I'm not a bot

MAKE(1) User Commands MAKE(1) make - GNU make utility to maintain groups of programs make [OPTION]... [TARGET]... The make utility will determine automatically which pieces of a large program need to be recompiled, and issue the commands to recompile them. The manual describes the GNU implementation of make, which was written by Richard Stallman and Roland McGrath, and is currently maintained by Paul Smith. Our examples show C programs, since they are very common, but you can use make with any programming language whose compiler can be run with a shell command. In fact, make is not limited to programs. You can use it to describe any task where some files must be updated automatically from others whenever the others change. To prepare to use make, you must write a file called the makefile that describes the relationships among files in your program, and provides commands for updating each file. In a program, typically the executable file is updated from object files, which are in turn made by compiling source files. Once a suitable makefile exists, each time you change some source files, this simple shell command: make suffices to perform all necessary recompilations. The make program uses the makefile description and the last-modification times of the files to decide which of the files need to be updated. For each of those files, it issues the commands recorded in the makefile. make executes commands in the makefile to update one or more targets, where target is typically a program. If no -f option is present, make will look for the makefiles GNUmakefile, makefile, and Makefile, in that order. Normally you should call your makefile either makefile or Makefile. (We recommend Makefile because it appears prominently near the beginning of a directory listing, right near other important files such as README.) The first name checked, GNUmakefile, is not recommended for most makefiles. You should use this name if you have a makefile that is specific to GNU make, and will not be understood by other versions of make. If makefile is '-', the standard input is read. make updates a target if it depends on prerequisite files that have been modified since the target was last modified, or if the target does not exist. -b, -m These options are ignored for compatibility with other versions of make. -B, --always-make Unconditionally make all targets. -C dir, --directory=dir Change to directory dir before reading the makefiles or doing anything else. If multiple -C options are specified, each is interpreted relative to the previous one: -C / -C etc is equivalent to -C /etc. This is typically used with recursive invocations of make. -d Print debugging information in addition to normal processing. The debugging information says which files are being considered for remaking, which file-times are being compared and with what results, which files actually need to be remade, which implicit rules are considered and which are applied---everything interesting about how make decides what to do. --debug[=FLAGS] Print debugging information in addition to normal processing. If the FLAGS are omitted, then the behavior is the same as if -d was specified. FLAGS may be any or all of the following names, comma- or space-separated. Only the first character is significant: the rest may be omitted: all for all debugging output (same as using -d), basic for basic debugging, verbose for more verbose basic debugging, implicit for showing implicit rule search operations, jobs for details on invocation of commands, makefile for debugging while remaking makefiles, print shows all recipes that are run even if they are silent, and why shows the reason make decided to rebuild each target. Use none to disable all previous debugging flags. -e, --environment-overrides Give variables taken from the environment precedence over variables from makefiles. -E string, --eval string Interpret string using the eval function, before parsing any makefiles. -f file, --file=file, --makefile=FILE Use file as a makefile. -i, --ignore-errors Ignore all errors in commands executed to remake files. -I dir, --include-dir=dir Specifies a directory dir to search for included makefiles. If several -I options are used to specify several directories, the directories are searched in the order specified. Unlike the arguments to other flags of make, directories given with -I flags may come directly after the flag: -Idir is allowed, as well as -I dir. This syntax is allowed for compatibility with the C preprocessor's -I flag. -j [jobs], --jobs[=jobs] Specifies the number of jobs (commands) to run simultaneously. If there is more than one -j option, the last one is effective. If the -j option is given without an argument, make will not limit the number of jobs that can run simultaneously. --jobserver-style=style The style of jobserver to use. The style may be one of fifo, pipe, or sem (Windows only). -k, --keep-going Continue as much as possible after an error. While the target that failed, and those that depend on it, cannot be remade, the other dependencies of these targets can be processed all the same. -l [load], --load-average[=load] Specifies that no new jobs (commands) should be started if there are others jobs running and the load average is at least load (a floating-point number). With no argument, removes a previous load limit. -L, --check-symlink-times Use the latest mtime between symlinks and target. -n, --just-print, --dry-run, --recon Print the commands that would be executed, but do not execute them (except in certain circumstances). -o file, --old-file=file, --assume-old=file Do not remake the file file even if it is older than its dependencies, and do not remake anything on account of changes in file. Essentially the file is treated as very old and its rules are ignored. -O[type], --output-sync[=type] When running multiple jobs in parallel with -j, ensure the output of each job is collected together rather than interspersed with output from other jobs. If type is not specified or is target the output from the entire recipe for each target is grouped together. If type is line the output from each command line within a recipe is grouped together. If type is recurse output from an entire recursive make is grouped together. If type is none output synchronization is disabled. -p, --print-data-base Print the data base (rules and variable values) that results from reading the makefiles; then execute as usual or as otherwise specified. This also prints the version information given by the -v switch (see below). To print the data base without trying to remake any files, use make -p -f/dev/null. -q, --question ``Question mode''. Do not run any commands, or print anything; just return an exit status that is zero if the specified targets are already up to date, nonzero otherwise. -r, --no-builtin-rules Eliminate use of the built-in implicit rules. Also clear out the default list of suffixes for suffix rules. -R, --no-builtin-variables Don't define any built-in variables. -s, --silent, --quiet Silent operation; do not print the commands as they are executed. --no-silent Cancel the effect of the -s option. -S, --no-keep-going, --stop Cancel the effect of the -k option. -t, --touch Touch files (mark them up to date without really changing them) instead of running their commands. This is used to pretend that the commands were done, in order to fool future invocations of make. --trace Information about the disposition of each target is printed (why the target is being rebuilt and what commands are run to rebuild it). -v, --version Print the version of the make program plus a copyright, a list of authors and a notice that there is no warranty. -w, --print-directory Print a message containing the working directory before and after other processing. This may be useful for tracking down errors from complicated nests of recursive make commands. --no-print-directory Turn off -w, even if it was turned on implicitly. --shuffle[=MODE] Enable shuffling of goal and prerequisite ordering. MODE is one of none to disable shuffle mode, random to shuffle prerequisites in random order, reverse to consider prerequisites in reverse order, or an integer which enables random mode with a specific seed value. If MODE is omitted the default is random. -W file, --what-if=file, --new-file=file, --assume-new=file Pretend that the target file has just been modified. When used with the -n flag, this shows you what would happen if you were to modify that file. Without -n, it is almost the same as running a touch command on the given file before running make, except that the modification time is changed only in the imagination of make. --warn-undefined-variables Warn when an undefined variable is referenced. GNU make exits with a status of zero if all makefiles were successfully parsed and no targets that were built failed. A status of one will be returned if the -q flag was used and make determines that a target needs to be rebuilt. A status of two will be returned if any errors were encountered. The full documentation for make is maintained as a Texinfo manual. If the info and make programs are properly installed at your site, the command info make should give you access to the complete manual. See the chapter ``Problems and Bugs'' in The GNU Make Manual. This manual page contributed by Dennis Morse of Stanford University. Further updates contributed by Mike Frysinger. It has been reworked by Roland McGrath. Maintained by Paul Smith. Copyright 1992-1993, 1996-2022 Free Software Foundation, Inc. This file is part of GNU make. GNU Make is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version. GNU Make is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with this program. If not, see . This page is part of the make (GNU make) project. Information about the project can be found at . If you have a bug report for this manual page, see . This page was obtained from the tarball make-4.4.tar.gz fetched from on 2025-02-02. If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for the page, or you have corrections or improvements to the information in this COLOPHON (which is not part of the original manual page), send a mail to man-pages@man7.org Pages that refer to this page: dh_assistant(1), dpkg-buildflags(1), dpkg-buildpackage(1), pkgdata(1), time(1), adjtime(3), pmdatrace(3), debhelper-compat-upgrade-checklist(7), sched(7), suffixes(7) I built this guide because I could never quite wrap my head around Makefiles. They seemed awash with hidden rules and esoteric symbols, and asking simple questions didnt yield simple answers. To solve this, I sat down for several weekends and read everything I could about Makefiles. I've condensed the most critical knowledge into this guide. Each topic has a brief description and a self contained example that you can run yourself.If you mostly understand Make, consider checking out the Makefile Cookbook, which has a template for medium sized projects with ample comments about what each part of the Makefile is doing.Good luck, and I hope you are able to slay the confusing world of Makefiles!Getting Started Makefiles are used to help decide which parts of a large program need to be recompiled. In the vast majority of cases, C or C++ files are compiled. Other languages typically have their own tools that serve a similar purpose as Make. Make can also be used beyond compilation too, when you need a series of instructions to run depending on what files have changed. This tutorial will focus on the C/C++ compilation use case.Here's an example dependency graph that you might build with Make. If any file's dependencies changes, then the file will get recompiled: What alternatives are there to Make?Popular C/C++ alternative build systems are SCons, CMake, Bazel, and Ninja. Some code editors like Microsoft Visual Studio have their own built in build tools. For Java, there's Ant, Maven, and Gradle. Other languages like Go, Rust, and TypeScript have their own build tools.Interpreted languages like Python, Ruby, and raw Javascript don't require an analogue to Makefiles. The goal of Makefiles is to compile whatever files need to be compiled, based on what files have changed. But when files in interpreted languages change, nothing needs to get recompiled. When the program runs, the most recent version of the file is used.The versions and types of MakeThere are a variety of implementations of Make, but most of this guide will work on whatever version you're using. However, it's specifically written for GNU Make, which is the standard implementation on Linux and MacOS. All the examples work for Make versions 3 and 4, which are nearly equivalent other than some esoteric differences.Running the ExamplesTo run these examples, you'll need a terminal and "make" installed. For each example, put the contents in a file called Makefile, and in that directory run the command make. Let's start with the simplest of Makefiles:hello: echo "Hello, World"Note: Makefiles must be indented using TABs and not spaces or make will fail.Here is the output of running the above example: makeecho "Hello, World"Hello, WorldThat's it! If you're a bit confused, here's a video that goes through these steps, along with describing the basic structure of Makefiles. A Makefile consists of a set of rules. A rule generally looks like this:targets: prerequisitescommandcommandcommandThe targets are file names, separated by spaces. Typically, there is only one per rule.The commands are a series of steps typically used to make the target(s). These need to start with a tab character, not spaces.The prerequisites are also file names, separated by spaces. These files need to exist before the commands for the target are run. These are also called dependenciesThe essence of MakeLet's start with a hello world example:hello: echo "Hello, World" echo "This line will print if the file hello does not exist."There's already a lot to take in here. Let's break it down:We have one target called helloThis target has two commandsThis target has no prerequisitesWe'll then run make hello. As long as the hello file does not exist, the commands will run. If hello does exist, no commands will run.It's important to realize that I'm talking about hello as both a target and a file. That's because the two are directly tied together. Typically, when a target is run (aka when the commands of a target are run), the commands will create a file with the same name as the target. In this case, the hello target does not create the hello file.Let's create a more typical Makefile - one that compiles a single C file. But before we do, make a file called blah.c that has the following contents:int main() { return 0; }Then create the Makefile (called Makefile, as always):blah:cc blah.c -o blahThis time, try simply running make. Since there's no target supplied as an argument to the make command, the first target is run. In this case, there's only one target (blah). The first time you run this, blah will be created. The second time, you'll see make: 'blah' is up to date. That's because the blah file already exists. But there's a problem: if we modify blah.c and then run make, nothing gets recompiled.We solve this by adding a prerequisite:blah: blah.ccc blah.c -o blahWhen we run make again, the following set of steps happens:The first target is selected, because the first target is the default targetThis has a prerequisite of blah.cMake decides if it should run the blah target. It will only run if blah doesn't exist, or blah.c is newer than blahThis last step is critical, and is the essence of make. What it's attempting to do is decide if the prerequisites of blah have changed since blah was last compiled. That is, if blah.c is modified, running make should recompile the file. And conversely, if blah.c has not changed, then it should not be recompiled.To make this happen, it uses the filesystem timestamps as a proxy to determine if something has changed. This is a reasonable heuristic, because file timestamps typically will only change if the files aremodified. But it's important to realize that this isn't always the case. You could, for example, modify a file, and then change the modified timestamp of that file to something old. If you did, Make would incorrectly guess that the file hadn't changed and thus could be ignored.Whew, what a mouthful. Make sure that you understand this. It's the crux of Makefiles, and might take you a few minutes to properly understand. Play around with the above examples or watch the video above if things are still confusing.More quick examplesThe following Makefile ultimately runs all three targets. When you run make in the terminal, it will build a program called blah in a series of steps:Make selects the target blah, because the first target is the default targetblah requires blah.o, so make searches for the blah.o targetblah.o requires blah.c, so make searches for the blah.c targetblah.c has no dependencies, so the echo command is runThe cc -c command is then run, because all of the blah.o dependencies are finishedThe top cc command is run, because all the blah dependencies are finishedThat's it: blah is a compiled c programblah: blah.o cc blah.o -o blah blah.o: blah.c cc -c blah.c -o blah.o blah.c: echo "int main() { return 0; }" > blah.c If you delete blah.c, all three targets will be rerun. If you edit it (and thus change the timestamp to newer than blah.o), the first two targets will run. If you run touch blah.o (and thus change the timestamp to newer than blah), then only the first target will run. If you change nothing, none of the targets will run. Try it out!This next example doesn't do anything new, but is nontheless a good additional example. It will always run both targets, because some_file depends on other_file, which is never created.some_file: other_file echo "This will always run, and runs second" touch some_file other_file: echo "This will always run, and runs first"Make cleanclean is often used as a target that removes the output of other targets, but it is not a special word in Make. You can run make and make clean on this to create and delete some_file.Note that clean is doing two new things here:It's a target that is not first (the default), and not a prerequisite. That means it'll never run unless you explicitly call make cleanIt's not intended to be a filename. If you happen to have a file named clean, this target won't run, which is not what we want. See .PHONY later in this tutorial on how to fix thissome_file: touch some_file clean:rm -f some_fileVariablesVariables can only be strings. You'll typically want to use :=, but = also works. See Variables Pt 2.Here's an example of using variables:files := file1 file2some_file: $(files) echo "Look at this variable: " $(files) touch some_file file1: touch file1file2: touch file2 clean:rm -f file1 file2 some_fileSingle or double quotes have no meaning to Make. They are simply characters that are assigned to the variable. Quotes are useful to shell/bash, though, and you need them in commands like printf. In this example, the two commands behave the same:a := one twob := 'one two' all:printf '$a'printf $bReference variables using either ${} or $()x := dude all: echo $(x)echo ${x}echo $x Targets Making multiple targets and you want all of them to run? Make an all target.Since this is the first rule listed, it will run by default if make is called without specifying a target.all: one two three one: touch onetwo: touch twothree: touch three clean:rm -f one two threeMultiple targets When there are multiple targets for a rule, the commands will be run for each target. $@ is an automatic variable that contains the target name.all: f1.o f2.o f1.o f2.o: echo $@Automatic Variables and Wildcards Both * and % are called wildcards in Make, but they mean entirely different things. * searches your filesystem for matching filenames. I suggest that you always wrap it in the wildcard function, because otherwise you may fall into a common pitfall described below.print: $(wildcard *.c) ls -la $?* may be used in the target, prerequisites, or in the wildcard function.Danger: * may not be directly used in a variable definitionsDanger: When * matches no files, it is left as it is (unless run in the wildcard function)thing_wrong := *.o thing_right := $(wildcard *.o) all: one two three four one: $(thing_wrong) two: *.o three: $(thing_right) four: $(wildcard *.o)% Wildcard% is really useful, but is somewhat confusing because of the variety of situations it can be used in.When used in "matching" mode, it matches one or more characters in a string. This match is called the stem.When used in "replacing" mode, it takes the stem that was matched and replaces that in a string.% is most often used in rule definitions and in some specific functions.See these sections on examples of it being used:Static Pattern RulesPattern RulesString SubstitutionThe vpath DirectiveAutomatic Variables There are many automatic variables, but often only a few show up:hey: one two echo $@ echo $? echo $^ echo $< touch hey one: touch one two: touch two clean:rm -f hey one twoFancy Rules Make loves c compilation. And every time it expresses its love, things get confusing. Perhaps the most confusing part of Make is the magic/automatic rules that are made. Make calls these "implicit" rules. I don't personally agree with this design decision, and I don't recommend using them, but they're often used and are thus useful to know. Here's a list of implicit rules:Compiling a C program: n.o is made automatically from n.c with a command of the form $(CC) -c $(CPPFLAGS) $(CFLAGS) $^ -o $@Compiling a C++ program: n.o is made automatically from n.cc or n.cpp with a command of the form $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $^ -o $@Linking a single object file: n is made automatically from n.o by running the command $(CC) $(LDFLAGS) $^ $(LOADLIBES) $(LDLIBS) -o $@The important variables used by implicit rules are:CC: Program for compiling C programs; default ccCXX: Program for compiling C++ programs; default g++CFLAGS: Extra flags to give to the C compilerCXXFLAGS: Extra flags to give to the C++ compilerCPPFLAGS: Extra flags to give to the C preprocessorLDFLAGS: Extra flags to give to compilers when they are supposed to invoke the linkerLet's see how we can now build a C program without ever explicitly telling Make how to do the compilation:CC = gcc CFLAGS = -g blah: blah.o blah.c: echo "int main() { return 0; }" > blah.c clean:rm -f blah*Static Pattern Rules Static pattern rules are another way to write less in a Makefile. Here's their syntax:targets...: target-pattern: prereq-patterns ... commandsThe essence is that the given target is matched by the target-pattern (via a % wildcard). Whatever was matched is called the stem. The stem is then substituted into the prereq-pattern, to generate the target's prereqs.A typical use case is to compile .c files into .o files. Here's the manual way:objects = foo.o bar.o all.oall: $(objects) $(CC) $^ -o all foo.o: foo.c $(CC) -c foo.c -o foo.o bar.o: bar.c $(CC) -c bar.c -o bar.o all.o: all.c $(CC) -c all.c -o all.o all.c: echo "int main() { return 0; }" > all.c %.c: touch $@ clean:rm -f *.c *.o allHere's the more efficient way, using a static pattern rule:objects = foo.o bar.o all.oall: $(objects) $(CC) $^ -o all $(objects): %.o: %.c $(CC) -c $^ -o $@ all.c: echo "int main() { return 0; }" > all.c %.c: touch $@ clean:rm -f *.c *.o allStatic Pattern Rules and Filter While I introduce the filter function later on, it's common to use in static pattern rules, so I'll mention that here. The filter function can be used in Static pattern rules to match the correct files. In this example, I made up the .raw and .result extensions.obj_files = foo.result bar.o lose.osrc_files = foo.raw bar.c lose.c all: $(obj_files) $(filter %.o,$(obj_files)): %.o: %.c echo "target: $@ prereq: $ foo%: force @$(MAKE) -f Makefile $@force: ; If you say make foo, make will find GNUmakefile,read it, and see that to make foo, it needs to run the recipefrobnicate > foo. If you say make bar, make willfind no way to make bar in GNUmakefile, so it will use therecipe from the pattern rule: make -f Makefile bar. IfMakefile provides a rule for updating bar, makewill apply the rule. And likewise for any other target thatGNUmakefile does not say how to make.The way this works is that the pattern rule has a pattern of just%, so it matches any target whatever. The rule specifies aprerequisite force, to guarantee that the recipe will be run evenif the target file already exists. We give the force target anempty recipe to prevent make from searching for an implicit rule tobuild itotherwise it would apply the same match-anything rule toforce itself and create a prerequisite loop! GNU make does its work in two distinct phases. During thefirst phase it reads all the makefiles, included makefiles, etc. andinternalizes all the variables and their values and implicit andexplicit rules, and builds a dependency graph of all the targets andtheir prerequisites. During the second phase, make uses thisinternalized data to determine which targets need to be updated andrun the recipes necessary to update them.Its important to understand this two-phase approach because it has adirect impact on how variable and function expansion happens; this isoften a source of some confusion when writing makefiles. Below is asummary of the different constructs that can be found in a makefile,and the phase in which expansion happens for each part of theconstruct.We say that expansion is immediate if it happens during thefirst phase: make will expand that part of the construct as themakefile is parsed. We say that expansion is deferred if it isnot immediate. Expansion of a deferred construct part is delayeduntil the expansion is used: either when it is referenced in animmediate context, or when it is needed during the second phase.You may not be familiar with some of these constructs yet. You canreference this section as you become familiar with them, in laterchapters. Variable definitions are parsed as follows:immediate = deferredimmediate ?= deferredimmediate := immediateimmediate ::= immediateimmediate :::= immediate-with-escapeimmediate += deferred or immediateimmediate != immediate define immediate deferredendef define immediate = deferredendef define immediate ?