

Maemo Diablo Reference Manual for maemo 4.1

# **GNU Build System**

December 22, 2008

# Contents

<b>1</b>	<b>GNU Build System</b>	<b>2</b>
1.1	Introduction	2
1.2	GNU Make and Makefiles	2
1.2.1	Simplest Real Example	3
1.2.2	Anatomy of Makefile	6
1.2.3	Default Goal	7
1.2.4	On Names of Makefiles	7
1.2.5	Questions	8
1.2.6	Adding Make Goals	8
1.2.7	Making One Target at a Time	9
1.2.8	PHONY Keyword	9
1.2.9	Specifying Default Goal	10
1.2.10	Other Common Phony Goals	11
1.2.11	Variables in Makefiles	11
1.2.12	Variable Flavors	11
1.2.13	Recursive Variables	12
1.2.14	Simple Variables	13
1.2.15	Automatic Variables	14
1.2.16	Integrating with Pkg-Config	15
1.3	GNU Autotools	16
1.3.1	Brief History of Managing Portability	17
1.3.2	GNU Autoconf	18
1.3.3	Substitutions	22
1.3.4	Introducing Automake	24
1.3.5	Checking for Distribution Sanity	29
1.3.6	Cleaning up	29
1.3.7	Integration with Pkg-Config	30

# Chapter 1

## GNU Build System

### 1.1 Introduction

The following code examples are used in this chapter:

- [simple-make-files](#)
- [autoconf-automake](#)

### 1.2 GNU Make and Makefiles

The *make* program from the GNU project is a powerful tool to aid implementing automation in the software building process. Beside this, it can be used to automate any task that uses files and in which these files are transformed into some other form. Make by itself does not know what the files contain or what they represent, but using a simple syntax it can be taught how to handle them.

When developing software with gcc (and other tools), gcc will often be invoked repeatedly with the same parameters and flags. After changing one source file, it will be noticed that other output files need to be rebuilt, and even the whole application if some interface has changed between the functions. This might happen whenever declarations change, new parameters are added to function prototypes etc.

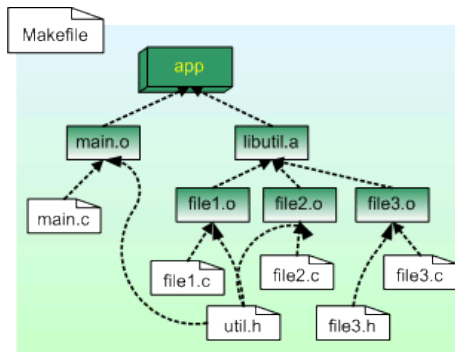
These tasks could, of course, be always performed manually, but after a while a nicer way is going to be more desirable.

GNU make is a software building automation tool that will execute repetitive tasks. It is controlled via a **Makefile** that contains lists of dependencies between different source files and output files. It also contains lists of commands that should be executed to satisfy these dependencies. Make uses the **timestamps** of files and the information of the files' existence to automate the rebuilding of applications (targets in make), as well as the rules that are specified in the Makefile.

Make can be used for other purposes as well. A target can easily be created for installing the built software on a destination computer, a target for generating documentation by using some automatic documentation generation tool, etc. Some people use make to keep multiple Linux systems up to date with the

newest software and various system configuration changes. In short, make is flexible enough to be generally useful.

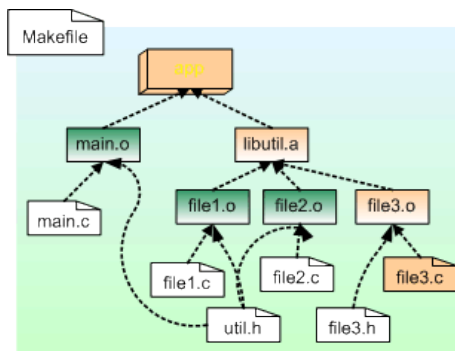
The dependencies between different files making up a software project:



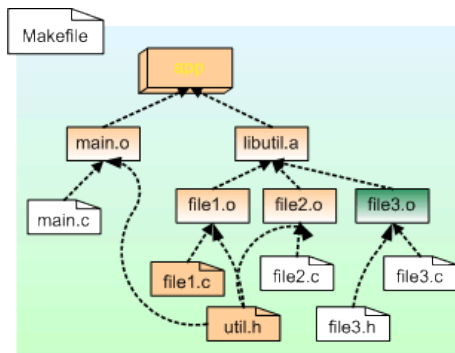
The aim of make is to satisfy the target. Each target has its own dependencies. A user will generally select a target for make to satisfy by supplying the target name on the command line. Make will start by checking, whether all of the dependencies exist and have an older timestamp than the target. If so, make will do nothing, as nothing has changed. However, since a header file (that an application is not dependent on directly) might change, make will propagate the changes to the 'root' of the target as shown in the picture above.

Make will rebuild all of the necessary dependencies and targets on the way towards the target. This way, make will only rebuild those dependencies that actually affect something, and thus, will save time and effort. In big projects, the amount of time saved is significant.

