remake Documentation

Release 4.3+dbg-1.4

Rocky Bernstein

Sep 05, 2023

Contents:

remake is a fork of and extends GNU Make. It adds, profiling, comprehensible tracing, extended error messages and a debugger

Although debugging GNU Makefiles is a little different than debugging, procedure-oriented programming languages, this debugger tries similar to other trepanning debuggers and *gdb* in general. So knowledge gained by learning this is transferable to those debuggers and vice versa.

An Emacs interface is available via realgud.

CHAPTER 1

Features

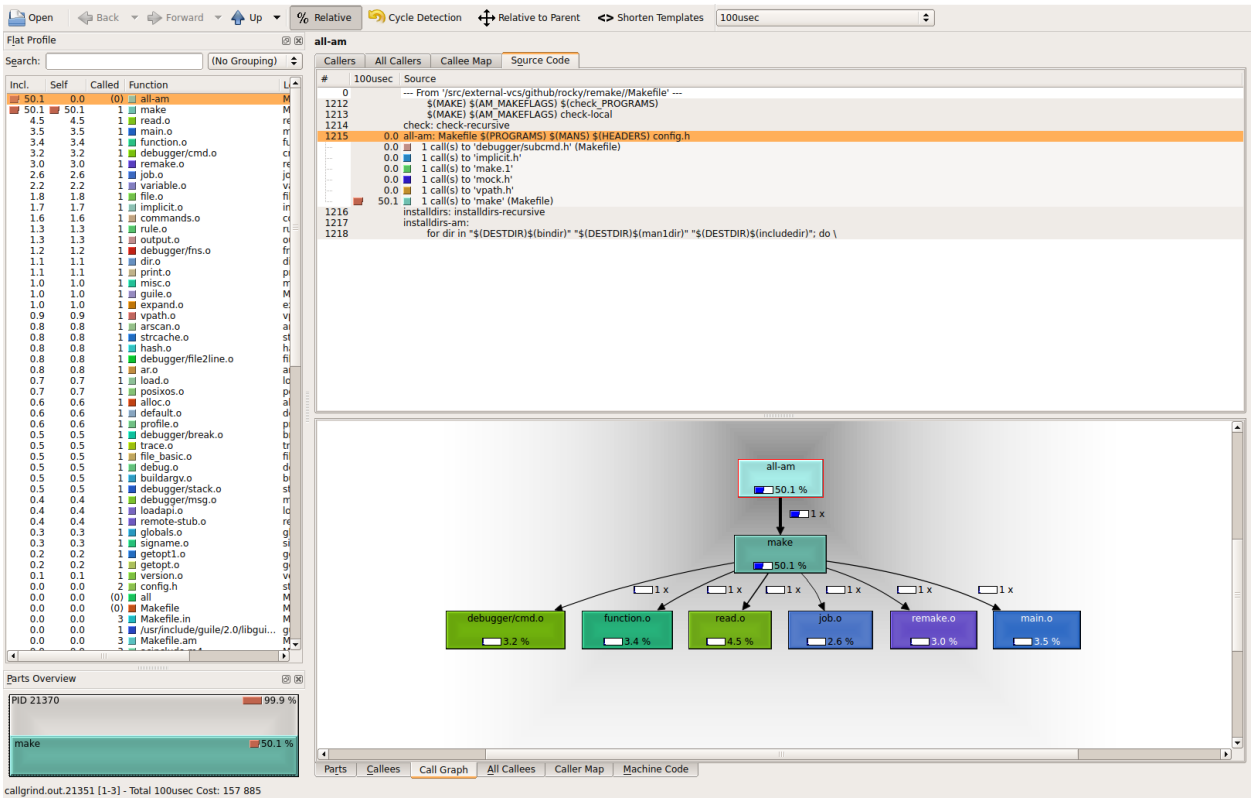
Although debugging GNU Makefiles is a little different than debugging, procedure-oriented programming languages, this debugger tries similar to other trepanning debuggers and *gdb* in general. So knowledge gained by learning this is transferable to those debuggers and vice versa.

1.1 Profiling

If you want to know where most of the time goes in building your system with Makefiles, there is a `--profile` option which times the targets.

This option creates Callgrind Profile Format output which can be read by KCachegrind, callgrind_annotate, or gprof2dot or other tools that understand this format.

You can get not only timings, but a graph of the target dependencies checked. Below is an image rendered from a profiling of a `remake` build:



1.2 Listing and Documenting Makefile Targets

Have you ever wanted `rake tasks` for GNU Make? That is, you have some strange `Makefile` and you want to see the targets, that you can run "make *target-name*" on?

