WIKIPEDIA

Make (software)

In software development, **Make** is a build automation tool that automatically builds executable programs and libraries from source code by reading files called *Makefiles* which specify how to derive the target program. Though integrated development environments and language-specific compiler features can also be used to manage a build process, Make remains widely used, especially in Unix and Unix-like operating systems.

Besides building programs, Make can be used to manage any project where some files must be updated automatically from others whenever the others change.

Make

Paradigm	macro, declarative
Designed by	Stuart Feldman
First appeared	April 1976
Implementation language	C
OS	Unix-like, Inferno
File formats	Makefile
Major implementations	
BSD, GNU, nmake	
Dialects	
BSD make, GNU make, Microsoft nmake	
Influenced	
Ant, Rake, MSBuild, and *others*	

Contents

Origin

There are now a number of dependency-tracking build utilities, but Make is one of the most widespread, primarily due to its inclusion in Unix, starting with the PWB/UNIX 1.0, which featured a variety of tools targeting software development tasks.[1] It was originally created by Stuart Feldman in April 1976 at Bell Labs.[2][3][1] Feldman received the 2003 ACM Software System Award for the authoring of this widespread tool.[4]

Feldman was inspired to write Make by the experience of a coworker in futilely debugging a program of his where the executable was accidentally not being updated with changes:

> Make originated with a visit from Steve Johnson (author of yacc, etc.), storming into my office, cursing the Fates that had caused him to waste a morning debugging a correct program (bug had been fixed, file hadn't been compiled, `cc *.o` was therefore unaffected). As I had spent a part of the previous evening coping with the same disaster on a project I was working on, the idea of a tool to solve it came up. It began with an elaborate idea of a dependency analyzer, boiled down to something much simpler, and turned into Make that weekend. Use of tools that were still wet was part of the culture. Makefiles were text files, not magically encoded binaries, because that was the Unix ethos: printable, debuggable, understandable stuff.
>
> — Stuart Feldman, *The Art of Unix Programming*, Eric S. Raymond 2003

Before Make's introduction, the Unix build system most commonly consisted of operating system dependent "make" and "install" shell scripts accompanying their program's source. Being able to combine the commands for the different targets into a single file and being able to abstract out dependency tracking and archive handling was an important step in the direction of modern build environments.

Derivatives

Make has gone through a number of rewrites, including a number of from-scratch variants which used the same file format and basic algorithmic principles and also provided a number of their own non-standard enhancements. Some of them are:

- Sun DevPro Make appeared in 1986 with SunOS-3.2. With SunOS-3.2, It was delivered as optional program; with SunOS-4.0, SunPro Make was made the default Make program.[5] In December 2006, Sun DevPro Make was made open source as part of the efforts to open-source Solaris.[6][7]
- dmake or Distributed Make that came with Sun Solaris Studio as its default Make, but not the default one on the Solaris Operating System (SunOS). It was originally required to build OpenOffice, but in 2009[8] the build system was rewritten to use GNU Make. While Apache OpenOffice still contains a mixture of both build systems,[9] the much more actively developed LibreOffice only uses the modernized "gbuild" now.[8]
- BSD Make (*pmake*,[10] *bmake*[11] or *fmake*[12]), which is derived from Adam de Boor's work on a version of Make capable of building targets in parallel, and survives with varying degrees of modification in FreeBSD,[11] NetBSD[13] and OpenBSD.[14] Distinctively, it has conditionals and iterative loops which are applied at the parsing stage and may be used to conditionally and programmatically construct the makefile,[15] including generation of targets at runtime.
- GNU Make (short *gmake*) is the standard implementation of Make for Linux and macOS.[16] It provides several extensions over the original Make, such as conditionals. It also provides many built-in functions which can be used to eliminate the need for shell-scripting in the makefile rules as well as to manipulate the variables set and used in the makefile.[17] For example, the *foreach* function can be used to iterate over a list of values, such as the names of files in a given directory.[18] GNU Make is required for building many software systems, including GCC (since version 3.4[19]), the Linux kernel,[20][21] Apache OpenOffice,[9] LibreOffice,[8] and Mozilla Firefox.[22]
- Rocky Bernstein's Remake[23] is a fork of GNU Make and provides several extensions over GNU Make, such as better location and error-location reporting, execution tracing, execution profiling, and it contains a debugger.
- Glenn Fowler's *nmake*[24] is unrelated to the Microsoft program of the same name. Its input is similar to Make, but not compatible. This program provides shortcuts and built-in features, which according to its developers reduces the size of makefiles by a factor of 10.

