

Maemo Diablo GNU Autotools Training Material

February 9, 2009

Contents

1	GNU Autotools	2
1.1	Introduction to GNU autotools	2
1.2	Brief history of managing portability	2
1.3	GNU autoconf	3
1.4	Substitutions	7
1.5	Introducing automake	9
1.6	Checking for distribution sanity	14
1.7	Cleaning up	15
1.8	Integration with pkg-config	15

Chapter 1

GNU Autotools

1.1 Introduction to GNU autotools

If you already feel comfortable using *GNU autotools* and creating your own *autotools* configuration files, feel free to skip this chapter.

Creating portable software written in the C language has historically been challenging. The portability issues are not restricted just to the differences between binary architectures, but also encompass differences in system libraries and different implementations of UNIX APIs in different systems. This chapter introduces GNU autotools as one solution for managing this complexity. There are also other tools available but you are most likely to encounter autotoolized projects and also Debian packaging supports autotools without too much effort.

1.2 Brief history of managing portability

When discussing portability challenges with C code, various issues will crop up:

- Same library function has differing prototypes in different UNIX-style systems.
- Same library function has differing implementations between systems (this is a hard problem).
- Same function can be found in different libraries in different UNIX systems, and because of this, we need to modify our linker parameters when we're making the final linking of our project.
- Besides these not so obvious problems, there is also of course the problem of potentially different architecture and hence different sizes for the standard C data types. This problem is best fixed by using GLib (in our case), so it will not be covered here.
- GLib also provides the necessary macros to do endianness detection and conversion.

The historical solutions to solve these problems can be roughly grouped as follows:

- The whole software is built with one big shell script which will try to pass the right options to different compilers and tools. Most of these scripts have luckily remained in-house and never seen the light of the world. They are very hard to maintain.
- Makefiles (GNU or other kind) allow some automation in building but do not directly solve any of the portability issues. The classical solution was to ship various makefiles already tailored for different UNIX systems and it was the end-user's responsibility to select the appropriate **Makefile** to use for their system. This quite often also required editing the **Makefile**, so the end-user needed to be knowledgeable in UNIX programming tools. A good example of this style is found in the game nethack (if you can find a suitably old version).
- We can automate some parts of the **Makefile** selection above by a suitable script that tries to guess the system on which it's running and select a suitable prepared **Makefile**. Maintaining such scripts is quite hard as there were historically so many different kinds of systems (different library versions, different compilers, and so on).
- We can go one step further, and automate the creation of Makefiles from such guessing script. Such scripts are normally called **configure** or **Configure** and are marked executable for your convenience. The name of the script doesn't automatically mean that the script is related to GNU autotools.
- Two major branches of such master configure scripts evolved, each operating a bit differently and suiting differing needs.
- These two scripts and a third guessing script were then combined into GNU autoconf. Since this happened many many years ago, most of the historical code has already been purged from GNU autoconf.

Besides autoconf, it became evident that a more general **Makefile** generation tool could be useful as part of the whole process. This is how GNU automake was born. It can also be used outside autoconf. We'll cover automake a bit later, but first we'll take a look at a simple autoconf configuration file (historically called as driver, but this is an obsolete term now).

1.3 GNU autoconf

autoconf is a tool that will use the GNU m4 macro preprocessor to process your configuration file and output a shell script based on the macros used in your file. Anything that m4 doesn't recognise as a macro will be passed verbatim to the output script. This means that you can include almost any shell script fragments that you want into your **configure.ac** (the modern name for the default configuration file for autoconf).