= deferredendef define immediate := immediateendef define immediate ::= immediateendef define immediate :::= immediate-with-escapeendef define immediate += deferred or immediateendef define immediate != immediateendef For the append operator +=, the right-hand side is consideredimmediate if the variable was previously set as a simple variable(:= or ::=), and deferred otherwise.For the immediate-with-escape operator :::=, the value onthe right-hand side is immediately expanded but then escaped (that is,all instances of $ in the result of the expansion are replacedwith $$).For the shell assignment operator !=, the right-hand side isevaluated immediately and handed to the shell. The result is storedin the variable named on the left, and that variable is considered arecursively expanded variable (and will thus be re-evaluated on eachreference). Conditional directives are parsed immediately. This means, forexample, that automatic variables cannot be used in conditionaldirectives, as automatic variables are not set until the recipe forthat rule is invoked. If you need to use automatic variables in aconditional directive you must move the condition into therecipe and use shell conditional syntax instead. A rule is always expanded the same way, regardless of the form:immediate : immediate ; deferred deferred That is, the target and prerequisite sections are expandedimmediately, and the recipe used to build the target is alwaysdeferred. This is true for explicit rules, pattern rules, suffixrules, static pattern rules, and simple prerequisite definitions. GNU make parses makefiles line-by-line. Parsing proceeds usingthe following steps: Read in a full logical line, including backslash-escaped lines(see Splitting Long Lines). Remove comments (see What Makefiles Contain). If the line begins with the recipe prefix character and we are in arule context, add the line to the current recipe and read the nextline (see Recipe Syntax). Expand elements of the line which appear in an immediateexpansion context (see How make Reads aMakefile). Scan the line for a separator character, such as : or =,to determine whether the line is a macro assignment or a rule(see Recipe Syntax). Internalize the resulting operation and read the next line. An important consequence of this is that a macro can expand to anentire rule, if it is one line long. This will work:myrule = target : ; echo built$(myrule) However, this will not work because make does not re-split linesafter it has expanded them:define myruletarget: echo builtendef$(myrule) The above makefile results in the definition of a target targetwith prerequisites echo and built, as if the makefilecontained target: echo built, rather than a rule with a recipe.Newlines still present in a line after expansion is complete areignored as normal whitespace.In order to properly expand a multi-line macro you must use theeval function: this causes the make parser to be run onthe results of the expanded macro (see The eval Function).Previously we learned that GNU make works in two distinctphases: a read-in phase and a target-update phase (see How make Reads a Makefile). GNU Make also hasthe ability to enable a second expansion of the prerequisites(only) for some or all targets defined in the makefile. In order forthis second expansion to occur, the special target.SECONDEXPANSION must be defined before the first prerequisitelist that makes use of this feature.If .SECONDEXPANSION is defined then when GNU make needs to checkthe prerequisites of a target, the prerequisites are expanded a secondtime. In most circumstances this secondary expansion will have no effect,since all variable and function references will have been expanded during theinitial parsing of the makefiles. In order to take advantage of the secondaryexpansion phase of the parser, then, its necessary to escape thevariable or function reference in the makefile. In this case the firstexpansion merely un-escapes the reference but doesnt expand it, and expansionis left to the secondary expansion phase. For example, consider thismakefile:.SECONDEXPANSION:ONEVAR = onefileTWOVAR = twofilemyfile: $(ONEVAR) $$(TWOVAR) After the first expansion phase the prerequisites list of themyfile target will be onefile and $(TWOVAR); thefirst (unescaped) variable reference to ONEVAR is expanded,while the second (escaped) variable reference is simply unescaped,without being recognized as a variable reference. Now during thesecondary expansion the first word is expanded again but since itcontains no variable or function references it remains the valueonefile, while the second word is now a normal reference to thevariable TWOVAR, which is expanded to the value twofile.The final result is that there are two prerequisites, onefileand twofile.Obviously, this is not a very interesting case since the same resultcould more easily have been achieved simply by having both variablesappear, unescaped, in the prerequisites list. One difference becomesapparent if the variables are reset; consider this example:.SECONDEXPANSION:AVAR = toponefile: $(AVAR)twofile: $$(AVAR)AVAR = bottom Here the prerequisite of onefile will be expanded immediately,and resolve to the value top, while the prerequisite oftwofile will not be full expanded until the secondary expansionand yield a value of bottom.This is marginally more exciting, but the true power of this featureonly becomes apparent when you discover that secondary expansionsalways take place within the scope of the automatic variables for thattarget. This means that you can use variables such as $@,$*, etc. during the second expansion and they will have theirexpected values, just as in the recipe. All you have to do is deferthe expansion by escaping the $. Also, secondary expansionoccurs for both explicit and implicit (pattern) rules. Knowing this,the possible uses for this feature increase dramatically. Forexample:.SECONDEXPANSION:main_OBJS := main.o try.o test.olib_OBJS := lib.o api.omain lib: $$($$@_OBJS) Here, after the initial expansion the prerequisites of both themain and lib targets will be $($@_OBJS). Duringthe secondary expansion, the $@ variable is set to the name ofthe target and so the expansion for the main target will yield$(main_OBJS), or main.o try.o test.o, while thesecondary expansion for the lib target will yield$(lib_OBJS), or lib.o api.o.You can also mix in functions here, as long as they are properly escaped:main_SRCS := main.c try.c test.clib_SRCS := lib.c api.c.SECONDEXPANSION:main lib: $$(patsubst %.c,%.o,$$($$@_SRCS)) This version allows users to specify source files rather than objectfiles, but gives the same resulting prerequisites list as the previousexample.Evaluation of automatic variables during the secondary expansionphase, especially of the target name variable $$@, behavessimilarly to evaluation within recipes. However, there are somesubtle differences and corner cases which come into play for thedifferent types of rule definitions that make understands. Thesubtleties of using the different automatic variables are describedbelow. During the secondary expansion of explicit rules, $$@ and$$% evaluate, respectively, to the file name of the target and,when the target is an archive member, the target member name. The$$< variable evaluates to the first prerequisite in the firstrule for this target. $$^ and $$+ evaluate to the listof all prerequisites of rules that have already appeared forthe same target ($$+ with repetitions and $$^without). The following example will help illustrate these behaviors:.SECONDEXPANSION:foo: foo.1 bar.1 $$< $$^ $$+ # line #1foo: foo.2 bar.2 $$< $$^ $$+ # line #2foo: foo.3 bar.3 $$< $$^ $$+ # line #3 In the first prerequisite list, all three variables ($$ foo echo $^ > bar echo $^ > biz During the execution of a grouped targets recipe, the automaticvariable $@ is set to the name of the particular target in thegroup which triggered the rule. Caution must be used if relying onthis variable in the recipe of a grouped target rule.Unlike independent targets, a grouped target rule must includea recipe. However, targets that are members of a grouped target mayalso appear in independent target rule definitions that do not haverecipes.Each target may have only one recipe associated with it. If a groupedtarget appears in either an independent target rule or in anothergrouped target rule with a recipe, you will get a warning and thelatter recipe will replace the former recipe. Additionally the targetwill be removed from the previous group and appear only in the newgroup.If you would like a target to appear in multiple groups, then you mustuse the double-colon grouped target separator, &:: whendeclaring all of the groups containing that target. Groupeddouble-colon targets are each considered independently, and eachgrouped double-colon rules recipe is executed at most once, if atleast one of its multiple targets requires updating. One file can be the target of several rules. All the prerequisitesmentioned in all the rules are merged into one list of prerequisites forthe target. If the target is older than any prerequisite from any rule,the recipe is executed.There can only be one recipe to be executed for a file. If more thanone rule gives a recipe for the same file, make uses the lastone given and prints an error message. (As a special case, if thefiles name begins with a dot, no error message is printed. This oddbehavior is only for compatibility with other implementations ofmake you should avoid using it). Occasionally it isuseful to have the same target invoke multiple recipes which aredefined in different parts of your makefile; you can usedouble-colon rules (see Double-Colon Rules) for this.An extra rule with just prerequisites can be used to give a few extraprerequisites to many files at once. For example, makefiles oftenhave a variable, such as objects, containing a list of all thecompiler output files in the system being made. An easy way to saythat all of them must be recompiled if config.h changes is towrite the following:objects = foo.o bar.ofoo.o : defs.hbar.o : defs.h test.h$(objects) : config.h This could be inserted or taken out without changing the rules that reallyspecify how to make the object files, making it a convenient form to use ifyou wish to add the additional prerequisite intermittently.Another wrinkle is that the additional prerequisites could bespecified with a variable that you set with a command line argument tomake (see Overriding Variables). For example,extradeps=$(objects) : $(extradeps) means that the command make extradeps=foo.h will considerfoo.h as a prerequisite of each object file, but plain makewill not.If none of the explicit rules for a target has a recipe, then makesearches for an applicable implicit rule to find onesee Using Implicit Rules). Static pattern rules are rules which specify multiple targets andconstruct the prerequisite names for each target based on the target name.They are more general than ordinary rules with multiple targets because thetargets do not have to have identical prerequisites. Their prerequisites mustbe analogous, but not necessarily identical.Syntax of Static Pattern RulesStatic Pattern Rules versus Implicit Rules Here is the syntax of a static pattern rule:targets : target-pattern: prereq-patterns recipe The targets list specifies the targets that the rule applies to.The targets can contain wildcard characters, just like the targets ofordinary rules (see Using Wildcard Characters in FileNames).The target-pattern and prereq-patterns say how to compute theprerequisites of each target. Each target is matched against thetarget-pattern to extract a part of the target name, called thestem. This stem is substituted into each of the prereq-patternsto make the prerequisite names (one from each prereq-pattern).Each pattern normally contains the character % just once. When thetarget-pattern matches a target, the % can match any part ofthe target name; this part is called the stem. The rest of thepattern must match exactly. For example, the target foo.o matchesthe pattern %.o, with foo as the stem. The targetsfoo.c and foo.out do not match that pattern.The prerequisite names for each target are made by substituting the stemfor the % in each prerequisite pattern. For example, if oneprerequisite pattern is %.c, then substitution of the stemfoo gives the prerequisite name foo.c. It is legitimateto write a prerequisite pattern that does not contain %; then thisprerequisite is the same for all targets.% characters in pattern rules can be quoted with precedingbackslashes (\). Backslashes that would otherwise quote %characters can be quoted with more backslashes. Backslashes that quote% characters or other backslashes are removed from the patternbefore it is compared to file names or has a stem substituted into it.Backslashes that are not in danger of quoting % characters gounmolested. For example, the pattern the\%weird\\%pattern\\ hasthe%weird\ preceding the operative % character, andpattern\\ following it. The final two backslashes are left alonebecause they cannot affect any % character.Here is an example, which compiles each of foo.o and bar.ofrom the corresponding .c file:objects = foo.o bar.oall: $(objects)$(objects): %.o: %.c $(CC) -c $(CFLAGS) $< -o $@ Here $< is the automatic variable that holds the name of theprerequisite and $@ is the automatic variable that holds the nameof the target; see Automatic Variables.Each target specified must match the target pattern; a warning is issuedfor each target that does not. If you have a list of files, only some ofwhich will match the pattern, you can use the filter function toremove non-matching file names (see Functions for String Substitution and Analysis):files = foo.elc bar.o lose.o$(filter %.o,$(files)): %.o: %.c $(CC) -c $(CFLAGS) $< -o $@$(filter %.elc,$(files)): %.elc: %.el emacs -f batch-byte-compile $< In this example the result of $(filter %.o,$(files)) isbar.o lose.o, and the first static pattern rule causes each ofthese object files to be updated by compiling the corresponding C sourcefile. The result of $(filter%.elc,$(files)) isfoo.elc, so that file is made from foo.el.Another example shows how to use $* in static pattern rules:bigoutput littleoutput : %output : text.g generate text.g -$* > $@ When the generate command is run, $* will expand to thestem, either big or little. A static pattern rule has much in common with an implicit rule defined as apattern rule (see Defining and Redefining Pattern Rules).Both have a pattern for the target and patterns for constructing thenames of prerequisites. The difference is in how make decideswhen the rule applies.An implicit rule can apply to any target that matches its pattern,but it does apply only when the target has no recipe otherwisespecified, and only when the prerequisites can be found. If more than oneimplicit rule appears applicable, only one applies; the choice depends onthe order of rules.By contrast, a static pattern rule applies to the precise list of targetsthat you specify in the rule. It cannot apply to any other target and itinvariably does apply to each of the targets specified. If two conflictingrules apply, and both have recipes, thats an error.The static pattern rule can be better than an implicit rule for thesereasons: You may wish to override the usual implicit rule for a fewfiles whose names cannot be categorized syntactically butcan be given in an explicit list. If you cannot be sure of the precise contents of the directoriesyou are using, you may not be sure which other irrelevant filesmight lead make to use the wrong implicit rule. The choicemight depend on the order in which the implicit rule search is done.With static pattern rules, there is no uncertainty: each rule appliesto precisely the targets specified. Double-colon rules are explicit rules written with ::instead of : after the target names. They are handleddifferently from ordinary rules when the same target appears in morethan one rule. Pattern rules with double-colons have an entirelydifferent meaning (see Match-Anything Pattern Rules).When a target appears in multiple rules, all the rules must be the sametype: all ordinary, or all double-colon. If they are double-colon, eachof them is independent of the others. Each double-colon rules recipeis executed if the target is older than any prerequisites of that rule.If there are no prerequisites for that rule, its recipe is alwaysexecuted (even if the target already exists). This can result inexecuting none, any, or all of the double-colon rules.Double-colon rules with the same target are in fact completely separatefrom one another. Each double-colon rule is processed individually, justas rules with different targets are processed.The double-colon rules for a target are executed in the order they appearin the makefile. However, the cases where double-colon rules really makesense are those where the order of executing the recipes would not matter.Double-colon rules are somewhat obscure and not often very useful; theyprovide a mechanism for cases in which the method used to update a targetdiffers depending on which prerequisite files caused the update, and suchcases are rare.Each double-colon rule should specify a recipe; if it does not, animplicit rule will be used if one applies.See Using Implicit Rules. In the makefile for a program, many of the rules you need to write oftensay only that some object file depends on some headerfile. For example, if main.c uses defs.h via an#include, you would write: You need this rule so that make knows that it must remakemain.o whenever defs.h changes. You can see that for alarge program you would have to write dozens of such rules in yourmakefile. And, you must always be very careful to update the makefileevery time you add or remove an #include.To avoid this hassle, most modern C compilers can write these rules foryou, by looking at the #include lines in the source files.Usually this is done with the -M option to the compiler.For example, the command: generates the output: Thus you no longer have to write all those rules yourself.The compiler will do it for you.Note that such a rule constitutes mentioning main.o in amakefile, so it can never be considered an intermediate file byimplicit rule search. This means that make wont ever removethe file after using it; see Chains of ImplicitRules.With old make programs, it was traditional practice to use thiscompiler feature to generate prerequisites on demand with a command likemake depend. That command would create a file dependcontaining all the automatically-generated prerequisites; then themakefile could use include to read them in (see Including Other Makefiles).In GNU make, the feature of remaking makefiles makes thispractice obsoleteyou need never tell make explicitly toregenerate the prerequisites, because it always regenerates any makefilethat is out of date. See How Makefiles Are Remade.The practice we recommend for automatic prerequisite generation is to haveone makefile corresponding to each source file. For each source filename.c there is a makefile name.d which listswhat files the object file name.o depends on. That wayonly the source files that have changed need to be rescanned to producethe new prerequisites.Here is the pattern rule to generate a file of prerequisites (i.e., a makefile)called name.d from a C source file called name.c:%.d: %.c @set -e; rm -f $@; \ $(CC) -M $(CPPFLAGS) $< > $@.$$$$; \ sed 's,\($*\)\.o[:]*,\1.o $@ : ,g' < $@.$$$$ > $@; \ rm -f $@.$$$$ See Defining and Redefining Pattern Rules, for information on defining pattern rules. The-e flag to the shell causes it to exit immediately if the$(CC) command (or any other command) fails (exits with anonzero status).With the GNU C compiler, you may wish to use the -MM flag insteadof -M. This omits prerequisites on system header files.See Options Controlling the Preprocessor in Using GNU CC, for details.The purpose of the sed command is to translate (for example): into:main.o main.d : main.c defs.hThis makes each .d file depend on all the source and header filesthat the corresponding .o file depends on. make thenknows it must regenerate the prerequisites whenever any of the source orheader files changes.Once youve defined the rule to remake the .d files,you then use the include directive to read them all in.See Including Other Makefiles. For example:sources = foo.c bar.cinclude $(sources:.c=.d) (This example uses a substitution variable reference to translate thelist of source files foo.c bar.c into a list of prerequisitemakefiles, foo.d bar.d. See Substitution References, for fullinformation on substitution references.) Since the .d files aremakefiles like any others, make will remake them as necessarywith no further work from you. See How Makefiles Are Remade.Note that the .d files contain target definitions; you shouldbe sure to place the include directive after the first,default goal in your makefiles or run the risk of having a randomobject file become the default goal.See How make Processes a Makefile. The recipe of a rule consists of one or more shell command lines tobe executed, one at a time, in the order they appear. Typically, theresult of executing these commands is that the target of the rule isbrought up to date.Users use many different shell programs, but recipes in makefiles arealways interpreted by /bin/sh unless the makefile specifiesotherwise. See Recipe Execution. Makefiles have the unusual property that there are really two distinctsyntaxes in one file. Most of the makefile uses make syntax(see Writing Makefiles). However, recipes are meantto be interpreted by the shell and so they are written using shellsyntax. The make program does not try to understand shellsyntax: it performs only a very few specific translations on thecontent of the recipe before handing it to the shell.Each line in the recipe must start with a tab (or the first characterin the value of the .RECIPEPREFIX variable; see Other Special Variables), except that the first recipe line may be attached to thetarget-and-prerequisites line with a semicolon in between. Anyline in the makefile that begins with a

done in the order in which the argumentsappear. When using characters that are special to make as function arguments,you may need to hide them. GNU make doesnt support escapingcharacters with backslashes or other escape sequences; however, becausearguments are split before they are expanded you can hide them by putting theminto
variables.Characters you may need to hide include: Commas Initial whitespace in the first argument Unmatched open parenthesis or brace An open parenthesis or brace if you dont want it to start a matched pair For example, you can define variables comma and space whosevalues are isolated comma and space characters, then substitute
thesevariables where such characters are wanted, like this:comma:= ,empty:=space:= $(empty) $(empty)foo:= a b cbar:= $(subst $(space),$(comma),$(foo))# bar is now a,b,c. Here the subst function replaces each space with a comma, throughthe value of foo, and substitutes the result. Here are some functions that operate on strings:$(subst
from,to,text) Performs a textual replacement on the text text: each occurrenceof from is replaced by to. The result is substituted forthe function call. For example,$(subst ee,EE,feet on the street) produces the value fEEt on the strEEt.$(patsubst pattern,replacement,text) Finds whitespace-separated words in text that matchpattern and replaces them
with replacement. Herepattern may contain a % which acts as a wildcard,matching any number of any characters within a word. Ifreplacement also contains a %, the % is replacedby the text that matched the % in pattern. Words that donot match the pattern are kept without change in the output. Only thefirst % in the pattern and replacement is
treatedthis way; any subsequent % is unchanged.% characters in patsubst function invocations can bequoted with preceding backslashes (\). Backslashes that wouldotherwise quote % characters can be quoted with more backslashes.Backslashes that quote % characters or other backslashes areremoved from the pattern before it is compared file
names or has a stemsubstituted into it. Backslashes that are not in danger of quoting% characters go unmolested. For example, the patternthe\%weird\\%pattern\\ has the%weird\ preceding theoperative % character, and pattern\\ following it. Thefinal two backslashes are left alone because they cannot affect any% character.Whitespace between
words is folded into single space characters;leading and trailing whitespace is discarded.For example,$(patsubst %.c,%.o,x.c.c bar.c) produces the value x.c.o bar.o.Substitution references (see SubstitutionReferences) are a simpler way to get the effect of the patsubstfunction:$(var:pattern=replacement) is equivalent to$(patsubst
pattern,replacement,$(var)) The second shorthand simplifies one of the most common uses ofpatsubst: replacing the suffix at the end of file names.$(var:suffix=replacement) is equivalent to$(patsubst %suffix,%replacement,$(var)) For example, you might have a list of object files:objects = foo.o bar.o baz.o To get the list of corresponding source files,
you could simply write: instead of using the general form:$(patsubst %.o,%.c,$(objects))$(strip string) Removes leading and trailing whitespace from string and replaceseach internal sequence of one or more whitespace characters with asingle space. Thus, $(strip a b c) results in abc.The function strip can be very useful when used in conjunctionwith
conditionals. When comparing something with the empty string using ifeq or ifneq, you usually want a string ofjust whitespace to match the empty string (see Conditional Parts of Makefiles).Thus, the following may fail to have the desired results:.PHONY: allifneq "$(needs_made)" ""all: $(needs_made)elseall:;@echo 'Nothing to make!'endif Replacing
the variable reference $(needs_made) with thefunction call $(strip$(needs_made)) in the ifneqdirective would make it more robust.$(findstring find,in) Searches in for an occurrence of find. If it occurs, thevalue is find; otherwise, the value is empty. You can use thisfunction in a conditional to test for the presence of a specificsubstring in a given string.
