Using Not AutoMake to develop platform independent build environments

CAIA Technical Report 050412A

Jason But
April 12th, 2004

Table of Contents

1. Introduction
2. Producing Platform Independent Code
   1. Writing Platform Independent Source Code
   2. Providing a Platform Independent Build Environment
3. The GNU AutoTools Suite
   1. Automake
   2. Autoconf
   3. Using Autoconf Without Automake
4. Not AutoMake (NAM)
5. Using NAM
   1. Required Files
      1. NAM_rules.mk.in
      2. bsd.mk
      3. gnu.mk
   2. Sample Files
      1. Makefile.in
      2. configure.in
   3. Building the Project
   4. Distributing the Project
   5. Examples
6. Download NAM
7. Conclusion

Introduction

While not all developed software - particularly tools developed for research purposes - is suitable for widespread distribution, it is still a good idea to develop software with a distinct goal of being able to compile and execute it on a variety of different platforms.

If we do intend to publically distribute code, then it becomes imperative to ensure that it will compile and execute on various platforms. This is even an issue when considering the same platform as different builds may have different names for their development tools, or may have libraries installed in different directories.

Recently, the general approach taken for distribution of platform independent code is the configure/make/make install cycle. In this technical report we discuss different ways to build this...
Using Not AutoMake to develop platform independent build environments

Introduction

In this document, I will discuss the GNU Autotools suite, why Automake is not as useful as Autoconf, and how to use Autoconf without Automake. I will then discuss our Not AutoMake (NAM) tool and how to use it to develop platform independent configure scripts.

Producing Platform Independent Code

There are two separate issues to overcome when developing platform independent software:

- Writing platform independent source code
- Providing a platform independent build environment

Writing Platform Independent Source Code

A complete treatise of writing platform independent source code is beyond the scope of this document, but some tips that can be used to help guide you are:

- Use libraries that are either standard or available on a wide range of platforms.
- Write standards compliant code (C/C++) - this increases the chance that your source code will work as expected with other compilers.
- If necessary, group platform dependent code into a small set of files with a common API - this allows you to compile in support for your platform of choice without changing the layout and implementation of common sections of code.

Providing a Platform Independent Build Environment

The build environment is what enables the end user to build and install your software in such a way that they can later use it. This can be complex as you cannot guarantee that certain tools and libraries will be available on the end user's platform. Similarly, the tools you require may be present but available under a different name. Finally, installation of your software after the build process can be highly platform dependent - this might include the location of files after installation and whether or not any of the files need to be registered.

While it is possible for each project to have its own way of configuring the build environment, and then building and installing the software, the configure/make/make install cycle is fast becoming the de-facto technique to do so. In this case:

- **configure** - A shell script that will scan the system, checking for all required tools and libraries and noting where they are installed. The script then builds a set of platform independent Makefiles that can be used to build and install the software in such a way that it is properly configured for the system under consideration. If the configure script fails to find any required tools or libraries, it typically fails with an appropriate error message.
- **make** - Uses the standard make tool to build the software using the Makefiles generated during the first stage. While make is a standard tool, its implementation is not and so the Makefiles under consideration must be written in a way that is compatible with all versions of make.
- **make install** - Uses the system make tool to install the compiled software into the correct location.

The primary difficulty in this scenario is how to write the configure script. In order for the script to function in all environments, we are restricted to using the standard Bourne Shell, the only scripting language which is practically guaranteed to exist on all platforms. While the shell itself may be present, this means we are limited in available functionality that more complex shells and scripting languages may offer.

The GNU AutoTools Suite

The GNU Autotools suite is a set of tools that aim to reduce the complexity in developing scripts to support the configure/make/make install software buildcycle. The suite consists of a number of tools but in terms of building application software, the tools of most interest are Automake and Autoconf.

The relationship between the tools in the GNU Autotools suite and their input and output files are shown
in Fig. 1. For most developers, we need only create a `configure.in` and a set of `Makefile.am` files prior to running `automake` and `autoconf` to generate a final `configure` script and set of `Makefile.in` files. Execution of `configure` will use the `Makefile.in` files as templates to generate the `Makefiles`.