- Microsoft *nmake*, a command-line tool which normally is part of Visual Studio.[25] It supports preprocessor directives such as includes and conditional expressions which use variables set on the command-line or within the makefiles.[26][27] Inference rules differ from Make; for example they can include search paths.[28] The Make tool supplied with Embarcadero products has a command-line option that "Causes MAKE to mimic Microsoft's NMAKE."[29]. Qt Project's *Jom* tool is a clone of nmake.[30]
- *Mk* replaced Make in Research Unix, starting from version 9.[31] A redesign of the original tool by Bell Labs programmer Andrew G. Hume, it features a different syntax. Mk became the standard build tool in Plan 9, Bell Labs' intended successor to Unix.[32]
- *Kati* is Google's replacement of GNU Make, used in Android OS builds. It translates the makefile into ninja for faster incremental builds.[33]

POSIX includes standardization of the basic features and operation of the Make utility, and is implemented with varying degrees of completeness in Unix-based versions of Make. In general, simple makefiles may be used between various versions of Make with reasonable success. GNU Make, Makepp and some versions of BSD Make default to looking first for files named "GNUmakefile",[34] "Makeppfile"[35] and "BSDmakefile"[36] respectively, which allows one to put makefiles which use implementation-defined behavior in separate locations.

Behavior

Make is typically used to build executable programs and libraries from source code. Generally though, Make is applicable to any process that involves executing arbitrary commands to transform a source file to a target result. For example, Make could be used to detect a change made to an image file (the source) and the transformation actions might be to convert the file to some specific format, copy the result into a content management system, and then send e-mail to a predefined set of users indicating that the above actions were performed.

Make is invoked with a list of target file names to build as command-line arguments:

```
make [TARGET ...]
```

Without arguments, Make builds the first target that appears in its makefile, which is traditionally a symbolic "phony" target named *all*.

Make decides whether a target needs to be regenerated by comparing file modification times.[37] This solves the problem of avoiding the building of files which are already up to date, but it fails when a file changes but its modification time stays in the past. Such changes could be caused by restoring an older version of a source file, or when a network filesystem is a source of files and its clock or time zone is not synchronized with the machine running Make. The user must handle this situation by forcing a complete build. Conversely, if a source file's modification time is in the future, it triggers unnecessary rebuilding, which may inconvenience users.

Makefiles are traditionally used for compiling code (*.c, *.cc, *.C, etc.), but they can also be used for providing commands to automate common tasks. One such makefile is called from the command line:

```
make                        # Without argument runs first TARGET
make help                   # Show available TARGETS
make dist                   # Make a release archive from current dir
```

Makefile

Make searches the current directory for the makefile to use, e.g. GNU Make searches files in order for a file named one of `GNUmakefile`, `makefile`, or `Makefile` and then runs the specified (or default) target(s) from (only) that file.

The makefile language is similar to declarative programming.[38][39][40] This class of language, in which necessary end conditions are described but the order in which actions are to be taken is not important, is sometimes confusing to programmers used to imperative programming.

One problem in build automation is the tailoring of a build process to a given platform. For instance, the compiler used on one platform might not accept the same options as the one used on another. This is not well handled by Make. This problem is typically handled by generating platform-specific build instructions, which in turn are processed by Make. Common tools for this process are Autoconf, CMake or GYP (or more advanced NG).

Makefiles may contain five kinds of things:[41]

1. An *explicit rule* says when and how to remake one or more files, called the rule's targets. It lists the other files that the targets depend on, called the prerequisites of the target, and may also give a recipe to use to create or update the targets.
2. An *implicit rule* says when and how to remake a class of files based on their names. It describes how a target may depend on a file with a name similar to the target and gives a recipe to create or update such a target.
3. A *variable definition* is a line that specifies a text string value for a variable that can be substituted into the text later.
4. A *directive* is an instruction for make to do something special while reading the makefile such as reading another makefile.
5. Lines starting with `#` are used for comments.

