GNU make - kickstart

Nicholas Mc Guire

Opentech EDV-Research GmbH

June 11, 2005

Contents

Version	Author	Date	Comment
1.0	Nicholas Mc Guire	16 Feb 2005	First shot

1. Introduction

Make is one of the core tools for the GNU project, it was originally written by Richard Stallman and Roland Mc Garth. The current make development, as of writing is being maintained by Paul D. Smith at `http://savannah.gnu.org/projects/make`. As usual for GNU projects there are mailing lists and repositories in place to help users and developers:

- make webpage: `http://www.gnu.org/software/make/make.html`
- documentation page: `http://www.gnu.org/software/make/manual/make.html`
- source archive: `ftp://www.gnu.org/pub/gnu/make/`
- latest release: `mkae-3.81.tar.gz`
- make user mailing list: `help-make@gnu.org`
- reporting bugs: `bug-make@gnu.org`

This tutorial is about GNU/Make - so it may or may not work with other versions of make - but as GNU/Make runs on about anything that can compile C-code there is no real reason not to use it anyway.

1.1. Core Concepts

Make is a POSIX.2 standard conform tool, some would say the tool, to determin which parts of a larger project need to be processed to build the final output - the goal of the project. To manage building of files that depend on a hierarchy of sources and require specific build instructions for each of the sources, make provides the necessary syntax to describe these hierarchies and dependancies so that changes at one point in the file hierarchy will trigger the update of all dependant parts and yield a build of a complete and consistent goal as if it had been built from scratch. Further it allows to manage the complexity of hierarchical build procedures in a way that allows exchange and modification of parts without uncontrolled side effects.

To achieve this, make provides a syntax for managing targets, prerequisites of these targets, a way of describing the build process via variables and pattern rules and the necessary argument handling to allow conditional builds.

If make is provided with the proper configuration files and the rules are correctly designed working on a multifile project looses its complexity with respect to version dependancies and keeping it consistent - make will take care of that . In this brief introduction to make we will try to outline the most commonly used functionalities of make. This tutorial is not trying to be complete, nor is it trying to replace the excellent documentation provided with make **??**, but it is trying to give a quick intro to make that can be covered in 2 to 3 hours of time - knowing that in industrial environments you simply don't get more time...

How does make descide ?

The two components that make will use to decide what to do are

- The makefile data base
- Built-in knowledge
- Timestamps

So, this tells us one important thing - make sure system times are correct on all of the systems involved in a software project (a common problem !) - other than that there is nothing to be said about the timestamps - the rest of this manual will be about the data base that make uses and how to describe it.

2. Invoking make

2.1. simple invocation

if you simply type in make without any further arguments - make will search for a set of files to be used as database by default:

- GNUMakefile:
- makefile
- Makefile

Not only are these three files probed for, but make actually tries to generate these files using its implicit rules. To see the full list of (almost absurd) attempts to create a valid GNUMakefile, makefile or Makefile, see the output of:

```
make -d
```

in an empty directory (just to make sure that it does not actually manage to find a valid Makefile !).

To see a list of (unfortunately not well documented) implicit rules that were used during the run of `make -d` use:

```
make -p /dev/null
```

If your database is in none of GNUMakefile, Makefile or makefile, then you can pass make the name on the commandline like:

```
make -f whatever.name
```

As a general rule it is though advisable to stick to the default names, with Makefile or makefile being used if you are using standard make capabilities and GNUMakefile if you are using any of the GNU specific extensions.

```
make -k
```

By default make will stop at the first error, this is sometimes not what you want as it may provide insight in the problem if make is allowed to continue even if the build of one part maybe failed. With the `-k` flag make will attempt to continue no matter what (note that you can selectively turn off error handling on a per command basis on make).

```
make -n
```

As with many UNIX commands, there is a "dry-run" option available for make as well. `make -n` will run through the make process but not actually do anything.

2.2. Parallel make

Especially when building large software projects, kdi, linux kernel or glibc, then the time spent waiting for I/O can become quite substantial. So if a build process involves building many file, then these builds can be done in parallel to maximize the throughput. For this make allows to use:

```
make -j #
```

passing it the number of subprocesses it should allow to run in parallel. Typical values on uni-processor systems with slow disks that show a clear improvement will be around 10, on SMP systems around 30. One has to experiment a bit where the maximum lies if one has frequent builds of large packages (you can time your build process with i.e. `time make -j 10 bzImage`).