There are two new options added to `remake` to assist this:

- `--tasks` gives a list of targets with `remake` descriptions
- `--targets` gives a list of *all* targets

A target with a `remake` description is just a one-line comment before the the target in the Makefile that describes what the target does and starts with `#:`

If you do this, when either of these options is shown it will also be shown with next to the target name when `--tasks` is run.

Here is an example. Consider this `Makefile`:

```
#: This is the main target
all:
    @echo all here

#: Test things
check:
    @echo check here

#: Build distribution
dist:
    @echo dist here
```

Let's run `remake --tasks`:

```
$ remake --tasks
all                  This is the main target
check                Test things
dist                 Build distribution
```

Many legacy[1] Makefiles don't have descriptive comment in them yet. So you can get a list of *all* targets using option `--targets`. But be warned, since GNU Make comes with lots of implicit rule defaults, this list can be quite large.

Here is an example of runnint `--targets` on the above file:

```
$ remake --targets -f comment.Makefile
          .C
          .C.o
          .DEFAULT
          ... # about 70 more lines!
          all       # This is the main target
          check     # Test things
          Makefile
          dist      # Build distribution
```

1.3 Searching for a Makefile in Parent Directories

When the `-c` flag is given (or `--search-parent`), if a Makefile or goal target isn't found in the current directory, `remake` will search in the parent directory for a Makefile. On finding a parent the closest parent directory with a Makefile, `remake` will set its current working directory to the directory where the Makefile was found.

In this respect the short option `-c`, is like `-C` except no directory need to be specified.

Here is a screenshot that shows `make` behavior versus `remake`:

1.4 Improved Execution Tracing

When the `-x` flag is given (or `--trace=normal`), any commands that are about to be run are shown as seen in the `Makefile` along with `set -x` tracing when run in a POSIX shell. Also, we override or rather ignore, any non-echo prefix `@` directive listed at the beginning of target commands.

If different granularity of tracing is desired the `--trace` option has other settings. See the relevant parts of this manual for more information.

And, if you the most flexibility in tracing there is a built-in debugger.

Here is a screenshot that shows tracing:

1.5 Debugger

Features of the debugger:

[1] As Ryan Davis explains: "legacy code" is any code you didn't write.

- Inspect target properties
- See the current target stack
- Set breakpoints
- Set and expand GNU Make variables
- Load in Makefiles
- write a shell script containing the target commands with GNU Make variables expanded away, so the shell code can be run (and debugged) outside of make.
- Enter debugger at the outset, call it from inside a Makefile, or enter it upon the first error

See *debugger* for more information on the built-in debugger.

1.6 For Developers

If you are interested in learning about how GNU Make works, you might find it easier to start out working with this code.

First, some Doxygen comments have been added.

Second, it has been simplified as a result of the removal of lesser-used OS's (from the standpoint of GNU Make use).

We don't even attempt to support:

- VMS (whether on VAX or the OpenVMS variant)
- DOS (with or without EMX and DJGCC),
- native MS/Windows,
- acornOS
- Amiga,
- OS2
- MINIX,
- RiscOS
- Xenix

This is 2020, not the late 1970-80's. Although GNU make is phasing some of these out, you can find C-preprocessor checks and C code in GNU Make for the above.

By eliminating support for the above, thousands of lines of code in support of the above has been removed.

And the remaining code is easier to read.

Sure, it has annoyed (and still annoys?) those who still work on and develop on the above. I get it. If it is any consolation, there is still GNU Make or GNU Make in older versions for such people.

However the *way* this code has been added makes already difficult-code to read even more difficult.

For example here is GNU Make 4.3 code from *job.c*

```
#if !defined(__MSDOS__) && !defined(_AMIGA) && !defined(WINDOWS32)
        remote_status_lose:
#endif
          pfatal_with_name ("remote_status");
```

(continues on next page)