Rules

A makefile consists of *rules*. Each rule begins with a textual *dependency line* which defines a target followed by a colon (:) and optionally an enumeration of components (files or other targets) on which the target depends. The dependency line is arranged so that the target (left hand of the colon) depends on components (right hand of the colon). It is common to refer to components as prerequisites of the target.[42]

```
target [target ...]: [component ...]
Tab ↹[command 1]
          .
          .
          .
Tab ↹[command n]
```

Usually each rule has a single unique target, rather than multiple targets.

For example, a C .o object file is created from .c files, so .c files come first (i.e. specific object file target depends on a C source file and header files). Because Make itself does not understand, recognize or distinguish different kinds of files, this opens up a possibility for human error. A forgotten or an extra dependency may not be immediately obvious and may result in subtle bugs in the generated software. It is possible to write makefiles which generate these dependencies by calling third-party tools, and some makefile generators, such as the Automake toolchain provided by the GNU Project, can do so automatically.

Each dependency line may be followed by a series of TAB indented command lines which define how to transform the components (usually source files) into the target (usually the "output"). If any of the prerequisites has a more recent modification time than the target, the command lines are run. The GNU Make documentation refers to the commands associated with a rule as a "recipe".

The first command may appear on the same line after the prerequisites, separated by a semicolon,

```
targets : prerequisites ; command
```

for example,

```
hello: ; @echo "hello"
```

Make can decide where to start through topological sorting.

Each command line must begin with a tab character to be recognized as a command. The tab is a whitespace character, but the space character does not have the same special meaning. This is problematic, since there may be no visual difference between a tab and a series of space characters. This aspect of the syntax of makefiles is often subject to criticism; it has been described by Eric S. Raymond as "one of the worst design botches in the history of Unix"[43] and *The Unix-Haters Handbook* said "using tabs as part of the syntax is like one of those pungee stick traps in *The Green Berets*". Feldman explains the choice as caused by a workaround for an early implementation difficulty preserved by a desire for backward compatibility with the very first users:

> Why the tab in column 1? Yacc was new, Lex was brand new. I hadn't tried either, so I figured this would be a good excuse to learn. After getting myself snarled up with my first stab at Lex, I just did something simple with the pattern newline-tab. It worked, it stayed. And then a few weeks later I had a user population of about a dozen, most of them friends, and I didn't want to screw up my embedded base. The rest, sadly, is history.
>
> — Stuart Feldman[43]

However, the GNU Make since version 3.82 allows to choose any symbol (one character) as the recipe prefix using the .RECIPEPREFIX special variable, for example:

```
.RECIPEPREFIX := :
all:
:@echo "recipe prefix symbol is set to '$(.RECIPEPREFIX)'"
```

Each command is executed by a separate shell or command-line interpreter instance. Since operating systems use different command-line interpreters this can lead to unportable makefiles. For instance, GNU Make (all POSIX Makes) by default executes commands with /bin/sh, where Unix commands like cp are normally used. In contrast to that, Microsoft's *nmake* executes commands with cmd.exe where batch commands like copy are available but not necessarily cp.

A rule may have no command lines defined. The dependency line can consist solely of components that refer to targets, for example:

```
realclean: clean distclean
```

The command lines of a rule are usually arranged so that they generate the target. An example: if `file.html` is newer, it is converted to text. The contents of the makefile:

```
file.txt: file.html
    lynx -dump file.html > file.txt
```

The above rule would be triggered when Make updates "file.txt". In the following invocation, Make would typically use this rule to update the "file.txt" target if "file.html" were newer.

```
make file.txt
```

Command lines can have one or more of the following three prefixes:

- a hyphen-minus (-), specifying that errors are ignored
- an at sign (@), specifying that the command is not printed to standard output before it is executed
- a plus sign (+), the command is executed even if Make is invoked in a "do not execute" mode

Ignoring errors and silencing echo can alternatively be obtained via the special targets `.IGNORE` and `.SILENT`.[44]

Microsoft's NMAKE has predefined rules that can be omitted from these makefiles, e.g. `c.obj $(CC)$(CFLAGS)`.

