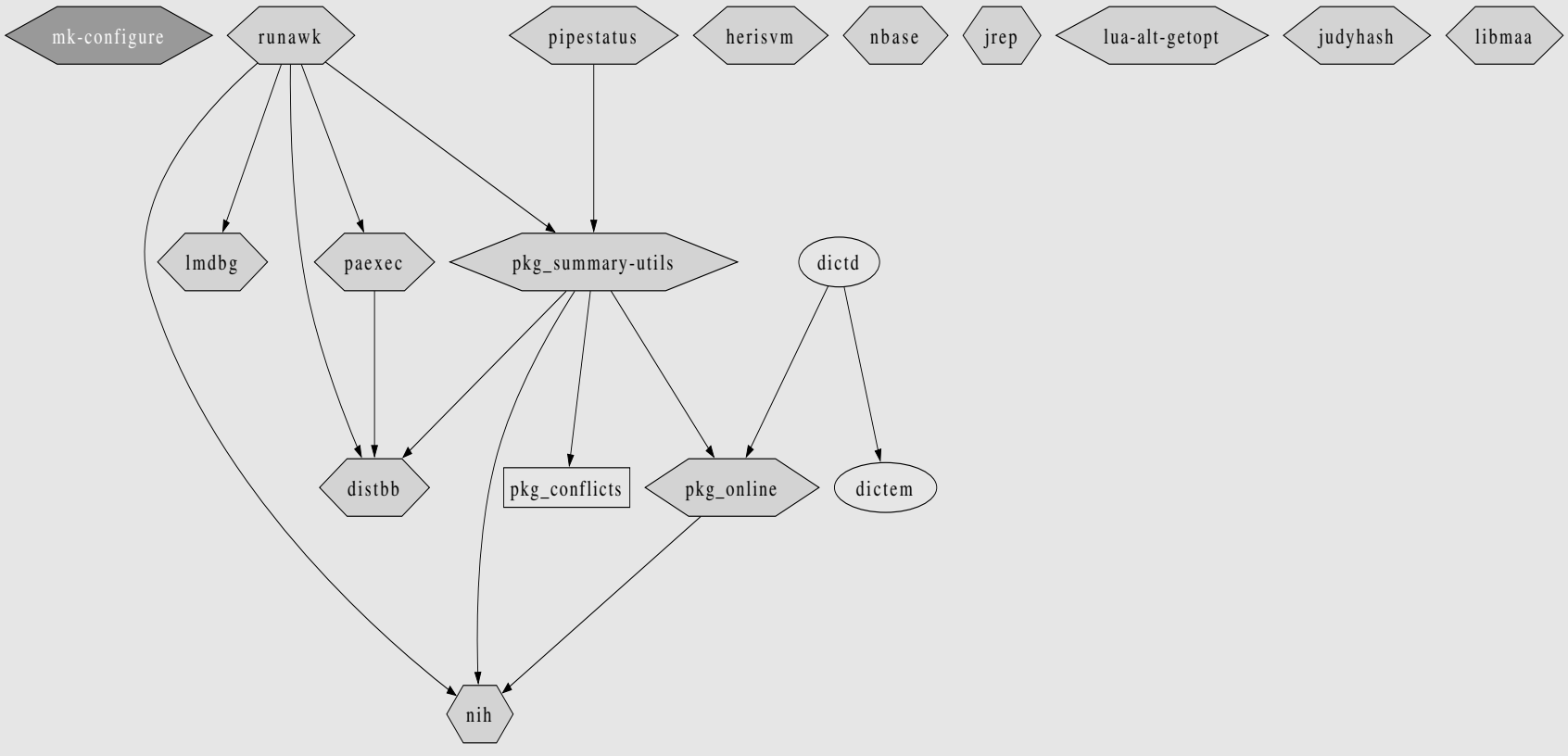
MK-CONFIGURE in real world

Real life samples of use

- Distributed fault tolerant bulk build tool for pkgsrc
  http://mova.org/~cheusov/pub/distbb/
  http://pkgsrc.se/wip/distbb/
- Client/server package search system for pkgsrc
  http://mova.org/~cheusov/pub/pkg_online/
  http://pkgsrc.se/wip/pkg_online-client/
  http://pkgsrc.se/wip/pkg_online-server/
- Any project based on traditional **bsd.{prog,lib,subdir}.mk** can easily be converted to mk-configure.

MK-CONFIGURE in real world

My opensource projects using mk-configure (filled hexagon), Mk files (box) and others (oval)



MK-C needs your help ;-)

- Packagers are welcome (Linux distros, OpenBSD etc.)
- MK-C distribution contains **a lot of regression tests and samples of use** (samples are used for testing too).
  Shell accounts on "exotic" UNIX-like platforms are needed (AIX, HP-UX, non-ELF BSDs, IRIX, Solaris, Hurd etc.) for testing and development.
- Review of the documentation. At the moment only mk-configure(7), samples/ and this presentation are available.
- sf.net provides two mailing lists:
  mk-configure-help and **mk-configure-discuss**.
- TODO file in the distribution is full of tasks.

The END.