To illustrate this, suppose that **file3.c** in the above picture is modified. After that, make is run, and it will automatically rebuild the necessary targets (**file3.o**, **libutil.a** and **app**):



Now suppose that another function is added to **file1.c**. Then also **util.h** would need to be modified accordingly. The picture shows that quite many objects and targets depend on this header file, so a sizable number of objects need to be rebuilt (but not all):



N.B. In the example pictures, there is a project with a custom static library, which will be linked against the test application.

## 1.2.1 Simplest Real Example

Before delving too deeply into the syntax of makefiles, it is instructive to first see make in action. For this, a simple project will be used, written in the C language.

In C, it is customary to write "header" files (conventional suffix for them is `.h`) and regular source files (`.c`). The header files describe calling conventions, APIs and structures that are to be made usable for the outside world. The `.c` files contain the implementation of these interfaces.

The first rule in this is: if something is changed in the interface file, then the binary file containing the code implementation (and other files that use the same interface) should be regenerated. Regeneration in this case means invoking `gcc` to create the binary file out of the C-language source file.

Make needs to be told two things at this point:

- If a file's content changes, which other files will be affected? Since a single interface will likely affect multiple other files, make uses a reversed ordering here. For each resulting file, it is necessary to list the files on which this one file depends on.
- What are the commands to regenerate the resulting files when the need arises?

This example deals with that as simply as possible. There is a project that consists of two source files and one header file. The contents of these files are listed below:

```
/**
 * The old faithful hello world.
 *
 * 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> /* EXIT_* */
#include "hello_api.h" /* sayhello */
```

```
int main(int argc, char **argv) {
    sayhello();

    return EXIT_SUCCESS;
}
```

Listing 1.1: simple-make-files/hello.c

```
/**
 * Implementation of sayhello.
 *
 * 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> /* printf */
#include "hello_api.h" /* sayhello declaration */

void sayhello(void) {
    printf("Hello world!\n");
}
```

Listing 1.2: simple-make-files/hello\_func.c

```
/**
 * Interface description for the hello_func module.
 *
 * 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_API_H
#define INCLUDE_HELLO_API_H
/* The above is protection against circular header inclusion. */

/* Function to print out "Hello world!\n". */
extern void sayhello(void);

#endif
/* ifndef INCLUDE_HELLO_API_H */
```

Listing 1.3: simple-make-files/hello\_api.h

So, in effect, there is the main application in **hello.c**, which uses a function that is implemented in **hello\_func.c** and declared in **hello\_api.h**. Building an application out of these files could be performed manually like this:

```
gcc -Wall hello.c hello_func.c -o hello
```

Or, it could be done in three stages:

```
gcc -Wall -c hello.c -o hello.o
gcc -Wall -c hello_func.c -o hello_func.o
gcc -Wall hello.o hello_func.o -o hello
```

In both cases, the resulting application (**hello**) will be the same.

In the first case, gcc is instructed to process all source files in one go, link the resulting object code and store that code (program in this case) into **hello**.

In the second case, gcc is instructed to create a binary object file for each of the source files. After that, gcc is instructed to link these output files (**hello.o** and **hello\_func.o**) together, and store the linked code into **hello**.

N.B. When gcc reads through the C source files, it will also read in the header files, since the C code uses the #include -preprocessor directive. This is because gcc will internally run all files ending with .c through cpp (the preprocessor) first.

The file describing this situation to make is as follows:

```
# define default target (first target = default)
# it depends on 'hello.o' (which will be created if necessary)
# and hello_func.o (same as hello.o)
hello: hello.o hello_func.o
    gcc -Wall hello.o hello_func.o -o hello

# define hello.o target
# it depends on hello.c (and is created from it)
# also depends on hello_api.h which would mean that
# changing the hello.h api would force make to rebuild
# this target (hello.o).
# gcc -c: compile only, do not link
hello.o: hello.c hello_api.h
    gcc -Wall -c hello.c -o hello.o

# define hello_func.o target
# it depends on hello_func.c (and is created from)
# and hello_api.h (since that's its declaration)
hello_func.o: hello_func.c hello_api.h
    gcc -Wall -c hello_func.c -o hello_func.o
#
```

Listing 1.4: simple-make-files/Makefile

This file is in the simplest form without using any variables or relying on built-in magic in GNU make. Later it will be shown that the same rules could be written in a much shorter way.