(continued from previous page)

```
        }
      else
        {
          /* No remote children.  Check for local children.  */
#if !defined(__MSDOS__) && !defined(_AMIGA) && !defined(WINDOWS32)
          if (any_local)
            {
#ifdef VMS
              /* Todo: This needs more untangling multi-process support */
              /* Just do single child process support now */
              vmsWaitForChildren (&status);
              pid = c->pid;

              /* VMS failure status can not be fully translated */
              status = $VMS_STATUS_SUCCESS (c->cstatus) ? 0 : (1 << 8);

              /* A Posix failure can be exactly translated */
              if ((c->cstatus & VMS_POSIX_EXIT_MASK) == VMS_POSIX_EXIT_MASK)
                status = (c->cstatus >> 3 & 255) << 8;
#else
#ifdef WAIT_NOHANG
              if (!block)
                pid = WAIT_NOHANG (&status);
              else
#endif
                EINTRLOOP (pid, wait (&status));
#endif /* !VMS */
```

Can you spot which code is used in the most-often POSIX unixy case? In some cases such as in the above, the most-often case is indented incorrectly because in of one of less-frequent cases it is say in an `else` clause (as appears above).

Note: If you have trouble parsing the above, the Pygments parser used in this document has trouble too. Even after adding mismatched braces in the above for context, I couldn't get Pygments to parse this after specifying C source with C-preprocessor directives. So I gave up, and opted for the slightly shorter source code without some enclosing braces.

I understand how this ugly code hard-to-read code most likely came about in GNU Make. Been there and done that myself too.

In the early days to gain traction and support, a project wants to support lots of different platforms and OS's, even obscure ones. To get going, you'll probably do that in the most expedient way.

But again, that was then and this is now.

If there are folks in the affected communities that would like `remake` added and are willing to code and do the testing, I am open to this. But *it needs to be added in a more modular way than was done in the past*.

Overall, I view this as a plus for developers who would like to extend GNU Make or understand the code.

CHAPTER 2

The Remake Debugger

When there problems in running GNU make, most of the time I can figure out what's wrong by switching to `remake` and looking at its call stack and extended error information.

When that is not sufficient, the `--trace` or `-x` option many times will fill in the gaps.

However there are situations when it is helpful to go deeper. So here and there is a full-fledged debugger built into `remake`.

`remake` can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- *examine* and query things: See the state of variables, see how they got expanded, where targets are defined, and look a the the state of targets
- *stop* at specified places such as targets, or when there is an error. In conjunction with this you can:
- *change* the internal state of things as though the Makefile were written differently
- *experiment* with Makefile code fragments possibly correcting the effects of one bug and go on to discover another.

Although you can use the `remake` debugger to debug Makefiles, it can also be used just as a front-end for learning more about Makefiles and how GNU `make` or `remake` processes a Makefile.

2.1 Entering the Debugger

Contents

2.1.1 Invoking the Debugger Initially

The simplest way to debug your program is to run `remake -X` or `remake --debugger`.

Example Debugger Sessions

In this example we'll use the Makefile from libcdio-paranoia

```
$  remake --debugger
...
Reading makefiles...
Updating makefiles....

-> (/tmp/remake/Makefile:1505)
Makefile: Makefile.in config.status
remake<0>
```

To be continued...

2.1.2 Calling the debugger within the Makefile

Sometimes it is not feasible to invoke the program from the debugger. Although the debugger tries to set things up to make it look like your program is called, sometimes the differences matter. Also the debugger adds overhead and slows down your program.

Another possibility then is to a function call into your Makefile to call the debugger at the spot you want to stop at.

Here is an Example:

```
foo: bar

debug:
      $(debugger "debug target break")

bar:
      $(debugger "first bar command")
      @echo hi

baz: debug
      @echo hello again
```

```
$ remake -f /tmp/foo.Makefile
debugger() function caled with parameter "bar called"
break
:o (/tmp/foo.Makefile:3)
bar
remake<0> where
=>#0  bar at /tmp/foo.Makefile:6
  #1  foo at /tmp/foo.Makefile:1
remake<0> quit
remake: That's all, folks...
```

(continues on next page)

(continued from previous page)

```
$ remake -f /tmp/foo.Makefile baz
debugger() function caled with parameter "debug target break"
break
:o (/src/external-vcs/github/rocky/remake/tmp/debugger.Makefile:3)
debug
remake<0> where
=>#0  debug at /tmp/foo.Makefile:3
  #1  baz at /tmp/foo.Makefile:10
remake<0>
```

2.1.3 Entering the debugger when `remake` encounters an error

This is done by supplying the `--post-mortem` or `-!` option on invocation.

Note that in contrast to the situations above. although you can examine state and evaluating expressions, execution has terminated. Therefore, some of the execution-specific commands are no longer applicable.