Thus, the two examples,$(findstring a,a b c)$(findstring a,b c) produce the values a and (the empty string),respectively. See Conditionals that Test Flags, for a practical application offindstring.$(filter pattern,text)Returns all whitespace-separated words in text that do matchany of the pattern words, removing any words that do notmatch. The patterns
are written using %, just like the patternsused in the patsubst function above.The filter function can be used to separate out different typesof strings (such as file names) in a variable. For example:sources := foo.c bar.c baz.s ugh.hfoo: $(sources) cc $(filter %.c %.s,$(sources)) -o foo says that foo depends of foo.c, bar.c,baz.s and ugh.h but only foo.c,
bar.c andbaz.s should be specified in the command to thecompiler.$(filter-out pattern,text) Returns all whitespace-separated words in text that do notmatch any of the pattern words, removing the words that domatch one or more. This is the exact opposite of the filterfunction.For example, given:objects=main1.o foo.o main2.o bar.omains=main1.o
main2.o the following generates a list which contains all the object files notin mains:$(filter-out $(mains),$(objects))$(sort list)Sorts the words of list in lexical order, removing duplicatewords. The output is a list of words separated by single spaces.Thus, returns the value bar foo lose.Incidentally, since sort removes duplicate words, you can useit for
this purpose even if you dont care about the sort order.$(word n,text) Returns the nth word of text. The legitimate values ofn start from 1. If n is bigger than the number of wordsin text, the value is empty. For example, returns bar.$(wordlist s,e,text) Returns the list of words in text starting with word s andending with word e (inclusive). The
legitimate values of sstart from 1; e may start from 0. If s is bigger than thenumber of words in text, the value is empty. If e isbigger than the number of words in text, words up to the end oftext are returned. If s is greater than e, nothingis returned. For example,$(wordlist 2, 3, foo bar baz) returns bar baz.$(words text) Returns the number of words in
text. Thus, the last word of textis $(word$(wordstext),text).$(firstword names) The argument names is regarded as a series of names, separatedby whitespace. The value is the first name in the series. The restof the names are ignored.For example, produces the result foo. Although $(firstwordtext) is the same as $(word 1,text), thefirstword function is
retained for its simplicity.$(lastword names) The argument names is regarded as a series of names, separatedby whitespace. The value is the last name in the series.For example, produces the result bar. Although $(lastwordtext) is the same as $(word $(words text),text),the lastword function was added for its simplicity and betterperformance. Here is
a realistic example of the use of subst andpatsubst. Suppose that a makefile uses the VPATH variableto specify a list of directories that make should search forprerequisite files(see VPATH Search Path for All Prerequisites).This example shows how totell the C compiler to search for header files in the same list ofdirectories.The value of VPATH is a list
of directories separated by colons,such as src:../headers. First, the subst function is used tochange the colons to spaces: This produces src ../headers. Then patsubst is used to turneach directory name into a -I flag. These can be added to thevalue of the variable CFLAGS, which is passed automatically to the Ccompiler, like this:override CFLAGS +=
$(patsubst %,-I%,$(subst :, ,$(VPATH))) The effect is to append the text -Isrc -I../headers to thepreviously given value of CFLAGS. The override directive isused so that the new value is assigned even if the previous value ofCFLAGS was specified with a command argument (see The override Directive). Several of the built-in expansion functions relate
specifically totaking apart file names or lists of file names.Each of the following functions performs a specific transformation on afile name. The argument of the function is regarded as a series of filenames, separated by whitespace. (Leading and trailing whitespace isignored.) Each file name in the series is transformed in the same way andthe results
are concatenated with single spaces between them.$(dir names) Extracts the directory-part of each file name in names. Thedirectory-part of the file name is everything up through (andincluding) the last slash in it. If the file name contains no slash,the directory part is the string ./. For example, produces the result src/ ./.$(notdir names) Extracts all
but the directory-part of each file name in names.If the file name contains no slash, it is left unchanged. Otherwise,everything through the last slash is removed from it.A file name that ends with a slash becomes an empty string. This isunfortunate, because it means that the result does not always have thesame number of whitespace-separated file
names as the argument had;but we do not see any other valid alternative.For example,$(notdir src/foo.c hacks) produces the result foo.c hacks.$(suffix names) Extracts the suffix of each file name in names. If the file namecontains a period, the suffix is everything starting with the lastperiod. Otherwise, the suffix is the empty string. This
frequentlymeans that the result will be empty when names is not, and ifnames contains multiple file names, the result may contain fewerfile names.For example,$(suffix src/foo.c src-1.0/bar.c hacks) produces the result .c .c.$(basename names) Extracts all but the suffix of each file name in names. If thefile name contains a period, the basename is
everything starting up to(and not including) the last period. Periods in the directory part areignored. If there is no period, the basename is the entire file name.For example,$(basename src/foo.c src-1.0/bar hacks) produces the result src/foo src-1.0/bar hacks.$(addsuffix suffix,names) The argument names is regarded as a series of names, separatedby
whitespace; suffix is used as a unit. The value ofsuffix is appended to the end of each individual name and theresulting larger names are concatenated with single spaces betweenthem. For example, produces the result foo.c bar.c.$(addprefix prefix,names) The argument names is regarded as a series of names, separatedby whitespace; prefix is used
as a unit. The value ofprefix is prepended to the front of each individual name and theresulting larger names are concatenated with single spaces betweenthem. For example,$(addprefix src/,foo bar) produces the result src/foo src/bar.$(join list1,list2) Concatenates the two arguments word by word: the two first words (onefrom each argument)
concatenated form the first word of the result, thetwo second words form the second word of the result, and so on. So thenth word of the result comes from the nth word of eachargument. If one argument has more words that the other, the extrawords are copied unchanged into the result.For example, $(join a b,.c .o) produces a.c b.o.Whitespace
between the words in the lists is not preserved; it isreplaced with a single space.This function can merge the results of the dir andnotdir functions, to produce the original list of files whichwas given to those two functions.$(wildcard pattern) The argument pattern is a file name pattern, typically containingwildcard characters (as in shell file name
patterns). The result ofwildcard is a space-separated list of the names of existing filesthat match the pattern.See Using Wildcard Characters in File Names.$(realpath names) For each file name in names return the canonical absolute name.A canonical name does not contain any . or .. components,nor any repeated path separators (/) or symlinks. In
case of afailure the empty string is returned. Consult the realpath(3)documentation for a list of possible failure causes.$(abspath names) For each file name in names return an absolute name that doesnot contain any . or .. components, nor any repeated pathseparators (/). Note that, in contrast to realpathfunction, abspath does not resolve symlinks
and does not requirethe file names to refer to an existing file or directory. Use thewildcard function to test for existence.There are four functions that provide conditional expansion. A keyaspect of these functions is that not all of the arguments areexpanded initially. Only those arguments which need to be expanded,will be expanded.$(if
condition,then-part[,else-part]) The if function provides support for conditional expansion in afunctional context (as opposed to the GNU make makefileconditionals such as ifeq (see Syntax ofConditionals)).The first argument, condition, first has all preceding andtrailing whitespace stripped, then is expanded. If it expands to anynon-empty string, then
the condition is considered to be true. If itexpands to an empty string, the condition is considered to be false.If the condition is true then the second argument, then-part, isevaluated and this is used as the result of the evaluation of the entireif function.If the condition is false then the third argument, else-part, isevaluated and this is the result of the if
function. If there isno third argument, the if function evaluates to nothing (theempty string).Note that only one of the then-part or the else-part will beevaluated, never both. Thus, either can contain side-effects (such asshell function calls, etc.)$(or condition1[,condition2[,condition3]]) The or function provides a short-circuiting OR operation.Each
argument is expanded, in order. If an argument expands to anon-empty string the processing stops and the result of the expansionis that string. If, after all arguments are expanded, all of them arefalse (empty), then the result of the expansion is the empty string.$(and condition1[,condition2[,condition3]]) The and function provides a short-circuiting
AND operation.Each argument is expanded, in order. If an argument expands to anempty string the processing stops and the result of the expansion isthe empty string. If all arguments expand to a non-empty string thenthe result of the expansion is the expansion of the last argument.$(intcmp lhs,rhs[,lt-part[,eq-part[,gt-part]]]) The intcmp function
provides support for numerical comparison ofintegers. This function has no counterpart among the GNU make makefileconditionals.The left-hand side, lhs, and right-hand side, rhs, are expandedand parsed as integral numbers in base 10. Expansion of the remainingarguments is controlled by how the numerical left-hand side compares to
thenumerical right-hand side.If there are no further arguments, then the function expands to empty if theleft-hand side and right-hand side do not compare equal, or to their numericalvalue if they do compare equal.Else if the left-hand side is strictly less than the right-hand side, theintcmp function evaluates to the expansion of the third argument,lt-
part. If both sides compare equal, then the intcmp functionevaluates to the expansion of the fourth argument, eq-part. If theleft-hand side is strictly greater than the right-hand side, then theintcmp function evaluates to the expansion of the fifth argument,gt-part.If gt-part is missing, it defaults to eq-part. If eq-partis missing, it defaults to the empty
string. Thus both $(intcmp9,7,hello) and $(intcmp 9,7,hello,world,) evaluate to the emptystring, while $(intcmp 9,7,hello,world) (notice the absence of a commaafter world) evaluates to world. The let function provides a means to limit the scope of avariable. The assignment of the named variables in a letexpression is in effect only within the text
provided by thelet expression, and this assignment doesnt impact that namedvariable in any outer scope.Additionally, the let function enables list unpacking byassigning all unassigned values to the last named variable.The syntax of the let function is:$(let var [var ...],[list],text) The first two arguments, var and list, are expanded beforeanything else is
done; note that the last argument, text, isnot expanded at the same time. Next, each word of theexpanded value of list is bound to each of the variable names,var, in turn, with the final variable name being bound to theremainder of the expanded list. In other words, the first wordof list is bound to the first variable var, the secondword to the second
variable var, and so on.If there are more variable names in var than there are words inlist, the remaining var variable names are set to theempty string. If there are fewer vars than words in listthen the last var is set to all remaining words in list.The variables in var are assigned as simply-expanded variablesduring the execution of let. See The Two
Flavorsof Variables.After all variables are thus bound, text is expanded to providethe result of the let function.For example, this macro reverses the order of the words in the listthat it is given as its first argument:reverse = $(let first rest,$1,\ $(if $(rest),$(call reverse,$(rest)))$(first))all: ; @echo $(call reverse,d c b a) will print a b c d. When first
called, let will expand$1 to d c b a. It will then assign first tod and assign rest to c b a. It will then expandthe if-statement, where $(rest) is not empty so we recursivelyinvoke the reverse function with the value of rest whichis now c b a. The recursive invocation of let assignsfirst to c and rest to b a. The recursioncontinues until let is called with just a
single value,a. Here first is a and rest is empty, so wedo not recurse but simply expand $(first) to a andreturn, which adds b, etc.After the reverse call is complete, the first andrest variables are no longer set. If variables by those namesexisted beforehand, they are not affected by the expansion of thereverse macro. The foreach function is similar to the
let function, but verydifferent from other functions. It causes one piece of text to be usedrepeatedly, each time with a different substitution performed on it. Theforeach function resembles the for command in theshell sh and the foreach command in the C-shell csh.The syntax of the foreach function is: The first two arguments, var and list, are
expanded beforeanything else is done; note that the last argument, text, isnot expanded at the same time. Then for each word of the expandedvalue of list, the variable named by the expanded value of varis set to that word, and text is expanded. Presumably textcontains references to that variable, so its expansion will be differenteach time.The result
is that text is expanded as many times as there arewhitespace-separated words in list. The multiple expansions oftext are concatenated, with spaces between them, to make the resultof foreach.This simple example sets the variable files to the list of all filesin the directories in the list dirs:dirs := a b c dfiles := $(foreach dir,$(dirs),$(wildcard $(dir)/*))
Here text is $(wildcard $(dir)/*). The first repetitionfinds the value a for dir, so it produces the same resultas $(wildcard a/*); the second repetition produces the resultof $(wildcard b/*); and the third, that of $(wildcard c/*).This example has the same result (except for setting dirs) asthe following example:files := $(wildcard a/* b/* c/* d/*) When text is
complicated, you can improve readability by giving ita name, with an additional variable:find_files = $(wildcard $(dir)/*)dirs := a b c dfiles := $(foreach dir,$(dirs),$(find_files)) Here we use the variable find_files this way. We use plain =to define a recursively-expanding variable, so that its value contains anactual function call to be re-expanded under
the control of foreach;a simply-expanded variable would not do, since wildcard would becalled only once at the time of defining find_files.Like the let function, the foreach function has no permanenteffect on the variable var; its value and flavor after theforeach function call are the same as they were beforehand. Theother values which are taken from
list are in effect onlytemporarily, during the execution of foreach. The variablevar is a simply-expanded variable during the execution offoreach. If var was undefined before the foreachfunction call, it is undefined after the call.See The Two Flavors of Variables.You must take care when using complex variable expressions that result invariable names
because many strange things are valid variable names, butare probably not what you intended. For example,files := $(foreach Esta-escrito-en-espanol!,b c ch,$(find_files)) might be useful if the value of find_files references the variablewhose name is Esta-escrito-en-espanol! (es un nombre bastante largo,no?), but it is more likely to be a mistake. The
file function allows the makefile to write to or read froma file. Two modes of writing are supported: overwrite, where the textis written to the beginning of the file and any existing content islost, and append, where the text is written to the end of the file,preserving the existing content. In both cases the file is created ifit does not exist. It is a fatal error
if the file cannot be openedfor writing, or if the write operation fails. The filefunction expands to the empty string when writing to a file.When reading from a file, the file function expands to theverbatim contents of the file, except that the final newline (if thereis one) will be stripped. Attempting to read from a non-existent fileexpands to the empty
string.The syntax of the file function is:$(file op filename[,text]) When the file function is evaluated all its arguments areexpanded first, then the file indicated by filename will beopened in the mode described by op.The operator op can be > to indicate the file will beoverwritten with new content, >> to indicate the currentcontents of the file will be
appended to, or < to indicate thecontents of the file will be read in. The filename specifiesthe file to be written to or read from. There may optionally bewhitespace between the operator and the file name.When reading files, it is an error to provide a text value.When writing files, text will be written to the file. Iftext does not already end in a newline a
final newline will bewritten (even if text is the empty string). If the textargument is not given at all, nothing will be written.For example, the file function can be useful if your buildsystem has a limited command line size and your recipe runs a commandthat can accept arguments from a file as well. Many commands use theconvention that an argument
prefixed with an @ specifies afile containing more arguments. Then you might write your recipe inthis way:program: $(OBJECTS) $(file >$@.in,$^) $(CMD) $(CMDFLAGS) @$@.in @rm $@.in If the command required each argument to be on a separate line of theinput file, you might write your recipe like this:program: $(OBJECTS) $(file >$@.in)
$(foreach O,,(file >>$@.in,$O)) $(CMD) $(CMDFLAGS) @$@.in @rm $@.in The call function is unique in that it can be used to create newparameterized functions. You can write a complex expression as thevalue of a variable, then use call to expand it with differentvalues.The syntax of the call function is:$(call variable,param,param,) When make
expands this function, it assigns each param totemporary variables $(1), $(2), etc. The variable$(0) will contain variable. There is no maximum number ofparameter arguments. There is no minimum, either, but it doesnt makesense to use call with no parameters.Then variable is expanded as a make variable in the contextof these temporary
assignments. Thus, any reference to $(1) inthe value of variable will resolve to the first param in theinvocation of call.Note that variable is the name of a variable, not areference to that variable. Therefore you would not normally usea $ or parentheses when writing it. (You can, however, use avariable reference in the name if you want the name not to
be aconstant.)If variable is the name of a built-in function, the built-in functionis always invoked (even if a make variable by that name alsoexists).The call function expands the param arguments beforeassigning them to temporary variables. This means that variablevalues containing references to built-in functions that have specialexpansion rules, like
foreach or if, may not work as youexpect.Some examples may make this clearer.This macro simply reverses its arguments:reverse = $(2) $(1)foo = $(call reverse,a,b) Here foo will contain b a.This one is slightly more interesting: it defines a macro to search forthe first instance of a program in PATH:pathsearch = $(firstword $(wildcard $(addsuffix
/$(1),$(subst :, ,$(PATH)))))LS := $(call pathsearch,ls) Now the variable LS contains /bin/ls or similar.The call function can be nested. Each recursive invocation getsits own local values for $(1), etc. that mask the values ofhigher-level call. For example, here is an implementation of amap function:map = $(foreach a,$(2),$(call $(1),$(a))) Now you can
map a function that normally takes only one argument,such as origin, to multiple values in one step:o = $(call map,origin,o map MAKE) and end up with o containing something like file file default.A final caution: be careful when adding whitespace to the arguments tocall. As with other functions, any whitespace contained in thesecond and subsequent
arguments is kept; this can cause strangeeffects. Its generally safest to remove all extraneous whitespace whenproviding parameters to call. The value function provides a way for you to use the value of avariable without having it expanded. Please note that thisdoes not undo expansions which have already occurred; for example ifyou create a simply
expanded variable its value is expanded during thedefinition; in that case the value function will return thesame result as using the variable directly.The syntax of the value function is: Note that variable is the name of a variable, not areference to that variable. Therefore you would not normallyuse a $ or parentheses when writing it. (You can,
however, usea variable reference in the name if you want the name not to be aconstant.)The result of this function is a string containing the value ofvariable, without any expansion occurring. For example, in thismakefile:FOO = $PATHall: @echo $(FOO) @echo $(value FOO) The first output line would be ATH, since the $P would beexpanded as a
make variable, while the second output line wouldbe the current value of your $PATH environment variable, sincethe value function avoided the expansion.The value function is most often used in conjunction with theeval function (see The eval Function). The eval function is very special: it allows you to define newmakefile constructs that are not
constant; which are the result ofevaluating other variables and functions. The argument to theeval function is expanded, then the results of that expansionare parsed as makefile syntax. The expanded results can define newmake variables, targets, implicit or explicit rules, etc.The result of the eval function is always the empty string;thus, it can be
placed virtually anywhere in a makefile withoutcausing syntax errors.Its important to realize that the eval argument is expandedtwice; first by the eval function, then the results ofthat expansion are expanded again when they are parsed as makefilesyntax. This means you may need to provide extra levels of escapingfor $ characters when using eval.
