FSK Consulting Incorporated
www.fsk-consulting.com
Qs-GnuBuildEnvironment.odt

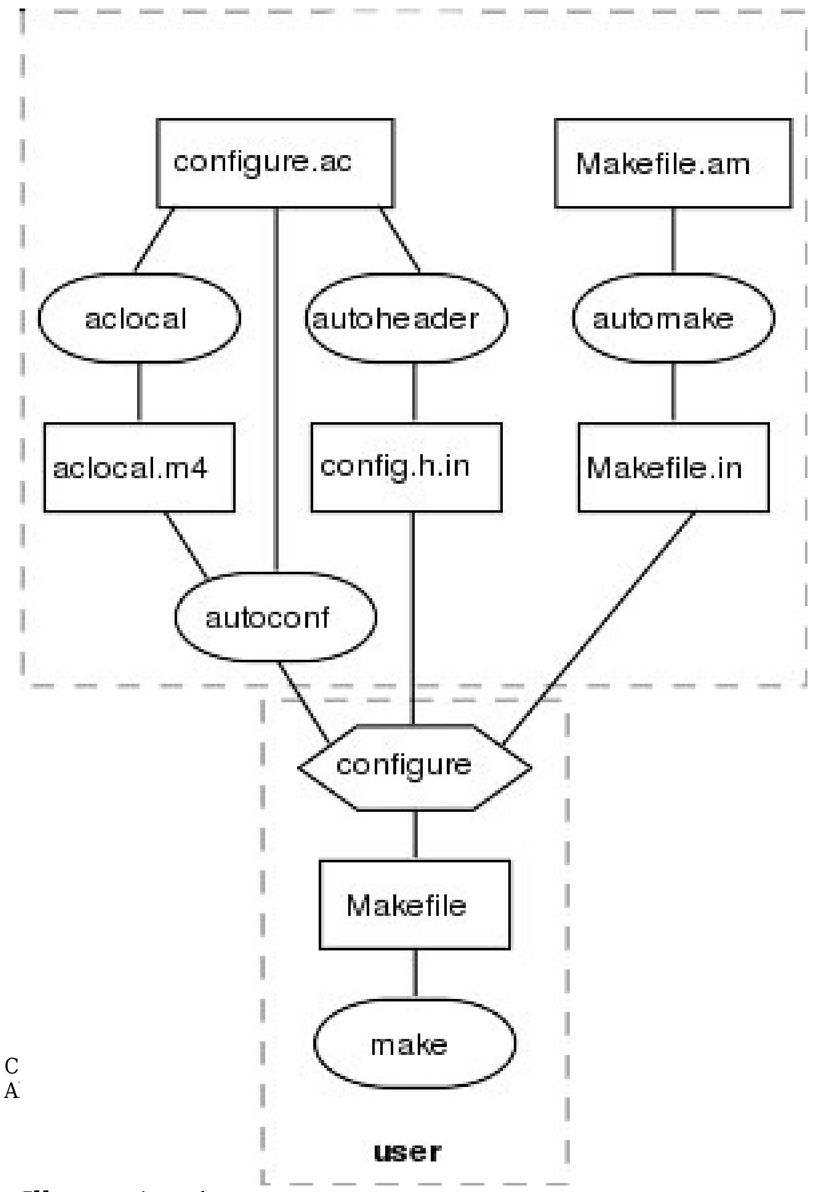
Introduction to the GNU Build System

Brief

Chances are that if you've ever obtained a software product and built it from source the instructions defined a 3-step installation procedure of 'configure', 'make', and 'make install' you've utilized the GNU Build Environment (also known as autotools). This environment has become the de facto standard in deploying software from source on Posix-like systems. The goal of this paper is to introduce you to the build environment enough to incorporate it into your own software products.

Introduction

The GNU Build System (also known as autotools) is a suite of tools authored by Gnu used to make portable software products. The build system will be viewed from two perspectives; 1) the user perspective as the user of the product, and 2) the developers perspective as the developer of the product. The remainder of this paper will discuss the system from both perspectives.



C
A

Illustration 1: http://en.wikipedia.org/wiki/Autotools/Autoconf.svg

Discussion

To begin understanding the GNU Build Environment, you must first view it from two perspectives; that of the user, and that of the developer.

User Perspective

After obtaining the software archive, the user simply wants to configure, build, test, and install the product. The easier this is to do, the more likely the user will use the product. A competing objective however is to not sacrifice flexibility. The GNU build environment offers both flexibility as well as simplicity from the user perspective. The user proceeds with a 3-step procedure.