2.2 Sample Debugger Sessions

2.2.1 An Extended Debug Session

In this session we will go into the debugger initially using the `--debugger` or `-X` option. We'll use the Makefile from the source code from cd-paranoia

Basic Information when stopped inside Debugger

```
$ remake -X
Reading makefiles...
Updating makefiles...
-> (/tmp/libcdio-paranoia/Makefile:428)
Makefile: Makefile.in config.status
remake<0>
```

The line immediately before the prompt `remake<0>`, we show the the target name, `Makefile` and its dependencies: `Makefile.in` and `config.status`.

The line before that has position information `(/tmp/libcdio-paranoia/Makefile:428)`. But at the beginning of the line is and arrow made up of two characters, `->`. This indicates that we have not done prerequisite checking for this target yet. Later we will come across other two-character icons like `++`. See *icons* for a complete list.

The zero in the prompt `remake<0>` is the command history number. If GNU Readline history support has it increments as we enter commands, otherwise it stays zero.

For each recursive call to `remake`, we'll add another pair of angle brackets <> around the number.

Some of the information is given in more verbose format using *info program*:

```
remake<0> info program
Starting directory `/tmp/libcdio-paranoia'
Program invocation:
    remake/make  -X
Recursion level: 0
Line 428 of "/tmp/libcdio-paranoia/Makefile"
Program stopped before rule-prequisite checking.
remake<1>
```

Notice that the prompt has incremented to 1 after entering the a command.

Stepping

We can use the *step,* command to progress a little in the interpretation or execution of the makefile:

```
remake<1> step
-> (/tmp/libcdio-paranoia/Makefile:415)
Makefile.in: Makefile.am m4/ld-version-script.m4 ...
remake<2> step
-> (/src/external-vcs/github/rocky/libcdio-paranoia/Makefile:443)
aclocal.m4: m4/ld-version-script.m4 ...
remake<3>
```

I have elided the list of dependencies listed above and substituted ellipses (`...`).

There is a slight difference between what you will find in the Makefile and the target output seen above. Below I'll list the what is in the Makefile versus what is line as shown above.

For line 415:

```
$(srcdir)/Makefile.in:  $(srcdir)/Makefile.am  $(am__configure_deps)
Makefile.in: Makefile.am m4/ld-version-script.m4 ...
```

while line 443:

```
$(ACLOCAL_M4):  $(am__aclocal_m4_deps)
aclocal.m4: m4/ld-version-script.m4 ...
```

In the debugger, variables have been expanded and file paths have been canonicalized. Therefore you see:

Makefile	remake output
$(srcdir)/Makefile.in	*Makefile.in*
$(ACLOCAL_M4)	*aclocal.m4 ...*

Let's recap where `remake` is in the process of running the Makefile. The first thing that seems to be done is that the `Makefile` dependencies need to checked. A dependency of `Makefile` is `Makefile.in` and that in turn depends

on target `aclocal.m4`. We have now stepped into and stopped at that target. At the `remake<3>` prompt then before checking for the dependencies of `aclocal.m4`.

You can see this dependency nesting that got us to this state using the *backtrace* command:

```
remake<3> backtrace
=>#0  aclocal.m4 at /tmp/libcdio-paranoia/Makefile:443
  #1  Makefile.in at /tmp/libcdio-paranoia/Makefile:415
  #2  Makefile at /tmp/libcdio-paranoia/Makefile:428
remake<4>
```

Stepping through the program can be illuminating as far as what is going on, especially when the Makefile has been derived in some way, as is the case here. This Makefile was created via `autotools`.

I had assumed that when I run `make` it looks for a default target and runs that. But as we see here, the first thing that goes on is to check to see if the Makefile is being used is itself out of date. If that is the situation, then the Makefile will get recreated and you start again.

However while all of this may be interesting, stepping can be a bit tedious.

In the next section, we talk about *breakpoints* which can get you to where you want to debug faster. To finish this session though use the *quit* command.

```
remake<4> quit
remake: That's all, folks...
```

Stopping with Continue

Let's say I am interested in what goes on when `make dist` is run. Again, I'll invoke the debugger initially.

```
$ remake -X
Reading makefiles...
Updating makefiles...
-> (/tmp/libcdio-paranoia/Makefile:428)
Makefile: Makefile.in config.status
remake<0>
```

Instead of stepping we can set a breakpoint on the `dist` target and continue running to that point in one command, using *continue*.

```
remake<0> continue dist
Breakpoint 1 on target `dist', mask 0x0f: file Makefile, line 703.
Updating goal targets...
-> (/src/external-vcs/github/rocky/libcdio-paranoia/Makefile:703)
dist:
remake<1>
```