**Automake**

`Automake` uses the `configure.in` and a set of `Makefile.am` files as input to help it generate a set of `Makefile.in` files. `Automake` extracts a list of `Makefile.am` files to consider and some `automake` specific macros from the `configure.in` file and uses this information to help it decide what contents to put into the `Makefile.in` files. The `Makefile.am` files contain a list of targets separated into different types and a list of sources that are required to build those targets. `Automake` then pieces together the required rules to build the specified targets from a library to `Makefile` segments to generate a templated set of `Makefile.in` files. Variables in the `Makefile.in` templates are later substituted by values determined when running the `configure` script.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>• <code>Makefile.am</code> format is simple and easy to write.</td>
<td>• Adding extra rules that must be built at certain stages is difficult.</td>
</tr>
<tr>
<td>• No need to worry about writing a platform independent <code>Makefile</code>.</td>
<td>• Generated <code>Makefile.in</code> template files are complex and difficult to understand.</td>
</tr>
<tr>
<td>• Large number of different types of targets supported</td>
<td>• The final <code>Makefiles</code> created using <code>automake</code> generated <code>Makefile.in</code> files are difficult to read</td>
</tr>
<tr>
<td></td>
<td>○ Understanding build process.</td>
</tr>
<tr>
<td></td>
<td>○ Debugging build/compile problems.</td>
</tr>
</tbody>
</table>

The biggest problem with `automake` is the quality of the generated `Makefiles`. While the execution of `make` is fully functional, the output is unwieldy and trying to debug the build process using the contents
of the Makefiles is not easy - as the contents of these Makefiles are not commented or spaced out. An example of an automake generated Makefile can be downloaded here.
As can be seen, each step in the build process is accompanied by a complex output string. On a typical terminal window, these build lines can often consume up to ten rows of the terminal display, with the command scrolling off the screen at a high rate. This type of output is unnecessary and unprofessional (in my humble opinion). A developer has issues of trying to spot compiler warning messages amongst the jumble of output, while trying to determine the progress of the build process is difficult since the text often scrolls by very quickly. This problem is often seen when building large open-source projects that compile large numbers (often over 100) source code files. Also, unless actually working on parameters, command scrolling off the screen at a high rate. This type of output is unnecessary and unprofessional (in my humble opinion). A developer has issues of trying to spot compiler warning messages amongst the jumble of output, while trying to determine the progress of the build process is difficult since the text often scrolls by very quickly. This problem is often seen when building large open-source projects that compile large numbers (often over 100) source code files. Also, unless actually working on parameters, command scrolling off the screen at a high rate. This type of output is unnecessary and unprofessional (in my humble opinion). A developer has issues of trying to spot compiler warning messages amongst the jumble of output, while trying to determine the progress of the build process is difficult since the text often scrolls by very quickly. This problem is often seen when building large open-source projects that compile large numbers (often over 100) source code files. Also, unless actually working on parameters, command scrolling off the screen at a high rate. This type of output is unnecessary and unprofessional (in my humble opinion). A developer has issues of trying to spot compiler warning messages amongst the jumble of output, while trying to determine the progress of the build process is difficult since the text often scrolls by very quickly. This problem is often seen when building large open-source projects that compile large numbers (often over 100) source code files. Also, unless actually working on parameters, command scrolling off the screen at a high rate. This type of output is unnecessary and unprofessional (in my humble opinion). A developer has issues of trying to spot compiler warning messages amongst the jumble of output, while trying to determine the progress of the build process is difficult since the text often scrolls by very quickly. This problem is often seen when building large open-source projects that compile large numbers (often over 100) source code files. Also, unless actually working on parameters, command scrolling off the screen at a high rate. This type of output is unnecessary and unprofessional (in my humble opinion). A developer has issues of trying to spot compiler warning messages amongst the jumble of output, while trying to determine the progress of the build process is difficult since the text often scrolls by very quickly. This problem is often seen when building large open-source projects that compile large numbers (often over 100) source code files. Also, unless actually working on parameters, command scrolling off the screen at a high rate. This type of output is unnecessary and unprofessional (in my humble opinion). A developer has issues of trying to spot compiler warning messages amongst the jumble of output, while trying to determine the progress of the build process is difficult since the text often scrolls by very quickly. This problem is often seen when building large open-source projects that compile large numbers (often over 100) source code files. Also, unless actually working on parameters, command scrolling off the screen at a high rate. This type of output is unnecessary and unprofessional (in my humble opinion). A developer has issues of trying to spot compiler warning messages amongst the jumble of output, while trying to determine the progress of the build process is difficult since the text often scrolls by very quickly. This problem is often seen when building large open-source projects that compile large numbers (often over 100) source code files. Also, unless actually working on parameters, command scrolling off the screen at a high rate. This type of output is unnecessary and unprofessional (in my humble opinion). A developer has issues of trying to spot compiler warning messages amongst the jumble of output, while trying to determine the progress of the build process is difficult since the text often scrolls by very quickly. This problem is often seen when building large open-source projects that compile large numbers (often over 100) source code files. Also, unless actually working on parameters, command scrolling off the screen at a high rate. This type of output is unnecessary and unprofessional (in my humble opinion). A developer has issues of trying to spot compiler warning messages amongst the jumble of output, while trying to determine the progress of the build process is difficult since the text often scrolls by very quickly. This problem is often seen when building large open-source projects that compile large numbers (often over 100) source code files. Also, unless actually working on parameters, command scrolling off the screen at a high rate. This type of output is unnecessary and unprofessional (in my humble opinion). A developer has issues of trying to spot compiler warning messages amongst the jumble of output, while trying to determine the progress of the build process is difficult since the text often scrolls by very quickly. This problem is often seen when building large open-source projects that compile large numbers (often over 100) source code files. Also, unless actually working on parameters, command scrolling off the screen at a high rate. This type of output is unnecessary and unprofessional (in my humble opinion). A developer has issues of trying to spot compiler warning messages amongst the jumble of output, while trying to determine the progress of the build process is difficult since the text
The code in the final `configure` script essentially runs a series of system tests, storing the results in shell variables. Upon conclusion, these variables are substituted into appropriate locations in the `Makefile.in` files to generate the final `Makefiles` for the build.