This makefile can be tested by running make in the directory that contains the makefile and the source files:

```
user@system:~$ make
gcc -Wall -c hello.c -o hello.o
gcc -Wall -c hello_func.c -o hello_func.o
gcc -Wall hello.o hello_func.o -o hello
```

The resulting files after running make:

```
user@system:~$ ls -la
total 58
drwxr-xr-x 3 user user 456 Aug 11 15:04 .
drwxr-xr-x 13 user user 576 Aug 11 14:56 ..
-rw-r--r-- 1 user user 699 Jun 1 14:48 Makefile
-rwxr-xr-x 1 user user 4822 Aug 11 15:04 hello
-rw-r--r-- 1 user user 150 Jun 1 14:48 hello.c
-rw-r--r-- 1 user user 824 Aug 11 15:04 hello.o
-rw-r--r-- 1 user user 220 Jun 1 14:48 hello_api.h
-rw-r--r-- 1 user user 130 Jun 1 14:48 hello_func.c
-rw-r--r-- 1 user user 912 Aug 11 15:04 hello_func.o
```

Executing the binary:

```
user@system:~$ ./hello
Hello world!
```

## 1.2.2 Anatomy of Makefile

From the simple example above, some of make's syntax can be deduced.

Here are the rules that can be learned:

- Comments are lines that start with the #-character. To be more precise, when make reads the makefile, it will ignore the #-character and any characters after it up to the end of the line. This means that comments can also be put at the end of lines, and make will ignore them, but this is considered bad practice as it would lead to subtle problems later on.
- The backslash character (\) can be used to escape the special meaning of the next character. The most important special character in makefiles is the dollar character (\$), which is used to access the contents of variables. There are also other special characters. To continue a line that is too long, the newline character can be escaped on that line. When the backslash is put at the end of the line, make will ignore the newline when reading input.
- Empty lines by themselves are ignored.
- A line that starts at column 0 (start of the line) and contains a colon character (:) is considered a rule. The name on the left side of the colon is created by the commands. This name is called a target. Any filenames specified after the colon are the files that the target depends on. They are called prerequisites (i.e. they are required to exist, before make decides to create the target).
- Lines starting with the tabulation character (tab) are commands that make will run to achieve the target.

In the printed material, the tabs are represented with whitespace, so be careful when reading the example makefiles. Note also that in reality, the command lines are considered as part of the rule.

Using these rules, it can now be deduced that:

- make will regenerate **hello** when either or both of its prerequisites change. So, if either **hello.o** or **hello\_func.o** change, make will regenerate **hello** by using the command: `gcc -Wall hello.o hello_func.o -o hello`
- If either **hello.c** or **hello\_api.h** change, make will regenerate **hello.o**.
- If either **hello\_func.c** or **hello\_api.h** change, make will regenerate **hello\_func.o**.

## 1.2.3 Default Goal

It should be noted that make was not explicitly told what it should do by default when run without any command line parameters (as was done above). How does it know that creating **hello** is the target for the run?

The first target given in a makefile is the default target. A target that achieves some higher-level goal (like linking all the components into the final application) is sometimes called a goal in GNU make documentation.

So, the first target in the makefile is the default goal when make is run without command line parameters.

N.B. In a magic-like way, make will automatically learn the dependencies between the various components and deduce that in order to create **hello**, it will also need to create **hello.o** and **hello\_func.o**. Make will regenerate the missing prerequisites when necessary. In fact, this is a quality that causes make do its magic.

Since the prerequisites for hello (hello.o and hello\_func.o) do not exist, and hello is the default goal, make will first create the files that the hello target needs. This can be evidenced from the screen capture of make running without command line parameters. It shows the execution of each command in the order that make decides is necessary to satisfy the default goal's prerequisites, and finally create the hello.

```
user@system:~$ make
gcc -Wall -c hello.c -o hello.o
gcc -Wall -c hello_func.c -o hello_func.o
gcc -Wall hello.o hello_func.o -o hello
```

## 1.2.4 On Names of Makefiles

The recommended name for the makefile is **Makefile**. This is not the first filename that GNU make will try to open, but it is the most portable one. In fact, the order of filenames that make attempts to open is: **GNUMakefile**, **Makefile** and finally **makefile**.

Unless one is sure that one's makefile will not work on other make systems (not GNU), one should refrain from using GNUMakefile. Makefile will be used here for the most of this material. The idea behind using Makefile instead of makefile is related to how shells and commands sort filenames, when contents of directories are requested. Since in ASCII the uppercase letters come before the lowercase letters, the important files are listed first. Makefiles are considered important, since they are the basis for building the project. (The collation order might be different in your locale and your environment.)

Make can explicitly be told which file to read by using the `-f` command line option. This option will be used in the next example.

## 1.2.5 Questions

Based on common sense, what should make do when it is run after:

- Deleting the hello file?
- Deleting the hello.o file?
- Modifying hello\_func.c?
- Modifying hello.c?
- Modifying hello\_api.h?
- Deleting the Makefile?

What would be done when writing the commands manually?

## 1.2.6 Adding Make Goals

Sometimes it is useful to add targets, whose execution does not result in a file, but instead causes some commands to be run. This is commonly used in makefiles of software projects to get rid of the binary files, so that building can be attempted from a clean state. These kinds of targets can also be justifiably called goals. GNU documentation uses the name "phony target" for these kinds of targets, since they do not result in creating a file, like normal targets do.