Now when I issue a `step`, I will step into the commands associated with the `dist` target:

```
remake<1> step
File 'dist' does not exist.
Must remake target 'dist'.
Makefile:704: target 'dist' does not exist
##>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
remake  dist-bzip2 dist-gzip am__post_remove_distdir='@:'
##<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
++ (/src/external-vcs/github/rocky/libcdio-paranoia/Makefile:703)
```

(continues on next page)

(continued from previous page)

```
dist
remake<2>
```

Notice that the event icon above is ++ which means I am stepping shell commands, here those associated with the Make target `dist`. Above the line with the event icon in between the two chevrons is the command that is *about* to be run.

To see the entire build commands, there is the *list* command. Here is that:

```
remake<2> list
/tmp/libcdio-paranoia/Makefile:705
dist:
#  recipe to execute (from 'Makefile', line 706):
    $(MAKE) $(AM_MAKEFLAGS) $(DIST_TARGETS) am__post_remove_distdir='@:'
    $(am__post_remove_distdir)
```

A form of the *target* command, *target @ command* does about the same thing. Note that in both cases variables are not expanded as the are in the trace output shown above between chevrons.

2.2.2 Debugging Make Variables

In the above session we have seen that output has variables expanded, while in the `list` and `target` commands variables were not expanded.

You can query any GNU Make variable that has been set in the program *without* variables inside expanded using the *print* command.

```
remake<2> print MAKE
(origin default) MAKE = $(MAKE_COMMAND)
```

The `(origin default)` means this is a built-in definition. Many variables that you will be interested in though, are set somewhere, and the variable is not a default it's location is also shown:

```
remake<3> print DATA
Makefile:168 (origin: makefile) DATA := libcdio_paranoia.pc libcdio_cdda.pc
```

The other kind of print which does full expansion of the variables is called `expand` or `x`. Here is an example

```
remake<4> expand MAKE
(origin default) MAKE := remake
```

Note that in printing expanded values we use `:=` while non-expanded values we use `=` This output matches the semantics of these assignment operators.

In fact, `expand` doesn't need a variable name, it will work with a string. For example:

```
remake<5> x $(MAKE) $(DIST_TARGETS)
remake dist-bzip2 dist-gzip
```

No location identification is given here since what I put in isn't a variable. Also note that for `expand` I add the dollar sign and parenthesis when there is other stuff. If you just want information about the variable you can leave that off.

However for `print` you *never* add the dollar sign; printing only prints *variables* not strings.

You can change values too using either the *set*, *set* or *setqx* commands. Let's see the difference between `set` and `setq`:

```
remake<6> set MAKE $(MAKE_COMMAND)
Variable MAKE now has value 'remake'
remake<7>  setq MAKE $(MAKE_COMMAND)
Variable MAKE now has value '$(MAKE_COMMAND)'
```

So with `set`, the value in the expression `$(MAKE_COMMAND)` is expanded before the variable definition is assigned. With `setq` the internal variables are kept unexpanded. Which you use or want is up to you.

Note the irregular syntax of `set` and `setq`. Don't put an equal sign between the variable and the expression. That is, `set MAKE = $(MAKE_COMMAND)` gives:

```
remake<8> set MAKE = $(MAKE_COMMAND)
Variable MAKE now has value '= remake'
```

which is probably not what you want. You can optionally put in the word "variable" when using `set` and "variable" is ignored. But it won't be if you use `setq`.

2.2.3 Debugging POSIX Shell Commands

Now consider the following sample Makefile `test2.mk`:

```
PACKAGE=make

all: $(PACKAGE).txt

$(PACKAGE).txt: ../doc/remake.texi
    makeinfo --no-headers $< > $@
```

Running this entering the debugger initially:

```
$ remake -X -f test2.mk
...
Reading makefiles...
updating makefiles....
Updating goal targets....
  /tmp/remake/src/test2.mk:3        File `all' does not exist.

-> (/tmp/test2.mk:5)
make.txt: ../doc/remake.texi
```

We could use the *target* command to show information about the current target, but that returns lots if information. So let us instead narrow the information to just the automatic variables that get set. The following commands do this are all mean the same thing: *target make.txt variables*, *target @ variables*, and *info locals*.

```
remake<1> target @ variables
@ := all
% :=
* :=
+ := make.txt
| :=
< := all
^ := make.txt
? :=
```