We'll start by a simple example and see how the basic configuration file works. Then we'll cover some limitations of *m4* syntax and how to avoid problems with the syntax.

```

# Specify the "application name" and application version
AC_INIT(hello, version-0.1)

# Since autoconf will pass through anything that it doesn't recognize
# into the final script ('configure'), we can use any valid shell
# statements here. Note that you should restrict your shell to
# standard features that are available in all UNIX shells, but in our
# case, we're content with the most used shell on Linux systems
# (bash).
echo -e "\n\nHello from configure (using echo)!\n\n"

# We can use a macro for this messages. This is much preferred as it
# is more portable.
AC_MSG_NOTICE([Hello from configure using msg-notice!])

# Check that the C Compiler works.
AC_PROG_CC

# Check what is the AWK-program on our system (and that one exists).
AC_PROG_AWK

# Check whether the 'cos' function can be found in library 'm'
# (standard C math library).
AC_CHECK_LIB(m, cos)

# Check for presence of system header 'unistd.h'.
# This will also test a lot of other system include files (it is
# semi-intelligent in determining which ones are required).
AC_CHECK_HEADER(unistd.h)

# You can also check for multiple system headers at the same time,
# but notice the different name of the test macro for this (plural).
AC_CHECK_HEADERS([math.h stdio.h])

# A way to implement conditional logic based on header file presence
# (we don't have a b0rk.h in our system).
AC_CHECK_HEADER(b0rk.h, [echo "b0rk.h present in system"], \
    [echo "b0rk.h not present in system"])

echo "But that doesn't stop us from continuing!"

echo "Directory to install binaries in is '$bindir'"
echo "Directory under which data files go is '$datadir'"
echo "For more variables, check 'config.log' after running configure"
echo "CFLAGS is '$CFLAGS'"
echo "LDFLAGS is '$LDFLAGS'"
echo "LIBS is '$LIBS'"
echo "CC is '$CC'"
echo "AWK is '$AWK'"

```

Listing 1.1: Contents of autoconf-automake/example1/configure.ac

The listing is verbosely commented, so it should be pretty self-evident what the different macros do. The macros that test for a feature or an include file will normally cause the generated configure script to generate small C code test programs that will be run as part of the configuration process. If these programs run successfully, the relevant test will succeed and the configuration process will continue to the next test.

The following convention holds with respect to the names of macros that are commonly used and available:

AC_* A macro that is included in autoconf or is meant for it.

AM_* A macro that is meant for automake.

Others The autoconf system can be expanded by writing your own macros which can be stored in your directory. Also some development packages install new macros for you to use. We'll see one example later on.

Let's now run autoconf. Without any parameters it will read **configure.ac** by default. If **configure.ac** doesn't exist, it will try to read **configure.in** instead. Note that using the name **configure.in** is considered obsolete and reason will become clear later on.

```
[sbox-DIABLO_X86: ~/example1] > autoconf
[sbox-DIABLO_X86: ~/example1] > ls -l
total 112
drwxr-xr-x 2 user user 4096 Sep 16 05:14 autom4te.cache
-rwxrwxr-x 1 user user 98683 Sep 16 05:14 configure
-rw-r--r-- 1 user user 1825 Sep 15 17:23 configure.ac
[sbox-DIABLO_X86: ~/example1] > ./configure

Hello from configure (using echo)!

configure: Hello from configure using msg-notice!
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ANSI C... none needed
checking for gawk... gawk
checking for cos in -lm... yes
checking how to run the C preprocessor... gcc -E
checking for egrep... grep -E
checking for ANSI C header files... yes
checking for sys/types.h... yes
checking for sys/stat.h... yes
checking for stdlib.h... yes
checking for string.h... yes
checking for memory.h... yes
checking for strings.h... yes
checking for inttypes.h... yes
checking for stdint.h... yes
checking for unistd.h... yes
checking for unistd.h... (cached) yes
checking math.h usability... yes
checking math.h presence... yes
checking for math.h... yes
checking stdio.h usability... yes
checking stdio.h presence... yes
checking for stdio.h... yes
checking b0rk.h usability... no
checking b0rk.h presence... no
checking for b0rk.h... no
b0rk.h not present in system
But that doesn't stop us from continuing!
Directory to install binaries in is '${exec_prefix}/bin'
Directory under which data files go is '${prefix}/share'
For more variables, check 'config.log' after running configure
CFLAGS is '-g -O2'
LDFLAGS is ''
LIBS is '-lm'
CC is 'gcc'
AWK is 'gawk'
```

Running autoconf

Autoconf will not output information about its progress. Only errors are output on stdout normally. Instead, it will create a new script in the same

directory called **configure** and will set it as executable for you.

The **configure** script is the result of all of the macro processing. If you look at the generated file with an editor or by using **less**, you'll note that it contains a lot of shell code.

Unless you are experienced in reading convoluted shell code, it is best not to try to understand what is attempted at the various stages. Normally we don't try fix or modify the generated file, since it would be overwritten anyway the next time that someone will run **autoconf**.

When we execute the generated script, you will see the output of various parts in the order that they were used in the configuration file.