The next step is to create a copy of the original makefile, and add a goal that will remove the binary object files and the application. N.B. Other parts of the makefile do not change.

```
# add a cleaning target (to get rid of the binaries)

# define default target (first target = default)
# it depends on 'hello.o' (which must exist)
# and hello_func.o
hello: hello.o hello_func.o
    gcc -Wall hello.o hello_func.o -o hello

# define hello.o target
# it depends on hello.c (and is created from)
# also depends on hello_api.h which would mean that
# changing the hello.h api would force make to rebuild
# this target (hello.o).
# gcc -c: compile only, do not link
hello.o: hello.c hello_api.h
    gcc -Wall -c hello.c -o hello.o

# define hello_func.o target
# it depends on hello_func.c (and is created from)
# and hello_api.h (since that's its declaration)
hello_func.o: hello_func.c hello_api.h
    gcc -Wall -c hello_func.c -o hello_func.o

# This is the definition of the target 'clean'
# Here we'll remove all the built binaries and
# all the object files that we might have generated
# Notice the -f flag for rm, it means "force" which
# in turn means that rm will try to remove the given
# files, and if there are none, then that's ok. Also
# it means that we have no writing permission to that
# file and have writing permission to the directory
# holding the file, rm will not then ask for permission
# interactively.
clean:
    rm -f hello hello.o hello_func.o
```

Listing 1.5: simple-make-files/Makefile.2

In order for make to use this file instead of the default Makefile, the `-f` command line parameter needs to be used as follows:

```
user@system:~$ make -f Makefile.2
gcc -Wall -c hello.c -o hello.o
gcc -Wall -c hello_func.c -o hello_func.o
gcc -Wall hello.o hello_func.o -o hello
```

To control which target make will process (instead of the default goal), target name needs to be given as command line parameter like this:

```
user@system:~$ make -f Makefile.2 clean
rm -f hello hello.o hello_func.o
```

No binary files remain in the directory:

```
user@system:~$ ls -la
total 42
drwxr-xr-x 3 user user 376 Aug 11 15:08 .
drwxr-xr-x 13 user user 576 Aug 11 14:56 ..
-rw-r--r-- 1 user user 699 Jun 1 14:48 Makefile
-rw-r--r-- 1 user user 1279 Jun 1 14:48 Makefile.2
-rw-r--r-- 1 user user 150 Jun 1 14:48 hello.c
-rw-r--r-- 1 user user 220 Jun 1 14:48 hello_api.h
-rw-r--r-- 1 user user 130 Jun 1 14:48 hello_func.c
```

## 1.2.7 Making One Target at a Time

Sometimes it is useful to ask make to process only one target. Similar to the clean case, it just needs the target name to be given on the command line:

```
user@system:~$ make -f Makefile.2 hello.o
gcc -Wall -c hello.c -o hello.o
```

Multiple target names can also be supplied as individual command line parameters:

```
user@system:~$ make -f Makefile.2 hello_func.o hello
gcc -Wall -c hello_func.c -o hello_func.o
gcc -Wall hello.o hello_func.o -o hello
```

This can be done with any number of targets, even phony ones. Make will try and complete all of them in the order that they are on the command line. Should any of the commands within the targets fail, make will abort at that point, and will not pursue the rest of the targets.

## 1.2.8 PHONY Keyword

Suppose that for some reason or another, a file called **clean** appears in the directory in which make is run with clean as the target. In this case, make would probably decide that since clean already exists, there is no need to run the commands leading to the target, and in this case, make would not run the rm command at all. Clearly this is something that needs to be avoided.

For these cases, GNU make provides a special target called **.PHONY** (note the leading dot). The real phony targets (clean) will be listed as a dependency to .PHONY. This will signal to make that it should never consider a file called clean to be the result of the phony target.

In fact, this is what most open source projects that use make do.

This leads in the following makefile (comments omitted for brevity):

```
hello: hello.o hello_func.o
    gcc -Wall hello.o hello_func.o -o hello

hello.o: hello.c hello_api.h
    gcc -Wall -c hello.c -o hello.o

hello_func.o: hello_func.c hello_api.h
    gcc -Wall -c hello_func.c -o hello_func.o

.PHONY: clean
clean:
```

```
rm -f hello hello.o hello_func.o
```

Listing 1.6: simple-make-files/Makefile.3

## 1.2.9 Specifying Default Goal

Make will use the first target in a makefile as its default goal. What if there is a need to explicitly set the default goal instead? Since it is not possible to actually change the "first target is the default goal" setting in make, this needs to be taken into account. So, a new target has to be added, and made sure that it will be processed as the first target in a makefile.