### Advantages
- Pre-existing macro set to check for the existence of:
  - Tools
  - Programs
  - Libraries
  - Headers
- M4 Macro Language to simplify understanding of `configure.in`
- Can directly use shell script in `configure.in`
- Can be used without `automake`

### Disadvantages
- M4 Macro Language means needing to learn yet another language
- The `autotools` suite of applications must be executed in the correct order to regenerate all required files and a correct `configure` script

The ability to easily check for the existence and locations of libraries installed on the system makes the developer task much easier. The final `configure` script can fail with an appropriate error message rather than the rather cryptic message printed at compile or link time during `make`. Also, as `configure` finds the locations of installed libraries, it is easier to ensure that they will be correctly used at build time.

It is also possible to not write a check for a required package in `configure.in` - in this case executing `configure` will be successful and the error will instead occur during the build (`make`) process.

For further assistance in writing `configure.in` files, please consult the [Autoconf Manual](http://autoconf.apache.org/download/manuals/autoconf.html).

**NOTE:** Autoconf will process either `configure.in` or `configure.ac` as input files. I prefer `configure.in` as it is consistent with the `Makefile.in` files we will also be generating.

### Using Autoconf Without Automake

I previously mentioned that it is possible to use the `autoconf` tool without using `automake`. In this case we need to manually generate the `Makefile.in` template files. As per the [Autoconf Manual](http://autoconf.apache.org/download/manuals/autoconf.html), variables defined in the `configure` script will be substituted into the `*.in` templates where the variable name is both prefixed and suffixed with the `@` character (eg. `@srcdir@` will be replaced with the value of the `srcdir` variable, containing a link to the directory containing the source).

The primary advantage is that any comments inserted into the `Makefile.in` templates will be migrated to the final `Makefiles`, similarly the structure of the template files will also be kept. This allows us to ensure that the final generated `Makefiles` will be neat and readable by any user looking to debug the build process.