Step 1 – Configure

This step is responsible for interrogating the users system and makes note of the systems environment, platform, architecture and the existence of required features that the software product requires. A plethora of configurable parameters are available to the user in specifying how the product will be installed. A notable example of a configurable parameter is the location of which the product will be installed. Each configurable parameter has a specified default and specifying an alternative is only necessary if the default is not acceptable. Successful execution of the configuration script results in a customized *Makefile* for the users system. This allows the user to proceed to the next step; building the software from source. If however the configuration script fails to locate an essential feature on the users system the user must resolve the issue(s) and repeat this step until successful.

Step 2 – Make

The generated *Makefile* from step-1 supports this step as well as the final installation step. The makefile is populated with the discovered specifics of the users system discovered in the previous step. The default makefile target builds the software allowing the user to simply type *make* to kick off the build. After a successful build, the user may choose to test the product and use it in the build directory. If the product meets the user's needs, the final step will install the product.

Step 3 – Make Install

The third and final step involves installing the product. This involves copying binaries,

Copyright 2007
All Rights Reserved

documentation, libraries and the like to their final destinations. While these destinations can be specified at time of configuration, the defaults generally specify a subdirectory of */usr*. Write permissions of this directory structure normally requires this step to be executed with superuser privileges (root on *nix variants).

Developer Perspective

The remainder of this section will guide you step-by-step through an example which incorporates the build environment.

We start by creating a subdirectory, which will be the location of our product. This is necessary because throughout the course of this procedure a good number of files will be generated as well as directories and keeping them in a subdirectory makes for easier cleanup and understanding.

```
localhost:~$ mkdir autotools/
localhost:~$ cd autotools/
```

Before we go further it is important to understand our primary goal, which is to give the user the previously defined experience. The initial interface for the user is the 'configure' script, all other interfaces (e.g. Makefile) are auto-generated from this script with input files that will present themselves shortly. The creation of the configuration script therefore is our goal of all to come.

To produce a configuration script for a software product you create a *configure.ac* file that contains invocations of the autoconf macros. The macros interrogate the system testing for features necessary for the software product. For reasons that will be explained later generate the *configure.ac* file as specified:

Step 1

```
localhost:~/autotools$ cat configure.ac
AC_PREREQ(2.59)
AC_INIT(FULL-PACKAGE-NAME, VERSION, BUG-REPORT-ADDRESS)
AM_INIT_AUTOMAKE(someApp,2.1)
AC_CONFIG_HEADER([config.h])
AC_PROG_CC
AC_OUTPUT(Makefile)
```

Also create the following files as specified:

```
localhost:~/autotools$ cat myMain.c
#include <stdio.h>
#include "myMath.h"

int main()
{
 const double radius=15.0;
 const double area =
                areaOfCircle(radius);
 printf("(%s:%d) the area of a circle” +
         “with radius %.2lf ft == %.2lf”+
         “ft^2\n",
        __FILE__,__LINE__,radius,area);
}

localhost:~/autotools$ cat myMath.h
#ifndef MYMATH_H
#define MYMATH_H
```

```
double areaOfCircle (double radius);
#endif

localhost:~/autotools$ cat myMath.c
#include "myMath.h"
#include <math.h>

double areaOfCircle (double radius)
{
  const double retVal =
                  pow(radius,2.0)*M_PI;
  return retVal;
}

localhost:~/autotools$ cat Makefile.am
bin_PROGRAMS=someApp
someApp_SOURCES=myMain.c myMath.c
```

Autoconf is a tool that generates the *configure* script from directives in the *configure.ac* file. Autoconf is a macro-expander, an application that repeatedly performs macros expansions on input, replacing macro declarations with macro bodies and at end produces a pure *sh* configuration script. Since *sh* is available on Posix-like systems this ensures the highest degree of portability. The next step is done specifically because autoconf is a macro expander.

Step 2

```
localhost:~/autotools$ aclocal
localhost:~/autotools$ ls
aclocal.m4 configure.ac  myMain.c  myMath.h autom4te.cache  Makefile.am   myMath.c
```

Notice after executing aclocal the existence of aclocal.m4 and autom4te.cache. The application aclocal creates the file aclocal.m4 by combining stock installed macros, user defined macros to define all the macros required by configure.ac. The autom4te.cache directory is simply a cache for the other tools so later invocations of autoheader, automake, or other tools will not require re-processing configure.ac. This can result in speed-ups of 30x or better. This cache directory may be deleted as you wish and will be regenerated upon execution of autoconf, our next step.