There is a `target` option to list just the shell commands of the target:

```
remake<2> target @ commands

make.txt:
#  commands to execute (from `test2.mk', line 6):
    makeinfo --no-headers $< > $@
```

We can see a full expansion of the command that is about to be run:

```
remake<5> target @ expand

#  commands to execute (from `test2.mk', line 6):
    makeinfo --no-headers $< > $@

#  commands to execute (from `test2.mk', line 6):
    -makeinfo --no-headers ../doc/remake.texi > make.txt
```

Now if we want to write out commands as a shell script which we might want to execute, we can use the *write* command:

```
(/tmp/remake/src/test2.mk:6): make.txt
remake<6> write
File "/tmp/make.txt.sh" written.
```

We can issue a shell command `cat -n /tmp/make.txt.sh` to see what was written. See *shell*.

```
remake<7> shell cat -n /tmp/make.txt.sh
#!/bin/sh
# cd /tmp/remake/src/
#/tmp/remake/src/test2.mk:5
makeinfo --no-headers ../doc/remake.texi > make.txt
```

If you issue step commands, the debugger runs the each command and stops. In this way, you can inspect the result of running that particular shell command and decide to continue or not.

```
remake<8> step

  Must remake target `make.txt'.
Invoking recipe from test2.mk:6 to update target `make.txt'.
##>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
makeinfo --no-headers ../doc/remake.texi > make.txt
##<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

++ (/tmp/test2.mk:5)
```

Notice that we've shown the expansion automatically. One subtle difference in the above output, is that we only show the *single* shell command that is about to be run when there are several commands. In our example though, there is only one command; so there is no a difference.

The ++ icon means that we are about to run that code.

```
make.txt
remake<9> @b{step}
  Successfully remade target file `make.txt'.

<- (/tmp/test2.mk:5)
make.txt
remake<10>
```

We ran the code, and are still at target `make.txt`. The `<-` icon means that have finished with this target and are about to return.

If you are at a target and want to continue to the end of the target you can use the command `finish` which is the same as `finish 0`.

2.2.4 Post-Mortem Debug Session

In this session we'll go into the debugger on encountering an error. For this the `--post-mortem` or `-!` option is used. We'll use the Makefile from the source code of this distribution.

to be continued...

2.3 Debugger Command Syntax

Command names and arguments are separated with spaces like POSIX shell syntax. Parenthesis around the arguments and commas between them are not used. If the character of a line starts with `#`, the command is ignored. (Actually, what is going on here is that it is a "comment" command.)

Within a single command, tokens are then white-space split. Again, this process disregards quotes or symbols that have meaning in GNU Make. Some commands like *expand*, have access to the untokenized string entered after the command name.

Resolving a command name involves possibly 2 steps. Some steps may be omitted depending on early success or some debugger settings:

1. The leading token is next looked up in the debugger alias table and the name may be substituted there.

2. After the above, The leading token is looked up a table of debugger commands. If an exact match is found, the command name and arguments are dispatched to that command.

GNU Readline is used to read commands, so it's capabilities are available, such as `vi` or `emacs` editing.

2.3.1 Event Icons

In the debugger, before showing position information there is a two-character event icon.

For example, in his line:

```
!! (/tmp/project/errors.Makefile:1)
^^ event icon is here
```

The `!!` indicates an error occurred and we have gone into post-mortem debugging.

Here is a list of event icons:

Icon	Event
`->`	Stopped before checking target prerequisites.
`..`	Stopped after checking target prerequisites.
`<-`	Stopped after running target commands.
`rd`	About to read a Makefile
`!!`	Error encountered and `--post-mortem flag given`. In post-mortem debugging.
`- -`	Ran a debugger step of a Makefile target and it's not one of the above.
`++`	Ran a debugger step in a POSIX command and it's not one of the above.
`:o`	A call to the debugger using the $(debugger) function in the Makefile
`||`	Finished making the goal target

2.4 Debugger Commands

Following *gdb* and the other trepanning debuggers, we classify commands into categories. Note though that some commands, like `quit`, and *run*, are in different categories and some categories are new, like `set`, `show`, and `info`.

2.4.1 Breakpoints (*break*, *delete*)

A *breakpoint* causes *remake* to stop when it reaches a Makefile target.

The debugger assigns a number to each breakpoint when you create it; these numbers are successive integers starting with 1. In many of the commands for controlling various features of breakpoints you use this number.