A further way of building in parallel is to use a parallel make implementation called `pmake`. `pmake` runs on PVM (Parallel Virtual Machine) or Mosix clusters and is able to read `make(1)` complient Makefiles or makefile. To utilize the full powerer of `pmake` you need to write pmakefiles (thats not the topic here though). pmake notes

3. Makefile

A `Makefile` is the primary data base file for the make program. Other files can be referenced in the Makefile, infact other Makefiles - so called sub-make processes, can be invoked from a top-level Makefile.

The content of a Makefile can be categorized by:

- explicid rules - or targets
- comments
- variables
- directives
- implicit rules

3.1. Targets

A Target is one step in the build process for the entire project - one step towards the goal of the make process. A target is described by:

```
target: prerequisites
<TAB    >target body - or rules
```

Note that the tabulator is a requirement as it is used as indicator for the sope of the body.

A target is update if any timestamp of a prerequisite is more recent than the timestamp of the target, unless a target is forced unconditionally (see Target Modifiers below).

The order of targets is simple. The first one found is built unless explicid requirest to build a different one is passed when invoking make. All other targets may be built depending on them being prerequisites to the first target. If a target is neither a prerequisite nor is requested as a command line argument to make, it simply is ignored. So what are prerequisites ?

The order of targets is simple. The first one found is built unless explicid requirest to build a different one is passed when invoking make. All other targets may be built depending on them being prerequisites to the first target. If a target is neither a prerequisite nor is requested as a command line argument to make, it simply is ignored. So what are prerequisites ?

3.1.1. Prerequesites

In a project that consists of multiple source files, not only the final outcome of the build process will be dependant on the content of each file, but also intermediate steps will be dependant on changes in a subset of the files. To describe these dependencies each target has the prerequisite filed allowing to describe these (static) dependencies.

```
hello.o: hello.c hello.h
```

This target to build the hello.o object file, will be called if any of the files hello.c or hello.h was changed - these targets are the prerequisites to build hello.o and they need to be up to date. If the time stamps of the prerequisites is not up to date then the target will be rebuilt. The prerequisites can be

- none
- a file
- a target

In our simple example hello.c and hello.h are most likely simply files, but this need not be the case, hello.h could be a target in the makefile that describes how to build hello.h from some XML file or what ever. Prerequisites are there to describe the hierarchical dependency tree of a target - each target posing one step in the hirarchy. The default behavior is: if there is a target name matching a prerequisite it will be called, otherwise it is assumed to be a file and the existance of the file is checked.

Empty prerequisites are typically used for targets that will be invoked by passing make the target name as an argument. Targets with empty prerequisites are typically used for maintenance purposes. The second case where we encounter empty prerequisites is when we rely on implicit rules - which is a bad idea and should not be done. So for our examples we will always assume that implicit rules are not being used and thus an empty prerequisite really means that there are no prerequisites to this target.

3.1.2. Target modifiers

For now we will ignore the details of the target body and assume that targets have simple prerequisites that are simply files, for the body we assume that it contains a trivial command for its rule, which will be executed if the target needs to be updated. For some targets we want to be able to change the behavior because the default behavior can lead to strange behaviors. Imagin we create a target clean, in which we want to do cleanup work. But if a file called clean exists the target would be considered up to date and make will ignore it.

Write up the following `Makefile`:

```
clean:
        echo "clean was called"
```

```
rtl13:~/ # make
clean was called
rtl13:~/ # touch clean
rlt13:~/ # make
make: ‘clean’ is up to date
rlt13:~/ # make clean
make: ‘clean’ is up to date
```

So the existance of a file with the name of the target is preventing make from processing this target. What we want when putting in adminstrative targets like ”clean” is to have them run on request unconditionally. To achieve that we must tell make to apply the usual rules - we mark it as bying phony.

```
.PHONY: clean
clean:
        echo "clean was called"
```

Now we can run the target independant of a file existing or not. Another use of .PHONY is when we look at submake processes - more on that later.

```
rlt13:~/ # make clean
clean was called
```

3.1.3. Standard Targets (productions)

Targets that don’t just build an individual file are refered to as productions, they commonly have a list of other targets as there prerequisites, or operate on a set of files/directories. There is no hard-coded rule for the names to use, but there are typcially used target names that the open-source developers are used to, and thus expect to be available.