### Not AutoMake (NAM)

The primary problem with not using `automake` is that the entire build process must now be manually created in the `Makefile.in` templates. In order to minimise the effort required to write these template files, I have developed Not AutoMake (NAM). NAM is a set of files that implement core project build functionality in a way that minimises the effort involved in writing `Makefile.in` file, it is based in spirit on the process used in the WINE project which also uses `autoconf` but not `automake`. The primary differences include comments, support for BSD Make as well as GNU Make and nice build output.

NAM offers the following features:

- Default build targets, all these targets are recursively called in any subdirectories processed by the `Makefile`
  - `all` - Build all targets
  - `clean` - Remove all temporary build files and final targets
  - `install` - Install software to the system
  - `uninstall` - Remove software from the system
- Recursive `make` in listed subdirectories
- C++ compilation of source code
- Linking of C++ archives (*.a) and executables
- Optional clean or verbose output during build
- Automatic generation of dependencies
- Automatic re-execution of `autoconf` and `configure` if necessary
  - `configure` in has changed
  - One of the templated *.in files have been modified
- Readable Makefiles
- Installation of executables in either `$(prefix)/usr/bin` or `$(prefix)/usr/sbin`
- Installation of man pages to system
- Application specific extensions to the `all`, `install` and `uninstall` targets

**Using NAM**

Details on downloading NAM are at the end of the document.

**Required Files**

NAM consists of three files, `NAM_rules.mk.in`, `bsd.mk` and `gnu.mk`. These three files should all be placed in the top level directory of the project to be built. You should not need to edit or modify these files in any way unless extending the functionality of NAM, in which case it would be appreciated if these changes are forwarded back to CAIA for incorporation into NAM.

**NAM_rules.mk.in**

Contains all the basic build rules to support the functionality provided by NAM, including regeneration of the Makefiles, recursive `make`, and all available targets. This file will be used as a template during execution of `configure` to generate `NAM_rules.mk`, which will subsequently be inserted into all other Makefiles for the project.

**bsd.mk**

Included from `NAM_rules.mk` during execution of `make` only if BSD Make is being executed. This includes some BSD make specific instructions to perform the tasks of:

- Setting the *verbosity* of make build output
- Determining dependencies for multiple targets
- Determining files to delete during `make clean`
- Including any automatically generated dependency includes
- Regenerating the Makefiles if necessary

**gnu.mk**

Included from `NAM_rules.mk` during execution of `make` only if GNU Make is being executed. This includes some GNU make specific instructions to perform the tasks of:

- Setting the *verbosity* of make build output
- Determining dependencies for multiple targets
- Determining files to delete during `make clean`
- Including any automatically generated dependency includes

**Sample Files**

A sample `Makefile.in` and `configure.in` are also provided to assist in setting up the build environment for your project. Your project should have a single `configure.in` file situated in the top level directory of the project and a number of `Makefile.in` files, one in each recursive level you wish to run `make` in. It is absolutely necessary for a `Makefile.in` to exist in the top level directory of the project.

**Makefile.in**
The contents of the sample Makefile.in are listed below.

```plaintext
# File: Makefile
# Makefile for the project.

# Variables required by the global make rules.
# TOPSRCDIR   - Points to the top level directory of the project.
# SRCDIR     - Points to the actual directory where the source code files for the Makefile exists.
# VPATH    - Directory to look for source files if not in the current directory. Must be equal to SRCDIR.
# TOPBUILDDIR - The top level directory we initially ran make from, used for generating nice output of the working directories.

TOPSRCDIR = @top_srcdir@
SRCDIR = @srcdir@
VPATH = @srcdir@
TOPBUILDDIR = ./@top_builddir@

# Recursively make in these subdirectories
SUBDIRS = subdir1 subdir2

# Compile flags, list include directories
INCLUDES = -I$(TOPSRCDIR)/extrainclude -I$(TOPSRCDIR)/include2

# List of programs and archives to link
PROGRAMS = prog1 prog2
ARCHIVES = libar1 libar2

# Source files for each target
prog1_SRCS = prog.cpp p1.cpp
prog2_SRCS = prog.cpp p2.cpp
libar1_SRCS = ar1.cpp ar2.cpp
libar2_SRCS = arA.cpp arB.cpp

# Extra libraries for link stage (only if needed)
prog1_LIBS = -lextralib
prog1_LDFLAGS = -L/path/to/extralib

# Precompiled headers
PRECOMP_HEADER = myheader.h

# Stuff to install
INSTALL_BIN = prog1
INSTALL_SBIN = prog2
INSTALL_MAN = prog1.5
@NAM_RULES@ 

# End of File: Makefile
```

The comments in this file are optional and can be removed, however all comments will be copied into the final Makefile so their presence can be useful.