The debugger assigns a number to each breakpoint when you create it; these numbers are successive integers starting with 1. In many of the commands for controlling various features of breakpoints you use this number.

Set a breakpoint (*break*)

break {*target* | *line-number*} [**all** | **run** | **prereq** | **end**]*

break

Set a breakpoint at a target or line number; also show breakpoints.

With a target name or a line number, set a break before running commands of that target or line number. Without argument, list all breakpoints.

For a given target, there are 3 places where one may want to stop at; that name can be given as a last option. The stopping points are:

- before target prerequisite checking: *prereq*
- after target prerequisite checking but before running commands: *run*
- after target is complete: *end*

Giving *all* will stop in all of the above places. The default behavior is *run*.

Examples:

```
break                # list all breakpoints
break 10             # Break on line 10 of the Makefile we are
                     # currently stopped at
break tests          # Break on the "tests" target
break tests prereq   # Break on the "tests" target before dependency checking is done
```

See also:

delete.

Remove breakpoints (*delete*)

delete [*bpnumber* [*bpnumber*...]]

Delete some breakpoints.

Arguments are breakpoint numbers with spaces in between. To delete all breakpoints, give no argument.

2.4.2 Examining data (*print*, *target*, *expand*, *write*)

Print Variable Information (*print*)

print [*variable*]

Expand *remake* variables.

Variable names should *not* be preceded with a dollar sign.

Note however that a more versatile print command is *examine* which can print arbitrary string expands which of course includes variable.

If you omit *variable*, the last expression again is displayed again..

Examples:

```
remake<0> print SHELL
Makefile:168 (origin: makefile) SHELL = /bin/sh

/tmp/remake/Makefile:243: Makefile.in
remake<1> print $MAKE   # don't use $
Can't find variable $MAKE

/tmp/remake/Makefile:243: Makefile.in
remake<1> print shell   # note case is significant
Can't find variable shell
```

See also:

expand, *info variables* <*info_variables*>.

Examining Targets (*target*)

target [*target-name*] [*info1* [*info2*...]]

Show information about a *target-name*.

target-name be the name of a target or it can be a variable like @ (the current target) or < (first dependency). If *target-name* is omitted use the current target.

When *remake* enters the debugger, *remake* spontaneously prints the line and target name that is under consideration and the location of this target. Likewise, when you select a target frame, the default target name is changed.

The following attributes names can be given after a target name:

- **attributes: rule attributes**
  - precious,
  - rule search,
  - and pattern stem
- commands: shell commands that need to be run to update the target
- depends: all target dependencies, i.e. order and non-order
- expand: like 'commands', but Makefile variables are expanded
- nonorder:" non-order dependencies", i.e. dependencies that are not ordered
- order: "order dependencies", i.e. dependencies that have to be run in a particular order
- previous: previous target name when there are multiple double-colons
- **state: target status:**
  - successully updated
  - needs to be updated
  - failed to be udated
  - invalid, an error of some sort occurred
- time: last modification time and whether file has been updated. If the target is not up to date you will see the message "File is very old." If a target is "phony", i.e. doesn't have file associated with it, the message "File does not exist." will appear instead of the time. In some cases you may see "Modification time never checked."
- variables: automatically set variables such as @ or <

Note that there are other automatic variables defined based on these. In particular those that have a *D* or *F* suffix, e.g. *$(@@D)*, or *$(*F)*. These however are not listed here but can shown in a *print* command or figured out from their corresponding single-letter variable name.

See also:

print.

Expand a Makefile string (*expand*)

expand *string*

Expands the string *string* given using *remake*'s internal variables. The expansion would be the same as if the string were given as a command inside the target.

Example:

```
remake<0> expand MAKE
(origin default) MAKE := /tmp/remake/src/./make


/tmp/remake/src/Makefile:264: Makefile.in

remake<1> print MAKE  # note the difference with the print
(origin default) MAKE = $(MAKE_COMMAND)

remake<2> expand $(MAKE)    # Note using $( ) doesn't matter here...
/tmp/remake/src/./make      # except in output format - no origin info

/tmp/remake/src/Makefile:264: Makefile.in

remake<2> p COMPILE
Makefile:104 (origin: makefile) COMPILE := $(CC) $(DEFS) $(DEFAULT_INCLUDES)

/tmp/remake/src/Makefile:264: Makefile.in
remake<10> @b{x compile starts: $(CC) $(DEFS) $(DEFAULT_INCLUDES)}
compile starts: gcc -DLOCALEDIR=\"\" -DLIBDIR=\"/usr/local/lib\" -DINCLUDEDIR=\"/usr/
↪local/include\" -DHAVE_CONFIG_H -I. -I..
```

See also:

print, *info variables* <*info_variables*>.

