Learning Autotools - autoconf, automake, & libtool

Learning Autotools - autoconf, automake, & libtool

Taken from
http://www.freesoftwaremagazine.com/books/autotools_a_guide_to_autoconf_automake_libtool

- Chapter 1
  - http://www.freesoftwaremagazine.com/articles/brief_introduction_to_gnu_autotools
- Chapter 2
  - http://www.freesoftwaremagazine.com/articles/gnu_coding_standards_applied_to_autotools
- Chapter 3
  - http://www.freesoftwaremagazine.com/articles/configuring_a_project_with_autoconf
- Chapter 4
  - http://www.freesoftwaremagazine.com/articles/automatically_writing_makefiles_with_autotools
- Chapter 5
  - http://www.freesoftwaremagazine.com/articles/building_shared_libraries_once_using_autotools
- Chapter 6
  - http://www.freesoftwaremagazine.com/articles/autotools_example
- Chapter 7
  - http://www.freesoftwaremagazine.com/articles/catalog_of_reusable_solutions
- Appendix A
  - http://www.freesoftwaremagazine.com/articles/overview_of_m4_overview
- References
  - http://www.freesoftwaremagazine.com/articles/reusing_autotools_solutions

Some helpful links

- Frank Schacherer's Make Cheat Sheet
  - http://www.schacherer.de/frank/technology/tools/make.html
- mxenoph Make Cheat Sheet
  - https://github.com/mxenoph/cheat_sheets/blob/master/make_cheatsheet.pdf
- Martin Vsteticka's Make Tutorial
  - http://martinvseticka.eu/temp/make/presentation.html#1
  -

Basic Skills

A brief introduction to the GNU Autotools

- If you're writing [free or open source] software targeting Unix or Linux systems, then you should be using the GNU Autotools.
- The purpose of the Autotools is two-fold
    - make life easer for your project
    - more portable--even to systems on which you've never tested, installed or even built your code
- About the only scenario where it makes sense NOT to use the Autotools is the one in which you are writing software for non-Unix platforms only--Microsoft Window comes to mind.

Your choice of language

- There are two factors that determine the importance of a computer language within the GNU community:
    - Are there any GNU packages written in the language?
    - Does the GNU compiler tool set support the language?
- Autoconf provides native support for the following languages based on these two criteria, where "native support" mean that Autoconf will compile, link and run source-level feature checks in these languages.:
    - C
    - C++
    - Objective C
    - Fortran
    - Fortran 77
    - Erlang
- If you want to build a Java package, you can configure Automake to do so, but you can't ask Autoconf to compile, link or run Java-based checks. Java simply isn't supported natively at this time by Autoconf.

Generating your package build system

- The GNU Autotools framework is comprised of three main packages
    - Autoconf
    - Automake
    - Libtool
- These packages were invented in that order, and evolved over time.
- Additionally, the tools in the Autotools packages can depend on or use utilities and functionality from the gettext, m4, sed, make and perl packages, as well as others.
- It's very important at this point to distinguish between a maintainer's system and an end-user's system.
    - The design goals of the Autotools specify that an Autotools-generated build

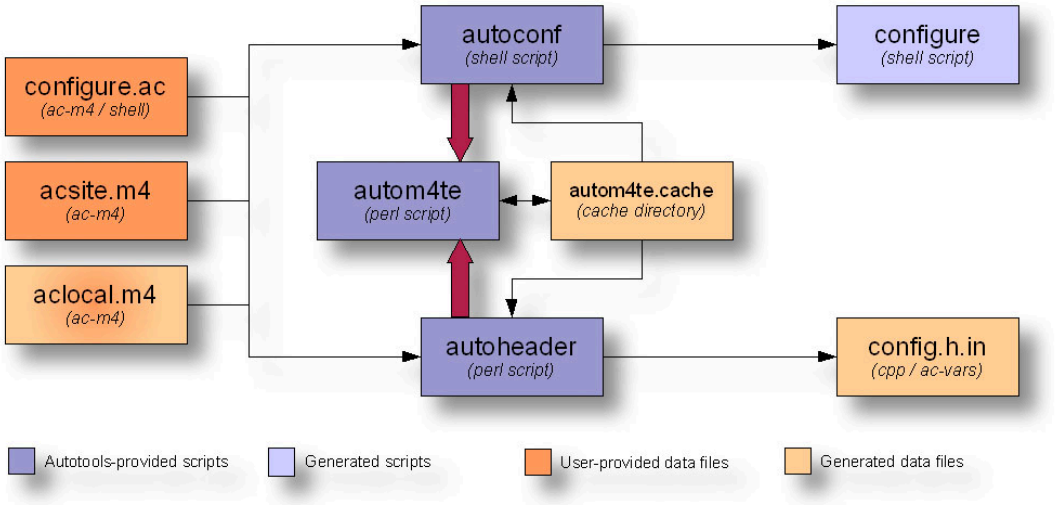
system rely only on readily available, preinstalled tools on the host machine.
    - Perl is only required on machines that maintainers use to create distributions, not on end-user machines that build packages from resulting release distributions packages.
    - A corollary to this is that end-users' machines need not have the Autotools installed.
- If you've ever downloaded, built and installed software from a "tarball"--a compressed archive with a .tar.gz, .tgz or .tar.bz2 extension--then you're probably aware of the fact that there is a common theme to this process. It usually looks something like this:
    - gzip -cd hackers-delight-1.0.tar.gz | tar -xvf -
    - ...
    - cd hackers-delight-1.0
    - ./configure
    - make all
    - sudo make install
- Most developers know and understand the purpose of the make utility, but what's the point of the configure script?
    - The use of configuration scripts (generally named configure) started a long time ago on Unix systems due to the variety imposed by the fast growing and divergent set of Unix and Unix-like platforms.
    - It's interesting to note that while Unix systems have generally followed the defacto-standard Unix kernel interface for decades, most software that does anything significant generally has to stretch outside of these more or less standardized boundaries.
    - Configuration scripts are hand-coded shell scripts designed to determine platform-specific characteristics, and to allow users to choose package options before running make.
    - This approach worked well for decades.
    - With the advent of dozens of Linux distributions, the explosion of feature permutations has made writing a decent portable configuration script very difficult--much more so than writing the makefiles for a new project.
    - Most people have come up with configuration scripts for their projects using a well-understood and pervasive technique--copy and modify a similar project's script.
    - By the early 90's it was becoming apparent to many developers that project configuration was going to become painful if something weren't done to ease the burden of writing massive shell scripts to manage configuration options--both those related to platform differences, and those related to package options.

Autoconf

- While configuration scripts were becoming longer and more complex, there were really only a few variables that needed to be specified by the user.
    - Most of these were simply choices to be made regarding components, features and options:
        - Where do I find libraries and header files?
        - Where do I want to install my finished product?