Step 3

```
localhost:~/autotools$ autoconf
localhost:~/autotools$ ls
aclocal.m4 configure Makefile.am  myMath.c autom4te.cache configure.ac  myMain.c myMath.h
```

After execution of autoconf, the configure script exists and we are a good way toward home. The GNU coding standards require the existence of a few files which include NEWS, README, AUTHORS, and ChangeLog. The absence of these files will cause a good deal of warnings in the future so we will generate them in this next step.

Step 4

```
localhost:~/autotools$ touch NEWS README AUTHORS ChangeLog
localhost:~/autotools$ ls
aclocal.m4  autom4te.cache  configure     Makefile.am  myMath.c  NEWS
AUTHORS     ChangeLog       configure.ac  myMain.c     myMath.h  README
```

Copyright 2007
All Rights Reserved

Step 5

The net step will create a template file for C-style #define statements for configure to use. The execution of autoheader will result in the generation of config.h.in file which will later be used by configure.ac.

```
localhost:~/autotools$ autoheader
localhost:~/autotools$ ls
aclocal.m4      ChangeLog    configure.ac  myMath.c  README AUTHORS  config.h.in
Makefile.am   myMath.h autom4te.cache  configure    myMain.c      NEWS
```

Step 6

One last step sparky and you've got an environment. Automake is a tool that automatically generates a Makefile.in that is a compliant with GNU Coding Standards.

```
localhost:~/autotools$ automake -a
lsconfigure.ac:3: installing `./missing'
configure.ac:3: installing `./install-sh'

Makefile.am: installing `./INSTALL'
Makefile.am: installing `./COPYING'
Makefile.am: installing `./depcomp'
localhost:~/autotools$ ls
aclocal.m4      config.h.in   depcomp      Makefile.in  myMath.h
AUTHORS         configure     INSTALL      missing      NEWS
autom4te.cache  configure.ac  install-sh   myMain.c     README
ChangeLog       COPYING       Makefile.am  myMath.c
```

Now without knowing the details of what is really going on, you've got a configure script that gives the user the experience we earlier described. We can now re-examine our steps in a bit more detail.

Step 1 – Revisited

Configure.ac

```
localhost:~/autotools$ cat -n configure.ac
1 AC_PREREQ(2.59)
2 AC_INIT(FULL-PACKAGE-NAME, VERSION, BUG-REPORT-ADDRESS)
3 AM_INIT_AUTOMAKE(someApp,2.1)
4 AC_CONFIG_HEADER([config.h])
5 AC_PROG_CC
6 AC_OUTPUT(Makefile)
7
```

Line 3 is of particular significance as it specifies the name of the product someApp and product version 2.1. The product and version are of particular interest when distributing the product. Line 4 specifies that you want a config.h.in file to be generated by autoheader. If this line does not exist autoheader will fail in creating the file.
I'll leave the details of the remaining lines undefined, you may look them up at your convenience.

Makefile.am

```
localhost:~/autotools$ cat -n Makefile.am

1  bin_PROGRAMS=someApp
2  someApp_SOURCES=myMain.c myMath.c
3
```

This file defines a template for automake to generate the Makefile.in file. Line 1 specifies the name of the application, the following line specifies the sources that define the application. These two lines indirectly result in the necessary Makefiles to distribute, build, and install the product. The

Copyright 2007
All Rights Reserved

needs to author your own Makefiles are no longer required since this method is far more portable.

Step 2 – Revisited

Not much to revisit in this step. It simply defines all macros required by configure.ac.

Step 3 – Revisited

Autoconf parses the configure.ac file and generates the configure script.

Step 4 – Revisited

The created files in step 4 are in compliance with the GNU Coding Standards. They should be populated as per conventions located at http://www.gnu.org/.

Step 5 – Revisited

The autoheader step acts on the directive on line 4 of the configure.ac which simply put says you to create a template header for configure (config.h.in). This template will later be used to generate a config.h file that can be imported in your applications source. Various macro definitions that identifiy the existence of features and files are located in this file. These directives can be used to make portable code.

Step 6 - Revisited

Automake helps create portable Makefile(s) which are then processed by the make utility. It takes the Makefile.am file as input and generates a Makefile.in as output. This file is later used by configure.

Conclusion

You've now configured a product to adhere to the GNU Build Environment.

Placing yourself in the user role, specifying 'configure', 'make', and 'make install' will result in installing the product.

Placing yourself in the developer role; specifying 'make dist' will result in a compressed tarball with your product, ready for distribution.

References

http://en.wikipedia.org/wiki/Autotools

http://sources.redhat.com/autobook/

http://www.gnu.org/prep/standards/

Copyright 2007
All Rights Reserved