The first four lines of the Makefile.in file are mandatory and MUST NOT be commented out or deleted. These lines ensure that variables will be set for each Makefile to describe the location of certain required directories.

The remaining variables may be commented out or deleted as required, these variables have the following uses:

- **SUBDIRS** - A list of all subdirectories to recurse into when running `make`
- **INCLUDES** - Compiler flags to define extra directories to search for included files
- **PROGRAMS** - A list of executables to link
- **ARCHIVES** - A list of temporary library archives (*.a) to link
- **xxx_SRCS** - A list of C++ source files to compile in order to build program **xxx**, where **xxx** is a program in the **PROGRAMS** or **ARCHIVES** variable
- **xxx_LIBS** - A list of libraries to use when linking program or archive **xxx**
- **xxx_LDFLAGS** - Linker flags to use when linking program or archive **xxx**
- **PRECOMP_HEADER** - A list of header files to compile using pre-compiled headers (note: this requires gcc 3.4 or greater)
- **INSTALL_BIN** - A list of executables to install to the **bin** directory
- **INSTALL_SBIN** - A list of executables to install to the **sbin** directory
- **INSTALL_MAN** - A list of man pages to install to the **man** directory

These variables should be declared **BEFORE** the line containing `@NAM_RULES@` (which includes the contents of `NAM_rules.mk`).

Any program specific targets and extra instructions to run for the **all**, **install** and **uninstall** targets **MUST** be listed after the line containing `@NAM_RULES@`, otherwise the default target of **all** will be replaced.

When writing rules you need to be aware how output is taken care of during build. When **make** is run in **clean**(default) or quiet **verbose** mode (see below), then the Makefile is processed in silent mode, as if `@` is prepended to all commands in all build rules. When **make** is run in **verbose** mode, all commands in all build rules will be echoed to the display.

- If you wish to echo a string to the display in all three modes, you should do so with the:
  ```
  @echo
  ```
  command. The actual command will never be echoed to the display, but its output will always be shown
- If you wish to echo a string to the display only in default output mode, you should do so with the:
  ```
  $(NICE_ECHO)
  ```
  command. This command resolves to `@echo` in default output mode and to `@:` in the other two output modes (meaning the command is not displayed AND there is no actual output either

configure.in

The contents of the sample configure.in are listed below.

```
# The minimum version of autoconf required to regenerate the configure script. #
AC_PREREQ(2.59)

# Initialise autoconf, set package name, version number and contact details. #
AC_INIT(my_prog, 0.1.2, [Contact Details])

AC_CONFIG_SRCDIR(src/myprog.cpp)

# Check for programs used to build my prog #
AC_PROG_MAKE_SET
AC_PROG_CXX
AC_PROG_RANLIB
AC_PROG_INSTALL
AC_LANG(C++)

# Check whether make sets the MAKE variable. #
# Check which C++ compiler we have (sets CXX and CXXFLAGS) #
# Set the language for all further tests to C++ #
AC_PROG_MAKE_SET
AC_PROG_CXX
AC_PROG_RANLIB
AC_PROG_INSTALL
AC_LANG(C++)
AC_ARG_PROGRAM

# Check how dependencies are created for the C++ compiler on this system #
AC_IF[<<-MV 2>/dev/null,
    [AC_SUBST(CPDPFLAGS, "-MM -MF \"\$@\".d\") [AC_SUBST(DEPDIR, "..d")] [AC_SUBST(DEPDIR, ".")]]

# Add macros to check for other programs here

# Checks for header files. #
# Add macros to check for header files here
```
This contains the minimal set of macros required in a configure.in file to work within the NAM architecture. You should add extra tests (macros) where indicated by the comments. You must also ensure that your Makefiles are listed AFTER NAM_rules.mk in the AC_CONFIG_FILES macro, otherwise an old copy of NAM_rules.mk will be included into your Makefile. For more information in writing configure.in macros, please consult the Autoconf Manual.