        - Which optional components do I want to build into my products?
    - With Autoconf, instead of modifying, debugging and losing sleep over literally thousands of lines of supposedly portable shell script, developers can write a short meta-script file, using a concise macro-based language, and let Autoconf generate a perfect configuration script.
    - A generated configuration script is more portable, more correct, and more maintainable than a hand-code version of the same script.
- Autoconf generated configure scripts provide a common set of options that are important to all portable, free, open source, and proprietary software projects running on LSB-compliant systems. (LSB = Linux Standard Base)
- The Autoconf package provides several programs.
    - Autoconf itself is written in Bourne shell script, while the others are perl scripts.
        - autoconf
        - autoheader
        - autom4te
        - autoreconf
        - autoscan
        - autoupdate
        - ifnames
- autoheader
    - The autoheader utility generates a C language header file template from configure.ac
    - This template file is usually called config.h.in
- autom4te
    - The automate utility is a cache manager used by most of the other Autotools.
    - In the early days of Autoconf there was really no need for such a cache, but because most of the Autotools use constructs found in configure.ac, the cache speeds up access by successive programs to configure.ac by about 40 percent or more
    - mainly used internally by the Autotools
    - the only sign you're given that it's working is the existence of an autom4te.cache directory in your top-level project directory
- autoreconf
    - The autoreconf program can be used to execute the configuration tools in the Autoconf, Automake and Libtool packages as required by the project
    - The purpose of autoreconf is to minimize the amount of regeneration that needs to be done, based on timestamps, features, and project state
    - Think of autoreconf as an Autotools bootstrap utility
    - If all you have is a configure.ac file, running autoreconf will run the tools you need in order to run configure and then make
- autoscan
    - The autoscan program is used to generate a reasonable configure.ac file for a new project
- autoupdate
    - The autoupdate utility is used to update your configure.ac or template (*.in) files to the syntax of the current version of the Autotools
- ifnames

    - The ifnames program is a small, and generally under-utilized program that accepts a list of source files names on the command line, and displays a list of C preprocessor definitions and their containing files on the stdout device
    - This utility is designed to help you determine what to put into your configure.ac and Makefile.am files for the sake of portability
    - If your project has already been written with some level of portability in mind, ifnames can help you find out where those attempts are located in your source tree, and what the names of potential portability defintions might be
- Of the tools in this list, only autoconf and autoheader are used directly by the project maintainer while generating a configure script, and actually, as we'll see later, only autoreconf really needs to be called directly
- The following diagram shows the interaction between input files and the Autoconf and autoheader programs to generate product files:



- The aclocal.m4 input file is both generated and user updated

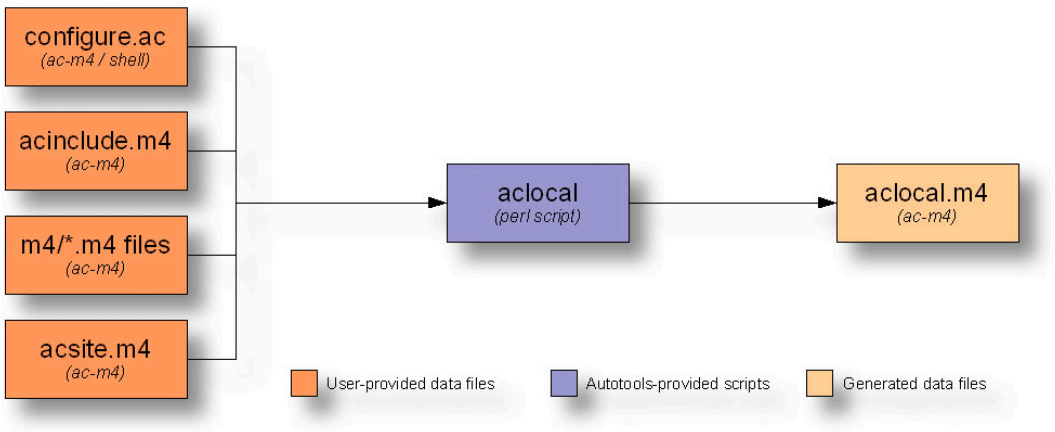
Automake

- So, what's so difficult about writing a makefile?
    - Well, actually, once you've done it a few times, writing a basic makefile for a new project is really rather trivial
    - The problems occur when you try to do more than just the basics
    - Llet's face it--what project maintainer has ever been satisfied with just a basic makefile?
- Standard make targets, which are to be expected
    - make install
    - make all
    - make clean
- Automake's job is to convert a much simplified specification of your project's build process into standard boilerplate makefile syntax that always works correctly the first

time, and provides all the standard functionality expected of a free software project
- In actuality, Automake creates projects that support guidelines defined in the GNU Coding Standards (https://www.gnu.org/prep/standards/)
- The Automake package provides the following tools in the form of perl scripts:
    - automake
    - aclocal
- The primary task of the Automake program is to generate standard makefile templates (named Makefile.in) from high-level build specification files (named Makefile.am)

Aclocal

- The aclocal utility is actually documented by the GNU manuals as a temporary work-around for a certain lack of flexibility in Autoconf
- Autoconf was designed and written first, and then a few years later, the idea for Automake was conceived as an add-on for Autoconf
- But Autoconf was really not designed to be extensible on the scale required by Automake
- The aclocal utility was designed to create a project's aclocal.m4 file, containing all the required Automake macros
- Since Automake's aclocal utility basically took over aclocal.m4 for its own purposes, it was also designed to read a new user-provided macro file called acinclude.m4
- Essentially, aclocal's job is to create an aclocal.m4 file by consolidating various macro files from installed Autotool packages and user-specified locations, such that Autoconf can find them all in one place
- The current recommendation is that you create a directory in your project directory called simply m4 (acinclude seems more appropriate to this author), and add macros in the form of individual .m4 files to this directory
    - All files in this directory will be gathered into aclocal.m4 before Autoconf processes your configure.ac file
    - Ultimately, aclocal will be replaced by functionality in Autoconf itself (we'll see)
- When used without Automake and Libtool, the aclocal.m4 file is written by hand, but when used in conjunction with Automake and Libtool, the file is generated by the aclocal utility, and acinclude.m4 is used to provide project-specific macros



Libtool

- The Libtool package allows you to build shared libraries on different Unix platforms without adding a lot of very platform-specific conditional code to your build system and source code
- Libtool not only provides a set of Autoconf macros that hide library naming differences in makefiles, but it also provides an optional library of dynamic loader functionality that can be added to your programs, allowing you to write more portable runtime dynamic shared object management code
- The libool package provides the following programs, libraries and header files:
    - libtool (program)
    - libtoolize (program)
    - ltdl (static and shared libraries)
    - ltdl.h (header file)