Macros

A makefile can contain definitions of macros. Macros are usually referred to as *variables* when they hold simple string definitions, like `{{{1}}}`. Macros in makefiles may be overridden in the command-line arguments passed to the Make utility. Environment variables are also available as macros.

Macros allow users to specify the programs invoked and other custom behavior during the build process. For example, the macro `CC` is frequently used in makefiles to refer to the location of a C compiler, and the user may wish to specify a particular compiler to use.

New macros (or simple "variables") are traditionally defined using capital letters:

```
MACRO = definition
```

A macro is used by expanding it. Traditionally this is done by enclosing its name inside `$()`. (Omitting the parentheses leads to Make interpreting the next letter after the `$` as the entire variable name.) An equivalent form uses curly braces rather than parentheses, i.e. `${}`, which is the style used in the BSDs.

```
NEW_MACRO = $(MACRO)-$(MACRO2)
```

Macros can be composed of shell commands by using the command substitution operator, denoted by backticks (`` ` ``).

```
YYYYMMDD  = ` date `
```

The content of the definition is stored "as is". Lazy evaluation is used, meaning that macros are normally expanded only when their expansions are actually required, such as when used in the command lines of a rule. An extended example:

```
PACKAGE   = package
VERSION   = ` date +"%Y.%m%d" `
ARCHIVE   = $(PACKAGE)-$(VERSION)

dist:
    #  Notice that only now macros are expanded for shell to interpret:
    #    tar -cf package-`date +"%Y%m%d"`.tar

    tar -cf $(ARCHIVE).tar .
```

The generic syntax for overriding macros on the command line is:

```
make MACRO="value" [MACRO="value" ...] TARGET [TARGET ...]
```

Makefiles can access any of a number of predefined *internal macros*, with `?` and `@` being the most common.

```
target: component1 component2
    # contains those components which need attention (i.e. they ARE YOUNGER than current
TARGET).
    echo $?
    # evaluates to current TARGET name from among those left of the colon.
    echo $@
```

A somewhat common syntax expansion is the use of `+=`, `?=`, and `!=` instead of the equal sign. It works on BSD and GNU makes alike.[45]

Suffix rules

Suffix rules have "targets" with names in the form `.FROM.TO` and are used to launch actions based on file extension. In the command lines of suffix rules, POSIX specifies[46] that the internal macro `$<` refers to the first prerequisite and `$@` refers to the target. In this example, which converts any HTML file into text, the shell redirection token `>` is part of the command line whereas `$<` is a macro referring to the HTML file:

```
.SUFFIXES: .txt .html

# From .html to .txt
.html.txt:
    lynx -dump $<   >   $@
```

When called from the command line, the above example expands.

```
$ make -n file.txt
lynx -dump file.html > file.txt
```

Pattern rules

Suffix rules cannot have any prerequisites of their own.[47] If they have any, they are treated as normal files with unusual names, not as suffix rules. GNU Make supports suffix rules for compatibility with old makefiles but otherwise encourages usage of *pattern rules*.[48]

A pattern rule looks like an ordinary rule, except that its target contains exactly one `%` character within the string. The target is considered a pattern for matching file names: the `%` can match any substring of zero or more characters,[49] while other characters match only themselves. The prerequisites likewise use `%` to show how their names relate to the target name.

The above example of a suffix rule would look like the following pattern rule:

```
# From %.html to %.txt
%.txt : %.html
	lynx -dump $< > $@
```

Other elements

Single-line comments are started with the hash symbol (#).

Some directives in makefiles can include other makefiles.