The valuefunction (see The value Function) can sometimes be useful in thesesituations, to circumvent unwanted expansions.Here is an example of how eval can be used; this examplecombines a number of concepts and other functions. Although it mightseem overly complex to use eval in this example, rather thanjust writing out the rules, consider two
things: first, the templatedefinition (in PROGRAM_template) could need to be much morecomplex than it is here; and second, you might put the complex,generic part of this example into another makefile, then includeit in all the individual makefiles. Now your individual makefiles arequite straightforward.PROGRAMS = server clientserver_OBJS =
server.o server_priv.o server_access.oserver_LIBS = priv protocolclient_OBJS = client.o client_api.o client_mem.oclient_LIBS = protocol# Everything after this is generic.PHONY: allall: $(PROGRAMS)define PROGRAM_template = $(1): $$($(1)_OBJS) $$($(1)_LIBS:%=-l%) ALL_OBJS += $$($(1)_OBJS)endef$(foreach prog,$(PROGRAMS),$(eval $(call
PROGRAM_template,$(prog))))$(PROGRAMS): $(LINK.o) $^ $(LDLIBS) -o $@clean: rm -f $(ALL_OBJS) $(PROGRAMS) The origin function is unlike most other functions in that it doesnot operate on the values of variables; it tells you something abouta variable. Specifically, it tells you where it came from.The syntax of the origin function is: Note that
variable is the name of a variable to inquire about,not a reference to that variable. Therefore you would not normallyuse a $ or parentheses when writing it. (You can, however, use avariable reference in the name if you want the name not to be a constant.)The result of this function is a string telling you how the variablevariable was
defined:undefinedif variable was never defined.defaultif variable has a default definition, as is usual with CCand so on. See Variables Used by Implicit Rules.Note that if you have redefined a default variable, the originfunction will return the origin of the later definition.environmentif variable was inherited from the environment provided
tomake.environment overrideif variable was inherited from the environment provided tomake, and is overriding a setting for variable in themakefile as a result of the -e option (see Summary of Options).fileif variable was defined in a makefile.command lineif variable was defined on the command line.overrideif variable was defined with an override
directive in amakefile (see The override Directive).automaticif variable is an automatic variable defined for the executionof the recipe for each rule (see Automatic Variables). This information is primarily useful (other than for your curiosity) todetermine if you want to believe the value of a variable. For example,suppose you have a makefile foo that
includes another makefilebar. You want a variable bletch to be defined in barif you run the command make-fbar, even if the environment containsa definition of bletch. However, if foo definedbletch before including bar, you do not want to override thatdefinition. This could be done by using an override directive infoo, giving that definition precedence
over the later definition inbar; unfortunately, the override directive would alsooverride any command line definitions. So, bar couldinclude:ifdef bletchifeq "$(origin bletch)" "environment"bletch = barf, gag, etc.endifendif If bletch has been defined from the environment, this will redefineit.If you want to override a previous definition of bletch if it
camefrom the environment, even under -e, you could instead write:ifneq "$(findstring environment,$(origin bletch))" ""bletch = barf, gag, etc.endif Here the redefinition takes place if $(origin bletch) returns eitherenvironment or environment override.See Functions for String Substitution and Analysis. The flavor function, like the origin function, does
notoperate on the values of variables but rather it tells you somethingabout a variable. Specifically, it tells you the flavor of avariable (see The Two Flavors of Variables).The syntax of the flavor function is: Note that variable is the name of a variable to inquire about,not a reference to that variable. Therefore you would not normallyuse a $ or
parentheses when writing it. (You can, however, use avariable reference in the name if you want the name not to be a constant.)The result of this function is a string that identifies the flavor of thevariable variable:undefinedif variable was never defined.recursiveif variable is a recursively expanded variable.simpleif variable is a simply expanded
variable. These functions control the way make runs. Generally, they are used toprovide information to the user of the makefile or to cause make to stopif some sort of environmental error is detected.$(error text) Generates a fatal error where the message is text. Note thatthe error is generated whenever this function is evaluated. So, ifyou put it
inside a recipe or on the right side of a recursivevariable assignment, it wont be evaluated until later. Thetext will be expanded before the error is generated.For example,ifdef ERROR1$(error error is $(ERROR1))endif will generate a fatal error during the read of the makefile if themake variable ERROR1 is defined. Or,ERR = $(error found an
error!).PHONY: errerr: ; $(ERR) will generate a fatal error while make is running, if theerr target is invoked.$(warning text) This function works similarly to the error function, above,except that make doesnt exit. Instead, text is expandedand the resulting message is displayed, but processing of the makefilecontinues.The result of the expansion of this
function is the empty string.$(info text) This function does nothing more than print its (expanded) argument(s)to standard output. No makefile name or line number is added. Theresult of the expansion of this function is the empty string. The shell function is unlike any other function other than thewildcard function(see The Function wildcard) in that
itcommunicates with the world outside of make.The shell function provides for make the same facility thatbackquotes (`) provide in most shells: it does command expansion.This means that it takes as an argument a shell command and expands to theoutput of the command. The only processing make does on the result isto convert each newline (or
carriage-return / newline pair) to a single space.If there is a trailing (carriage-return and) newline it will simply beremoved.The commands run by calls to the shell function are run whenthe function calls are expanded (see How make Readsa Makefile). Because this function involves spawning a new shell, you shouldcarefully consider the performance
implications of using the shellfunction within recursively expanded variables vs. simply expanded variables(see The Two Flavors of Variables).An alternative to the shell function is the != assignmentoperator; it provides a similar behavior but has subtle differences(see Setting Variables). The != assignment operator isincluded in newer POSIX
standards.After the shell function or != assignment operator isused, its exit status is placed in the .SHELLSTATUS variable.Here are some examples of the use of the shell function:contents := $(shell cat foo) sets contents to the contents of the file foo, with a space(rather than a newline) separating each line.files := $(shell echo *.c) sets files to the
expansion of *.c. Unless make isusing a very strange shell, this has the same result as$(wildcard *.c) (as long as at least one .c fileexists).All variables that are marked as export will also be passed to theshell started by the shell function. It is possible to create avariable expansion loop: consider this makefile:export HI = $(shell echo hi)all: ; @echo $$HI
When make wants to run the recipe it must add the variable HI tothe environment; to do so it must be expanded. The value of this variablerequires an invocation of the shell function, and to invoke it we mustcreate its environment. Since HI is exported, we need to expand it tocreate its environment. And so on. In this obscure case make will usethe
value of the variable from the environment provided to make, orelse the empty string if there was none, rather than looping or issuing anerror. This is often what you want; for example:export PATH = $(shell echo /usr/local/bin:$$PATH) However, it would be simpler and more efficient to use a simply-expandedvariable here (:=) in the first place. If
GNU make is built with support for GNU Guile as an embeddedextension language then the guile function will be available.The guile function takes one argument which is first expandedby make in the normal fashion, then passed to the GNU Guileevaluator. The result of the evaluator is converted into a string andused as the expansion of the guile
function in the makefile.See GNU Guile Integration for details onwriting extensions to make in Guile.You can determine whether GNU Guile support is available by checkingthe .FEATURES variable for the word guile. A makefile that says how to recompile a program can be used in morethan one way. The simplest use is to recompile every file that is
outof date. Usually, makefiles are written so that if you runmake with no arguments, it does just that.But you might want to update only some of the files; you might want to usea different compiler or different compiler options; you might want just tofind out which files are out of date without changing them.By giving arguments when you run make,
you can do any of thesethings and many others.The exit status of make is always one of three values:0The exit status is zero if make is successful.2The exit status is two if make encounters any errors.It will print messages describing the particular errors.1The exit status is one if you use the -q flag and makedetermines that some target is not already
up to date.See Instead of Executing Recipes. The way to specify the name of the makefile is with the -f or--file option (--makefile also works). For example,-f altmake says to use the file altmake as the makefile.If you use the -f flag several times and follow each -fwith an argument, all the specified files are used jointly asmakefiles.If you do not use the -f
or --file flag, the default isto try GNUmakefile, makefile, and Makefile, inthat order, and use the first of these three which exists or can be made(see Writing Makefiles). The goals are the targets that make should strive ultimatelyto update. Other targets are updated as well if they appear asprerequisites of goals, or prerequisites of prerequisites of
goals, etc.By default, the goal is the first target in the makefile (not countingtargets that start with a period). Therefore, makefiles are usuallywritten so that the first target is for compiling the entire program orprograms they describe. If the first rule in the makefile has severaltargets, only the first target in the rule becomes the default goal, notthe
whole list. You can manage the selection of the default goal fromwithin your makefile using the .DEFAULT_GOAL variable(see Other Special Variables).You can also specify a different goal or goals with command linearguments to make. Use the name of the goal as an argument.If you specify several goals, make processes each of them inturn, in the
order you name them.Any target in the makefile may be specified as a goal (unless itstarts with - or contains an =, in which case it will beparsed as a switch or variable definition, respectively). Eventargets not in the makefile may be specified, if make can findimplicit rules that say how to make them.Make will set the special variable
MAKECMDGOALS to thelist of goals you specified on the command line. If no goals were givenon the command line, this variable is empty. Note that this variableshould be used only in special circumstances.An example of appropriate use is to avoid including .d filesduring clean rules (see Generating Prerequisites Automatically), somake wont create
them only to immediately remove themagain:sources = foo.c bar.cifeq (,$(filter clean,$(MAKECMDGOALS))include $(sources:.c=.d)endif One use of specifying a goal is if you want to compile only a part ofthe program, or only one of several programs. Specify as a goal eachfile that you wish to remake. For example, consider a directory
containingseveral programs, with a makefile that starts like this:.PHONY: allall: size nm ld ar as If you are working on the program size, you might want to saymakesize so that only the files of that program are recompiled.Another use of specifying a goal is to make files that are not normallymade. For example, there may be a file of debugging output,
or aversion of the program that is compiled specially for testing, which hasa rule in the makefile but is not a prerequisite of the default goal.Another use of specifying a goal is to run the recipe associated witha phony target (see Phony Targets) or empty target (see Empty Target Files to Record Events). Many makefiles containa phony target named
clean which deletes everything except sourcefiles. Naturally, this is done only if you request it explicitly withmakeclean. Following is a list of typical phony and emptytarget names. See Standard Targets for Users, for a detailed list of all thestandard target names which GNU software packages use.all Make all the top-level targets the makefile knows
about.clean Delete all files that are normally created by running make.mostlyclean Like clean, but may refrain from deleting a few files that peoplenormally dont want to recompile. For example, the mostlycleantarget for GCC does not delete libgcc.a, because recompiling itis rarely necessary and takes a lot of time.distclean realcleanclobberAny of
these targets might be defined to delete more files thanclean does. For example, this would delete configuration filesor links that you would normally create as preparation for compilation,even if the makefile itself cannot create these files.install Copy the executable file into a directory that users typically searchfor commands; copy any auxiliary files
that the executable uses intothe directories where it will look for them.print Print listings of the source files that have changed.tar Create a tar file of the source files.shar Create a shell archive (shar file) of the source files.dist Create a distribution file of the source files. This mightbe a tar file, or a shar file, or a compressed version of one of theabove,
or even more than one of the above.TAGS Update a tags table for this program.check testPerform self tests on the program this makefile builds. The makefile tells make how to tell whether a target is up to date,and how to update each target. But updating the targets is not alwayswhat you want. Certain options specify other activities for make.-n --
just-print--dry-run--recon No-op. Causes make to print the recipes that are needed tomake the targets up to date, but not actually execute them. Note thatsome recipes are still executed, even with this flag (see How the MAKE Variable Works). Also any recipesneeded to update included makefiles are still executed(see How Makefiles Are Remade).-t --
touch Touch. Marks targets as up to date without actually changingthem. In other words, make pretends to update the targets butdoes not really change their contents; instead only their modifiedtimes are updated.-q --question Question. Silently check whether the targets are up to date, butdo not execute recipes; the exit code shows whether any
updates areneeded.-W file --what-if=file--assume-new=file--new-file=file What if. Each -W flag is followed by a file name. The givenfiles modification times are recorded by make as being the presenttime, although the actual modification times remain the same.You can use the -W flag in conjunction with the -n flagto see what would happen if you were
to modify specific files. With the -n flag, make prints the recipe that it wouldnormally execute but usually does not execute it.With the -t flag, make ignores the recipes in the rulesand uses (in effect) the command touch for each target that needs tobe remade. The touch command is also printed, unless -s or.SILENT is used. For speed, make does not
actually invokethe program touch. It does the work directly.With the -q flag, make prints nothing and executes norecipes, but the exit status code it returns is zero if and only if thetargets to be considered are already up to date. If the exit status isone, then some updating needs to be done. If make encounters anerror, the exit status is two, so you can
distinguish an error from atarget that is not up to date.It is an error to use more than one of these three flags in the sameinvocation of make.The -n, -t, and -q options do not affect recipelines that begin with + characters or contain the strings$(MAKE) or ${MAKE}. Note that only the line containingthe + character or the strings $(MAKE) or
${MAKE}is run regardless of these options. Other lines in the same rule arenot run unless they too begin with + or contain $(MAKE) or${MAKE} (See How the MAKE Variable Works.)The -t flag prevents phony targets (see Phony Targets) frombeing updated, unless there are recipe lines beginning with +or containing $(MAKE) or ${MAKE}.The -W
flag provides two features: If you also use the -n or -q flag, you can see whatmake would do if you were to modify some files. Without the -n or -q flag, when make is actuallyexecuting recipes, the -W flag can direct make to act asif some files had been modified, without actually running the recipesfor those files. Note that the options -p and -v allow you
to obtain otherinformation about make or about the makefiles in use(see Summary of Options). Sometimes you may have changed a source file but you do not want torecompile all the files that depend on it. For example, suppose you adda macro or a declaration to a header file that many other files dependon. Being conservative, make assumes that
any change in theheader file requires recompilation of all dependent files, but you knowthat they do not need to be recompiled and you would rather not wastethe time waiting for them to compile.If you anticipate the problem before changing the header file, you canuse the -t flag. This flag tells make not to run therecipes in the rules, but rather to
mark the target up to date bychanging its last-modification date. You would follow this procedure: Use the command make to recompile the source files that reallyneed recompilation, ensuring that the object files are up-to-datebefore you begin. Make the changes in the header files. Use the command make -t to mark all the object files asup to date.