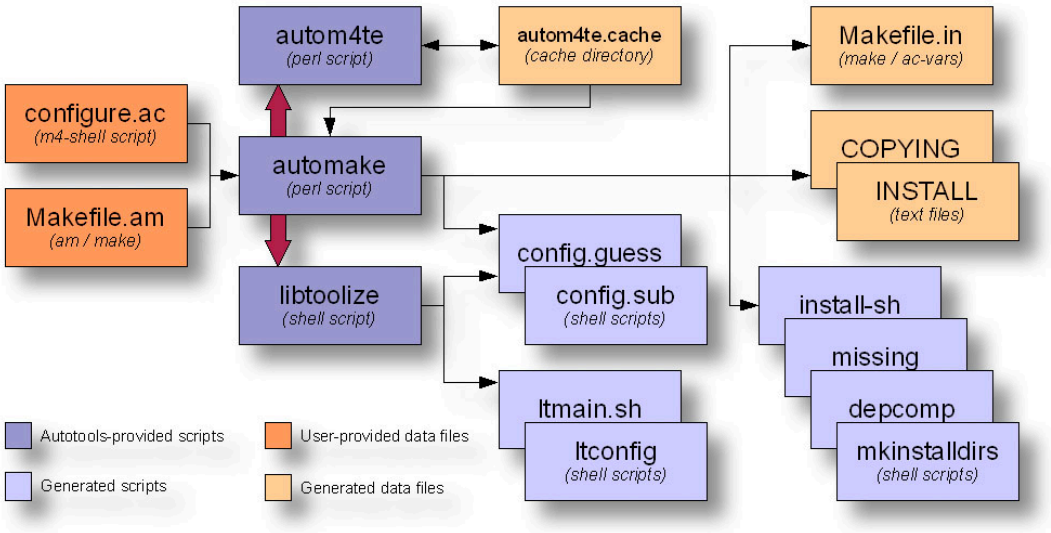
Libtoolize

- The libtoolize shell script is used to prepare your project to use Libtool
- In reality, libtoolize generates a custom version of the libtool script in your project directory
- This script is then executed at the appropriate time by Automake-generated makefiles

The Libtool C API - ltdl

- The Libtool package also provides the ltdl library and header files, which provide a consistent run-time shared object manager across platforms
- The ltdl library may be linked statically or dynamically into your programs, giving them a consistent runtime shared library access interface from one platform to another

- The following data flow diagram illustrates the interaction between Automake and Libtool scripts and input files to create products used by users to configure and build your project:



- Automake and Libtool are both standard pluggable options that can be added to configure.ac with a few simple macro calls.

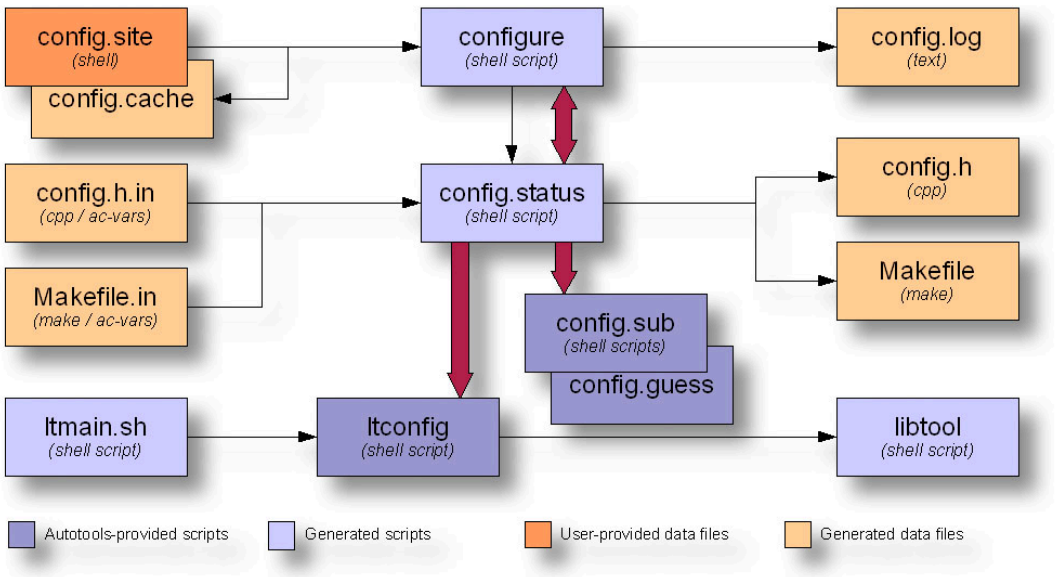
Building Your Package

- You ought to be concerned that your users' build experience is much simpler than yours.
- It wouldn't hurt a bit if you got some benefit from this concern, as well

Running Configure

- Once the Autotools have finished their work, you're left with a shell script called configure, and one or more Makefiles.in files.
    - These product files are intended to be packages with project release distribution packages
    - Your users download these packages, unpack them, and run configure and make
    - The configure script generates Makefiles from the Makefile.in files
    - It also generates a config.h header file from the config.h.in file built by autoheader
- So why didn't the Autotools just generate the makefiles directly to be shipped with your release?
    - One reason is that without makefiles, you can't run make.
        - This means that you're forced to run configure first, after you download

and unpack a project distribution package
        - Makefile.in files are nearly identical to the makefiles you might write by hand, except that you didn't have to
        - And they do a lot more than most people are willing to hand code into a set of makefiles
    - Another reason is that the configure script may then insert platform-characteristics and user-specified optional features directly into your makefiles, making them more specifically tailored to the platforms on which they are being used
- The following diagram illustrates the interaction between configure and the scripts that it executes during the build process to create your Makefiles and your config.h header file:

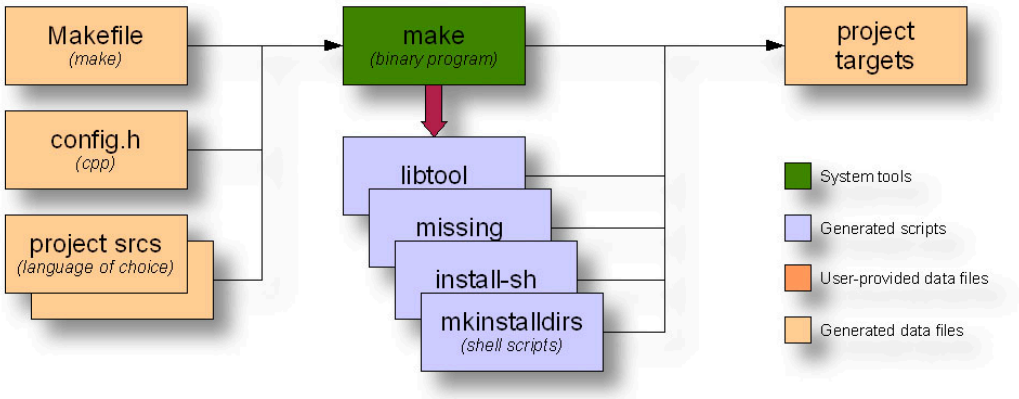


- The configure script appears to have this weird sort of incestuous relationship with another script called config.status
    - As it turns out, the only file (besides a log file) that configure generates is config.status
    - The configure script's function is to determine platform characteristics and features available, as specified in configure.ac
    - Once it has this information, it generates config.status such that it contains all of the check results, and then calls it
    - The newly generated config.status file uses the check information (now embedded within it) to generate platform-specific config.h and makefiles, as well as any other files specified for instantiation in configure.ac
    - As the double ended red arrow shows, config.status can also call configure
    - When used with the --recheck option, config.status will call configure with the same command line options with which it was originally generated
- The configure script also generates a log file called config.log, which contains very

useful information about why a particular execution of configure failed on your user's platform
- A nice feature of config.log is that it logs how configure was executed--which command line options were used
- If you need to re-generate makefiles and config.h header files for some reason, just type ./config.status in the project build directory