In order to achieve this, a new phony target should be created, and the wanted targets should be listed as the phony target's prerequisites. Any name can be used for the target, but all is a very common name for this use. The only thing that needs to be remembered is that this target needs to be the first one in the makefile.

```
.PHONY: all
all: hello

hello: hello.o hello_func.o
    gcc -Wall hello.o hello_func.o -o hello

hello.o: hello.c hello_api.h
    gcc -Wall -c hello.c -o hello.o

hello_func.o: hello_func.c hello_api.h
    gcc -Wall -c hello_func.c -o hello_func.o

.PHONY: clean
clean:
    rm -f hello hello.o hello_func.o
```

Listing 1.7: simple-make-files/Makefile.4

Something peculiar can be seen in the listing above. Since the first target is the default goal for make, would that not make .PHONY now to be the default target? Since .PHONY is a special target in GNU make, this is safe. Also, because of compatibility reasons, GNU make will ignore any targets that start with a leading dot (.) when looking for the default goal.

## 1.2.10 Other Common Phony Goals

Many other phony goals will be encountered in makefiles that projects use.

Some of the more common ones include:

- **install:** Prerequisites for install is the software application binary (or binaries). The commands (normally *install* is used on Linux) will specify which binary file to place under which directory on the system (/usr/bin, /usr/local/bin, etc).
- **distclean:** Similar to clean; removes object files and such, but removes other files as well (sounds scary). Normally this is used to implement the removal of Makefile in the case that it was created by some automatic tool (*configure* for example).

- package: Prerequisites are as in install, but instead of installing, creates an archive file (*tar*) with the files in question. This archive can then be used to distribute the software.

Other common phony targets can be found in the [GNU make manual \[1\]](#).

### 1.2.11 Variables in Makefiles

So far these files have been listing filenames explicitly. Writing makefiles this way can get rather tedious, if not error prone.

This is why all make programs (not just the GNU variant) provide variables. Some other make programs call them macros.

Variables work almost as they do inside regular UNIX command shells. They are a simple mechanism, by which a piece of text can be associated with a name that can be referenced later on in multiple places in the makefiles. Make variables are based on text replacement, just like shell variables are.

### 1.2.12 Variable Flavors

The variables that can be used in makefiles come in two basic styles or flavors.

The default flavor is where referencing a variable will cause make to expand the variable contents at the point of reference. This means that if the value of the variable is some text in which other variables are referenced, their contents will also be replaced automatically. This flavor is called recursive.

The other flavor is simple, meaning that the content of a variable is evaluated only once, when the variable is defined.

Deciding on which flavor to use might be important, when the evaluation of variable contents needs to be repeatable and fast. In these cases, simple variables are often the correct solution.

### 1.2.13 Recursive Variables

Here are the rules for creating recursive variables:

- Names must contain only ASCII alphanumeric characters and underscores (to preserve portability).
- Only lowercase letters should be used in the names of the variables. Make is case sensitive, and variable names written in capital letters are used when it is necessary to communicate the variables outside make, or their origin is from outside (environment variables). This is, however, only a convention, and there will also be variables that are local to the makefile, but still written in uppercase.
- Values may contain any text. The text will be evaluated by make when the variable is used, not when it is defined.
- Long lines may be broken up with a backslash character (\) and newline combination. This same mechanism can be used to continue other long lines as well (not just variables). Do not put any whitespace after the backslash.

- A variable that is being defined should not be referenced in its text portion. This would result in an endless loop whenever this variable is used. If this happens GNU make will stop to an error.

Now the previously used makefile will be reused, but some variables will be introduced. This also shows the syntax of defining the variables, and how to use them (reference them):

```
# define the command that we use for compilation
CC = gcc -Wall

# which targets do we want to build by default?
# note that 'targets' is just a variable, its name
# does not have any special meaning to make
targets = hello

# first target defined will be the default target
# we use the variable to hold the dependencies
.PHONY: all
all: $(targets)

hello: hello.o hello_func.o
    $(CC) hello.o hello_func.o -o hello

hello.o: hello.c hello_api.h
    $(CC) -c hello.c -o hello.o

hello_func.o: hello_func.c hello_api.h
    $(CC) -c hello_func.c -o hello_func.o

# we'll make our cleaning target more powerful
# we remove the targets that we build by default and also
# remove all object files
.PHONY: clean
clean:
    rm -f $(targets) *.o
```

Listing 1.8: simple-make-files/Makefile.5

The CC variable is the standard variable to hold the name of the C compiler executable. GNU make comes preloaded with a list of tools that can be accessed in similar way (as will be shown with \$(RM) shortly, but there are others). Most UNIX tools related to building software already have similar pre-defined variables in GNU make. Here one of them will be overridden for no other reason than to demonstrate how it is done. Overriding variables like this is not recommended, since the user running make later might want to use some other compiler, and would have to edit the makefile to do so.

### 1.2.14 Simple Variables

Suppose you have a makefile like this:

```
CC = gcc -Wall

# we want to add something to the end of the variable
CC = $(CC) -g

hello.o: hello.c hello_api.h
    $(CC) -c hello.c -o hello.o
```

---

Listing 1.9: simple-make-files/Makefile.6

What might seem quite logical to a human reader, will not seem very logical to make.

Since the contents of recursive variables are evaluated when they are referenced, it can be seen that the above fragment will lead to an infinite loop.