The next time you run make, the changes in theheader files will not cause any recompilation. If you have already changed the header file at a time when some filesdo need recompilation, it is too late to do this. Instead, you canuse the -ofile flag, which marks a specified file asold (see Summary of Options). This meansthat the file itself will not be
remade, and nothing else will beremade on its account. Follow this procedure: Recompile the source files that need compilation for reasons independentof the particular header file, with make -o headerfile.If several header files are involved, use a separate -o optionfor each header file. Touch all the object files with make -t. An argument that contains
= specifies the value of a variable:v=x sets the value of the variable v to x.If you specify a value in this way, all ordinary assignments of the samevariable in the makefile are ignored; we say they have beenoverridden by the command line argument.The most common way to use this facility is to pass extra flags tocompilers. For example, in a properly
written makefile, the variableCFLAGS is included in each recipe that runs the C compiler, so afile foo.c would be compiled something like this: Thus, whatever value you set for CFLAGS affects each compilationthat occurs. The makefile probably specifies the usual value forCFLAGS, like this: Each time you run make, you can override this value if
youwish. For example, if you say make CFLAGS='-g -O', each Ccompilation will be done with cc -c -g -O. (This alsoillustrates how you can use quoting in the shell to enclose spaces andother special characters in the value of a variable when you overrideit.)The variable CFLAGS is only one of many standard variables thatexist just so that you can
change them this way. See Variables Used by Implicit Rules, for a complete list.You can also program the makefile to look at additional variables of yourown, giving the user the ability to control other aspects of how themakefile works by changing the variables.When you override a variable with a command line argument, you candefine either a
recursively-expanded variable or a simply-expandedvariable. The examples shown above make a recursively-expandedvariable; to make a simply-expanded variable, write := or::= instead of =. But, unless you want to include avariable reference or function call in the value that youspecify, it makes no difference which kind of variable you create.There
is one way that the makefile can change a variable that you haveoverridden. This is to use the override directive, which is a linethat looks like this: override variable = value(see The override Directive). Normally, when an error happens in executing a shell command, makegives up immediately, returning a nonzero status. No further recipes
areexecuted for any target. The error implies that the goal cannot becorrectly remade, and make reports this as soon as it knows.When you are compiling a program that you have just changed, this is notwhat you want. Instead, you would rather that make try compilingevery file that can be tried, to show you as many compilation errorsas possible.On
these occasions, you should use the -k or--keep-going flag. This tells make to continue toconsider the other prerequisites of the pending targets, remaking themif necessary, before it gives up and returns nonzero status. Forexample, after an error in compiling one object file, make -kwill continue compiling other object files even though it
alreadyknows that linking them will be impossible. In addition to continuingafter failed shell commands, make -k will continue as much aspossible after discovering that it does not know how to make a targetor prerequisite file. This will always cause an error message, butwithout -k, it is a fatal error (see Summary of Options).The usual behavior of
make assumes that your purpose is to get thegoals up to date; once make learns that this is impossible, it mightas well report the failure immediately. The -k flag says that thereal purpose is to test as much as possible of the changes made in theprogram, perhaps to find several independent problems so that you cancorrect them all before the next
attempt to compile. This is why EmacsM-x compile command passes the -k flag by default. In some situations, make will need to create its own temporary files.These files must not be disturbed while make is running, including allrecursively-invoked instances of make.If the environment variable MAKE_TMPDIR is set then all temporary filescreated by
make will be placed there.If MAKE_TMPDIR is not set, then the standard location for temporaryfiles for the current operating system will be used. For POSIX systems thiswill be the location set in the TMPDIR environment variable, or elsethe systems default location (e.g., /tmp) is used. On Windows,first TMP then TEMP will be checked, then
TMPDIR, andfinally the system default temporary file location will be used.Note that this directory must already exist or make will fail:make will not attempt to create it.These variables cannot be set from within a makefile: GNU makemust have access to this location before it begins reading the makefiles. Certain standard ways of remaking target
files are used very often. Forexample, one customary way to make an object file is from a C source fileusing the C compiler, cc.Implicit rules tell make how to use customary techniques sothat you do not have to specify them in detail when you want to usethem. For example, there is an implicit rule for C compilation. Filenames determine which
implicit rules are run. For example, Ccompilation typically takes a .c file and makes a .o file.So make applies the implicit rule for C compilation when it seesthis combination of file name endings.A chain of implicit rules can apply in sequence; for example, makewill remake a .o file from a .y file by way of a .c file.The built-in implicit rules use several
variables in their recipes sothat, by changing the values of the variables, you can change the way theimplicit rule works. For example, the variable CFLAGS controls theflags given to the C compiler by the implicit rule for C compilation.You can define your own implicit rules by writing pattern rules.Suffix rules are a more limited way to define implicit
rules.Pattern rules are more general and clearer, but suffix rules areretained for compatibility. To allow make to find a customary method for updating a targetfile, all you have to do is refrain from specifying recipes yourself.Either write a rule with no recipe, or dont write a rule at all.Then make will figure out which implicit rule to use based onwhich
kind of source file exists or can be made.For example, suppose the makefile looks like this:foo : foo.o bar.o cc -o foo foo.o bar.o $(CFLAGS) $(LDFLAGS) Because you mention foo.o but do not give a rule for it, makewill automatically look for an implicit rule that tells how to update it.This happens whether or not the file foo.o currently exists.If an
implicit rule is found, it can supply both a recipe and one ormore prerequisites (the source files). You would want to write a rulefor foo.o with no recipe if you need to specify additionalprerequisites, such as header files, that the implicit rule cannotsupply.Each implicit rule has a target pattern and prerequisite patterns. There maybe many implicit
rules with the same target pattern. For example, numerousrules make .o files: one, from a .c file with the C compiler;another, from a .p file with the Pascal compiler; and so on. The rulethat actually applies is the one whose prerequisites exist or can be made.So, if you have a file foo.c, make will run the C compiler;otherwise, if you have a file foo.p,
make will run the Pascalcompiler; and so on.Of course, when you write the makefile, you know which implicit rule youwant make to use, and you know it will choose that one because youknow which possible prerequisite files are supposed to exist.See Catalogue of Built-In Rules,for a catalogue of all the predefined implicit rules.Above, we said an
implicit rule applies if the required prerequisites existor can be made. A file can be made if it is mentioned explicitly inthe makefile as a target or a prerequisite, or if an implicit rule can berecursively found for how to make it. When an implicit prerequisite is theresult of another implicit rule, we say that chaining is occurring.See Chains of Implicit
Rules.In general, make searches for an implicit rule for each target, andfor each double-colon rule, that has no recipe. A file that is mentionedonly as a prerequisite is considered a target whose rule specifies nothing,so implicit rule search happens for it. See Implicit Rule Search Algorithm, for thedetails of how the search is done.Note that explicit
prerequisites do not influence implicit rule search.For example, consider this explicit rule: The prerequisite on foo.p does not necessarily mean thatmake will remake foo.o according to the implicit rule tomake an object file, a .o file, from a Pascal source file, a.p file. For example, if foo.c also exists, the implicitrule to make an object file from a C
source file is used instead,because it appears before the Pascal rule in the list of predefinedimplicit rules (see Catalogue of Built-InRules).If you do not want an implicit rule to be used for a target that has norecipe, you can give that target an empty recipe by writing a semicolon(see Defining Empty Recipes). Here is a catalogue of predefined implicit
rules which are alwaysavailable unless the makefile explicitly overrides or cancels them.See Canceling Implicit Rules, for information oncanceling or overriding an implicit rule. The -r or--no-builtin-rules option cancels all predefined rules.This manual only documents the default rules available on POSIX-basedoperating systems. Other operating
systems, such as VMS, Windows,OS/2, etc. may have different sets of default rules. To see the fulllist of default rules and variables available in your version of GNUmake, run make -p in a directory with no makefile.Not all of these rules will always be defined, even when the -roption is not given. Many of the predefined implicit rules areimplemented in
make as suffix rules, so which ones will bedefined depends on the suffix list (the list of prerequisites ofthe special target .SUFFIXES). The default suffix list is:.out, .a, .ln, .o, .c, .cc,.C, .cpp, .p, .f, .F, .m,.r, .y, .l, .ym, .lm, .s,.S, .mod, .sym, .def, .h,.info, .dvi, .tex, .texinfo, .texi,.txinfo, .w, .ch .web, .sh,.elc, .el. All of the implicit rules described belowwhose
prerequisites have one of these suffixes are actually suffixrules. If you modify the suffix list, the only predefined suffixrules in effect will be those named by one or two of the suffixes thatare on the list you specify; rules whose suffixes fail to be on thelist are disabled. See Old-Fashioned Suffix Rules,for full details on suffix rules.Compiling C programs
n.o is made automatically from n.c witha recipe of the form (CC)(CPPFLAGS)$(CFLAGS)-c.Compiling C++ programs n.o is made automatically from n.cc,n.cpp, or n.C with a recipe of the form(CXX)(CPPFLAGS)$(CXXFLAGS)-c. We encourage you to use thesuffix .cc or .cpp for C++ source files instead of .C tobetter support case-insensitive file
systems.Compiling Pascal programs n.o is made automatically from n.pwith the recipe $(PC) $(PFLAGS) -c.Compiling Fortran and Ratfor programs n.o is made automatically from n.r,n.F or n.f by running theFortran compiler. The precise recipe used is as follows:.f$(FC) $(FFLAGS) -c..F$(FC) $(FFLAGS) $(CPPFLAGS) -c..r$(FC) $(FFLAGS) $(RFLAGS)
-c.Preprocessing Fortran and Ratfor programsn.f is made automatically from n.r orn.F. This rule runs just the preprocessor to convert aRatfor or preprocessable Fortran program into a strict Fortranprogram. The precise recipe used is as follows:.F$(FC) $(CPPFLAGS) $(FFLAGS) -F..r$(FC) $(FFLAGS) $(RFLAGS) -F.Compiling Modula-2 programs
n.sym is made from n.def with a recipe of the form$(M2C)$(M2FLAGS)$(DEFFLAGS). n.o is made fromn.mod; the form is: $(M2C)$(M2FLAGS)$(MODFLAGS).Assembling and preprocessing assembler programs n.o is made automatically from n.s byrunning the assembler, as. The precise recipe is$(AS) $(ASFLAGS).n.s is made automatically from n.S
byrunning the C preprocessor, cpp. The precise recipe is(CPP)(CPPFLAGS).Linking a single object file n is made automatically from n.o by running the Ccompiler to link the program. The precise recipe used is (CC)(LDFLAGS)n.o$(LOADLIBES)$(LDLIBS).This rule does the right thing for a simple program with only onesource file. It will also do
the right thing if there are multipleobject files (presumably coming from various other source files), oneof which has a name matching that of the executable file. Thus, when x.c, y.c and z.c all exist will execute:cc -c x.c -o x.occ -c y.c -o y.occ -c z.c -o z.occ x.o y.o z.o -o xrm -f x.orm -f y.orm -f z.o In more complicated cases, such as when there is no
object file whosename derives from the executable file name, you must write an explicitrecipe for linking.Each kind of file automatically made into .o object files willbe automatically linked by using the compiler ($(CC),$(FC) or $(PC); the C compiler $(CC) is used toassemble .s files) without the -c option. This could bedone by using the .o object files as
intermediates, but it isfaster to do the compiling and linking in one step, so thats how itsdone.Yacc for C programs n.c is made automatically from n.y byrunning Yacc with the recipe $(YACC) $(YFLAGS).Lex for C programs n.c is made automatically from n.l byrunning Lex. The actual recipe is $(LEX) $(LFLAGS).Lex for Ratfor programsn.r is made
automatically from n.l byrunning Lex. The actual recipe is $(LEX) $(LFLAGS).The convention of using the same suffix .l for all Lex filesregardless of whether they produce C code or Ratfor code makes itimpossible for make to determine automatically which of the twolanguages you are using in any particular case. If make iscalled upon to remake an
object file from a .l file, it mustguess which compiler to use. It will guess the C compiler, becausethat is more common. If you are using Ratfor, make sure makeknows this by mentioning n.r in the makefile. Or, if youare using Ratfor exclusively, with no C files, remove .c fromthe list of implicit rule suffixes with:.SUFFIXES:.SUFFIXES: .o .r .f .l Making
Lint Libraries from C, Yacc, or Lex programs n.ln is made from n.c by running lint. The precise recipe is $(LINT)$(LINTFLAGS)$(CPPFLAGS)-i.The same recipe is used on the C code produced fromn.y or n.l.TeX and Web n.dvi is made from n.tex with the recipe$(TEX). n.tex is made from n.web with$(WEAVE), or from n.w (and from n.ch ifit exists or
can be made) with $(CWEAVE). n.p ismade from n.web with $(TANGLE) and n.cis made from n.w (and from n.ch if it existsor can be made) with $(CTANGLE).Texinfo and Info n.dvi is made from n.texinfo,n.texi, or n.txinfo, with the recipe$(TEXI2DVI)$(TEXI2DVI_FLAGS). n.info is made fromn.texinfo, n.texi, or n.txinfo, withthe recipe
$(MAKEINFO)$(MAKEINFO_FLAGS).Any file n is extracted if necessary from an RCS filenamed either n,v or RCS/n,v. The preciserecipe used is (CO)(COFLAGS). n will not beextracted from RCS if it already exists, even if the RCS file isnewer. The rules for RCS are terminal(see Match-Anything Pattern Rules),so RCS files cannot be generated from
another source; they mustactually exist.Any file n is extracted if necessary from an SCCS filenamed either s.n or SCCS/s.n. The preciserecipe used is (GET)(GFLAGS). The rules for SCCS areterminal (see Match-Anything Pattern Rules),so SCCS files cannot be generated from another source; they mustactually exist.For the benefit of SCCS, a file n is
copied fromn.sh and made executable (by everyone). This is forshell scripts that are checked into SCCS. Since RCS preserves theexecution permission of a file, you do not need to use this featurewith RCS.We recommend that you avoid using of SCCS. RCS is widely held to besuperior, and is also free. By choosing free software in place ofcomparable
(or inferior) proprietary software, you support the freesoftware movement. Usually, you want to change only the variables listed in the tableabove, which are documented in the following section.However, the recipes in built-in implicit rules actually usevariables such as COMPILE.c, LINK.p, andPREPROCESS.S, whose values contain the recipes listed
above.make follows the convention that the rule to compile a.x source file uses the variable COMPILE.x.Similarly, the rule to produce an executable from a .xfile uses LINK.x; and the rule to preprocess a.x file uses PREPROCESS.x.Every rule that produces an object file uses the variableOUTPUT_OPTION. make defines this variable either tocontain -o
$@, or to be empty, depending on a compile-timeoption. You need the -o option to ensure that the output goesinto the right file when the source file is in a different directory,as when using VPATH (see Searching Directories for Prerequisites). However,compilers on some systems do not accept a -o switch for objectfiles. If you use such a system, and
use VPATH, somecompilations will put their output in the wrong place.A possible workaround for this problem is to give OUTPUT_OPTIONthe value ;mv$*.o$@. The recipes in built-in implicit rules make liberal use of certainpredefined variables. You can alter the values of these variables inthe makefile, with arguments to make, or in the environment
toalter how the implicit rules work without redefining the rulesthemselves. You can cancel all variables used by implicit rules withthe -R or --no-builtin-variables option.For example, the recipe used to compile a C source file actually says$(CC) -c $(CFLAGS) $(CPPFLAGS). The default values of the variablesused are cc and nothing, resulting in the
command cc -c. Byredefining CC to ncc, you could cause ncc to beused for all C compilations performed by the implicit rule. By redefiningCFLAGS to be -g, you could pass the -g option toeach compilation. All implicit rules that do C compilation use$(CC) to get the program name for the compiler and allinclude $(CFLAGS) among the arguments given
to the compiler.The variables used in implicit rules fall into two classes: those that arenames of programs (like CC) and those that contain arguments for theprograms (like CFLAGS). (The name of a program may also containsome command arguments, but it must start with an actual executable programname.) If a variable value contains more than
one argument, separate themwith spaces.The following tables describe of some of the more commonly-used predefinedvariables. This list is not exhaustive, and the default values shown here maynot be what make selects for your environment. To see thecomplete list of predefined variables for your instance of GNU make youcan run make -p in a
directory with no makefiles.Here is a table of some of the more common variables used as names ofprograms in built-in rules:AR Archive-maintaining program; default ar.AS Program for compiling assembly files; default as.CC Program for compiling C programs; default cc.CXX Program for compiling C++ programs; default g++.CPP Program for
running the C preprocessor, with results to standard output;default $(CC) -E.FC Program for compiling or preprocessing Fortran and Ratfor programs;default f77.M2C Program to use to compile Modula-2 source code; default m2c.PC Program for compiling Pascal programs; default pc.CO Program for extracting a file from RCS; default co.GET
Program for extracting a file from SCCS; default get.LEX Program to use to turn Lex grammars into source code; default lex.YACC Program to use to turn Yacc grammars into source code; default yacc.LINT Program to use to run lint on source code; default lint.MAKEINFO Program to convert a Texinfo source file into an Info file; defaultmakeinfo.TEX
Program to make TeX DVI files from TeX source;default tex.TEXI2DVI Program to make TeX DVI files from Texinfo source;default texi2dvi.WEAVE Program to translate Web into TeX; default weave.CWEAVE Program to translate C Web into TeX; default cweave.TANGLE Program to translate Web into Pascal; default tangle.CTANGLE Program to
translate C Web into C; default ctangle.RM Command to remove a file; default rm -f. Here is a table of variables whose values are additional arguments for theprograms above. The default values for all of these is the emptystring, unless otherwise noted.ARFLAGS Flags to give the archive-maintaining program; default rv.ASFLAGS Extra flags to give
to the assembler (when explicitlyinvoked on a .s or .S file).CFLAGS Extra flags to give to the C compiler.CXXFLAGS Extra flags to give to the C++ compiler.COFLAGS Extra flags to give to the RCS co program.CPPFLAGS Extra flags to give to the C preprocessor and programsthat use it (the C and Fortran compilers).FFLAGS Extra flags to give to the
Fortran compiler.GFLAGS Extra flags to give to the SCCS get program.LDFLAGS Extra flags to give to compilers when they are supposed to invoke the linker,ld, such as -L. Libraries (-lfoo) should beadded to the LDLIBS variable instead.LDLIBS Library flags or names given to compilers when they are supposed toinvoke the linker, ld. LOADLIBES is a