Running make

- Finally, you run make.
  - Just plain old make
  - In fact, the Autotools designers went to a LOT of trouble to ensure that you didn't need any special version or brand of make.
  - You don't need GNU make--you can use Solaris make, or BSD Unix make if you wish
- The following diagram depicts the interaction between the make utility and the generated makefiles during the build process to create your project products:



Let's practice ...

Project Structure
We'll start with a simple example project, and build on it as we continue our exploration of source-level software distribution. OSS projects generally have some sort of catchy name--often they're named after some past hero or ancient god, or even some made-up word--perhaps an acronym that can be pronounced like a real word. I'll call this the jupiter project, mainly because that way I don't have to come up with functionality that matches my project name! For jupiter, I'll create a project directory structure something like this:

```
$ cd dev
$ mkdir -p jupiter/src
$ touch jupiter/Makefile
```

```
$ touch jupiter/src/Makefile
$ touch jupiter/src/main.c
$ cd jupiter
```

We'll start with support for the most basic of targets in any software project: all and clean. As we progress, it'll become clear that we need to add a few more important targets to this list, but for now, these will get us going. The top-level Makefile does very little at this point, merely passing requests for all and clean down to src/Makefile recursively. In fact, this is a fairly common type of build system, known as a recursive build system. Here are the contents of each of the three files in our project:

Makefile

```
all clean jupiter:
	$(MAKE) -C src $@
```

src/Makefile

```
all: jupiter

jupiter: main.c
	gcc -g -O0 -o $@ $+

clean:
	-rm jupiter
```

src/main.c

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char * argv[])
{
	printf("Hello from %s!\n", argv[0]);
	return 0;
}
```

Some makefile basics
Here are a few things to keep in mind
Besides comments, which begin with a HASH mark, there are only three types of entities in a makefile:

- variable assignments
- rules
- commands

Commands always start with a TAB character.

- Any line in a makefile beginning with a TAB character is ALWAYS considered by make to be a command
- A list of one or more commands should always be associated with a preceeding rule

The general layout of a makefile is:

```
var1=val1
var2=val2
...
rule1
        cmd1a
        cmd1b
        ...
rule2
        cmd2a
        cmd2b
        ...
```

Variable assignments may take place at any point in the makefile, however you should be aware that make reads each makefile twice

- The first pass gathers variables and rules into tables, and the second pass resolves dependencies defined by the rules
- So regardless of where you put your variable definitions, make will act as though they'd all been declared at the top, in the order you specified them throughout the makefile.

The make utility is a rule-based command engine

- The rules indicate when and which commands should be executed
- When you prefix a line with a TAB character, you're telling make that you want it to execute these statements from a shell according to the rules specified on the line above.

Variables in makefiles are nearly identical to shell or environment variables

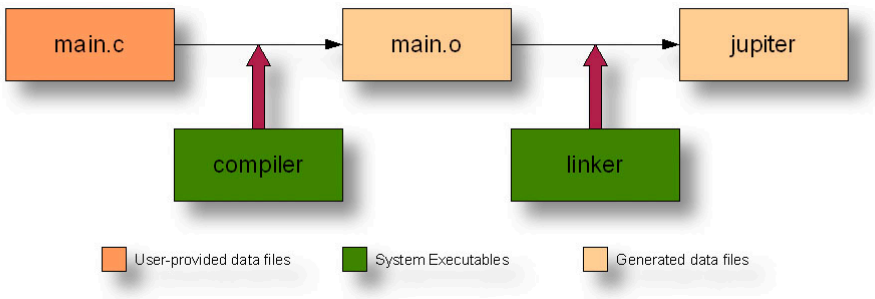
- In Bourne shell syntax, you'd reference a variable in this manner: ${my_var}
- In a makefile, the same syntax applies, except you would use parentheses instead of french braces: $(my_var)
- As in shell syntax, the delimiters are optional, but should be used to avoid ambiguous syntax when necessary
- Thus, $my_var is functionally equivalent to $(my_var)
- If you ever want to use a shell variable inside a make command, you need to escape

the DOLLAR sign by doubling it. For instance, $${shell_var}
- In general, it's a good idea not to depend on environment variables in your build process, although it's okay to use certain variables conditionally, if they're present.

Lines in the example makefiles above that are not variable assignments (don't contain an EQUAL sign), and are not commands (are not prefixed with a TAB character), are all rules of one type or another.

The rules used in the examples above are known as "common" make rules, containing a single COLON character

- The COLON character separates targets on the left from dependencies on the right
- Targets are products--generally file system entities that can be produced by running one or more commands, such as a C compiler or a linker
- Dependencies are source objects, or objects from which targets may be created
    - These may be computer language source files, or anything really that can be used by a command to generate a target object
- For example
    - C compiler takes dependency main.c as input, and generates target main.o
    - A linker takes dependency main.o as input, and generates a named executable target, jupiter, in these examples:



Next revision of the simple Makefile from above:

```
jupiter: main.o
        ld main.o ... -o jupiter

main.o: main.c
        gcc -c -g -O2 -o main.o main.c
```

- This sample makefile contains two rules
    - The first says that jupiter depends on main.o
    - The second says that main.o depends on main.c.

- Ultimately, of course, jupiter depends on main.c, but main.o is a necessary intermediate dependency in this case, because there are two steps to the process--compile and link--with an intermediate result in between. For each rule, there is an associated list of commands that make uses to build the target from the list of dependencies.

Next revision of this simple Makefile:

```
sources = main.c print.c display.c

jupiter: $(sources)
        gcc -g -O2 -o jupiter $(sources)
```

- Of course, there is an easier way in the case of this example
    - gcc (as with most compilers) will call the linker for you--which, as you can probably tell from the elipsis in my example above, is very desirable
    - This alleviates the need for one of the rules, and provides a convenient way of adding more dependent files to the single remaining rule
- In this example, I've added a make variable to reduce redundancy
    - We now have a list of source files that is referenced in two places
    - But, it seems a shame to be required to reference this list twice in this manner, when the make utility knows which rule and which command it's dealing with at any moment during the process
    - Additionally, there may be other objects in the dependency list that are not in the sources variable
    - It would be nice to be able to reference the entire dependency list without duplicating that list.
        - As it happens, there are various "automatic" variables that can be used to reference portions of the controlling rule during the execution of a command
        - For example $(@) (or the more common syntax $@) references the current target, while $+ references the current list of dependencies

Next revision of this simple Makefile:

```
sources = main.c print.c display.c

jupiter: $(sources)
        gcc -g -O2 -o $@ $+
```

- If you enter "make" on the command line, the make utility will look for the first target in a file named "Makefile" in the current directory, and try to build it using the rules defined in that file
- If you specify a different target on the command line, make will attempt to build that

target instead

Targets need not be files only

- They can also be so-called "phony targets", defined for convenience, as in the case of all and clean
- These targets don't refer to true products in the file system, but rather to particular outcomes--the directory is "cleaned", or "all" desirable targets are built, etc.
    - In the same way that dependencies may be listed on the right side of the COLON, rules for multiple targets with the same dependencies may be combined by listing targets on the left side of the COLON, in this manner:

      ```
      all clean jupiter:
              $(MAKE) -C src $@
      ```
    - The -C command-line option tells make to change to the specified directory before looking for a makefile to run

Creating a source distribution archive

It's great to be able to type "make all" or "make clean" from the command line to build and clean up this project. But in order to get the jupiter project source code to our users, we're going to have to create and distribute a source archive.

Building a source distribution archive is usually relegated to the dist target, so we'll add one

- Normally, the rule of thumb is to take advantage of the recursive nature of the build system, by allowing each directory to manage its own portions of a global process
- An example of this is how we passed control of building jupiter down to the src directory, where the jupiter source code is located
- However, the process of building a compressed archive from a directory structure isn't really a recusive process
    - well, okay, yes it is, but the recursive portions of the process are tucked away inside the tar utility
- This being the case, we'll just add the dist target to our top-level makefile:

Top-level Makefile

```
package = jupiter
version = 1.0
tarname = $(package)
distdir = $(tarname)-$(version)

all clean jupiter:
        $(MAKE) -C src $@
```

```
dist: $(distdir).tar.gz

$(distdir).tar.gz: $(distdir)
        tar chof - $(distdir) |\
          gzip -9 -c >$(distdir).tar.gz
        rm -rf $(distdir)

$(distdir):
        mkdir -p $(distdir)/src
        cp Makefile $(distdir)
        cp src/Makefile $(distdir)/src
        cp src/main.c $(distdir)/src

.PHONY: all clean dist
```

- In this version of the top-level Makefile, we've added a new construct, the .PHONY rule.
- At least it seems like a rule
    - it contains a COLON character, anyway
- The .PHONY rule is a special kind of rule called a "dot-rule", which is built into make.
    - The make utility understands several different dot-rules
    - The purpose of the .PHONY rule is simply to tell make that certain targets don't generate file system objects, so make won't go looking for product files in the file system that are named after these targets.
    - Normally, the make utility determines which commands to run by comparing the time stamps of the associated rule products to those of their dependencies in the file system, but phony targets don't have associated file system objects.
- We've added the new dist target in the form of three rules for the sake of readability, modularity and maintenance
    - This is a great rule of thumb to following in any software engineering process:
        - Build large processes from smaller ones, and reuse the smaller processes where it makes sense to do so

The dist target depends on the existance of the ultimate goal, a source-level compressed archive package, `jupiter-1.0.tar.gz`, also known as a "tarball"

- We've added a make variable for the version number to ease the process of updating the project version later
- We've used another variable for the package name for the sake of possibly porting this makefile to another project
- We've also logically split the functions of package name and tar name, in case we want them to be different later - the default tar name is the package name
- Finally, we've combined references to these variables into a distdir variable to reduce duplication and complexity in the makefile

The rule that builds the tarball indicates how this should be done with a command that uses the gzip and tar utilities to create the file

- But, notice also that the rule has a dependency - the directory to be archived
- We don't want everything in our project directory hierarchy to go into our tarsal - only exactly those files that are necessary for the distribution
- Basically, this means any file required to build and install our project
- We certainly don't want object files and executables from our last build attempt to end up in the archive, so we have to build a directory containing exactly what we want to ship
- This pretty much mandates the use of individual cp commands, unfortunately.
- Since there's a rule in the makefile that tells how this directory should be created, make runs the commands for this rule before running the commands for the current rule
- The make utility runs rules to build dependencies recursively until the requested target's commands can be run

Forcing a rule to run

There's a subtle flaw in the $(distdir) target that may not be obvious, but it will rear its ugly head at the worst times

- If the archive directory already exists when you type make dist, then make won't try to create it
- Try this:

```
$ mkdir jupiter-1.0
$ make dist
tar chof - jupiter-1.0 | gzip -9 -c >jupiter-1.0...
rm -rf jupiter-1.0 &> /dev/null
```

- Notice that the dist target didn't copy any files - it just built an archive out of the existing jupiter-1.0 directory, which was empty
    - Our end-users would have gotten a real surpise when they unpacked this tarball!
    - The problem is that the $(distdir) target is a real target with no dependencies, which means that make will consider it up-to-date as long as it exists in the file system
    - We could add $(distdir) to the .PHONY rule, but this would be a lie - it's not a phony target, it's just that we want to force it to be rebuilt every time
    - The proper way to ensure it gets rebuilt is to have it not exist before make attempts to build it
        - A common method for accomplishing this task to to create a true phony target that will run every time, and add it to the dependency chain at or above the $(distdir) target
        - For obvious reasons, a commonly used name for this sort of target is "FORCE":

Update Top-level Makefile

```
package = jupiter
version = 1.0
tarname = $(package)
distdir = $(tarname)-$(version)

all clean jupiter:
        $(MAKE) -C src $@

dist: $(distdir).tar.gz

$(distdir).tar.gz: FORCE $(distdir)
        tar chof - $(distdir) |\
          gzip -9 -c >$(distdir).tar.gz
        rm -rf $(distdir)

$(distdir):
        mkdir -p $(distdir)/src
        cp Makefile $(distdir)
        cp src/Makefile $(distdir)/src
        cp src/main.c $(distdir)/src

FORCE:
        -rm $(distdir).tar.gz &> /dev/null
        -rm -rf $(distdir) &> /dev/null