Note that **autoconf** (because of **m4**) is an imperative language which means that it will execute the commands one by one when it detects them. This is in contrast to declarative languages, like **Makefiles**.

M4-syntax notes:

- Public **autoconf** M4 macros all start with **A[CST]_***.
- Private macros start with an underscore, but they shouldn't be used since they will change from one **autoconf** version to the other (using undocumented features is bad style anyway).
- **m4** uses **[** and **]** to quote arguments to functions, but in most cases we can leave them out as well. It's best to avoid using braces unless the macro doesn't seem to work properly otherwise. When writing new macros, using braces will become more important, but this material does not cover creating custom macros.
- If you absolutely need to pass braces to the generated script, you have three choices:
 - **@<:@** is same as **[**, and **@>:@** is same as **]**
 - **[[]]** will expand into **[]** (most of time)
 - avoid **[** and **]** (most command line tools and shell don't really require them)
- Since **m4** will use braces for its own needs, we cannot use the **[** command to test things in our scripts, but instead must use **test** (which is more clear anyway). This is why we escape the braces with the rules given above if we really need them to be output into the generated shell script.

If you introduce bad shell syntax into your configuration file, the bad syntax will cause errors only when you will run the generated script file (not when **autoconf** generates the script). In general, **autoconf** will almost always succeed, but you might find that the generated script will not. It's not always simple to know which error in the shell script corresponds to which line in the original **configure.ac** but you will learn with experience.

You might notice a line that reports the result of testing for **unistd.h**. It will appear twice, the first time because it's the default test to run whenever we start testing for headers and a second time because we test for it explicitly. The second test output contains text (cached), which means that the test has been already run and the result has been saved into a cache (the mysterious **autom4te.cache** directory). This means that for large projects which might do

the same tests over and over, the tests are only run once and this will make running the script quite a bit faster.

The last lines output above contain the values of variables. When the configure script runs, it will automatically create shell variables for you that you can use in your shell code fragments. The macros for checking what programs are available for compiling should illustrate that point. Here we used `awk` as an example.

The configure script will take the initial values for variables from your environment but also contains a lot of options that you can give to your script and using those will introduce new variables that you can also use. Anyone compiling modern open source projects will be familiar with options like `--prefix` and other similar ones. Both of these cases are illustrated below:

```
[sbox-DIABLO_X86: ~/example1] > CFLAGS='-O2 -Wall' ./configure --prefix=/usr
...
Directory to install binaries in is '${exec_prefix}/bin'
Directory under which data files go is '${prefix}/share'
For more variables, check 'config.log' after running configure
CFLAGS is '-O2 -Wall'
LDFLAGS is ''
LIBS is '-lm '
CC is 'gcc'
AWK is 'gawk'
```

Modifying configure runs

It might not seem that giving the prefix changes anything, but this is because the shell doesn't expand the value in this particular case. It would expand the value later on if we'd use the variable in our script for doing something. If you want to see some effect, you can try passing the `--datadir` option (because we print that out explicitly).

If you're interested in what other variables are available for configure, you can read the generated **config.log** file, since the variables are listed at the end of that file.

1.4 Substitutions

Besides creating the configure script, autoconf can do other useful things as well. Some people say that autoconf is at least as powerful as emacs, if not more so! Unfortunately with all this power comes awfully lot of complexity when things don't quite work.

Sometimes it is useful to use our variables within text files that are otherwise non-related to our **configure.ac**. These might be configuration files or files that will be used in some part of the building process later on. For this, autoconf provides a mechanism called *substitution*. There is a special macro that will read in an external file, replace all instances of variable names in it, and then store the resulting file into a new file. The convention in naming the input files is to add a suffix `.in` to the names. The name of generated output file will be the same, but without this suffix. Note that the substitution will be done when you run the generated configure script, not when autoconf is run.

The generated configure script will replace all occurrences of the variable name surrounded with '@' characters with the variable value when it reads through each of the input files.