Line continuation is indicated with a backslash `\` character at the end of a line.

```
   target: component \
           component
   Tab ↹command ;          \
   Tab ↹command |          \
   Tab ↹piped-command
```

Example makefiles

The makefile:

```
PACKAGE   = package
VERSION   = ` date "+%Y.%m%d%" `
RELEASE_DIR  = ..
RELEASE_FILE = $(PACKAGE)-$(VERSION)

# Notice that the variable LOGNAME comes from the environment in
# POSIX shells.
#
# target: all - Default target. Does nothing.
all:
	echo "Hello $(LOGNAME), nothing to do by default"
		# sometimes: echo "Hello ${LOGNAME}, nothing to do by default"
	echo "Try 'make help'"

# target: help - Display callable targets.
help:
	egrep "^# target:" [Mm]akefile

# target: list - List source files
list:
	# Won't work. Each command is in separate shell
	cd src
	ls

	# Correct, continuation of the same shell
	cd src; \
    ls

# target: dist - Make a release.
dist:
    tar -cf  $(RELEASE_DIR)/$(RELEASE_FILE) && \
    gzip -9  $(RELEASE_DIR)/$(RELEASE_FILE).tar
```

Below is a very simple makefile that by default (the "all" rule is listed first) compiles a source file called "helloworld.c" using the system's C compiler and also provides a "clean" target to remove the generated files if the user desires to start over. The `$@` and `$<` are two of the so-called internal macros (also known as automatic variables) and stand for the target name and "implicit" source, respectively. In the example below, `$^` expands to a space delimited list of the prerequisites. There are a number of other internal macros.[46][50]

```
CFLAGS ?= -g

all: helloworld

helloworld: helloworld.o
    # Commands start with TAB not spaces
    $(CC) $(LDFLAGS) -o $@ $^

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

clean: FRC
    $(RM) helloworld helloworld.o

# This pseudo target causes all targets that depend on FRC
# to be remade even in case a file with the name of the target exists.
# This works with any make implementation under the assumption that
# there is no file FRC in the current directory.
FRC:
```

Many systems come with predefined Make rules and macros to specify common tasks such as compilation based on file suffix. This lets users omit the actual (often unportable) instructions of how to generate the target from the source(s). On such a system the above makefile could be modified as follows:

```
all: helloworld

helloworld: helloworld.o
    $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $^

clean: FRC
    $(RM) helloworld helloworld.o

# This is an explicit suffix rule. It may be omitted on systems
# that handle simple rules like this automatically.
.c.o:
    $(CC) $(CFLAGS) -c $<

FRC:
.SUFFIXES: .c
```

That "helloworld.o" depends on "helloworld.c" is now automatically handled by Make. In such a simple example as the one illustrated here this hardly matters, but the real power of suffix rules becomes evident when the number of source files in a software project starts to grow. One only has to write a rule for the linking step and declare the object files as prerequisites. Make will then implicitly determine how to make all the object files and look for changes in all the source files.

Simple suffix rules work well as long as the source files do not depend on each other and on other files such as header files. Another route to simplify the build process is to use so-called pattern matching rules that can be combined with compiler-assisted dependency generation. As a final example requiring the gcc compiler and GNU Make, here is a generic makefile that compiles all C files in a folder to the corresponding object files and then links them to the final executable. Before compilation takes place, dependencies are gathered in makefile-friendly format into a hidden file ".depend" that is then included to the makefile. Portable programs ought to avoid constructs used below.

```
# Generic GNUMakefile

# Just a snippet to stop executing under other make(1) commands
# that won't understand these lines
ifneq (,)
This makefile requires GNU Make.
endif

PROGRAM = foo
C_FILES := $(wildcard *.c)
OBJS := $(patsubst %.c, %.o, $(C_FILES))
CC = cc
CFLAGS = -Wall -pedantic
LDFLAGS =
LDLIBS = -lm

all: $(PROGRAM)

$(PROGRAM): .depend $(OBJS)
	$(CC) $(CFLAGS) $(OBJS) $(LDFLAGS) -o $(PROGRAM) $(LDLIBS)

depend: .depend

.depend: cmd = gcc -MM -MF depend $(var); cat depend >> .depend;
.depend:
	@echo "Generating dependencies..."
	@$(foreach var, $(C_FILES), $(cmd))
	@rm -f depend

-include .depend

# These are the pattern matching rules. In addition to the automatic
# variables used here, the variable $* that matches whatever % stands for
# can be useful in special cases.
%.o: %.c
	$(CC) $(CFLAGS) -c $< -o $@

%: %.o
	$(CC) $(CFLAGS) -o $@ $<

clean:
	rm -f .depend $(OBJS)

.PHONY: clean depend
```

See also

- List of build automation software
- Dependency graph

References

1. Thompson, T. J. (November 1980). "Designer's Workbench: Providing a Production Environment". *Bell System Technical Journal*. **59** (9): 1811–1825. doi:10.1002/j.1538-7305.1980.tb03063.x (https://doi.org/10.1002%2Fj.1538-7305.1980.tb03063.x). S2CID 27213583 (https://api.semanticscholar.org/CorpusID:27213583). "In the general

maintenance of DWB, we have used the Source Code Control System and make utility provided by the PWB/UNIX* interactive operating system."