.PHONY: FORCE all clean dist
```

The FORCE rule's commands are executed every time because FORCE is a phony target

- By making FORCE a dependency of the tarball, we're given the opportunity to delete any previously created files and directories before make begins to evaluate whether or not these targets' commands should be executed
- This is really much cleaner, because we can now remove the "pre-cleanup" commands from all of the rules, except for FORCE, where they really belong

There are actually more accurate ways of doing this

- we could make the $(distdir) target dependent on all of the files in the archive directory
- If any of these files are newer than the directory, the target would be executed
- This scheme would require an elaborate shell script containing sed commands or non-portable GNU make functions to replace file paths in the dependency list for the copy commands
- For our purposes, this implementation is adequate
- Perhaps it would be worth the effort if our project were huge, and creating an archive directory required copying and/or generating thousands of files

The use of a leading DASH character on some of the rm commands is interesting

- A leading DASH character tells make to not care about the status code of the associated command
- Normally make will stop execution with an error message on the first command that returns a non-zero status code to the shell
- I use a leading DASH character on the rm commands in the FORCE rule because I want to delete previously created product files that may or may not exist, and rm will return an error if I attempt to delete a non-existent file
- Note that I explicitly did NOT use a leading DASH on the rm command in the $(distdir) rule
- This is because this rm command must succeed, or something is very wrong, as the preceeding command should have created a tarball from this directory

Another such leading character that you may encounter is the ATSIGN (@) character

- A command prefixed with an ATSIGN tells make not to print the command as it executes it
- Normally make will print each command as it's executed
- A leading ATSIGN tells make that you don't want to see this command
- This is a common thing to do on echo statements
    - you don't want make to print echo statements, because then your message will be printed twice, and that's just ugly

Automatically testing a distribution
There is a way of performing a sort of self-check on the dist target

- We can create yet another phony target called "distcheck" that does exactly what our users will do
    - unpack the tarball, and build the project
- We can do this in a new temporary directory
- If the build process fails, then the distcheck target will break, telling us that we forgot something crucial in our distribution

Updated portion of the top-level Makefile

```
...
distcheck: $(distdir).tar.gz
        gzip -cd $+ | tar xvf -
        $(MAKE) -C $(distdir) all clean
        rm -rf $(distdir)
        @echo "*** Package $(distdir).tar.gz\
          ready for distribution."
...
.PHONY: FORCE all clean dist distcheck
```

Here, we've added the distcheck target to the top-level makefile

- Since the distcheck target depends on the tarball itself, it will first build a tarball using the same targets used by the dist target
- It will then execute the distcheck commands, which are to unpack the tarball it just built and run "make all clean" on the resulting directory
- This will build both the all and clean targets, successively
- If that process succeeds, it will print out a message, telling us that we can sleep well knowing that our users will probably not have a problem with this tarball

Now all we have to do is remember to run "make distcheck" before we post our tarballs for public distribution!

Unit testing anyone?
Some people think unit testing is evil, but really - -the only honest rationale they can come up with for not doing it is laziness

- Let's face it, proper unit testing is hard work, but it pays off in the end
- Those who do it have learned a lesson (usually as children) about the value of delayed gratification

A good build system is no exception

- It should encorporate proper unit testing
- The commonly used target for testing a build is the check target, so we'll go ahead and add the check target in the usual manner
- The test should probably go in src/Makefile because jupiter is built in src/Makefile, so we'll have to pass the check target down from the top-level makefile

But what commands do we put in the check rule?

- Well, jupiter is a pretty simple program
    - it prints out a message, "Hello from <path>jupiter!"
    - where <path> is variable, depending on the location from which jupiter was executed
- We could check to see that jupiter actually does output such a string
- We'll use the grep utility to test our assertion

Makefile

```
...
all clean check jupiter:
        $(MAKE) -C src $@
```

```
...
.PHONY: FORCE all clean check dist distcheck
```

src/Makefile

```
...
check: all
        ./jupiter | grep "Hello from .*jupiter!"
        @echo "*** ALL TESTS PASSED ***"
...
.PHONY: all clean check
```

Note that check is dependent on all

- We can't really test our products unless they've been built
- We can ensure they're up to date by creating such a dependency
- Now make will run commands for all if it needs to before running the commands for check

There's one more thing we could do to enhance our build system a bit

- We can add the check target to the make command in our distcheck target
- Adding it right between the all and clean targets seems appropriate

Update top-level Makefile

```
...
distcheck: $(distdir).tar.gz
        gzip -cd $+ | tar xvf -
        $(MAKE) -C $(distdir) all check clean
        rm -rf $(distdir)
        @echo "*** Package $(distdir).tar.gz\
          ready for distribution."
...
```

Now, when we run "make distcheck", our entire build system will be tested before packaging is considered successful.

Installing products
We lack one important feature - installation

- In the case of the jupiter project, this is fairly trivial - there's only one executable, and most users could probably guess that this file should be copied into either the /usr/bin or /usr/local/bin directory
- More complex projects, however could cause our users some real consternation when it comes to where to put user and system binaries, libraries, header files, and

documentation, including man pages, info pages, pdf files, and README, INSTALL and COPYRIGHT files
- Do we really want our users to have to figure all that out?

We'll just create an install target that manages putting things where they go, once they're built properly

- Why not just make installation part of the all target?
- A few reasons, really.
    - First, build and installation are separate logical concepts.
        - Remember the rule: Break up large processes into smaller ones and reuse the smaller ones where you can.
    - The second reason is a matter of rights.
        - Users have rights to build in their own home directories, but installation often requires root-level rights to copy files into system directories.
    - Finally, there are several reasons why a user may wish to build, but not install

While creating a distribution package may not be an inherently recursive process, installation certainly is

- so we'll allow each subdirectory in our project to manage installation of its own components
- To do this, we need to modify both makefiles
- The top-level makefile is easy
- Since there are no products to be installed in the top-level directory, we'll just pass on the responsibility to src/Makefile in the usual way

Update top-level Makefile

```
...
all clean check install jupiter:
        $(MAKE) -C src $@
...
.PHONY: FORCE all clean check dist distcheck
.PHONY: install
```

src/Makefile

```
...
install:
        cp jupiter /usr/bin
        chown root:root /usr/bin/jupiter
        chmod +x /usr/bin/jupiter

.PHONY: all clean check install
```

In the top-level makefile

- we've added install to the list of targets passed down to src/Makefile
- In both files we've added install to the phony target list

Now our users can just type the following sequence of commands, and have our project built and installed with the correct system attributes and ownership on their platforms:

```
$ tar -zxvf jupiter-1.0.tar.gz
$ cd jupiter-1.0
$ make all
$ sudo make install
```

Updating the makefile definitions to provide the end-user the choice of changing the location of the installation for final executables and other final files:

Updated top-level Makefile

```
...
export prefix=/usr/local

all clean install jupiter:
        $(MAKE) -C src $@
...
```

src/Makefile

```
...
install:
        install -d $(prefix)/bin
        install -m 0755 jupiter $(prefix)/bin
```

You may have noticed that I've declared and assigned the prefix variable in the top-level makefile, but I've referenced it in src/Makefile

- This is possible because we used the export modifier in the top-level makefile to export this make variable to the shell that make spawns when it executes itself in the src directory
- This is a nice feature of make because it allows us to define all of our user variables in one obvious location--at the top of the top-level makefile
- W've now declared the prefix variable to be /usr/local, which is very nice for those who want jupiter to be installed in /usr/local/bin, but not so nice for those who just want it installed in /usr/bin
- Fortunately, make allows the definition of make variables on the command line, in this manner:

```
$ sudo make prefix=/usr install
```

Variables defined on the command line override those defined in the makefile

- Thus, users who want to install jupiter into their /usr/bin directory now have the option of specifying this on the make command line when they install jupiter

Uninstalling a package
What if a user doesn't like our package after it's been installed, and she just wants to get it off her system?

- This is fairly likely with the jupiter package, as it's rather useless and takes up valuable space in her bin directory
- In the case of your projects however, it's more likely that she wants to install a newer version of your project cleanly, or she wants to change from the test build she downloaded from your website to a professionally packaged version of your project provided by her Linux distribution.
- We really should have an uninstall target, for these and other reasons:

Updated top-level Makefile

```
...
all clean install uninstall jupiter:
        $(MAKE) -C src $@
...
.PHONY: FORCE all clean dist distcheck
.PHONY: install uninstall
```

src/Makefile

```
...
uninstall:
        -rm $(prefix)/bin/jupiter

.PHONY: all clean check install uninstall
```

We now have two places to update when changing our installation processes - the install and uninstall targets

- Unfortunately, this is really about the best we can hope for when writing our own makefiles, without resorting to fairly complex shell script commands
- We'll soon show you how this example can be rewritten in a much simpler way using Automake

Finally, while we're at it, let's add testing the install and uninstall targets to our distcheck target:

Updated top-level Makefile

```
...
distcheck: $(distdir).tar.gz
        gzip -cd $+ | tar xvf -
        $(MAKE) -C $(distdir) all check
        $(MAKE) -C $(distdir) prefix=\
         $${PWD}/$(distdir)/_inst install uninstall
        $(MAKE) -C $(distdir) clean
        rm -rf $(distdir)
        @echo "*** Package $(distdir).tar.gz\
          ready for distribution."
...
```

To do this properly, we had to break up the $(MAKE) commands into three different steps, so that we could add the proper prefix to the install and uninstall targets without affecting the other targets

- We'll have more to say on this topic in a few minutes.
- Note also that I used a double DOLLAR sign on the $${PWD} variable reference
    - This was done in order to ensure that make passed the reference to the shell with the rest of the command line
    - We wanted this variable to be dereferenced by the shell, rather than the make utility
    - Technically, we didn't have to do this because the PWD variable was initialized for make from the environment, but it serves as a good example of this process

An aside - The Filesystem Hierarchy Standard
By the way, where are we getting these directory names from?

- What if some Unix system out there doesn't use /usr or /usr/local?
- Well, in the first place, this is another reason for providing the prefix variable - to handle those sorts of situations
- However, most Unix and Unix-like systems nowadays follow the Filesystem Hierarchy Standard (FHS), as closely as possible
- The FHS defines a number of "standard places", including the following root-level directories:
    - /bin
    - /etc
    - /home
    - /opt
    - /sbin
    - /srv
    - /tmp

    - /usr
    - /var
- This list is not exhaustive
- In addition, the FHS defines several standard locations beneath these root-level directories
    - For instance, the /usr directory should contain the following sub-directories
        - /usr/bin
        - /usr/include
        - /usr/lib
        - /usr/local
        - /usr/sbin
        - /usr/share
        - /usr/src

Supporting standard targets and variables
In addition to those we've already mentioned, the GNU Coding Standards document lists some important targets and variables that you should support in your projects, mainly because everyone else does and your users will expect them

- Some of the chapters in the GNU Coding Standards should be taken with a grain of salt (unless you're actually working on a GNU sponsored project, in which case you're probably not reading this book because you need to)
- Standard targets defined by the GNU Coding Standards document include:
    - all
    - install
    - install-html
    - install-dvi
    - install-pdf
    - install-ps
    - uninstall
    - install-strip
    - clean
    - distclean
    - mostlyclean
    - maintainer-clean
    - TAGS
    - info
    - dvi
    - html
    - pdf
    - ps
    - dist
    - check
    - installcheck
    - installdirs
- Note that you don't need to support all of these targets, but you should consider supporting those which make sense for your project

    - For example, if you build and install HTML pages in your project, then you should probably consider supporting the html and install-html targets.
- Autotools projects support these, and more
    - Some of these are useful to users, while others are only useful to maintainers.

Variables that your project should support (as you see fit) include the following

- We've added the default values for these variables on the right
- You'll note that most of these variables are defined in terms of a few of them, and ultimately only one of them, prefix
- The reason for this is (again) flexibility to the end user. I call these "prefix variables", for lack of a more standard name:

```
prefix          = /usr/local
exec-prefix     = $(prefix)
bindir          = $(exec_prefix)/bin
sbindir         = $(exec_prefix)/sbin
libexecdir      = $(exec_prefix)/libexec
datarootdir     = $(prefix)/share
datadir         = $(datarootdir)
sysconfdir      = $(prefix)/etc
sharedstatedir  = $(prefix)/com
localstatedir   = $(prefix)/var
includedir      = $(prefix)/include
oldincludedir   = /usr/include
docdir          = $(datarootdir)/doc/$(package)
infodir         = $(datarootdir)/info
htmldir         = $(docdir)
dvidir          = $(docdir)
pdfdir          = $(docdir)
psdir           = $(docdir)
libdir          = $(exec_prefix)/lib
lispdir         = $(datarootdir)/emacs/site-lisp
localedir       = $(datarootdir)/locale
mandir          = $(datarootdir)/man
manNdir         = $(mandir)/manN  (N = 1..9)
manext          = .1
manNext         = .N              (N = 1..9)
srcdir          = (compiled project root)
```

To support the variables and targets that we've used so far in the jupiter project, we need to add the bindir variable, in this manner:

Update top-level Makefile

```
...
```

```
export prefix = /usr/local
export exec_prefix = $(prefix)
export bindir = $(exec_prefix)/bin
...
```

src/Makefile

```
...
install:
        install -d $(bindir)
        install -m 0755 jupiter $(bindir)

uninstall:
        -rm $(bindir)/jupiter
...
```

The ability to change these variables like this is particularly useful to a Linux distribution packager, who needs to install packages into very specific system locations:

```
$ make prefix=/usr sysconfdir=/etc install
...
```

Getting your project into a Linux distro
To support staged installation, all you really need to do is provide a variable named "DESTDIR" in your build system that is a sort of super-prefix to all of your installed products

- To show you how this is done, we'll add staged installation support to the jupiter project
- This is so trivial, it only requires three changes to src/Makefile:

src/Makefile

```
...
install:
        install -d $(DESTDIR)$(bindir)
        install -m 0755 jupiter $(DESTDIR)$(bindir)

uninstall:
        -rm $(DESTDIR)$(bindir)/jupiter
...
```

For the sake of symmetry and to be complete, it doesn't hurt to add $(DESTDIR) to uninstall

- Besides, we need it to be complete for the sake of the distcheck target, which we'll now modify to take advantage of our staged installation functionality:

Updated top-level Makefile

