# Autotools: a practitioner's guide to Autoconf, Automake and Libtool

Short URL: http://fsmsh.com/2753 [1]

There are few people who would deny that Autoconf, Automake and Libtool have revolutionized the free software world. While there are many thousands of Autotools advocates, some developers absolutely *hate* the Autotools, with a passion. Why? Let me try to explain with an analogy.

In the early 1990's I was working on the final stages of my bachelor's degree in computer science at Brigham Young University. I took a 400-level computer graphics class, wherein I was introduced to C++, and the object-oriented programming paradigm. For the next 5 years, I had a love-hate relationship with C++. I was a pretty good C coder by that time, and I thought I could easily pick up C++, as close in syntax as it was to C. How wrong I was. I fought late into the night, more often than I'd care to recall, with the C++ compiler over performance issues.

The problem was that the most fundamental differences between C++ and C are not obvious to the casual observer. Most of these differences are buried deep within the C++ language specification, rather than on the surface, in the language syntax. The C++ compiler generates code beneath the covers at a level never even conceived of by C compiler writers. This level of code generation provides functionality in a few lines of C++ code that requires dozens of lines of C code. Oh, yes--you can write object-oriented software in C. But you are required to manage all of the details yourself. In C++, these details are taken care of for you by the compiler. The advantages should be clear.

But this high-level functionality comes at a price--you have to learn to understand what the compiler is doing for you, so you can write your code in a way that complements it. Not surprisingly, often the most intuitive thing to do in this situation for the new C++ programmer is to inadvertently write code that works against the underlying infrastructure generated by the compiler.

And therein lies the problem. Just as there were many programmers then (I won't call them software engineers--that title comes with experience, not from a college degree) complaining of the nightmare that was C++, so likewise there are many programmers today complaining of the nightmare that is the Autotools. The differences between make and Automake are very similar to the differences between C and C++. The most basic single-line Makefile.am generates a Makefile.in file (an Autoconf template) containing nearly 350 lines of make script.

# Who should read this book

This book is written for the open source software package maintainer. I'm purposely not using the terms "free software" or "proprietary software that's free". The use of the term "open source" is critical in this context. You see, open source defines a type of software distribution channel. One in which the primary method of obtaining software functionality is downloading a source archive, unpacking, building and installing the built products on your system. Free software may be published in binary form. Proprietary software may be given away. But open source software implies source-level distribution.

Source-level distribution relegates a particular portion of the responsibility of software development to the end-user that has traditionally been assumed by the software developer. But end-users are not developers, so most of them won't know how to properly build your package. What to do, what to do... The most widely adopted approach from the earliest days of the open source movement was to make the package build process as simple as possible for the end user, such that she could perform a few well-known steps and have your package cleanly installed on her system.

Most packages are built using makefiles, and the make utility is as pervasive a tool as anything else that's available. It's very easy to type `make`--but that's not the problem. The problem crops up when the package doesn't build successfully, because of some unanticipated difference between the user's system and the developer's system.

Thus was born the ubiquitous configure script--initially a simple shell script that configured the end-user's environment so that the make utility could successfully build a source package on the end-user's system. Hand-coded configure scripts helped, but they weren't the final answer. They fixed about 65

percent of the problems resulting from system configuration differences--and they were a pain in the neck to write properly. Dozens of changes were made incrementally over a period of years, until the script would work properly on most systems anyone cared about. But the entire process was clearly in need of an upgrade.

Do you have any idea of the number of build-breaking differences there are between existing systems today? Neither do I, but there is a handful of developers in the world who know a large percentage of these differences. Between them and the free software community, the Autotools were born. The Autotools were designed to create configure scripts and makefiles that work correctly and provide significant chunks of valuable end-user functionality under most circumstances, and on most systems--even systems not initially considered (or even known about) by the package maintainer.