This is best illustrated with a small example. The input file contents are listed after the autoconf configuration file. In this example we'll only do substitution for one file, but it's also possible to process multiple files using the substitution mechanism.

```
# An example showing how to use a variable-based substitution.

AC_INIT(hello, version-0.1)

AC_PROG_CC
AC_PROG_AWK
AC_CHECK_HEADER(unistd.h)
AC_CHECK_HEADERS([math.h stdio.h])
AC_CHECK_HEADER(b0rk.h, [echo "b0rk.h present in system"], \
    [echo "b0rk.h not present in system"])

echo "But that doesn't stop us from continuing!"

# Run the test-output.txt(.in) through autoconf substitution logic.
AC_OUTPUT(test-output.txt)
```

Listing 1.2: Contents of autoconf-automake/example2/configure.ac

```
This file will go through autoconf variable substitution.

The output file will be named as 'test-output.txt' (the '.in'-suffix
is stripped).

All names surrounded by '@'-characters will be replaced by the values
of the variables (if present) by the configure script when it is run
by end user.

Prefix: @prefix@
C compiler: @CC@

This is a name for which there is no variable.
Stuff: @stuff@
```

Listing 1.3: Contents of autoconf-automake/example2/test-output.txt.in

We then run autoconf and configure:

```

[sbox-DIABLO_X86: ~/example2] > autoconf
[sbox-DIABLO_X86: ~/example2] > ./configure
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ANSI C... none needed
checking for gawk... gawk
checking how to run the C preprocessor... gcc -E
checking for egrep... grep -E
checking for ANSI C header files... yes
checking for sys/types.h... yes
checking for sys/stat.h... yes
checking for stdlib.h... yes
checking for string.h... yes
checking for memory.h... yes
checking for strings.h... yes
checking for inttypes.h... yes
checking for stdint.h... yes
checking for unistd.h... yes
checking for unistd.h... (cached) yes
checking math.h usability... yes
checking math.h presence... yes
checking for math.h... yes
checking stdio.h usability... yes
checking stdio.h presence... yes
checking for stdio.h... yes
checking b0rk.h usability... no
checking b0rk.h presence... no
checking for b0rk.h... no
b0rk.h not present in system
But that doesn't stop us from continuing!
configure: creating ./config.status
config.status: creating test-output.txt
[sbox-DIABLO_X86: ~/example2] > cat test-output.txt
This file will go through autoconf variable substitution.

The output file will be named as 'test-output.txt' (the '.in'-suffix
is stripped).

All names surrounded by '@'-characters will be replaced by the values
of the variables (if present) by the configure script when it is run
by end user.

Prefix: /usr/local
C compiler: gcc

This is a name for which there is no variable.
Stuff: @stuff@

```

File contents substitution example

We'll use this feature later on. When you run your own version of the file, you might notice the creation of file called **config.status**. It is the file that actually does the substitution for external files, so if your configuration is otherwise complex and you only want to re-run the substitution of the output files, you can run the **config.status** script.

1.5 Introducing automake

We'll next create a small project that consists of yet another hello world. This time we have a file implementing the application (main), a header file describing the API (printHello()) and the implementation of the function.

As it happens, GNU automake is designed so that it can be easily integrated into autoconf. We'll utilise this in the next example so that we don't need to write our **Makefile** by hand anymore. Instead, the **Makefile** will be generated by the configure script and it will contain the necessary settings for the system on which **configure** is run.

In order for this to work, we'll need two things:

- We need a configuration file for automake (conventionally `Makefile.am`).
- We need to tell our autoconf that it should create `Makefile.in` based on `Makefile.am` by using automake.

We'll start with the autoconf configuration file:

```
# Any source file name related to our project is ok here.
AC_INIT(helloapp, 0.1)

# We're using automake, so we init it next. The name of the macro
# starts with 'AM' which means that it is related to automake ('AC'
# is related to autoconf).
# Initiating automake means more or less generating the .in file from
# the .am file although it can also be generated at other steps.
AM_INIT_AUTOMAKE

# Compiler check.
AC_PROG_CC
# Check for 'install' program.
AC_PROG_INSTALL
# Generate the Makefile from Makefile.in (using substitution logic).
AC_OUTPUT(Makefile)
```