2. "V7/usr/src/cmd/make/ident.c" (https://archive.today/20130901165315/http://minnie.tuhs.org/cgi-bin/utree.pl?file=V7%2Fusr%2Fsrc%2Fcmd%2Fmake%2Fident.c). *tuhs.org*. 1 September 2013. Archived from the original (http://minnie.tuhs.org/cgi-bin/utree.pl?file=V7%2Fusr%2Fsrc%2Fcmd%2Fmake%2Fident.c) on 1 September 2013. Retrieved 18 March 2018.
3. Feldman, S. I. (April 1979). "Make --- A Program for Maintaining Computer Programs". *Software: Practice and Experience*. **9** (4): 255–265. CiteSeerX 10.1.1.39.7058 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.7058). doi:10.1002/spe.4380090402 (https://doi.org/10.1002%2Fspe.4380090402).
4. Matthew Doar (2005). *Practical Development Environments*. O'Reilly Media. p. 94. ISBN 978-0-596-00796-6.
5. "Google Groups" (http://arquivo.pt/wayback/20110122130054/https://groups.google.com/forum/). *arquivo.pt*. Archived from the original on 22 January 2011. Retrieved 18 March 2018.
6. "OpenSolaris at Two (Jim Grisanzio)" (https://web.archive.org/web/20131212131557/https://blogs.oracle.com/jimgris/entry/opensolaris_at_two). 12 December 2013. Archived from the original (https://blogs.oracle.com/jimgris/entry/opensolaris_at_two) on 12 December 2013. Retrieved 18 March 2018.
7. Grisanzio, Jim. The OpenSolaris Story (http://www.guug.de/veranstaltungen/osdevcon2007/slides/grisanzio-opensolaris-story-guug.pdf).
8. "Development/Gbuild - The Document Foundation Wiki" (https://wiki.documentfoundation.org/Development/Gbuild). *wiki.documentfoundation.org*. Retrieved 18 March 2018.
9. "Apache OpenOffice Building Guide - Apache OpenOffice Wiki" (https://wiki.openoffice.org/wiki/Documentation/Building_Guide_AOO#Make_Systems_Used_by_Apache_OpenOffice). *wiki.openoffice.org*. Retrieved 18 March 2018.
10. *FreeBSD 2.0.5 Make Source Code* (https://svnweb.freebsd.org/base/stable/2.0.5/usr.bin/make/make.h?revision=8869&view=markup), 1993
11. https://www.freebsd.org/cgi/man.cgi?query=bmake&sektion=1
12. https://manpages.debian.org/jessie/freebsd-buildutils/fmake.1
13. "make" (https://netbsd.gw.com/cgi-bin/man-cgi?make++NetBSD-current). *NetBSD Manual Pages*. Retrieved 9 July 2020.
14. "make(1) - OpenBSD manual pages" (https://man.openbsd.org/make.1#HISTORY). *man.openbsd.org*. Retrieved 18 March 2018.
15. "make" (https://www.freebsd.org/cgi/man.cgi?query=make&manpath=FreeBSD+12.1-RELEASE+and+Ports#INCLUDE%09STATEMENTS,_CONDITIONALS_AND_FOR_LOOPS). *FreeBSD*. Retrieved 9 July 2020. "Makefile inclusion, conditional structures and for loops reminiscent of the C programming language are provided in make."
16. Arnold Robbins (2005), *Unix in a Nutshell, Fourth Edition* (http://www.linuxdevcenter.com/pub/a/linux/excerpts/9780596100292/gnu-make-utility.html), O'Reilly
17. "8. Functions for Transforming Text" (https://www.gnu.org/software/make/manual/html_node/Functions.html), *GNU make*, Free Software Foundation, 2013
18. "8.5 The foreach Function" (https://www.gnu.org/software/make/manual/html_node/Foreach-Function.html#Foreach-Function), *GNU make*, Free Software Foundation, 2013
19. "GCC 3.4 Release Series Changes, New Features, and Fixes" (https://www.gnu.org/software/gcc/gcc-3.4/changes.html). Free Software Foundation. 2006.
20. Javier Martinez Canillas (December 26, 2012). "Kbuild: the Linux Kernel Build System" (http://www.linuxjournal.com/content/kbuild-linux-kernel-build-system). *Linux Journal*.
21. Greg Kroah-Hartman (2006), *Linux Kernel in a Nutshell* (http://www.linuxtopia.org/online_books/linux_kernel/kernel_configuration/ch03.html), O'Reilly