How can this be corrected? Make actually provides two mechanisms for this. This is the solution with simple variables:

```
CC := gcc -Wall
# we want to add something to the end of the variable
CC := $(CC) -g
hello.o: hello.c hello_api.h
        $(CC) -c hello.c -o hello.o
```

Listing 1.10: simple-make-files/Makefile.7

Notice that the equals character (=) has been changed into := .

This is how simple variables are created. Whenever a simple variable is referenced, make will just replace the text that is contained in the variable without evaluating it. It will do the evaluation only when the variable is defined. In the above example, CC is created with the content of gcc -Wall (which is evaluated, but is just plain text), and when the CC variable is defined next time, make will evaluate \$(CC) -g which will be replaced with gcc -Wall -g, as one might expect.

So, the only two differences between the variable flavors are:

- When defining simple variables, := is used.
- make evaluates the contents, when the simple variable is defined, not when it is referenced later.

Most of the time it is advisable to use the recursive variable flavor, since it does what is wanted.

There was a mention about two ways of appending text to an existing variable. The other mechanism is the += operation as follows:

```
CC = gcc -Wall
# we want to add something to the end of the variable
CC += -g
hello.o: hello.c hello_api.h
        $(CC) -c hello.c -o hello.o
```

Listing 1.11: simple-make-files/Makefile.8

The prepending operation can also be used with simple variables. Make will not change the type of variable on the left side of the += operator.

### 1.2.15 Automatic Variables

There is a pre-defined set of variables inside make that can be used to avoid repetitive typing, when writing out the commands in a rule.

This is a list of the most useful ones:

- `<`: replaced by the first prerequisite of the rule.
- `^`: replaced by the list of all prerequisites of the rule.
- `@`: replaced by the target name.
- `?`: list of prerequisites that are newer than the target is (if target does not exist, they are all considered newer).

By rewriting the makefile using automatic variables, the result is:

```
# add the warning flag to CFLAGS-variable
CFLAGS += -Wall

targets = hello

.PHONY: all
all: $(targets)

hello: hello.o hello_func.o
      $(CC) $^ -o $@

hello.o: hello.c hello_api.h
      $(CC) $(CFLAGS) -c $< -o $@

hello_func.o: hello_func.c hello_api.h
      $(CC) $(CFLAGS) -c $< -o $@

.PHONY: clean
clean:
      $(RM) $(targets) *.o
```

Listing 1.12: simple-make-files/Makefile.9

Notice the cases when `^` is used instead of `<` in the above snippet. It is not desired to pass the header files for compilation to the compiler, since the source file already includes the header files. For this reason, `<` is used. On the other hand, when linking programs and there are multiple files to put into the executable, `^` would normally be used.

This relies on a couple of things:

- `$(RM)` and `$(CC)` will be replaced with paths to the system compiler and removal commands. `$(CFLAGS)` is a variable that contains a list of options to pass whenever make will invoke the C compiler.

It can also be noticed that all these variable names are in uppercase. This signifies that they have been communicated from the system environment to make. In fact, if creating an environment variable called `CFLAGS`, make will create it internally for any makefile that it will process. This is the mechanism to communicate variables into makefiles from outside.

Writing variables in uppercase is a convention signaling external variables, or environmental variables, so this is the reason why lowercase should be used in own private variables within a Makefile.

### 1.2.16 Integrating with Pkg-Config

The above has provided the knowledge to write a simple Makefile for the Hello World example. In order to get the the result of `pkg-config`, the GNU make

\$(shell command params) function will be used here. Its function is similar to the backtick operation of the shell.

```
# define a list of pkg-config packages we want to use
pkg_packages := gtk+-2.0 hildon-1

# get the necessary flags for compiling
PKG_CFLAGS := $(shell pkg-config --cflags $(pkg_packages))
# get the necessary flags for linking
PKG_LDFLAGS := $(shell pkg-config --libs $(pkg_packages))

# additional flags
# -Wall: warnings
# -g: debugging
ADD_CFLAGS := -Wall -g

# combine the flags (so that CFLAGS/LDFLAGS from the command line
# still work).
CFLAGS := $(PKG_CFLAGS) $(ADD_CFLAGS) $(CFLAGS)
LDFLAGS := $(PKG_LDFLAGS) $(LDFLAGS)

targets = hildon_helloworld-1

.PHONY: all
all: $(targets)

hildon_helloworld-1: hildon_helloworld-1.o
$(CC) $^ -o $@ $(LDFLAGS)

hildon_helloworld-1.o: hildon_helloworld-1.c
$(CC) $(CFLAGS) -c $< -o $@

.PHONY: clean
clean:
$(RM) $(targets) *.o
```

Listing 1.13: simple-make-files/Makefile.10

One might wonder, where the CC and RM variables come from. They certainly were not declared anywhere in the Makefile. GNU make comes with a list of pre-defined variables for many programs and their arguments. GNU make also contains a preset list of pattern rules (which will not be dealt here), but basically these are pre-defined rules that are used when (for example) one needs an .o file from an .c file. The rules that were manually specified above are actually the same rules that GNU make already contains, so the Makefile can be made even more compact by only specifying the dependencies between files.