deprecated (butstill supported) alternative to LDLIBS. Non-library linkerflags, such as -L, should go in the LDFLAGS variable.LFLAGS Extra flags to give to Lex.YFLAGS Extra flags to give to Yacc.PFLAGS Extra flags to give to the Pascal compiler.RFLAGS Extra flags to give to the Fortran compiler for Ratfor programs.LINTFLAGS Extra flags to give
to lint. Sometimes a file can be made by a sequence of implicit rules. For example,a file n.o could be made from n.y by runningfirst Yacc and then cc. Such a sequence is called a chain.If the file n.c exists, or is mentioned in the makefile, nospecial searching is required: make finds that the object file canbe made by C compilation from n.c; later on,
when consideringhow to make n.c, the rule for running Yacc isused. Ultimately both n.c and n.o areupdated.However, even if n.c does not exist and is not mentioned,make knows how to envision it as the missing link betweenn.o and n.y! In this case, n.c iscalled an intermediate file. Once make has decided to use theintermediate file, it is entered in the
data base as if it had beenmentioned in the makefile, along with the implicit rule that says how tocreate it.Intermediate files are remade using their rules just like all otherfiles. But intermediate files are treated differently in two ways.The first difference is what happens if the intermediate file does notexist. If an ordinary file b does not exist, and
makeconsiders a target that depends on b, it invariably createsb and then updates the target from b. But if b isan intermediate file, then make can leave well enough alone:it wont create b unless one of its prerequisites is out ofdate. This means the target depending on b wont be rebuilteither, unless there is some other reason to update that target:
forexample the target doesnt exist or a different prerequisite is newerthan the target.The second difference is that if make does create b inorder to update something else, it deletes b later on after it is nolonger needed. Therefore, an intermediate file which did not exist beforemake also does not exist after make. make reports thedeletion to you by
printing a rm command showing which file it isdeleting.You can explicitly mark a file as intermediate by listing it as a prerequisiteof the special target .INTERMEDIATE. This takes effect even if thefile is mentioned explicitly in some other way.A file cannot be intermediate if it is mentioned in the makefile as a targetor prerequisite, so one way to avoid
the deletion of intermediate files is byadding it as a prerequisite to some target. However, doing so can cause maketo do extra work when searching pattern rules (see Implicit Rule Search Algorithm).As an alternative, listing a file as a prerequisite of the special target.NOTINTERMEDIATE forces it to not be considered intermediate (just asany other
mention of the file will do). Also, listing the target pattern of apattern rule as a prerequisite of .NOTINTERMEDIATE ensures that notargets generated using that pattern rule are considered intermediate.You can disable intermediate files completely in your makefile byproviding .NOTINTERMEDIATE as a target with no prerequisites:in that case it
applies to every file in the makefile.If you do not want make to create a file merely because it doesnot already exist, but you also do not want make toautomatically delete the file, you can mark it as a secondaryfile. To do this, list it as a prerequisite of the special target.SECONDARY. Marking a file as secondary also marks it asintermediate.A chain can
involve more than two implicit rules. For example, it ispossible to make a file foo from RCS/foo.y,v by running RCS,Yacc and cc. Then both foo.y and foo.c areintermediate files that are deleted at the end.No single implicit rule can appear more than once in a chain. This meansthat make will not even consider such a ridiculous thing as makingfoo from
foo.o.o by running the linker twice. Thisconstraint has the added benefit of preventing any infinite loop in thesearch for an implicit rule chain.There are some special implicit rules to optimize certain cases that wouldotherwise be handled by rule chains. For example, making foo fromfoo.c could be handled by compiling and linking with
separatechained rules, using foo.o as an intermediate file. But whatactually happens is that a special rule for this case does the compilationand linking with a single cc command. The optimized rule is used inpreference to the step-by-step chain because it comes earlier in theordering of rules.Finally, for performance reasons make will not consider
non-terminalmatch-anything rules (i.e., %:) when searching for a rule tobuild a prerequisite of an implicit rule (see Match-Anything Pattern Rules). You define an implicit rule by writing a pattern rule. A patternrule looks like an ordinary rule, except that its target contains thecharacter % (exactly one of them). The target is considered apattern for
matching file names; the % can match any nonemptysubstring, while other characters match only themselves. The prerequisiteslikewise use % to show how their names relate to the target name.Thus, a pattern rule %.o : %.c says how to make any filestem.o from another file stem.c.Note that expansion using % in pattern rules occursafter any variable
or function expansions, which take placewhen the makefile is read. See How to UseVariables, and Functions for Transforming Text. A pattern rule contains the character % (exactly one of them)in the target; otherwise, it looks exactly like an ordinary rule. Thetarget is a pattern for matching file names; the % matches anynonempty substring, while
other characters match only themselves.For example, %.c as a pattern matches any file name that ends in.c. s.%.c as a pattern matches any file name that startswith s., ends in .c and is at least five characters long.(There must be at least one character to match the %.) The substringthat the % matches is called the stem.% in a prerequisite of a pattern
rule stands for the same stemthat was matched by the % in the target. In order for thepattern rule to apply, its target pattern must match the file nameunder consideration and all of its prerequisites (after patternsubstitution) must name files that exist or can be made. These filesbecome prerequisites of the target.Thus, a rule of the form specifies how
to make a file n.o, with another filen.c as its prerequisite, provided that n.cexists or can be made.There may also be prerequisites that do not use %; such a prerequisiteattaches to every file made by this pattern rule. These unvaryingprerequisites are useful occasionally.A pattern rule need not have any prerequisites that contain %, orin fact any
prerequisites at all. Such a rule is effectively a generalwildcard. It provides a way to make any file that matches the targetpattern. See Defining Last-Resort Default Rules.More than one pattern rule may match a target. In this casemake will choose the best fit rule. See How Patterns Match.Pattern rules may have more than one target; however, every
target mustcontain a % character. Multiple target patterns in pattern rules arealways treated as grouped targets (see Multiple Targetsin a Rule) regardless of whether they use the : or &:separator.There is one exception: if a pattern target is out of date or doesnot exist and the makefile does not need to build it, then it will not causethe other targets
to be considered out of date. Note that this historicalexception will be removed in future versions of GNU make and should notbe relied on. If this situation is detected make will generate awarning pattern recipe did not update peer target; however, makecannot detect all such situations. Please be sure that your recipe updatesall the target patterns
when it runs. Here are some examples of pattern rules actually predefined inmake. First, the rule that compiles .c files into .ofiles:%.o : %.c $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@ defines a rule that can make any file x.o fromx.c. The recipe uses the automatic variables $@ and$< to substitute the names of the target file and the source filein

each case where the rule applies (see Automatic Variables).Here is a second built-in rule:% :: RCS/%,v $(CO) $(COFLAGS) $< defines a rule that can make any file x whatsoever from acorresponding file x,v in the sub-directory RCS. Sincethe target is %, this rule will apply to any file whatever, providedthe appropriate prerequisite file exists. The double colon makes the ruleterminal, which means that its prerequisite may not be an intermediatefile (see Match-Anything Pattern Rules).This pattern rule has two targets:%.tab.c %.tab.h: %.y bison -d $< This tells make that the recipe bison -d x.y willmake both x.tab.c and x.tab.h. If the filefoo depends on the files parse.tab.o and scan.oand the file scan.o depends on the file parse.tab.h,when parse.y is changed, the recipe bison -d parse.ywill be executed only once, and the prerequisites of bothparse.tab.o and scan.o will be satisfied. (Presumablythe file parse.tab.o will be recompiled from parse.tab.cand the file scan.o from scan.c, while foo islinked from parse.tab.o, scan.o, and its otherprerequisites, and it will execute happily ever after.) Suppose you are writing a pattern rule to compile a .c file into a.o file: how do you write the cc command so that it operateson the right source file name? You cannot write the name in the recipe,because the name is different each time the implicit rule is applied.What you do is use a special feature of make, the automaticvariables. These variables have values computed afresh for each rule thatis executed, based on the target and prerequisites of the rule. In thisexample, you would use $@ for the object file name and $filenm, floc->lineno); /* Register the function with make name "mk-temp". */ gmk_add_function ("mk-temp", gen_tmpfile, 1, 1, 1); return 1;} Next, we will write a Makefile that can build this shared object, loadit, and use it:all: @echo Temporary file: $(mk-temp tmpfile.)load mk_temp.somk_temp.so: mk_temp.c $(CC) -shared -fPIC -o $@ $< On MS-Windows, due to peculiarities of how shared objects areproduced, the compiler needs to scan the import library producedwhen building make, typically calledlibgnumake-version.dll.a, where version is theversion of the load object API. So the recipe to produce a sharedobject will look on Windows like this (assuming the API version is 1):mk_temp.dll: mk_temp.c $(CC) -shared -o $@ $< -lgnumake-1 Now when you run make youll see something like:$ makemk_temp plugin loaded from Makefile:4cc -shared -fPIC -o mk_temp.so mk_temp.cTemporary filename: tmpfile.A7JEwd GNU make is often one component in a larger system of tools,including integrated development environments, compiler toolchains,and others. The role of make is to start commands anddetermine whether they succeeded or not: no special integration isneeded to accomplish that. However, sometimes it is convenient tobind make more tightly with other parts of the system, bothhigher-level (tools that invoke make) and lower-level (toolsthat make invokes).Sharing Job Slots with GNU makeSynchronized Terminal Output GNU make has the ability to run multiple recipes in parallel(see Parallel Execution) and to cap the total number ofparallel jobs even across recursive invocations of make(see Communicating Options to aSub-make). Tools that make invokes which are also ableto run multiple operations in parallel, either using multiple threadsor multiple processes, can be enhanced to participate in GNUmakes job management facility to ensure that the total numberof active threads/processes running on the system does not exceed themaximum number of slots provided to GNU make. GNU make uses a method called the jobserver to control thenumber of active jobs across recursive invocations. The actualimplementation of the jobserver varies across different operatingsystems, but some fundamental aspects are always true.First, make will provide information necessary for accessing thejobserver through the environment to its children, in the MAKEFLAGSenvironment variable. Tools which want to participate in the jobserverprotocol will need to parse this environment variable and find the wordstarting with --jobserver-auth=. The value of this option willdescribe how to communicate with the jobserver. The interpretation of thisvalue is described in the sections below.Be aware that the MAKEFLAGS variable may contain multiple instances ofthe --jobserver-auth= option. Only the last instance isrelevant.Second, every command make starts has one implicit job slotreserved for it before it starts. Any tool which wants to participatein the jobserver protocol should assume it can always run one jobwithout having to contact the jobserver at all.Finally, its critical that tools that participate in the jobserverprotocol return the exact number of slots they obtained from thejobserver back to the jobserver before they exit, even under errorconditions. Remember that the implicit job slot should notbe returned to the jobserver! Returning too few slots means thatthose slots will be lost for the rest of the build process; returningtoo many slots means that extra slots will be available. Thetop-level make command will print an error message at the endof the build if it detects an incorrect number of slots available inthe jobserver.As an example, suppose you are implementing a linker which providesfor multithreaded operation. You would like to enhance the linker sothat if it is invoked by GNU make it can participate in thejobserver protocol to control how many threads are used during link.First you will need to modify the linker to determine if theMAKEFLAGS environment variable is set. Next you will need toparse the value of that variable to determine if the jobserver isavailable, and how to access it. If it is available then you canaccess it to obtain job slots controlling how much parallelism yourtool can use. Once done your tool must return those job slots back tothe jobserver.POSIX Jobserver InteractionWindows Jobserver Interaction On POSIX systems the jobserver is implemented in one of two ways: on systemsthat support it, GNU make will create a named pipe and use that for thejobserver. In this case the auth option will have the form--jobserver-auth=fifo:PATH where PATH is the pathname of thenamed pipe. To access the jobserver you should open the named pipe path andread/write to it as described below.If the system doesnt support named pipes, or if the user provided the--jobserver-style option and specified pipe, then the jobserverwill be implemented as a simple UNIX pipe. In this case the auth option willhave the form --jobserver-auth=R,W where R and W arenon-negative integers representing file descriptors: R is the read filedescriptor and W is the write file descriptor. If either or both ofthese file descriptors are negative, it means the jobserver is disabled forthis process.When using a simple pipe, only command lines that make understands tobe recursive invocations of make (see How theMAKE Variable Works) will have access to the jobserver. When writingmakefiles you must be sure to mark the command as recursive (most commonly byprefixing the command line with the + indicator (see Recursive Use of make). Note that the read side of the jobserverpipe is set to blocking mode. This should not be changed.In both implementations of the jobserver, the pipe will be pre-loaded with onesingle-character token for each available job. To obtain an extra slot youmust read a single character from the jobserver; to release a slot you mustwrite a single character back into the jobserver.Its important that when you release the job slot, you write back the samecharacter you read. Dont assume that all tokens are the same character;different characters may have different meanings to GNU make. Theorder is not important, since make has no idea in what order jobs willcomplete anyway.There are various error conditions you must consider to ensure yourimplementation is robust: If you have a command-line argument controlling the parallel operation of yourtool, consider whether your tool should detect situations where both thejobserver and the command-line argument are specified, and how it shouldreact. If your tool does not recognize the format of the --jobserver-authstring, it should assume the jobserver is using a different style and itcannot connect. If your tool determines that the --jobserver-auth option references asimple pipe but that the file descriptors specified are closed, this meansthat the calling make process did not think that your tool was arecursive make invocation (e.g., the command line was not prefixed witha + character). You should notify your users of this situation. Your tool should be sure to write back the tokens it read, even under errorconditions. This includes not only errors in your tool but also outsideinfluences such as interrupts (SIGINT), etc. You may want to installsignal handlers to manage this write-back. Your tool may also examine the first word of the MAKEFLAGS variable andlook for the character n. If this character is present thenmake was invoked with the -n option and your tool may want tostop without performing any operations. Normally GNU make will invoke all commands with access to thesame standard and error outputs that make itself was startedwith. A number of tools will detect whether the output is a terminalor not-a-terminal, and use this information to change the outputstyle. For example if the output goes to a terminal the tool may addcontrol characters that set color, or even change the location of thecursor. If the output is not going to a terminal then these specialcontrol characters are not emitted so that they dont corrupt logfiles, etc.The --output-sync (see Output DuringParallel Execution) option will defeat the terminal detection. Whenoutput synchronization is enabled GNU make arranges for allcommand output to be written to a file, so that its output can bewritten as a block without interference from other commands. Thismeans that all tools invoked by make will believe that theiroutput is not going to be displayed on a terminal, even when it willbe (because make will display it there after the command iscompleted).In order to facilitate tools which would like to determine whether ornot their output will be displayed on a terminal, GNU make willset the MAKE_TERMOUT and MAKE_TERMERR environmentvariables before invoking any commands. Tools which would like todetermine whether standard or error output (respectively) will bedisplayed on a terminal can check these environment variables todetermine if they exist and contain a non-empty value. If so the toolcan assume that the output will (eventually) be displayed on aterminal. If the variables are not set or have an empty value, thenthe tool should fall back to its normal methods of detecting whetheroutput is going to a terminal or not.The content of the variables can be parsed to determine the type ofterminal which will be used to display the output.Similarly, environments which invoke make and would like tocapture the output and eventually display it on a terminal (or somedisplay which can interpret terminal control characters) can set thesevariables before invoking make. GNU make will notmodify these environment variables if they already exist when itstarts. Here is a summary of the features of GNU make, for comparisonwith and credit to other versions of make. We consider thefeatures of make in 4.2 BSD systems as a baseline. If you areconcerned with writing portable makefiles, you should not use thefeatures of make listed here, nor the ones in Incompatibilities and Missing Features.Many features come from the version of make in System V. The VPATH variable and its special meaning.See Searching Directories for Prerequisites.This feature exists in System V make, but is undocumented.It is documented in 4.3 BSD make (which says it mimics System VsVPATH feature). Included makefiles. See Including Other Makefiles.Allowing multiple files to be included with a single directive is a GNUextension. Variables are read from and communicated via the environment.See Variables from the Environment. Options passed through the variable MAKEFLAGS to recursiveinvocations of make.See Communicating Options to a Sub-make. The automatic variable $% is set to the member namein an archive reference. See Automatic Variables. The automatic variables $@, $*, $ foo.1 will fail when the build directory is not the source directory, becausefoo.man and sedscript are in the source directory.When using GNU make, relying on VPATH to find the sourcefile will work in the case where there is a single dependency file,since the make automatic variable $< will represent thesource file wherever it is. (Many versions of make set $ $@ GNU distributions usually contain some files which are not sourcefilesfor example, Info files, and the output from Autoconf, Automake,Bison or Flex. Since these files normally appear in the sourcedirectory, they should always appear in the source directory, not in thebuild directory. So Makefile rules to update them should put theupdated files in the