So, returning to that passionate hate felt by some developers toward the Autotools: If you get your head screwed on straight about the primary purpose of the Autotools, then hate quickly turns into respect--and even appreciation. Often the root of such hate is a simple misunderstanding of the rationale behind the Autotools. The purpose of the Autotools is not to make life simpler for the package maintainer (although it really does in the long run). *The purpose of the Autotools is to make life simpler for the end-user.*

To drive my point home, I'll wager that you'll never see a Linux distribution packager spouting hateful sentiment on the Autotools mailing lists. These people are in a class of engineers by themselves. They're generally quiet on mailing lists--asking an occasional well-considered question when they really need to--but lurking and learning, for the most part. Packagers grasp the advantages of the Autotools immediately. They embrace them by studying them until they know them like an expert C++ programmer knows his compiler. They don't write many Autoconf input scripts, but they do patch a lot of them.

How do you become such an expert? I recommend you start with this book. I've organized it in the best way I know how to help you get your head around the functionality provided by the Autotools. But don't stop there. Pick up the manuals. They're free, and links are provided in the References section of this book, but they're easy to find with a simple internet query. I've left a LOT of details out of this book, because my purpose is to quickly get you up to speed on understanding and using the Autotools. The Autotools manuals are well-written and concise, but more importantly, they're complete. After reading this book,

they should be a cake walk.

Then study open source and free software packages that use the Autotools. See what other experts have done. Learning by example is an excellent way to begin to retain the information you've read. Finally, instrument some of your own projects with the Autotools. Doing is by far the best way to learn. The initial reading will reduce the frustration of this exercise to something bearable.

Above all, remember why you're doing this--because you want your end-user's experience with your package to be as delightful as possible. No open source project was ever successful until it had a well-established user base, and you don't get there by alienating your users. You do it by creating a user build, installation and operation experience that shines. You'll still need to handle the operation experience, of course, but Autotools can provide a *great* multi-platform build and installation experience--with far less effort on your part.

# The book that was never to be

I've wondered often during the last eight years how strange it is that the *only* third-party book on Autotools that I've been able to discover is the New Rider's 2000 publication of GNU AUTOCONF, AUTOMAKE and LIBTOOL, affectionately known in the community as "The Goat Book".

I've been in this industry for 25 years, and I've worked with free software for quite some time now. I've learned a lot about free software maintenance and development--most of it, unfortunately, by trial and error. Had there *been* other books on the topic, I would have snatched them all up immediately, rather than spend hours--even days sometimes--trying to get the Autotools to do something I could have done in a makefile in a few minutes.

I've been told by publishers that there is simply no market for such a book. In fact, one editor told me that he himself had tried unsuccessfully to entice authors to write this book a few years ago. His authors wouldn't finish the project, and the publisher's market analysis indicated that there was very little interest in the book.

No interest?! Let's analyze this picture: There are nearly 200,000 free software projects on sourceforge.net alone. If only 10 percent of those are still active, that's still 20,000 live projects. If 80 percent of those are Linux or Unix based

packages, that's 16,000 free software packages that might use the Autotools. And that's only sourceforge.net. Each of those packages has at least one maintainer--often two or three. Each of those maintainers *probably* uses (or has tried to use) the Autotools. Many of them have a fairly solid understanding of the Autotools by now, but at what expense in time and effort did they gain this understanding?

Publishers believe that free software developers tend to disdain written documentation--perhaps they're right. Interestingly, books on Perl sell like Perl's going out of style--which is actually somewhat true these days--and yet people are still buying enough Perl books to keep their publishers happy. All of this explains why there are ten books on the shelf with animal pictures on the cover for perl, but literally nothing for free software developers.

The authors of the Goat Book, Gary Vaughan, Ben Elliston, Tom Tromey and Ian Lance Taylor, are well known in the industry, to say the least--indeed, they're probably the best people I know of to write a book on the Autotools. But, as fast as free software moves these days, a book published in 2000 might as well have been published in 1980. Nevertheless, because of the need for *any* book on this subject, the Goat Book [2] is still being sold new in bookstores. In fairness to the authors, they have maintained an online version [3] through February of 2006.

