# Beyond Makefiles: Autotools and the GNU Build System

**Patrick Nichols**
Software Engineer
CISL Consulting Group

**UCAR SEA**
December 10, 2015

# Why do we need more tools?

**Diversity!**

1. Compilers, programming languages and support libraries.
    a. Even with the same compiler different versions may present different functions and language features (Think Fortran90 vs 2003 or C++11 vs C++14).

2. Accelerators (at present, each accelerator needs it's own "language")

3. Operating systems

4. Different environments (where are the system libraries?)

**Very Difficult to develop Portable, Uniform Builds**

# Why do we need more tools?

**Why can't we just use make?**

1. Need plenty of if/else constructs
2. Requires user to define several Environmental variables or create really complicated scripts
3. Substitution macros/functions
4. Each developer would need to do this for their own package
5. Each Environment needs to be defined before hand

*Before Autotools, people actually did this!*

# History

**1976**- first version of Make AT&T Stuart Feldman

**1992**- Configure developed by several authors

Metaconfig for Perl

Cygnus

Imake

GNU Autoconf

**1994**- Automake

**1996**- Libtool for shared libraries

**1998**- Windows support through Cygwin

# Autotools-GNU Build System

Familiar to most users who build packages:

Configure, Make, Make check, Make install

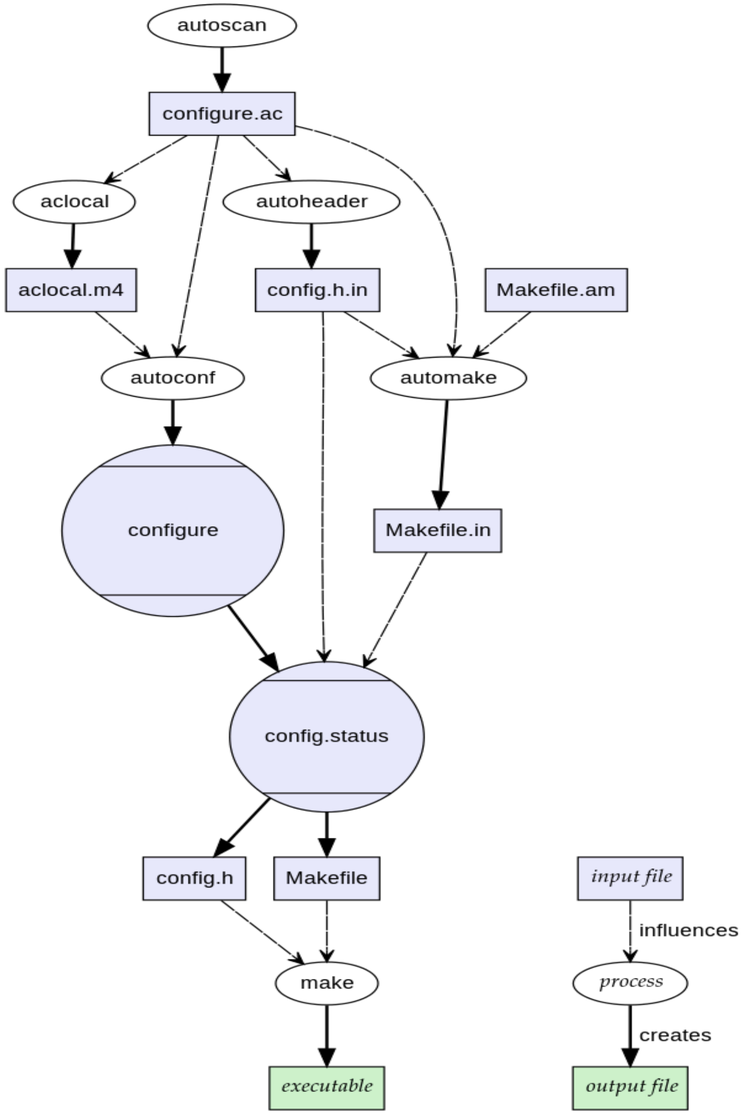
The Autotools suite:

Autoconf - creates configure file

Automake - create makefile from by running configure

Libtool - creates shared libraries

Gettext - multilingual package



Original author:Jdthood

## GNU Build System

autoscan

autoconf

automake

make

Key input files:

1. configure.ac
2. Makefile.am

# Autoconfiscating a Package

1. Create a Makefile.am (much simpler than a Makefile usually) for each directory in the build tree
2. Run autoscan to create a configure.scan file
3. Modify the configure.scan to make a configure.ac file
    a. Look for avx/mmx dependencies
    b. Look for libs that autoscan missed or does not know about
    c. Compiler dependent flags
4. Run autoconf or autoreconf to produce configure file
    autoreconf -fvi
    (autoreconf is your friend!)
1. Run configure to create Makefile
2. Run make

# Default Make Targets

- make
- make install
- make check
- make clean
- make uninstall **(note cmake does not have this!)**
- make distclean
- make installstrip
- make dist

***All of these are created for you!***

Makefile writing simplified?

# Directory Structure for Installation

| **Directory Variable** | **Directory** |
|---|---|
| prefix | /usr/local |
| exec-prefix | prefix |
| bin | exec-prefix/bin |
| lib | exec-prefix/lib |
| | |
| includedir | prefix/include |
| datarootdir | prefix/share |
| datadir | datarootdir |
| mandir | datarootdir/man |
| infodir | datarootdir/info |

# Makefile.am

```
WHERE_TARGETYPE = TARGET
WHERE ->BIN prefix/bin
        LIB prefix/lib
        CUSTOM prefix/custom
TARGET_TYPES -> PROGRAMS
                LIBRARIES
                LTLIBRARIES
                SCRIPTS
                DATA
                HEADERS
```

# Makefile.am

WHERE could also be:

NOINST - don't install it (e.g. just link it into something else or use for testing)

CHECK - build for tests on "make check"

After this one must put in : TARGET_SOURCES = (FILES TO BE COMPILED)

or one can use AM_DEFAULT_SOURCE_EXT = .f90

# Makefile.am (executables)

```
BIN_PROGRAMS = exec1 exec2
EXEC1_SOURCES = exec1a.c exec1b.c
EXEC1_LDADD= -lsomelib

#ifeq ($(CXX),gcc)
EXEC1_CFLAGS= -mtune=native
#endif

SUBDIRS = sub1
```

# Makefile.am

```
BIN_PROGRAMS= foo bar
FOO_SOURCES = foo.c foo2.c
BAR_SOURCES = bar.c bar2.c
LIB_LIBRARIES = mylib.a
```

Can define custom compilation flags for each target

foo_CFLAGS (foo_FFLAGS for fortran), foo_LDFLAGS, foo_LDADD,

foo_LINK,foo_COMPILE

# Makefile.am (Libraries)

**STATIC:**

```
LIB_LIBRARIES = mylib.a
mylib_A_SOURCES = mysrc.f mysrc2.f
```

**SHARED:**

```
LIB_LTLIBRARIES = mylib.la
mylib_la_LDFLAGS = -version-info 1:2:3 -rpath /opt/lib
```

Creates static and shared versions appropriate for OS

# Makefile.am (tests)

```
CHECK_PROGRAMS= test1 test2
AUTOMAKE_OPTIONS= dejagnu
TESTS = $(CHECK_PROGRAMS)
```

This runs when you do “make check”

Debugging can be problematic for libtool libraries

# Makefile.am (include files)

```
putilsdir= $(includedir)/putils
putils_HEADERS = header.h header2.h
```

You can easily control what headers are installed and where they are installed

Fortran modules can also installed this way (Intel and gcc look in CPATH at some point)

# Configure Script

Runs m4 shell script interpreter and macro expander

If you can not find a macro for your use, you can write one using the m4 shell script language. Hundreds of macros already exist: http://www.gnu.org/software/autoconf-archive/The-Macros.html#The-Macros