source directory.However, if a file does not appear in the distribution, then theMakefile should not put it in the source directory, because building aprogram in ordinary circumstances should not modify the source directoryin any way.Try to make the build and installation targets, at least (and all theirsubtargets) work correctly with a parallel make. Write the Makefile commands (and any shell scripts, such asconfigure) to run under sh (both the traditional Bourneshell and the POSIX shell), not csh. Dont use anyspecial features of ksh or bash, or POSIX featuresnot widely supported in traditional Bourne sh.The configure script and the Makefile rules for building andinstallation should not use any utilities directly except these:awk cat cmp cp diff echo expr false grep install-info ln lsmkdir mv printf pwd rm rmdir sed sleep sort tar test touch tr true Compression programs such as gzip can be used in thedist rule.Generally, stick to the widely-supported (usuallyPOSIX-specified) options and features of these programs. Forexample, dont use mkdir -p, convenient as it may be, because afew systems dont support it at all and with others, it is not safefor parallel execution. For a list of known incompatibilities, seePortable Shell Programming in Autoconf. It is a good idea to avoid creating symbolic links in makefiles, since afew file systems dont support them.The Makefile rules for building and installation can also use compilersand related programs, but should do so via make variables so that theuser can substitute alternatives. Here are some of the programs wemean:ar bison cc flex install ld ldconfig lexmake makeinfo ranlib texi2dvi yacc Use the following make variables to run those programs:$(AR) $(BISON) $(CC) $(FLEX) $(INSTALL) $(LD) $(LDCONFIG) (LEX)(MAKE) $(MAKEINFO) $(RANLIB) $(TEXI2DVI) $(YACC) When you use ranlib or ldconfig, you should make surenothing bad happens if the system does not have the program in question.Arrange to ignore an error from that command, and print a message beforethe command to tell the user that failure of this command does not meana problem. (The Autoconf AC_PROG_RANLIB macro can help withthis.)If you use symbolic links, you should implement a fallback for systemsthat dont have symbolic links.Additional utilities that can be used via Make variables are: It is ok to use other utilities in Makefile portions (or scripts)intended only for particular systems where you know those utilitiesexist. Makefiles should provide variables for overriding certain commands, options,and so on.In particular, you should run most utility programs via variables.Thus, if you use Bison, have a variable named BISON whose defaultvalue is set with BISON = bison, and refer to it with$(BISON) whenever you need to use Bison.File management utilities such as ln, rm, mv, andso on, need not be referred to through variables in this way, since usersdont need to replace them with other programs.Each program-name variable should come with an options variable that isused to supply options to the program. Append FLAGS to theprogram-name variable name to get the options variable nameforexample, BISONFLAGS. (The names CFLAGS for the Ccompiler, YFLAGS for yacc, and LFLAGS for lex, areexceptions to this rule, but we keep them because they are standard.)Use CPPFLAGS in any compilation command that runs thepreprocessor, and use LDFLAGS in any compilation command thatdoes linking as well as in any direct use of ld.If there are C compiler options that must be used for propercompilation of certain files, do not include them in CFLAGS.Users expect to be able to specify CFLAGS freely themselves.Instead, arrange to pass the necessary options to the C compilerindependently of CFLAGS, by writing them explicitly in thecompilation commands or by defining an implicit rule, like this:CFLAGS = -gALL_CFLAGS = -I. $(CFLAGS).c.o: $(CC) -c $(CPPFLAGS) $(ALL_CFLAGS) $< Do include the -g option in CFLAGS, because that is notrequired for proper compilation. You can consider it a defaultthat is only recommended. If the package is set up so that it iscompiled with GCC by default, then you might as well include -Oin the default value of CFLAGS as well.Put CFLAGS last in the compilation command, after other variablescontaining compiler options, so the user can use CFLAGS tooverride the others.CFLAGS should be used in every invocation of the C compiler,both those which do compilation and those which do linking.Every Makefile should define the variable INSTALL, which is thebasic command for installing a file into the system.Every Makefile should also define the variables INSTALL_PROGRAMand INSTALL_DATA. (The default for INSTALL_PROGRAM shouldbe $(INSTALL); the default for INSTALL_DATA should be${INSTALL} -m 644.) Then it should use those variables as thecommands for actual installation, for executables and non-executablesrespectively. Minimal use of these variables is as follows:$(INSTALL_PROGRAM) foo $(bindir)/foo$(INSTALL_DATA) libfoo.a $(libdir)/libfoo.a However, it is preferable to support a DESTDIR prefix on thetarget files, as explained in the next section.It is acceptable, but not required, to install multiple files in onecommand, with the final argument being a directory, as in:$(INSTALL_PROGRAM) foo bar baz $(bindir) DESTDIR is a variable prepended to each installed target file,like this:$(INSTALL_PROGRAM) foo $(DESTDIR)$(bindir)/foo$(INSTALL_DATA) libfoo.a $(DESTDIR)$(libdir)/libfoo.a The DESTDIR variable is specified by the user on the makecommand line as an absolute file name. For example:make DESTDIR=/tmp/stage install DESTDIR should be supported only in the install* anduninstall* targets, as those are the only targets where it isuseful.If your installation step would normally install/usr/local/bin/foo and /usr/local/lib/libfoo.a, then aninstallation invoked as in the example above would install/tmp/stage/usr/local/bin/foo and/tmp/stage/usr/local/lib/libfoo.a instead.Prepending the variable DESTDIR to each target in this wayprovides for staged installs, where the installed files are notplaced directly into their expected location but are instead copiedinto a temporary location (DESTDIR). However, installed filesmaintain their relative directory structure and any embedded file nameswill not be modified.You should not set the value of DESTDIR in your Makefileat all; then the files are installed into their expected locations bydefault. Also, specifying DESTDIR should not change theoperation of the software in any way, so its value should not beincluded in any file contents.DESTDIR support is commonly used in package creation. It isalso helpful to users who want to understand what a given package willinstall where, and to allow users who dont normally have permissionsto install into protected areas to build and install before gainingthose permissions. Finally, it can be useful with tools such asstow, where code is installed in one place but made to appearto be installed somewhere else using symbolic links or special mountoperations. So, we strongly recommend GNU packages supportDESTDIR, though it is not an absolute requirement. Installation directories should always be named by variables, so it iseasy to install in a nonstandard place. The standard names for thesevariables and the values they should have in GNU packages aredescribed below. They are based on a standard file system layout;variants of it are used in GNU/Linux and other modern operatingsystems.Installers are expected to override these values when callingmake (e.g., make prefix=/usr install) orconfigure (e.g., configure --prefix=/usr). GNUpackages should not try to guess which value should be appropriate forthese variables on the system they are being installed onto: use thedefault settings specified here so that all GNU packages behaveidentically, allowing the installer to achieve any desired layout.All installation directories, and their parent directories, should becreated (if necessary) before they are installed into.These first two variables set the root for the installation. All theother installation directories should be subdirectories of one ofthese two, and nothing should be directly installed into these twodirectories.prefix A prefix used in constructing the default values of the variables listedbelow. The default value of prefix should be /usr/local.When building the complete GNU system, the prefix will be empty and/usr will be a symbolic link to /.(If you are using Autoconf, write it as @prefix@.)Running make install with a different value of prefix fromthe one used to build the program should not recompile theprogram.exec_prefix A prefix used in constructing the default values of some of thevariables listed below. The default value of exec_prefix shouldbe $(prefix).(If you are using Autoconf, write it as @exec_prefix@.)Generally, $(exec_prefix) is used for directories that containmachine-specific files (such as executables and subroutine libraries),while $(prefix) is used directly for other directories.Running make install with a different value of exec_prefixfrom the one used to build the program should not recompile theprogram. Executable programs are installed in one of the following directories.bindir The directory for installing executable programs that users can run.This should normally be /usr/local/bin, but write it as$(exec_prefix)/bin.(If you are using Autoconf, write it as @bindir@.)sbindir The directory for installing executable programs that can be run fromthe shell, but are only generally useful to system administrators. Thisshould normally be /usr/local/sbin, but write it as$(exec_prefix)/sbin.(If you are using Autoconf, write it as @sbindir@.)libexecdir The directory for installing executable programs to be run by otherprograms rather than by users. This directory should normally be/usr/local/libexec, but write it as $(exec_prefix)/libexec.(If you are using Autoconf, write it as @libexecdir@.)The definition of libexecdir is the same for all packages, soyou should install your data in a subdirectory thereof. Most packagesinstall their data under $(libexecdir)/package-name/,possibly within additional subdirectories thereof, such as$(libexecdir)/package-name/machine/version. Data files used by the program during its execution are divided intocategories in two ways. Some files are normally modified by programs; others are never normallymodified (though users may edit some of these). Some files are architecture-independent and can be shared by allmachines at a site; some are architecture-dependent and can be sharedonly by machines of the same kind and operating system; others may neverbe shared between two machines. This makes for six different possibilities. However, we want todiscourage the use of architecture-dependent files, aside from objectfiles and libraries. It is much cleaner to make other data filesarchitecture-independent, and it is generally not hard.Here are the variables Makefiles should use to specify directoriesto put these various kinds of files in:datarootdirThe root of the directory tree for read-only architecture-independentdata files. This should normally be /usr/local/share, butwrite it as $(prefix)/share. (If you are using Autoconf, writeit as @datarootdir@.) datadirs default value isbased on this variable; so are infodir, mandir, andothers.datadirThe directory for installing idiosyncratic read-onlyarchitecture-independent data files for this program. This is usuallythe same place as datarootdir, but we use the two separatevariables so that you can move these program-specific files withoutaltering the location for Info files, man pages, etc.This should normally be /usr/local/share, but write it as$(datarootdir). (If you are using Autoconf, write it as@datadir@.)The definition of datadir is the same for all packages, so youshould install your data in a subdirectory thereof. Most packagesinstall their data under $(datadir)/package-name/.sysconfdirThe directory for installing read-only data files that pertain to asingle machinethat is to say, files for configuring a host. Mailerand network configuration files, /etc/passwd, and so forth belonghere. All the files in this directory should be ordinary ASCII textfiles. This directory should normally be /usr/local/etc, butwrite it as $(prefix)/etc.(If you are using Autoconf, write it as @sysconfdir@.)Do not install executables here in this directory (they probably belongin $(libexecdir) or $(sbindir)). Also do not installfiles that are modified in the normal course of their use (programswhose purpose is to change the configuration of the system excluded).Those probably belong in $(localstatedir).sharedstatedirThe directory for installing architecture-independent data files whichthe programs modify while they run. This should normally be/usr/local/com, but write it as $(prefix)/com.(If you are using Autoconf, write it as @sharedstatedir@.)localstatedirThe directory for installing data files which the programs modify whilethey run, and that pertain to one specific machine. Users should neverneed to modify files in this directory to configure the packagesoperation; put such configuration information in separate files that goin $(datadir) or $(sysconfdir). $(localstatedir)should normally be /usr/local/var, but write it as$(prefix)/var.(If you are using Autoconf, write it as @localstatedir@.)runstatedirThe directory for installing data files which the programs modifywhile they run, that pertain to one specific machine, and which neednot persist longer than the execution of the programwhich isgenerally long-lived, for example, until the next reboot. PID filesfor system daemons are a typical use. In addition, this directoryshould not be cleaned except perhaps at reboot, while the general/tmp (TMPDIR) may be cleaned arbitrarily. This shouldnormally be /var/run, but write it as$(localstatedir)/run. Having it as a separate variable allowsthe use of /run if desired, for example. (If you are usingAutoconf 2.70 or later, write it as @runstatedir@.) These variables specify the directory for installing certain specifictypes of files, if your program has them. Every GNU package shouldhave Info files, so every program needs infodir, but not allneed libdir or lispdir.includedirThe directory for installing header files to be included by userprograms with the C #include preprocessor directive. Thisshould normally be /usr/local/include, but write it as$(prefix)/include.(If you are using Autoconf, write it as @includedir@.)Most compilers other than GCC do not look for header files in directory/usr/local/include. So installing the header files this way isonly useful with GCC. Sometimes this is not a problem because somelibraries are only really intended to work with GCC. But some librariesare intended to work with other compilers. They should install theirheader files in two places, one specified by includedir and onespecified by oldincludedir.oldincludedirThe directory for installing #include header files for use withcompilers other than GCC. This should normally be /usr/include.(If you are using Autoconf, you can write it as @oldincludedir@.)The Makefile commands should check whether the value ofoldincludedir is empty. If it is, they should not try to useit; they should cancel the second installation of the header files.A package should not replace an existing header in this directory unlessthe header came from the same package. Thus, if your Foo packageprovides a header file foo.h, then it should install the headerfile in the oldincludedir directory if either (1) there is nofoo.h there or (2) the foo.h that exists came from the Foopackage.To tell whether foo.h came from the Foo package, put a magicstring in the filepart of a commentand grep for that string.docdirThe directory for installing documentation files (other than Info) forthis package. By default, it should be/usr/local/share/doc/yourpkg, but it should be written as$(datarootdir)/doc/yourpkg. (If you are using Autoconf,write it as @docdir@.) The yourpkg subdirectory, whichmay include a version number, prevents collisions among files withcommon names, such as README.infodirThe directory for installing the Info files for this package. Bydefault, it should be/usr/local/share/info, but it should bewritten as $(datarootdir)/info. (If you are using Autoconf,write it as @infodir@.) infodir is separate fromdocdir for compatibility with existing practice.htmldirdvidirpdfdirpsdirDirectories for installing documentation files in the particularformat. They should all be set to $(docdir) by default. (Ifyou are using Autoconf, write them as @htmldir@,@dvidir@, etc.) Packages which supply several translationsof their documentation should install them in$(htmldir)/ll, $(pdfdir)/ll, etc. wherell is a locale abbreviation such as en or pt_BR.libdirThe directory for object files and libraries of object code. Do notinstall executables here, they probably ought to go in $(libexecdir)instead. The value of libdir should normally be/usr/local/lib, but write it as $(exec_prefix)/lib.(If you are using Autoconf, write it as @libdir@.)lispdirThe directory for installing any Emacs Lisp files in this package. Bydefault, it should be /usr/local/share/emacs/site-lisp, but itshould be written as $(datarootdir)/emacs/site-lisp.If you are using Autoconf, write the default as @lispdir@.In order to make @lispdir@ work, you need the following linesin your configure.ac file:lispdir='${datarootdir}/emacs/site-lisp'AC_SUBST(lispdir)localedirThe directory for installing locale-specific message catalogs for thispackage. By default, it should be /usr/local/share/locale, butit should be written as $(datarootdir)/locale. (If you areusing Autoconf, write it as @localedir@.) This directoryusually has a subdirectory per locale. Unix-style man pages are installed in one of the following:mandirThe top-level directory for installing the man pages (if any) for thispackage. It will normally be /usr/local/share/man, but youshould write it as $(datarootdir)/man. (If you are usingAutoconf, write it as @mandir@.)man1dirThe directory for installing section 1 man pages. Write it as$(mandir)/man1.man2dirThe directory for installing section 2 man pages. Write it as$(mandir)/man2Dont make the primary documentation for any GNU software be aman page. Write a manual in Texinfo instead. Man pages are just forthe sake of people running GNU software on Unix, which is a secondaryapplication only.manextThe file name extension for the installed man page. This should containa period followed by the appropriate digit; it should normally be .1.man1extThe file name extension for installed section 1 man pages.man2extThe file name extension for installed section 2 man pages.Use these names instead of manext if the package needs to install manpages in more than one section of the manual. And finally, you should set the following variable:srcdirThe directory for the sources being compiled. The value of thisvariable is normally inserted by the configure shell script.(If you are using Autoconf, use srcdir = @srcdir@.) For example:# Common prefix for installation directories.# NOTE: This directory must exist when you start the install.prefix = /usr/localdatarootdir = $(prefix)/sharedatadir = $(datarootdir)exec_prefix = $(prefix)# Where to put the executable for the command 'gcc'.bindir = $(exec_prefix)/bin# Where to put the directories used by the compiler.libexecdir = $(exec_prefix)/libexec# Where to put the Info files.infodir = $(datarootdir)/info If your program installs a large number of files into one of thestandard user-specified directories, it might be useful to group theminto a subdirectory particular to that program. If you do this, youshould write the install rule to create these subdirectories.Do not expect the user to include the subdirectory name in the value ofany of the variables listed above. The idea of having a uniform set ofvariable names for installation directories is to enable the user tospecify the exact same values for several different GNU packages. Inorder for this to be useful, all the packages must be designed so thatthey will work sensibly when the user does so.At times, not all of these variables may be implemented in the currentrelease of Autoconf and/or Automake; but as of Autoconf2.60, webelieve all of them are. When any are missing, the descriptions hereserve as specifications for what Autoconf will implement. As aprogrammer, you can either use a development version of Autoconf oravoid using these variables until a stable release is made whichsupports them. All GNU programs should have the following targets in their Makefiles:allCompile the entire program. This should be the default target. Thistarget need not rebuild any documentation files; Info files shouldnormally be included in the distribution, and DVI (and otherdocumentation format) files should be made only when explicitly askedfor.By default, the Make rules should compile and link with -g, sothat executable programs have debugging symbols. Otherwise, you aressentially helpless in the face of a crash, and it is often far fromeasy to reproduce with a fresh build.installCompile the program and copy the executables, libraries, and so on tothe file names where they should reside for actual use. If there is asimple test to verify that a program is properly installed, this targetshould run that test.Do not strip executables when installing them. This helps eventualdebugging that may be needed later, and nowadays disk space is cheapand dynamic loaders typically ensure debug sections are not loaded duringnormal execution. Users that need stripped binaries may invoke theinstall-strip target to do that.If possible, write the install target rule so that it does notmodify anything in the directory where the program was built, providedmake all has just been done. This is convenient for building theprogram under one user name and installing it under another.The commands should create all the directories in which files are to beinstalled, if they dont already exist. This includes the directoriesspecified as the values of the variables prefix andexec_prefix, as well as all subdirectories that are needed.One way to do this is by means of an installdirs targetas described below.Use - before any command for installing a man page, so thatmake will ignore any errors. This is in case there are systemsthat dont have the Unix man page documentation system installed.The way to install Info files is to copy them into $(infodir)with $(INSTALL_DATA) (see Variables for Specifying Commands), and then runthe install-info program if it is present. install-infois a program that edits the Info dir file to add or update themenu entry for the given Info file; it is part of the Texinfo package.Here is a sample rule to install an Info file that also tries tohandle some additional situations, such as install-info notbeing present.do-install-info: foo.info installdirs $(NORMAL_INSTALL)# Prefer an info file in . to one in srcdir. if test -f foo.info; then d=.; \ else d="$(srcdir)"; fi; \ $(INSTALL_DATA) $$d/foo.info \ "$(DESTDIR)$(infodir)/foo.info"# Run install-info only if it exists.