The biggest gripe I have with the Goat Book is the same gripe I have with the GNU manuals themselves. I'm talking about the shear volume of information that is assumed to be understood by the reader. The Goat Book is written in a very non-linear fashion, so it's difficult to learn anything from it. It's a great reference, but a terrible tutorial. Perhaps the authors were targeting an audience that had already graduated to more advanced topics. In either case, the Goat Book, while being very complete from a content perspective, is definitely not a great learning resource for the beginner.

And yet a large percentage of their readership today are young people just starting out with Unix and Linux, and most of their issues center around Unix utilities not generally associated with the Autotools. Take `sed`, for example: What a dream of a tool to work with--I love it! More to the point however, a solid understanding of the *basic* functionality of `sed`, `m4`, shell script and other utilities is critical to understanding the proper use of the Autotools. The Goat Book does cover the m4 macro processor in great detail, but it's not clear to the uninitiated that one might do well to start with Chapter 21. Understanding how

something works under the covers is often a good way to master a topic, but a general introduction at an appropriate point in higher-level discussions can make all the difference to a beginner.

Existing GNU documentation is more often reference material than solution-oriented instruction. What we need is a cookbook-style approach, covering real problems found in real projects. As each recipe is mastered, the reader makes small intuitive leaps--I call them minor epiphanies. Put enough of these under your belt and overall mastery of the Autotools is ultimately inevitable.

Let me give you another analogy: I'd been away from math classes for about three years when I took my first college calculus course. I struggled the entire semester with little progress. I understood the theory, but I had trouble with the homework. I just didn't have the background I needed. So the next semester, I took college algebra and trigonometry as half-semester classes each ("on the block", to use the vernacular). At the end of that semester I tried calculus again. This time I did very well--finishing the class with a solid A grade. What was missing the first time? Just basic math skills. You'd think it wouldn't have made that much difference, but it really does.

The same concept applies to understanding the Autotools. You need a solid understanding of the tools upon which the Autotools are built in order to become proficient with the Autotools themselves. For example, here's a message I came across a few days ago while I was perusing the Autoconf mailing list:

```
>>> If I do this:
>>>
>>> AC_CHECK_FUNC(
>>>   [chokeme],
>>>   [],
>>>   []
>>> )
>>>
>>> It will yield shell code that ends in:
>>>
>>> if
>>>  :
>>> else
>>>
>>> fi
```

```
>>>
>>> Which produces a configure script that dies
>>> with:
>>> "syntax error near unexpected token `fi'"
>>>
>>> Is this an autoconf bug, or user error on
>>> my part?

>> The else part is not empty, it consists of
>> explicit whitespace. When collecting arguments
>> only unquoted leading whitespace is skipped by
>> m4, trailing whitespace (quoted or not) is
>> preserved. You need to put the closing paren
>> immediately after the closing quote of the
>> argument.

> Is that something I should always do? I've been
> consistently putting the closing paren on its
> own line. Is that a "never"?

You can instead use dnl to ignore the trailing
whitespace, provided the closing paren is in
column 1.
```

Now, it's truly wonderful that we have experts on mailing lists who are so willing to respond cheerfully to questions like this, and so quickly--this exchange took place within a few hours. However, without looking, I submit that similar questions have probably been asked dozens of times in the last 5 years. Not because mailing list posters don't read the archives (although I'll admit that they probably don't often do so), but rather because this problem can rear its ugly head in many different ways, none of which look remotely related to each other in the eyes of the uninitiated.

Here are some of the problems with the response to this request: Does the original poster (OP) even know what m4 is? If so, does he realize he's running it when he executes "autoconf" to generate his configure script? Alright, suppose he does; either way, he's clearly not an m4 expert or he wouldn't have needed help with this issue to begin with.