Building the Project

When your project is ready to go, you can build it by invoking the configure script to prepare the Makefiles initially. I prefer to invoke the build in a different subtree of the project so that object and other files do not get mixed up with the source code for the project. To do this for a project in a directory called my_proj you can execute the following steps:

```
cd my_proj
autoconf
mkdir build
cd build
./configure
```

Executing autoconf will generate the configure script initially, while running configure from the build subdirectory will cause the top level directory for the build to be in the build directory, this is where you will find the top level Makefile.

You can now execute make from this directory to build and install the software. It is no longer necessary to rerun autoconf as the Makefile you have just generated will ensure this occurs if required (due to a change in any of the *.in files).

To build the software with clean (default) output:

```
make
```

or

```
make VERBOSE=1
```

To build the software with verbose output:

```
make VERBOSE=2
```

To build the software with silent output:

```
make VERBOSE=3
```

To install the software with clean (default) output:

```
make install
```

or

```
make VERBOSE=1 install
```
To clean the build tree with verbose output:

make VERBOSE=2 clean

The same verbosity rule is applied to other targets.

Sample output during execution of make for the netsniff project is shown below.

```
make[1]: Entering directory `/home/jbut/Development/test/build/src'
make[2]: Entering directory `/home/jbut/Development/test/build/src/Crypto-PAn.1.0'
g++ -g -O2 -DHAVE_CONFIG_H -t -I./src/Crypto-PAn.1.0/include -MMD -MP -MF `dirname protocols/ppppacket.o`/.deps/`basename protocols/ppppacket`.d -c -o protocols/ppppacket.o ../../src/protocols/ppppacket.cpp
make[2]: Leaving directory `/home/jbut/Development/test/build/src/Crypto-PAn.1.0'
ranlib libCryptoPAn.a
ar cru libCryptoPAn.a panonymizer.o rijndael.o
make[2]: Leaving directory `/home/jbut/Development/test/build/src/Crypto-PAn.1.0'
```

Sample output during execution of make VERBOISE=2 for the netsniff project is shown below.

```
gmake[1]: Entering directory `/home/jbut/Development/test/build/src'
gmake[2]: Entering directory `/home/jbut/Development/test/build/src/Crypto-PAn.1.0'
g++ -g -O2 -DHAVE_CONFIG_H -t -I./src/Crypto-PAn.1.0/include -MMD -MP -MF `dirname panonymizer.o`.deps/`basename panonymizer`.d -c -o panonymizer.o ../../src/Crypto-PAn.1.0/panonymizer.cpp
g++ -g -O2 -DHAVE_CONFIG_H -t -I./src/Crypto-PAn.1.0/include -MMD -MP -MF `dirname rijndael.o`.deps/`basename rijndael`.d -c -o rijndael.o ../../../src/Crypto-PAn.0/rijndael.cpp
```

Leaving scripts
Making in scripts
Leaving src
Making in src
Leaving src/Crypto-PAn.0
Making in src/Crypto-PAn.0
Making in src
Making in src/Crypto-PAn.1.0
```
Using Not AutoMake to develop platform independent build environments

Examples

You should distribute your project with a prebuilt configure script, not every platform will have autoconf installed.

Distributing the Project

You should distribute your project with a prebuilt configure script, not every platform will have autoconf installed.

Examples
The pkthisto and netsniff projects provide a NAM based build environment. They can be downloaded as samples of how to use NAM in a real project.

**Download NAM**

The NAM set of files is maintained and made available [here](#).

**Conclusion**

While autoconf can be used to facilitate the building of Makefiles for building platform independent software, the use of automake leaves a result that appears unprofessional and is difficult to understand. This precipitated the development of Not AutoMake (NAM). NAM allows the use of autoconf without automake, with simple specification input Makefile.in files, and resultant Makefiles that are both easy to read and follow, while producing a much nicer output during the build process.