- all
- clean
- dist
- distclean
- allclean
- .

3.2. Comments

Not much to say on comments. A comment is anything from a # to the end of line. So you can put a comment on a line by itself by putting the # as the first character on that line or after some statement.

```
# A comment line

hello: hello.c # a comment after a statement
```

It is legal in make to do:

```
junk:
        echo "junk was called"
# a badly placed comment
        echo " more junk output"
```

the problem with this is that the two echo lines are preceeded by a mandatory <TAB> - which is not mandated for comment lines - but you probably don't want to ask for confusion this way (as forgetting the <TAB> in front of one of the echo commands would case that part of the target body to fail with a `***missing seperator.  Stop.`).

3.3. Variables

Variables in `make` are names representing a text string, there are four types of variables in Make:

- simple expanded or imediate

- recursive expanded or deferred
- automatic
- implicit

The use of these is unfortunately the most common cause for confusion for people stating to use `make` - and it is one of the common causes for ”strange behvior”.

The thing to remember about makes variables is that they are strings - never anything else. A variable that is assigned a value of 1 is assigned the string 1 and not an integer of numeric value 1.

3.3.1. Imediate and Deferred

There are two fundamentaly different types of user assigned variables. Variables that are expanded imediatlely when read and those that are passed to the rule and expanded when processing the rule. why these tow types ? If you think of the semingly trivial problem of:

```
VAR=VALUE1
VAR=VAR VALUE2
```

Tgus wizkd bot work if VAR is not assigned imediately, but if everything must be assigned imediately then:

```
CFLAGS=$(INCLUDE) -O2
INCLUDE=-I/src
```

would not work. To make this work the expansion would need to be deferred to the point in the build process were it is actually being used and expansion would need to take place there - with the obvious problem that the first example would then fail due to an infinite loop!

A second problem is that, especially with large software projects, the performance of deferred expansion is a problem, if you consider a variable that is assigned the output of a command, and every reference to that variable then actually calls the command then performance can be hit severely.

For this reason GNU Make provides the two distinct types (see the examples for a full scale rant on what to do and what not to do).

3.3.2. Modifying Variables

Variables cannot only be defined by putting them in a makefile, but there is a hirarchy of places `make` will search for, to resolve variables. By default the hirarchy is (from lowers to highest):

- Implicid variable
- The environment
- Declaration in the Makefile by assigment
- On the command line
- via override directive in the Makefile

A short session using the following file stored as M2 will show you this:

```
override INCLUDE=-I/src
MYFLAGS=-Os

.PHONY: dummy
dummy:
@echo $(CC) $(MYFLAGS) $(INCLUDE)
```

First we will just make sure that none of these variables are actually present in your current environment:

```
rtl13: ~ # unset CC
rtl13: ~ # unset MYFLAGS
rtl13: ~ # make -f M2
cc -Os -I/src
rtl13: ~ # export CC=gcc
rtl13: ~ # make -f M2
gcc -Os -I/src
rtl13: ~ # make -f M2 CC=powerpc-linux-gcc
powerpc-linux-gcc -Os -I/src
rtl13: ~ # make -f M2 CC=powerpc-linux-gcc MYFLAGS=-O2
powerpc-linux-gcc -O2 -I/src
rtl13: ~ # make -f M2 INCLUDE=/opt
cc -Os -I/src
```

It is fairly easy to end up with a Makefile that works fine on boxA and two years later you must rebuild the project and dig out the old tree, and it does not work any longer because you were relying on some environment variable that was set ! This means that if you use make, then the environment you are working in needs to be stored in your source/doc management system. No need to be complicated, it should be enough to simply:

```
rtl13 ˜ # env > build_env-XCL.rtl13
rtl13 ˜ # cvs add build_env-XCL.rtl13
rtl13 ˜ # cvs commit
```

Not spending these 45 seconds can be quite expensive in the long run.

3.4. Automatic Variables

3.5. Implicid Variables

3.6. Directives

3.7. Implicid rules - Patterm Rules

4. make functions

4.1. Writing your own functions

(see Sequences)

5. External Calls

5.0.1. Executing Commands

ecoing / silent /interrupted

5.0.2. Calling Sub-make

5.0.3. shell invocation

5.0.4. Interactive Makefile

5.1. reporting bugs

References

[GNU] GNU's not UNIX,
`http://www.gnu.org/`, `ftp://ftp.gnu.org/`