This leads to the following, slightly simpler version:

```
# define a list of pkg-config packages we want to use
pkg_packages := gtk+-2.0 hildon-1

PKG_CFLAGS := $(shell pkg-config --cflags $(pkg_packages))
PKG_LDFLAGS := $(shell pkg-config --libs $(pkg_packages))

ADD_CFLAGS := -Wall -g

# combine the flags
CFLAGS := $(PKG_CFLAGS) $(ADD_CFLAGS) $(CFLAGS)
LDFLAGS := $(PKG_LDFLAGS) $(LDFLAGS)
```

```

targets = hildon_helloworld-1

.PHONY: all
all: $(targets)

# we can omit the rules and just specify the dependencies
# for our simple project this doesn't seem like a big win
# but for larger projects where you have multiple object
# files, this will save considerable time.
hildon_helloworld-1: hildon_helloworld-1.o
hildon_helloworld-1.o: hildon_helloworld-1.c

.PHONY: clean
clean:
    $(RM) $(targets) *.o

```

Listing 1.14: simple-make-files/Makefile.11

## 1.3 GNU Autotools

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 section introduces GNU autotools as one solution for managing this complexity. There are also other tools available, but the most likely to be encountered are autotoolized projects and also Debian packaging supports autotools.

### 1.3.1 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, the linker parameters need to be modified when making the final linking of the 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 (as in this 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 looking at a suitably old version).
- Some parts of the Makefile selection above can be automated by a suitable script that tries to guess the system on which it is running and select a suitably prepared Makefile. Maintaining such scripts is quite hard, as there were historically so many different kinds of systems (different library versions, different compilers, etc.).
- Even the creation of Makefiles can be automated from such guessing script. Such scripts are normally called **configure** or **Configure**, and are marked executable for the sake of convenience. The name of the script does not 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 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. Automake will be covered a bit later, but first there will be a look at a simple autoconf configuration file (historically called a driver, but this is an obsolete term now).

### 1.3.2 GNU Autoconf

Autoconf is a tool that will use the GNU m4 macro preprocessor to process the configuration file and output a shell script based on the macros used in the file. Anything that m4 does not recognize as a macro, will be passed verbatim to the output script. This means that almost any wanted shell script fragments can be included into the **configure.ac** (the modern name for the default configuration file for autoconf).

The first subject here is a simple example to see how the basic configuration file works. Then some limitations of m4 syntax will be covered, and hints on how to avoid problems with the syntax will be given.

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

```

# Since autoconf will pass through anything that it does not 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 are 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 do not 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 does not 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 1.15: 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 true in 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 own macros which can be stored in one's own directory. Also some development packages install new macros to use. One example of this will be covered later on.

The next step is to run *autoconf*. Without any parameters, it will read **configure.ac** by default. If **configure.ac** does not exist, it will try to read **configure.in** instead. N.B. Using the name **configure.in** is considered obsolete; the reason for this will be explained later.

```
[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'
```

Autoconf will not output information about its progress. Normally, only errors are output on stdout. Instead, it will create a new script called **configure** in the same directory, and set it as executable.

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

Unless one is experienced in reading convoluted shell code, it is best not

to try to understand what is attempted at the various stages. Normally the generated file is not fixed or modified manually, since it would be overwritten anyway the next time that someone runs autoconf.

When the generated script is executed, the output of various parts can be seen in the order that they were used in the configuration file.

N.B. 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 should not 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 quoting is not needed. It is best to avoid using brackets, unless the macro does not seem to work properly otherwise. When writing new macros, using brackets will become more important, but this material does not cover creating custom macros.
- If it is absolutely necessary to pass brackets to the generated script, there are three choices:
  - @<:@ is same as [, and @>:@ is same as ]
  - [ ] will expand into [] (most of time)
  - avoid [ and ] (most command line tools and shell do not really require them)
- Since m4 will use brackets for its own needs, the [ command cannot be used to test things in scripts, but instead test has to be used (which is more clear anyway). This is why the brackets are escaped with the rules given above, if they really are needed to be output into the generated shell script.

If bad shell syntax is introduced into the configuration file, the bad syntax will cause errors only when the generated script file is run (not when autoconf generates the script). In general, autoconf will almost always succeed, but the generated script might not. It is not always simple to know which error in the shell script corresponds to which line in the original configure.ac, but experience will teach that.

There can be noticed a line that reports the result of testing for **unistd.h**. It will appear twice: the first time because it is the default test to run whenever testing for headers, and the second time because it was explicitly tested for. 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 line's output above contains the values of variables. When the configure script runs, it will automatically create shell variables that can be

used in shell code fragments. The macros for checking what programs are available for compiling should illustrate that point. Here awk was used as an example.

The configure script will take the initial values for variables from the environment, but also contains a lot of options that can be given to the script, and using those will introduce new variables that can also be used. Anyone compiling modern open source projects will be familiar with options like prefix and similar. 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'
```

It might seem that giving the prefix does not change anything, but this is because the shell does not expand the value in this particular case. It would expand the value later on, if the variable was used in the script for doing something. In order to see some effect, one can try passing the datadir-option (because that is printed out explicitly).