22. "Build Instructions" (https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Build_Instructions).
23. Rocky Bernstein. "Remake – GNU Make with comprehensible tracing and a debugger" (http://bashdb.sourceforge.net/remake/).
24. Glenn Fowler (January 4, 2012). "nmake Overview" (https://web.archive.org/web/20150902010526/http://www2.research.att.com/~astopen/nmake/nmake.html). Information and Software Systems Research, AT&T Labs Research. Archived from the original (http://www2.research.att.com/~astopen/nmake/nmake.html) on September 2, 2015. Retrieved May 26, 2014.
25. "NMAKE Reference Visual Studio 2015" (http://msdn.microsoft.com/en-us/library/dd9y37ha.aspx). Microsoft. 2015.
26. "Makefile Preprocessing Directives" (http://msdn.microsoft.com/en-us/library/7y32zxwh.aspx). 2014.
27. "Makefile Preprocessing Operators" (http://msdn.microsoft.com/en-us/library/8t2e8d78.aspx). Microsoft. 2014.
28. "Search Paths in Rules" (http://msdn.microsoft.com/en-us/library/hk9ztb8x.aspx). Microsoft. 2014.
29. "MAKE" (http://docs.embarcadero.com/products/rad_studio/radstudio2007/RS2007_helpupdates/HUpdate4/EN/html/devwin32/make_xml.html). CodeGear(TM). 2008.
30. "Jom - Qt Wiki" (https://wiki.qt.io/Jom). Qt Project. 2021.
31. McIlroy, M. D. (1987). *A Research Unix reader: annotated excerpts from the Programmer's Manual, 1971–1986* (http://www.9grid.fr/www.9grid.fr/misc/unix-reader.pdf) (PDF) (Technical report). Bell Labs. CSTR 139.
32. Hume, Andrew G.; Flandrena, Bob (2002). "Maintaining files on Plan 9 with Mk" (http://9p.io/sys/doc/mk.html). *Plan 9 Programmer's Manual*. AT&T Bell Laboratories. Archived (https://web.archive.org/web/20150711105955/http://9p.io/sys/doc/mk.html) from the original on July 11, 2015.
33. "google/kati: An experimental GNU make clone" (https://github.com/google/kati). *GitHub*. 30 November 2020.
34. "GNU 'make' " (https://www.gnu.org/software/make/manual/make.html#Makefile-Names). Free Software Foundation.
35. "Makepp" (http://makepp.sourceforge.net/2.0/makepp_command.html#f_makefile).
36. "Free BSD make" (https://www.freebsd.org/cgi/man.cgi?query=make&apropos=0&sektion=0&manpath=FreeBSD+6.2-RELEASE&format=html#DESCRIPTION).
37. How to sort Linux ls command file output (http://alvinalexander.com/linux/unix-linux-ls-command-file-sort-sorting) Archived (https://web.archive.org/web/20160913020855/http://alvinalexander.com/linux/unix-linux-ls-command-file-sort-sorting) September 13, 2016, at the Wayback Machine
38. an overview on dsls (http://phoenix.labri.fr/wiki/doku.php?id=an_overview_on_dsls) Archived (https://web.archive.org/web/20071023021126/http://phoenix.labri.fr/wiki/doku.php?id=an_overview_on_dsls) October 23, 2007, at the Wayback Machine, 2007/02/27, phoenix wiki
39. Re: Choreography and REST (http://lists.w3.org/Archives/Public/www-ws-arch/2002Aug/0105.html) Archived (https://web.archive.org/web/20160912144047/http://lists.w3.org/Archives/Public/www-ws-arch/2002Aug/0105.html) September 12, 2016, at the Wayback Machine, from Christopher B Ferris on 2002-08-09
40. Target Junior Makefiles (http://www.robots.ox.ac.uk/~tgtjr/makefiles.html) Archived (https://web.archive.org/web/20100107082837/http://www.robots.ox.ac.uk/~tgtjr/makefiles.html) January 7, 2010, at the Wayback Machine, Andrew W. Fitzgibbon and William A. Hoffman
41. 3.1 What Makefiles Contain (https://www.gnu.org/software/make/manual/html_node/Makefile-Contents.html), *GNU make*, Free Software Foundation
42. "Prerequisite Types (GNU make)" (https://www.gnu.org/software/make/manual/html_node/Prerequisite-Types.html). *GNU.org*. GNU Project. Retrieved 15 December 2020.