```
...
distcheck: $(distdir).tar.gz
        gzip -cd $+ | tar xvf -
        $(MAKE) -C $(distdir) all check
        $(MAKE) -C $(distdir) DESTDIR=\
         $${PWD}/$(distdir)/_inst install uninstall
        $(MAKE) -C $(distdir) clean
        rm -rf $(distdir)
        @echo "*** Package $(distdir).tar.gz\
          ready for distribution."
...
```

Changing the prefix variable to the DESTDIR variable in the second $(MAKE) line above allows us to test a complete install directory hierarchy properly, as we'll see shortly here.

Never recompile from an install target in your makefiles. Otherwise your users won't be able to access your staged installation features when using prefix overrides.

Another reason for this is to allow the user to install into a **grouped location**, and then create **links** to the actual files in the proper locations. Some people like to do this, especially when they are testing out a package, and want to keep track of all of its components.

Standard user variables
The GNU Coding Standards document defines a set of variables that are sort of sacred to the user

- That is, these variables should be used by a GNU build system, but never modified by a GNU build system
- The following list is by no means comprehensive, and ironically, there isn't a comprehensive list to be found in the GCS document
- These are called "user variables", and they include the following for C and C++ programs:

```
CC          - the C compiler
CFLAGS      - C compiler flags
CXX         - the C++ compiler
CXXFLAGS    - C++ compiler flags
LDFLAGS     - linker flags
CPPFLAGS    - C preprocessor flags
...
```

For our purposes, these few are sufficient, but for a more complex makefile, you should become familiar with the larger list so that you can use them as the occasion arises

- To use these in our makefiles, we'll just replace "gcc" with $(CC), and then set CC to the gcc compiler at the top of the makefile
- We'll do the same for CFLAGS and CPPFLAGS, although this last one will contain nothing by default:

src/Makefile

```
...
CC     = gcc
CFLAGS = -g -O2
...
jupiter: main.c
        $(CC) $(CFLAGS) $(CPPFLAGS) -o $@ $+
...
```

The reason this works is that the make utility allows such variables to be overridden by options on the command line:

```
$ make CC=gcc3 CFLAGS='-g -O0' CPPFLAGS=-dtest
```

Remember that environment variables do not automatically override those set in the makefile

- To get the functionality we want, we could use a little GNU make-specific syntax in our makefile:

```
CC     ?= gcc
CFLAGS ?= -g -O2
```

The "?=" operation is a GNU Make-specific operator, which will only set the variable in the makefile if it hasn't already been set elsewhere

- This means we can now override these particular variable settings by setting them in the environment
- But don't forget that this **will only work in** GNU Make.

Configuring your package
The primary tasks of a typical configure script are to:

- generate files from templates containing replacement variables,
- generate a C language header file (often called config.h) for inclusion by project source code,
- set user options for a particular make environment - such as debug flags, etc.,
- set various package options as environment variables,
- and test for the existance of tools, libraries, and header files.

A makefile template contains configuration variables in an easily recognized (and substituted) format

- The configure script replaces these variables with values determined during configuration - either from command line options specified by the user, or from a thorough analysis of the platform environment
- Often this analysis entails such things as checking for the existence of certain system or package include files and libraries, searching various file system paths for required utilities and tools, and even running small programs designed to indicate the feature set of the shell, C compiler, or desired libraries
- The tool of choice here for variable replacement has, in the past, been the sed stream editor
    - A simple sed command can replace all of the configuration variables in a makefile template in a single pass through the file
    - In the latest version of Autoconf (2.62, as of this writing) prefers awk to sed for this process
    - The awk utility is almost as pervasive as sed these days, and it much more powerful with respect to the operations it can perform on a stream of data
    - For the purposes of the jupiter project, either one of these tools would suffice.

Chapter 3:
Configuring your project with Autoconf

Before Automake, Autoconf was used alone, and many legacy open source projects have never really made the transition to the full Autotools suite. As a result, it would not be uncommon to find an open source project containing a file called configure.in (the older naming convention used by Autoconf) and hand-written Makefile.in templates.

Configure scripts, the Autoconf way

Let's spend some time just focusing on the use of Autoconf alone.

- The input to Autoconf is the shell script.
- The macro language used is called M4
    - M4 is a general purpose macro language processor that was originally written by none other than Brian Kernighan and Dennis Ritchie in 1977
    - The "M" means macro; 4 means "4 letters of a-c-r-o"
    - M4 macros are similar in many ways to macros defined in C language source files for the C preprocessor, which is also a text replacement tool.
- The design goals of Autoconf included primarily that it should run on all systems without the addition of complex tool chains and utility sets.
- Autoconf dependencies
    - M4
    - sed
    - awk

    - perl
- The master configuration file, configure.ac, is a shell script filled with M4 syntax
- We'll cover more details of M4 later on in this document

This section builds on the Jupiter project started in Chapter 2.

The smallest configure.ac file

```
AC_INIT([jupiter], [1.0])
AC_OUTPUT
```

Notes:

- `AC_INIT`
    - found in $PREFIX/share/autoconf/autoconf/general.m4
        - where $PREFIX can be either
            - /usr
            - /usr/local
- `AC_OUTPUT`
    - $PREFIX/share/autoconf/autoconf/status.m4
- The square brackets around the parameters are used by Autoconf as a quoting mechanism.
- Use Autoconf quotes ([and]) around every argument to ensure that the expected macro expansions are generated.
- M4 macros may or may not take parameters.
    - When they do, then a set of parentheses must be used when passing the arguments.
- The opening parenthesis must immediately follow the macro name, with no intervening white space.
- M4 has the ability to specify optional parameters, in which case, you may omit the parentheses if you choose not to pass a parameter.
- The result of passing this configure.ac file through Autoconf is essentially the same file (now called configure), only with these two macros fully expanded.
- These two M4 macros expand into a file containing over 2200 lines of Bourne shell script that's over 60K bytes in size! Interestingly, you wouldn't really know this by looking at their definitions.
    - They're both fairly short--only a dozen or two lines each.
    - The reason for this apparent disparity is simple - they're written in a modular fashion, each macro expanding several others, which in turn expand several others, and so on.

Executing Autoconf:

- Just execute `autoconf` in the same directory as your `configure.ac` file.
- However, let's instead use `autoreconf` command.
    - Running `autoreconf` has exactly the same effect as running autoconf, except

that `autoreconf` will also do "the right thing" when you start adding Automake and Libtool functionality to your build system.
    - `autoreconf` is the recommended method for executing the Autotools tool chain, and it's smart enough to only execute the tools that you need, in the order that you need them, and with the options that you need.

```
user@host:~/dev/autotools/chap3$ autoreconf
user@host:~/dev/autotools/chap3$ ls -lp
autom4te.cache/
configure
configure.ac
```

TBD… Work in progress...