Listing 1.4: Contents of `autoconf-automake/example3/configure.ac`

Then we present the configuration file for automake:

```
# Automake rule primer:
# 1) Left side_ tells what kind of target this will be.
# 2) _right side tells what kind of dependencies are listed.
#
# As an example, below:
# 1) bin = binaries
# 2) PROGRAMS lists the programs to generate Makefile.ins for.
bin_PROGRAMS = helloapp

# Listing source dependencies:
#
# The left side_ gives the name of the application to which the
# dependencies are related to.
# _right side gives again the type of dependencies.
#
# Here we then list the source files that are necessary to build the
# helloapp -binary.
helloapp_SOURCES = helloapp.c hello.c hello.h

# For other files that cannot be automatically deduced by automake,
# you need to use the EXTRA_DIST rule which should list the files
# that should be included. Files can also be in other directories or
# even whole directories can be included this way (not recommended).
#
# EXTRA_DIST = some.service.file.service some.desktop.file.desktop
```

Listing 1.5: Contents of autoconf-automake/example3/Makefile.am

This material tries to introduce just enough of automake for it to be useful for small projects. Because of this, detailed syntax of Makefile.am is not explained. Based on the above description automake will know what kind of Makefile.in to create and autoconf will take it over from there and fill in the missing pieces.

The source files for the project are as simple as possible, but note the implementation of `printHello`. Is there something fishy going on?

```
/**
 * This maemo code example is licensed under a MIT-style license,
 * that can be found in the file called "License" in the same
 * directory as this file.
 * Copyright (c) 2007-2008 Nokia Corporation. All rights reserved.
 */

#include <stdlib.h>
#include "hello.h"

int main(int argc, char** argv) {

    printHello();

    return EXIT_SUCCESS;
}
```

Listing 1.6: The main application in autoconf-automake/example3/helloapp.c

```
/**
 * This maemo code example is licensed under a MIT-style license,
 * that can be found in the file called "License" in the same
 * directory as this file.
 * Copyright (c) 2007-2008 Nokia Corporation. All rights reserved.
 */

#ifndef INCLUDE_HELLO_H
#define INCLUDE_HELLO_H

extern void printHello(void);

#endif
```

Listing 1.7: Prototypes of projects in autoconf-automake/example3/hello.h

```
/**
 * This maemo code example is licensed under a MIT-style license,
 * that can be found in the file called "License" in the same
 * directory as this file.
 * Copyright (c) 2007-2008 Nokia Corporation. All rights reserved.
 */

#include <stdio.h>
#include "hello.h"

void printHello(void) {
    /* Note that these are available as defines now. */
    printf((" PACKAGE " " VERSION ")\n");
    printf("Hello world!\n");
}
```

```
}
```

Listing 1.8: Implementation of printHello in autoconf-automake/example3/hello.c

The `PACKAGE` and `VERSION` defines will be passed to the build process automatically and we'll see their use later. For now, we're armed and dangerous, so let's try our luck:

```
[sbox-DIABLO_X86: ~/example3] > autoconf
configure.ac:10: error: possibly undefined macro: AM_INIT_AUTOMAKE
If this token and others are legitimate, please use m4_pattern_allow.
See the Autoconf documentation.
```

Uh oh

Seems that we were a bit too dangerous. `autoconf` is complaining about a macro for which it cannot find a definition (actually `m4` is the program that does the complaining). What gives? The problem is that by default, `autoconf` only knows about built-in macros. If we want to use a macro for integration or a macro that comes with another package, we need to tell `autoconf` about it. Luckily this process is quite painless.

We'll need to create a local **aclocal.m4** file into the same directory with our **configure.ac**. When it will start, `autoconf` will read this file and it's enough to put the necessary macros there.

We'll use an utility program called **aclocal**, which will scan our **configure.ac** and copy the relevant macros into the local **aclocal.m4**. The directory for all the extension macros is usually **/usr/share/aclocal/** which you might want to check out at some point.

We'll run **aclocal**, and then try to run `autoconf` again. Note that the messages that are introduced into our process from now on are inevitable since some macros use obsolete features or have incomplete syntax and thus trigger warnings. There is no easy solution to this other than to fix the macros themselves.

```
[sbox-DIABLO_X86: ~/example3] > aclocal
/scratchbox/tools/share/aclocal/pkg.m4:5: warning:
underquoted definition of PKG_CHECK_MODULES
run info '(automake)Extending aclocal' or see
http://sources.redhat.com/automake/automake.html#Extending%20aclocal
/usr/share/aclocal/pkg.m4:5: warning:
underquoted definition of PKG_CHECK_MODULES
/usr/share/aclocal/gconf-2.m4:8: warning:
underquoted definition of AM_GCONF_SOURCE_2
/usr/share/aclocal/audiofile.m4:12: warning:
underquoted definition of AM_PATH_AUDIOFILE
[sbox-DIABLO_X86: ~/example3] > autoconf
[sbox-DIABLO_X86: ~/example3] > ./configure
configure: error: cannot find install-sh or install.sh in
~/example3 ~/ ../..
```

Much better, but we're not quite there yet.