43. "Chapter 15. Tools: make: Automating Your Recipes", *The Art of Unix Programming*, Eric S. Raymond 2003
44. `make` (https://www.opengroup.org/onlinepubs/9699919799/utilities/make.html) – Commands & Utilities Reference, The Single UNIX Specification, Issue 7 from The Open Group
45. `make(1)` (https://www.freebsd.org/cgi/man.cgi?query=make&sektion=1) – FreeBSD General Commands Manual
46. "make" (http://www.opengroup.org/onlinepubs/009695399/utilities/make.html#tag_04_84_13_07). *www.opengroup.org*. Retrieved 18 March 2018.
47. "GNU make manual: suffix rules" (https://www.gnu.org/software/make/manual/html_node/Suffix-Rules.html#Suffix-Rules). Free Software Foundation.
48. "GNU make manual: pattern rules" (https://www.gnu.org/software/make/manual/html_node/Pattern-Rules.html#Pattern-Rules). Free Software Foundation.
49. See section *Pattern Matching Rules* in the SunPro man page (http://www.lehman.cuny.edu/cgi-bin/man-cgi?make+1) Archived (https://web.archive.org/web/20140529103302/http://www.lehman.cuny.edu/cgi-bin/man-cgi?make%201) May 29, 2014, at the Wayback Machine
50. Automatic Variables (https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html#Automatic-Variables) Archived (https://web.archive.org/web/20160425225736/https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html#Automatic-Variables) April 25, 2016, at the Wayback Machine GNU `make'

External links

- GNU Make homepage (https://www.gnu.org/software/make/)
- Practical Makefiles, by Example (http://nuclear.mutantstargoat.com/articles/make/)
- Writing and Debugging a Makefile (https://web.archive.org/web/20160319035817/https://www.freebsd.org/doc/en/books/pmake/writeanddebug.html)
- "Ask Mr. Make" series of article about GNU Make (https://web.archive.org/web/20160323113355/https://www.cmcrossroads.com/keyword/type/3169)
- *Managing Projects with GNU make -- 3.xth edition* (https://web.archive.org/web/20161001175323/http://www.wanderinghorse.net/computing/make/)
- *What is wrong with make?* (https://web.archive.org/web/20160730053857/http://freecode.com/articles/what-is-wrong-with-make)
- *What’s Wrong With GNU make?* (https://web.archive.org/web/20160813235106/http://www.conifersystems.com/whitepapers/gnu-make/)
- *Recursive Make Considered Harmful* (https://web.archive.org/web/20150330111905/http://miller.emu.id.au/pmiller/books/rmch/)
- Advanced Auto-Dependency Generation (https://web.archive.org/web/20161006202049/http://make.mad-scientist.net/papers/advanced-auto-dependency-generation/).
- *Using NMake* (https://web.archive.org/web/20160424041207/http://c2.com/cgi/wiki?UsingNmake)
- *Make7 - A portable open source make utility* (http://seed7.sourceforge.net/scrshots/make7.htm) written in Seed7
- *Microsoft's NMAKE predefined rules* (https://web.archive.org/web/20130815213042/http://msdn.microsoft.com/en-us/library/cx06ysxh.aspx).