Configure can be done recursively

Configure can be evoked from an external directory

# Configure.ac (boiler plate)

```
AC_INIT([amhello], [1.0], [bug-report@address]
AM_INIT_AUTOMAKE([foreign -Wall -Werror])
# Check for special options AC_ARG_WITH, AC_ARG_ENABLE
# Checks for programs AC_PROG_CC
#Checks for libraries AC_CHECK_LIBS
# Checks for header files AC_CHECK_HEADERS
# Checks for typedefs, structures, and compiler characteristics
# Checks for library functions
AC_CONFIG_HEADERS([config.h]
AC_CONFIG_FILES([Makefile src/Makefile])
AC_OUTPUT
```

# Configure.ac

AC_MSG_ERROR(ERROR-DESCRIPTION, [EXIT-STATUS])

Print ERROR-DESCRIPTION (also to config.log) and abort ‘configure’

AC_MSG_WARN(ERROR-DESCRIPTION)

Just print message warning user

AC_CHECK_LIB(LIBRARY, FUNCT, [ACT-IF-FOUND], [ACT-IF-NOT])

Can leave [ACT-IF-FOUND] blank and AC_HAVE_LIBNAME will be defined

Can leave [FUNCT] blank to just check linking with library

AC_CHECK_HEADERS([stdlib.h])

AC_CHECK_FUNCS([FUNCTION])

AX_F90_MODULE(MODULE, MODULE-REGEXP, FUNCTION-BODY
[, SEARCH-PATH [, ACTION-IF-FOUND [, ACTION-IF-NOT-FOUND]]])

# Adapting to the Environment

Write config.h.in with macros to substitute for functions that do not exist for your environment or point to substitute functions

```
#ifndef HAVE_MAINFUNC

#define mainfunc() someotherfunct()

#endif
```

Put in #if/else in code to adapt to what is available

# Old Style

```
#include <sys/time.h>

struct stopwatch {
    struct timeval ts,tf;
    double acc_time;
    void start() { gettimeofday(&ts,0x0);};
};
```

# New Style

```
#include “config.h”
#ifdef HAVE_CLOCKGETTIME
#include <ctime>
struct stopwatch {
        struct timespec ts,tf;
        double acc_time;
        void start() { clock_gettime(CLOCK_MONOTONIC,&ts);};
};
#elif HAVE_GETTIMEOFDAY
#include <sys/time.h>
struct stopwatch {
         struct timeval ts,tf;
         double acc_time
         void start() { gettimeofday(&ts,0x0);};
};
#else
use clock() function and clock_t ts,tf
#endif
```

# Environmental Variables

**Autoconf in general:**

CXX, CC, FC, CFLAGS (etc for  other languages), LDFLAGS, LD_LIBRARY_PATH

**Most compilers can use:**

LIBRARY_PATH (*better than LD_LIBRARY_PATH*)

CPATH,C_INCLUDE_PATH,F_INCLUDE_PATH, CPLUS_INCLUDE_PATH, F_MODULE_PATH, etc.

## Advantages

- Very Well Documented
- A multitude of features and “tricks”
- Easy to do multiple builds from the same source directory
- Easy for users when everything works well (configure,make,make install)
- Most of the autotools suite does not need to be present for the casual user

## Disadvantages

- Slower install than just using make
- Need a POSIX environment
- Complexity
- Hard for the casual user to fix things when it goes wrong
- Still depends on some environmental variables to function (CXX,CC,FC and so on)

# So what else is out there?

- CMake (similar problems to Autoconf but without much documentation). Main advantage is ccmake curses/gui interface and it works in Windows environment(?)
- Maven (complicated as Autoconf and very java centric)
- Ant (another java tool)
- SCons (python tool which requires some code writing to work)

**Perhaps what is needed is another program on top of the GNU build system**

Thanks for attending!
Any Questions?