If you now list the contents of the directory, you might be wondering where is the **Makefile.in**? We need to run `automake` manually, so that the file will be created. At the same time we also need to introduce the missing files to the directory (like the **install.sh** that **configure** seems to complain about).

This is done by executing `automake -ac`, which will create the **Makefile.in** and also copy the missing files into their proper places. **This step will also copy**

a file called **COPYING** into your directory which by default will contain the **GPL**. So, if you're going to distribute your software under some other license, this might be the correct moment to replace the license file with the one that you really want to use.

```
[sbox-DIABLO_X86: ~/example3] > automake -ac
configure.ac: installing './install-sh'
configure.ac: installing './missing'
Makefile.am: installing './depcomp'
[sbox-DIABLO_X86: ~/example3] > ./configure
checking for a BSD-compatible install... /scratchbox/tools/bin/install -c
checking whether build environment is sane... yes
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ANSI C... none needed
checking for style of include used by make... GNU
checking dependency style of gcc... gcc3
checking for a BSD-compatible install... /scratchbox/tools/bin/install -c
configure: creating ./config.status
config.status: creating Makefile
config.status: executing depfiles commands
```

Configure completes

Notice the second to last line of the output which tells us that **autoconf** just created **Makefile** for us (based on the **Makefile.in** that **automake** created).

You might grow tired of doing all these steps manually each time when you want to make sure that all generated files are really generated. You are not alone. Most developers will create a script called **autogen.sh** which will implement the necessary bootstrap-procedures for them. Below is a file that is suitable for our use. Real projects might have more steps because of localisation and other requirements.

```
#!/bin/sh
#
# An utility script to setup the autoconf environment for the first
# time. Normally this script would be run when checking out a
# development version of the software from SVN/version control.
# Regular users expect to download .tar.gz/tar.bz2 source code
# instead, and those should come with with 'configure' script so that
# users do not require the autoconf/automake tools.
#
# Scan configure.ac and copy the necessary macros into aclocal.m4.
aclocal

# Generate Makefile.in from Makefile.am (and copy necessary support
# files, because of -ac).
automake -ac

# This step is not normally necessary, but documented here for your
# convenience. The files listed below need to be present to stop
# automake from complaining during various phases of operation.
#
# You also should consider maintaining these files separately once
```

```
# you release your project into the wild.
#
# touch NEWS README AUTHORS ChangeLog

# Run autoconf (will create the 'configure'-script).
autoconf

echo 'Ready to go (run configure)'
```

Listing 1.9: A typical simple autogen.sh (autoconf-automake/example3/autogen.sh)

You might notice the commented line with touch and wonder why it is commented. There is a target called `distcheck` that automake will create in our **Makefile** and this target checks whether the distribution tarball contains all the necessary files. The files listed on the touch-line are necessary (even if empty), so you'll need to create those at some point. Without these files, the penultimate **Makefile** will complain when you run the `distcheck`-target.

We'll now build our project and test it out:

```
[sbox-DIABLO_X86: ~/example3] > make
if gcc -DPACKAGE_NAME=\"helloapp\" -DPACKAGE_TARNAME=\"helloapp\"
-DPACKAGE_VERSION=\"0.1\" -DPACKAGE_STRING=\"helloapp 0.1\"
-DPACKAGE_BUGREPORT=\"\" -DPACKAGE=\"helloapp\" -DVERSION=\"0.1\"
-I. -Iexample3 -g -O2 -MT helloapp.o -MD -MP -MF ".deps/helloapp.Tpo"
-c -o helloapp.o helloapp.c; \ then \
mv -f ".deps/helloapp.Tpo" ".deps/helloapp.Po";
else rm -f ".deps/helloapp.Tpo"; exit 1;
fi
if gcc -DPACKAGE_NAME=\"helloapp\" -DPACKAGE_TARNAME=\"helloapp\"
-DPACKAGE_VERSION=\"0.1\" -DPACKAGE_STRING=\"helloapp 0.1\"
-DPACKAGE_BUGREPORT=\"\" -DPACKAGE=\"helloapp\" -DVERSION=\"0.1\"
-I. -Iexample3 -g -O2 -MT hello.o -MD -MP -MF ".deps/hello.Tpo"
-c -o hello.o hello.c; \ then \
mv -f ".deps/hello.Tpo" ".deps/hello.Po";
else rm -f ".deps/hello.Tpo"; exit 1;
fi
gcc -g -O2 -o helloapp helloapp.o hello.o
[sbox-DIABLO_X86: ~/example3] > ./helloapp
(helloapp 0.1)
Hello world!
[sbox-DIABLO_X86: ~/example3] > make clean
test -z "helloapp" || rm -f helloapp
rm -f *.o
```