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

### 1.3.3 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 also lot of complexity. Sometimes it may be difficult to find out why things do not quite work.

Sometimes it is useful to use the variables inside text files that are not directly related to the **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 as 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 the suffix. N.B. The substitution will be done when the generated configure script is run, not when autoconf is run.

The generated configure script will replace all occurrences of the variable name surrounded with 'at' (@) 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, the substitution will only be made for one file, but it is 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 1.16: 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 the names surrounded by '@'-characters will be replaced by the
values
of the variables (if present) by the configure script, when it is run
by the end user.

Prefix: @prefix@
C compiler: @CC@

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

```

Listing 1.17: 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@

```

This feature will be used later on. When running one's own version of the file, one might notice the creation of file called **config.status**. It is the file that actually does the substitution for external files, so if the configuration is otherwise complex, and one only wants to re-run the substitution of the output files, one can run the config.status script.

### 1.3.4 Introducing Automake

The next step is to create a small project that consists of yet another hello world. This time, there will be 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. This will be utilized in the next example, so that the Makefile does not have to be written 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, two things are necessary:

- A configuration file for automake (conventionally Makefile.am).
- Telling the autoconf that it should create Makefile.in based on Makefile.am by using automake.

The example starts 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 1.18: autoconf-automake/example3/configure.ac

Then the configuration file is presented 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 1.19: 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, the 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 notice the implementation of printHello.

```
/**
 * 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 1.20: 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 1.21: 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 1.22: autoconf-automake/example3/hello.c

The PACKAGE and VERSION defines will be passed to the build process automatically, and their use will be shown later.

```
[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.
```

Autoconf is complaining about a macro, for which it cannot find a definition (actually m4 is the program that does the complaining). The problem is that by default, autoconf only knows about built-in macros. When using a macro for integration, or a macro that comes with another package, autoconf needs to be told about it. Luckily, this process is quite painless.

A local **aclocal.m4** file needs to be created into the same directory with the **configure.ac**. When starting, autoconf will read this file, and it is enough to put the necessary macros there.

For this, a utility program called **aclocal** will be used, which will scan the **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 should be checked out at some point.

Now it is time to run **aclocal**, and then try and run **autoconf** again. N.B.: The messages that are introduced into the 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 PRG_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 PRG_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 ~/ ~/..
```

If one now lists the contents of the directory, one might be wondering, where the **Makefile.in** is. Automake needs to be run manually, so that the file will be created. At the same time, the missing files need to be introduced to the directory (such as 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 the directory, which by default will contain the GPL.* So, if the software is going to be distributed under some other license, this might be the correct moment to replace the license file with the appropriate one.

```

[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

```

Notice the second to last line of the output, telling that autoconf just created a Makefile (based on the Makefile.in that *automake* created).

It can prove to be tedious to perform all these steps manually each time when wanting to make sure that all generated files are really generated. Most developers create a script called **autogen.sh**, which will implement the necessary bootstrap procedures for them. Below is a file that is suitable for this example. Real projects might have more steps because of localization and other requirements.

```

#!/bin/sh
#
# A 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 1.23: autoconf-automake/example3/autogen.sh

In the above code, the line with touch is commented. This might raise a question. There is a target called distcheck that automake creates in the 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 they need to be created at some point. Without these files, the penultimate Makefile will complain, when running the distcheck-target.

Build the project now 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
```

### 1.3.5 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 the source, including the configure script and other necessary files (which are specified in Makefile.am).

To test whether it is possible to build the 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 and build the software with default make target. If it fails, the relevant error will be given.

It is recommended to make the distcheck target each time before making a dist target, so that one can be sure that the distribution tarball can be used outside the source tree. This step is especially critical later, when making Debian packages.

### 1.3.6 Cleaning up

For the sake of convenience, the example3 directory also includes a script called antigen.sh, which will try its best to get rid of all generated files (it will be necessary to autogen.sh the project afterwards).

Having a clean-up script is not very common in open source projects, but it is especially useful when starting autotools development, as it allows testing the toolchain easily from scratch.

The contents of `antigen.sh` that is suitable for simple projects:

```
#!/bin/sh
#
# A 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 1.24: `autoconf-automake/example3/antigen.sh`

### 1.3.7 Integration with Pkg-Config

The last part that is covered for autoconf is how to integrate pkg-config support into the 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.

Here only one will be used, `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 1.25: 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).

One should also check the validity of the package names by using:

```
pkg-config --list-all
```

to make sure one does not try to get the wrong library information.



# Bibliography

[1] GNU 'make'. <http://www.gnu.org/software/make/manual/make.html>.