# Use 'if' instead of just prepending '-' to the# line so we notice real errors from install-info.# Use '$(SHELL) -c' because some shells do not# fail gracefully when there is an unknown command. $(POST_INSTALL) if $(SHELL) -c 'install-info --version' \ >/dev/null 2>&1; then \ install-info --dir-file="$(DESTDIR)$(infodir)/dir" \ "$(DESTDIR)$(infodir)/foo.info"; \ else true; fi When writing the install target, you must classify all thecommands into three categories: normal ones, pre-installationcommands and post-installation commands. See Install Command Categories.install-htmlinstall-dviinstall-pdfinstall-psThese targets install documentation in formats other than Info;theyre intended to be called explicitly by the person installing thepackage, if that format is desired. GNU prefers Info files, so thesemust be installed by the install target.When you have many documentation files to install, we recommend thatyou avoid collisions and clutter by arranging for these targets toinstall in subdirectories of the appropriate installation directory,such as htmldir. As one example, if your package has multiplemanuals, and you wish to install HTML documentation with many files(such as the split mode output by makeinfo --html), youllcertainly want to use subdirectories, or two nodes with the same namein different manuals will overwrite each other.Please make these install-format targets invoke thecommands for the format target, for example, by makingformat a dependency.uninstallDelete all the installed filesthe copies that the installand install-* targets create.This rule should not modify the directories where compilation is done,only the directories where files are installed.The uninstallation commands are divided into three categories, just likethe installation commands. See Install Command Categories.install-stripLike install, but strip the executable files while installingthem. In simple cases, this target can use the install target ina simple way:install-strip: $(MAKE) INSTALL_PROGRAM='$(INSTALL_PROGRAM) -s' \ install But if the package installs scripts as well as real executables, theinstall-strip target cant just refer to the installtarget; it has to strip the executables but not the scripts.install-strip should not strip the executables in the builddirectory which are being copied for installation. It should only stripthe copies that are installed.Normally we do not recommend stripping an executable unless you are surethe program has no bugs. However, it can be reasonable to install astripped executable for actual execution while saving the unstrippedexecutable elsewhere in case there is a bug.cleanDelete all files in the current directory that are normally created bybuilding the program. Also delete files in other directories if theyare created by this makefile. However, dont delete the files thatrecord the configuration. Also preserve files that could be made bybuilding, but normally arent because the distribution comes withthem. There is no need to delete parent directories that were createdwith mkdir -p, since they could have existed anyway.Delete .dvi files here if they are not part of the distribution.distcleanDelete all files in the current directory (or created by thismakefile) that are created by configuring or building the program. Ifyou have unpacked the source and built the program without creatingany other files, make distclean should leave only the filesthat were in the distribution. However, there is no need to deleteparent directories that were created with mkdir -p, since theycould have existed anyway.mostlycleanLike clean, but may refrain from deleting a few files that peoplenormally dont want to recompile. For example, the mostlycleantarget for GCC does not delete libgcc.a, because recompiling itis rarely necessary and takes a lot of time.maintainer-cleanDelete almost everything that can be reconstructed with this Makefile.This typically includes everything deleted by distclean, plusmore: C source files produced by Bison, tags tables, Info files, andso on.The reason we say almost everything is that running the commandmake maintainer-clean should not delete configure evenif configure can be remade using a rule in the Makefile. Moregenerally, make maintainer-clean should not delete anythingthat needs to exist in order to run configure and then begin tobuild the program. Also, there is no need to delete parentdirectories that were created with mkdir -p, since they couldhave existed anyway. These are the only exceptions;maintainer-clean should delete everything else that can berebuilt.The maintainer-clean target is intended to be used by a maintainer ofthe package, not by ordinary users. You may need special tools toreconstruct some of the files that make maintainer-clean deletes.Since these files are normally included in the distribution, we donttake care to make them easy to reconstruct. If you find you need tounpack the full distribution again, dont blame us.To help make users aware of this, the commands for the specialmaintainer-clean target should start with these two:@echo 'This command is intended for maintainers to use; it'@echo 'deletes files that may need special tools to rebuild.'TAGSUpdate a tags table for this program.infoGenerate any Info files needed. The best way to write the rules is asfollows:info: foo.infofoo.info: foo.texi chap1.texi chap2.texi $(MAKEINFO) $(srcdir)/foo.texi You must define the variable MAKEINFO in the Makefile. It shouldrun the makeinfo program, which is part of the Texinfodistribution.Normally a GNU distribution comes with Info files, and that means theInfo files are present in the source directory. Therefore, the Makerule for an info file should update it in the source directory. Whenusers build the package, ordinarily Make will not update the Info filesbecause they will already be up to date.dvihtmlpdfpsGenerate documentation files in the given format. These targetsshould always exist, but any or all can be a no-op if the given outputformat cannot be generated. These targets should not be dependenciesof the all target; the user must manually invoke them.Heres an example rule for generating DVI files from Texinfo:dvi: foo.dvifoo.dvi: foo.texi chap1.texi chap2.texi $(TEXI2DVI) $(srcdir)/foo.texi You must define the variable TEXI2DVI in the Makefile. Itshould run the program texi2dvi, which is part of the Texinfodistribution. (texi2dvi uses TeX to do the real work offormatting. TeX is not distributed with Texinfo.) Alternatively,write only the dependencies, and allow GNU make to provide thecommand.Heres another example, this one for generating HTML from Texinfo:html: foo.htmlfoo.html: foo.texi chap1.texi chap2.texi $(TEXI2HTML) $(srcdir)/foo.texi Again, you would define the variable TEXI2HTML in the Makefile;for example, it might run makeinfo --no-split --html(makeinfo is part of the Texinfo distribution).distCreate a distribution tar file for this program. The tar file should beset up so that the file names in the tar file start with a subdirectoryname which is the name of the package it is a distribution for. Thisname can include the version number.For example, the distribution tar file of GCC version 1.40 unpacks intoa subdirectory named gcc-1.40.The easiest way to do this is to create a subdirectory appropriatelynamed, use ln or cp to install the proper files in it, andthen tar that subdirectory.Compress the tar file with gzip. For example, the actualdistribution file for GCC version 1.40 is called gcc-1.40.tar.gz.It is ok to support other free compression formats as well.The dist target should explicitly depend on all non-source filesthat are in the distribution, to make sure they are up to date in thedistribution.See Making Releases in GNU Coding Standards.checkPerform self-tests (if any). The user must build the program beforerunning the tests, but need not install the program; you should writethe self-tests so that they work when the program is built but notinstalled. The following targets are suggested as conventional names, for programsin which they are useful.installcheckPerform installation tests (if any). The user must build and installthe program before running the tests. You should not assume that$(bindir) is in the search path.installdirsIts useful to add a target named installdirs to create thedirectories where files are installed, and their parent directories.There is a script called mkinstalldirs which is convenient forthis; you can find it in the Gnulib package.You can use a rule like this:# Make sure all installation directories (e.g. $(bindir))# actually exist by making them if necessary.installdirs: mkinstalldirs $(srcdir)/mkinstalldirs $(bindir) $(datadir) \ $(libdir) $(infodir) \ $(mandir) or, if you wish to support DESTDIR (strongly encouraged),# Make sure all installation directories (e.g. $(bindir))# actually exist by making them if necessary.installdirs: mkinstalldirs $(srcdir)/mkinstalldirs \ $(DESTDIR)$(bindir) $(DESTDIR)$(datadir) \ $(DESTDIR)$(libdir) $(DESTDIR)$(infodir) \ $(DESTDIR)$(mandir) This rule should not modify the directories where compilation is done.It should do nothing but create installation directories. When writing the install target, you must classify all thecommands into three categories: normal ones, pre-installationcommands and post-installation commands.Normal commands move files into their proper places, and set theirmodes. They may not alter any files except the ones that come entirelyfrom the package they belong to.Pre-installation and post-installation commands may alter other files;in particular, they can edit global configuration files or data bases.Pre-installation commands are typically executed before the normalcommands, and post-installation commands are typically run after thenormal commands.The most common use for a post-installation command is to runinstall-info. This cannot be done with a normal command, sinceit alters a file (the Info directory) which does not come entirely andsolely from the package being installed. It is a post-installationcommand because it needs to be done after the normal command whichinstalls the packages Info files.Most programs dont need any pre-installation commands, but we have thefeature just in case it is needed.To classify the commands in the install rule into these threecategories, insert category lines among them. A category linespecifies the category for the commands that follow.A category line consists of a tab and a reference to a special Makevariable, plus an optional comment at the end. There are threevariables you can use, one for each category; the variable namespecifies the category. Category lines are no-ops in ordinary executionbecause these three Make variables are normally undefined (and youshould not define them in the makefile).Here are the three possible category lines, each with a comment thatexplains what it means: $(PRE_INSTALL) # Pre-install commands follow. $(POST_INSTALL) # Post-install commands follow. $(NORMAL_INSTALL) # Normal commands follow. If you dont use a category line at the beginning of the installrule, all the commands are classified as normal until the first categoryline. If you dont use any category lines, all the commands areclassified as normal.These are the category lines for uninstall: $(PRE_UNINSTALL) # Pre-uninstall commands follow. $(POST_UNINSTALL) # Post-uninstall commands follow. $(NORMAL_UNINSTALL) # Normal commands follow. Typically, a pre-uninstall command would be used for deleting entriesfrom the Info directory.If the install or uninstall target has any dependencieswhich act as subroutines of installation, then you should starteach dependencys commands with a category line, and start themain targets commands with a category line also. This way, you canensure that each command is placed in the right category regardless ofwhich of the dependencies actually run.Pre-installation and post-installation commands should not run anyprograms except for these:[basename bash cat chgrp chmod chown cmp cp dd diff echoexpand expr false find getopt grep gunzip gziphostname install install-info kill ldconfig ln ls md5summkdir mkfifo mknod mv printenv pwd rm rmdir sed sort teetest touch true uname xargs yesThe reason for distinguishing the commands in this way is for the sakeof making binary packages. Typically a binary package contains all theexecutables and other files that need to be installed, and has its ownmethod of installing themso it does not need to run the normalinstallation commands. But installing the binary package does need toexecute the pre-installation and post-installation commands.Programs to build binary packages work by extracting thepre-installation and post-installation commands. Here is one way ofextracting the pre-installation commands (the -s option tomake is needed to silence messages about enteringsubdirectories):make -s -n install -o all \ PRE_INSTALL=pre-install \ POST_INSTALL=post-install \ NORMAL_INSTALL=normal-install \ | gawk -f pre-install.awk where the file pre-install.awk could contain this:$0 ~ /^(normal-install|post-install)[\t]*$/ {on = 0}on {print $0}$0 ~ /^pre-install[\t]*$/ {on = 1} Here is the makefile for the GNU tar program. This is amoderately complex makefile. The first line uses a #! settingto allow the makefile to be executed directly.Because it is the first target, the default goal is all. Aninteresting feature of this makefile is that testpad.h is asource file automatically created by the testpad program,itself compiled from testpad.c.If you type make or make all, then make createsthe tar executable, the rmt daemon that providesremote tape access, and the tar.info Info file.If you type make install, then make not only createstar, rmt, and tar.info, but also installsthem.If you type make clean, then make removes the .ofiles, and the tar, rmt, testpad,testpad.h, and core files.If you type make distclean, then make not only removesthe same files as does make clean but also theTAGS, Makefile, and config.status files.(Although it is not evident, this makefile (andconfig.status) is generated by the user with theconfigure program, which is provided in the tardistribution, but is not shown here.)If you type make realclean, then make removes the samefiles as does make distclean and also removes the Info filesgenerated from tar.texinfo.In addition, there are targets shar and dist that createdistribution kits.#!/usr/bin/make -f# Generated automatically from Makefile.in by configure.# Un*x Makefile for GNU tar program.# Copyright (C) 1991 Free Software Foundation, Inc.# This program is free software; you can redistribute# it and/or modify it under the terms of the GNU# General Public License SHELL = /bin/sh#### Start of system configuration section. ####srcdir = .# If you use gcc, you should either run the# fixincludes script that comes with it or else use# gcc with the -traditional option. Otherwise ioctl# calls will be compiled incorrectly on some systems.CC = gcc -OYACC = bison -yINSTALL = /usr/local/bin/install -cINSTALLDATA = /usr/local/bin/install -c -m 644# Things you might add to DEFS:# -DSTDC_HEADERS If you have ANSI C headers and# libraries.# -DPOSIX If you have POSIX.1 headers and# libraries.# -DBSD42 If you have sys/dir.h (unless# you use -DPOSIX), sys/file.h,# and st_blocks in `struct stat'.# -DUSG If you have System V/ANSI C# string and memory functions# and headers, sys/sysmacros.h,# fcntl.h, getcwd, no valloc,# and ndir.h (unless# you use -DDIRENT).# -DNO_MEMORY_H If USG or STDC_HEADERS but do not# include memory.h.# -DDIRENT If USG and you have dirent.h# instead of ndir.h.# -DSIGTYPE=int If your signal handlers# return int, not void.# -DNO_MTIO If you lack sys/mtio.h# (magtape ioctls).# -DNO_REMOTE If you do not have a remote shell# or rexec.# -DUSE_REXEC To use rexec for remote tape# operations instead of# forking rsh or remsh.# -DVPRINTF_MISSING If you lack vprintf function# (but have _doprnt).# -DDOPRNT_MISSING If you lack _doprnt function.# Also need to define# -DVPRINTF_MISSING.# -DFTIME_MISSING If you lack ftime system call.# -DSTRSTR_MISSING If you lack strstr function.# -DVALLOC_MISSING If you lack valloc function.# -DMKDIR_MISSING If you lack mkdir and# rmdir system calls.# -DRENAME_MISSING If you lack rename system call.# -DFTRUNCATE_MISSING If you lack ftruncate# system call.# -DV7 On Version 7 Unix (not# tested in a long time).# -DEMUL_OPEN3 If you lack a 3-argument version# of open, and want to emulate it# with system calls you do have.# -DNO_OPEN3 If you lack the 3-argument open# and want to disable the tar -k# option instead of emulating open.# -DXENIX If you have sys/inode.h# and need it 94 to be included.DEFS = -DSIGTYPE=int -DDIRENT -DSTRSTR_MISSING \ -DVPRINTF_MISSING -DBSD42# Set this to rtapelib.o unless you defined NO_REMOTE,# in which case make it empty.RTAPELIB = rtapelib.oLIBS =DEF_AR_FILE = /dev/rmt8DEFBLOCKING = 20CDEBUG = -gCFLAGS = $(CDEBUG) -I. -I$(srcdir) $(DEFS) \ -DDEF_AR_FILE=\"$(DEF_AR_FILE)\" \ -DDEFBLOCKING=$(DEFBLOCKING)LDFLAGS = -gprefix = /usr/local# Prefix for each installed program,# normally empty or `g'.binprefix =# The directory to install tar in.bindir = $(prefix)/bin# The directory to install the info files in.infodir = $(prefix)/info#### End of system configuration section. ####SRCS_C = tar.c create.c extract.c buffer.c \ getoldopt.c update.c gnu.c mangle.c \ version.c list.c names.c diffarch.c \ port.c wildmat.c getopt.c getopt1.c \ regex.cSRCS_Y = getdate.ySRCS = $(SRCS_C) $(SRCS_Y)OBJS = $(SRCS_C:.c=.o) $(SRCS_Y:.y=.o) $(RTAPELIB)AUX = README COPYING ChangeLog Makefile.in \ makefile.pc configure configure.in \ tar.texinfo tar.info* texinfo.tex \ tar.h port.h open3.h getopt.h regex.h \ rmt.h rmt.c rtapelib.c alloca.c \ msd_dir.h msd_dir.c tcexparg.c \ level-0 level-1 backup-specs testpad.c.PHONY: allall: tar rmt tar.infotar: $(OBJS) $(CC) $(LDFLAGS) -o $@ $(OBJS) $(LIBS)rmt: rmt.c $(CC) $(CFLAGS) $(LDFLAGS) -o $@ rmt.ctar.info: tar.texinfo makeinfo tar.texinfo.PHONY: installinstall: all $(INSTALL) tar $(bindir)/$(binprefix)tar -test ! -f rmt || $(INSTALL) rmt /etc/rmt $(INSTALLDATA) $(srcdir)/tar.info* $(infodir)$(OBJS): tar.h port.h testpad.hregex.o buffer.o tar.o: regex.h# getdate.y has 8 shift/reduce conflicts.testpad.h: testpad ./testpadtestpad: testpad.o $(CC) -o $@ testpad.oTAGS: $(SRCS) etags $(SRCS).PHONY: cleanclean: rm -f *.o tar rmt testpad testpad.h core.PHONY: distcleandistclean: clean rm -f TAGS Makefile config.status.PHONY: realcleanrealclean: distclean rm -f tar.info*.PHONY: sharshar: $(SRCS) $(AUX) shar $(SRCS) $(AUX) | compress \ > tar-`sed -e '/version_string/!d' \ -e 's/[^0-9.]*\([0-9.]*\).*/\1/' \ -e q version.c`.shar.Z.PHONY: distdist: $(SRCS) $(AUX) echo tar-`sed \ -e '/version_string/!d' \ -e 's/[^0-9.]*\([0-9.]*\).*/\1/' \ -e q version.c` > .fname -rm -rf `cat .fname` mkdir `cat .fname` ln $(SRCS) $(AUX) `cat .fname` tar chZf `cat .fname`.tar.Z `cat .fname` -rm -rf `cat .fname` .fnametar.zoo: $(SRCS) $(AUX) -rm -rf tmp.dir -mkdir tmp.dir -rm tar.zoo for X in $(SRCS) $(AUX) ; do \ echo $$X ; \ sed 's/$$/^M/' $$X \ > tmp.dir/$$X ; done cd tmp.dir ; zoo aM ../tar.zoo * -rm -rf tmp.dir

Ginger beer plant how to make. Making ginger beer. Making ginger beer with yeast. How to make your own ginger beer.