Building, PACKAGE and VERSION defines and cleaning

1.6 Checking for distribution sanity

The generated **Makefile** contains various targets that can be used when creating distribution tarballs (tar-files containing the source code and necessary files to build the software). The most important of these is the `dist`-target, which will by default create a **.tar.gz**-file out of your source including the **configure**-script and other necessary files (which are specified in **Makefile.am**).

To test whether it is possible to build your software from such distribution tarball, execute the `distcheck`-target. It will first create a distribution tarball, then extract it in a new subdirectory, run **configure** there and try to build the software with default make target. If it fails, you'll get the relevant error.

It's recommended to do the `distcheck` target each time before doing a `dist` target so that you can be sure that the distribution tarball can be used outside

your source tree. This step is especially critical when we'll start making Debian packages later on.

1.7 Cleaning up

For your convenience, the **example3** directory also includes a script called **antigen.sh**, which will try its best to get rid of all generated files (you'll need to **autogen.sh** the project afterwards).

Having a cleanup script is not very common in open source projects, but it's especially useful when starting autotools development, as it allows you test your toolchain easily from scratch.

The contents of **antigen.sh** which is suitable for simple projects:

```
#!/bin/sh
#
# An utility script to remove all generated files.
#
# Running autogen.sh will be required after running this script since
# the 'configure' script will also be removed.
#
# This script is mainly useful when testing autoconf/automake changes
# and as a part of their development process.
#
# If there's a Makefile, then run the 'distclean' target first (which
# will also remove the Makefile).
if test -f Makefile; then
    make distclean
fi

# Remove all tar-files (assuming there are some packages).
rm -f *.tar.* *.tgz

# Also remove the autotools cache directory.
rm -Rf autom4te.cache

# Remove rest of the generated files.
rm -f Makefile.in aclocal.m4 configure depcomp install-sh missing
```

Listing 1.10: An utility script to clean up all generated files (autoconf-automake/example3/antigen.sh)

1.8 Integration with pkg-config

The last part that we cover for autoconf is how to integrate **pkg-config** support into your projects when using **configure.ac**.

pkg-config comes with a macro package (**pkg.m4**) which contains some useful macros for integration. The best documentation for these can be found in the **pkg-config** manual pages.

We'll use only one, **PKG_CHECK_MODULES** which should be used like this:

```
# Check whether the necessary pkg-config packages are present. The
# PKG_CHECK_MODULES macro is supplied by pkg-config
# (/usr/share/aclocal/).
#
```



```

# The first parameter will be the variable name prefix that will be
# used to create two variables: one to hold the CFLAGS required by
# the packages, and one to hold the LDFLAGS (LIBS) required by the
# packages. The variable name prefix (HHW) can be chosen freely.
PKG_CHECK_MODULES(HHW, gtk+-2.0 hildon-1 hildon-fm-2 gnome-vfs-2.0 \
gconf-2.0 libosso)
# At this point HHW_CFLAGS will contain the necessary compiler flags
# and HHW_LIBS will contain the linker options necessary for all the
# packages listed above.
#
# Add the pkg-config supplied values to the ones that are used by
# Makefile or supplied by the user running ./configure.
CFLAGS="$HHW_CFLAGS $CFLAGS"
LIBS="$HHW_LIBS $LIBS"

```

Listing 1.11: Part of configure.ac for pkg-config integration

The proper placing for this code is after all the AC_PROG_*-checks and before the AC_OUTPUT-macros (so that the build flags may affect the substituted files).

You should also check the validity of the package names by using `pkg-config --list-all` to make sure you don't try to get the wrong library information.