Does the OP understand the concept of quoting as it relates to m4 or to Autoconf? Perhaps he's always simply copied one configure.ac script to another,

modifying as little as possible to get it to work with a new project. Given the high-level nature of configure.ac, this is entirely possible (I've done it myself). If so, he may just assume that the square brackets are necessary around each parameter in an Autoconf macro. Given the nature of the question, I'd say the OP believes that the entirety of each parameter is contained within the brackets, so this assumption is not at all improbable.

Another problem is seen in the final response where the OP is told, "...instead use dnl to ignore the trailing whitespace..." If the OP didn't understand m4 whitespace rules, he probably doesn't know about the m4 built-in macro, `dnl`. If that's the case, then this response made no sense to him whatsoever. Even if he did understand what he was to do--perhaps based on having seen `dnl` being used in other configure.ac scripts, apparently as a secondary form of comment delimiter--he probably doesn't understand the full impact or use of this macro. Regardless, you can bet there are other mailing list readers who experienced far more confusion over this exchange.

This book attempts to alleviate some of the confusion and reduce the existing learning curve by presenting the Autotools in a manner conducive to an open source beginner learning how to use them.

# How this book is organized

Chapter 1 presents a general overview of the packages that are considered part of the GNU Autotools. This chapter describes the interaction between these packages, and the files consumed by and generated by each one. In each case, I've provided a graphic depiction of the flow of data from hand-coded input files, to final output files. Don't worry if you feel overwhelmed after reading Chapter 1. The details will become clear later. I recommend that you give this chapter a quick read to start with, and then come back to it later, after you've read the rest of this book.

Chapter 2 covers free software project structure and organization. This chapter also goes into detail on the GNU coding standards and the Filesystem Hierarchy Standard documents, both of which have played vital roles in the design of the Autotools. It presents some fundamental tenets upon which the design of each of the Autotools is based. With these concepts, you'll be prepared to understand some of the most fundamental rationale behind architectural decisions made by the Autotools developers. This chapter designs a simple project (jupiter) from

start to finish using a hand-coded configure script and makefiles. It builds on jupiter in a step-wise fashion, as we begin to discover useful functionality to make our's and our end-users' tasks simpler, relative to the jupiter project. The project is built on principles taken from these two documents. As a side benefit, the GNU manuals for the Autotools should begin to make a lot more sense to you.

Chapters 3, 4 and 5 cover the basic purpose and use of the GNU Autoconf, Automake and Libtool packages, respectively. If you already have a basic familiarity with these packages, you can probably skip these chapters, but please feel free to revisit them if you find yourself in over your head with the remaining chapters.

Chapter 6 takes an existing complex open source project (FLAIM) through the process of converting from a hand-coded build system to an Autotools build system, from start to finish. The example provided by this chapter will use the concepts presented in previous chapters to take it from the original hand-coded makefiles to a complete Autotools project, implementing all of the features provided by the original build system. This process should help you to understand how you might "autoconfiscate" one of your own existing complex projects.

Chapter 7 is a compilation of tips and tricks or resusable solutions that I've come across during my experience with the Autotools. I could have shoe-horned this information into more or less appropriate locations in the preceding chapters. I chose not to do this for two reasons: First, I didn't want to clutter the main text with side issues--one of my goals in writing this book was to make it readable. Second, I didn't want to reduce the importance of these items by slipping them in somewhere. My hope is that by enumerating them within their own chapter, they become more accessible to you.

Appendix A provides an overview of those features of the M4 macro processor that are relevant to obtaining a solid understanding of Autoconf.

Finally, the References section includes relevant links to the best material on Autotools available on the internet, including manuals and tutorials.

1. http://fsmsh.com/2753

2. http://www.amazon.com/GNU-Autoconf-Automake-Libtool-Circle/dp/1578701902
3. http://sourceware.org/autobook