Write the Commands of a Target (*write*)

write [*target* [[*filename*|***here**]]]

Use this to write the command portion of a target with *remake's internal variables expanded. If a filename is given that is the file where the expanded commands are written. If the filename is 'here* then it is not written to a file but output inside the debugger as other debugger commands behave. And if no file name is given a filename based on the target name is created.

Examples:

```
$ remake -X -f tests/spec/example/simple.Makefile
Reading makefiles...
Updating makefiles....
Updating goal targets....
-> (/tmp/remake/tests/spec/example/simple.Makefile:2)
all:
remake<0> write
File "/tmp/all.sh" written.
remake<1> w all here
#!/bin/sh
## /src/external-vcs/github/rocky/remake/tests/spec/example/simple.Makefile:2
## all:

#cd /src/external-vcs/github/rocky/remake
echo all here
```

2.4.3 Specifying and examining files (*edit*, *list*, *load*)

Edit Makefile (*edit*)

edit

Edit Make at the current target location.

The editing program of your choice is invoked with the current line set to the active line in the program.

You can customize to use any editor you want by using the *EDITOR* environment variable. The only restriction is that your editor, .e.g. *ex*, recognizes the following command-line syntax:

```
ex +*number* *filename*
```

The optional numeric value *+number* specifies the number of the line in the file where to start editing. For example, to configure *remake* to use the *emacs* editor, you could use these commands with the in a POSIX shell:

```
EDITOR=/usr/bin/emacs
export EDITOR
remake ...
```

See also:

list

List Makefile target (*list*)

list [*target*]

list *line-number* | **-**

List target dependencies and commands for *target* or *line-number*

Without a target name or line number, use the current target. A target name of - will use the parent target on the target stack.

Examples:

```
remake<0> list
/tmp/remake/tests/spec/example/simple.Makefile:2
all:
#  recipe to execute (from '/tmp/remake/tests/spec/example/simple.Makefile', line 3):
    echo all here

remake<1> list -
** We don't seem to have a parent target.
```

See also:

ref:*target* <*target*>, *edit*.

Read and Evaluate Makefile (*load*)

load *file-glob*

Read in and evaluate GNU Makefile *file-glob*..

file-glob should resolve after glob expansion to single GNU Makefile. Target dependencies are updated after reading in the file.

Here are several possible uses of this command.

In debug sessions you can fix the source code and the run *load* to have the code reread in, to test out the fix.

Another use is to have pecific "debug"-oriented Makefiles that aren't normally used, but when you want to trace things are avialable. This is an aspect of aspect-oriented programming

2.4.4 Information from the Debugged Session (*break*, *files*, *line*, *program*, *rules*, *target*, *targets*, *tasks*, *variables*)

info [*info-subcommand*]

Gets various pieces of information about the program being debugged.

You can give unique prefix of the name of a subcommand to get information about just that subcommand.

Type *info* for a list of info subcommands and what they do. Type `help info` for a summary list of info subcommands.

List all Breakpoints (*info break*)

info break

Show status of user-settable breakpoints.

The columns in a line show are as follows:

- The "Num" column is the breakpoint number which can be used in a *delete* command.
- The "Disp" column contains one of "keep", "del", the disposition of the breakpoint after it gets hit.
- The "mask" at which points within the target that we stop
- The "Where" column indicates where the breakpoint is located.

Example:

```
remake<1> info break
Num Type           Disp Enb Mask Target  Location
  1 breakpoint     keep   y 0x07 help at /tmp/remake/docs/Makefile:12
```

Show breakpoints.

See also:

break, *delete*

Show Read-in Files (*info files*)

info files

Show read-in Makefiles. The last is the one initially named.

See also:

Read and Evaluate Makefile (load)

Show Target Stack Frame (*info frame*)

info frame

Show target-stack frame.

See also:

backtrace

Show the Current Line (‘info line)

info line

Show information about the current line.

Example:

```
remake<1> info line
Line 12 of "/tmp/remake/docs/Makefile"
```

See also:

info program, *info target*

Show Makefile Information (*info program*)

info program

Show program information and why we are stopped

- Reason the program is stopped.
- The next line to be run

Example:

```
zshdb<1> info program
Program stopped.
It stopped after being stepped.
Next statement to be run is:
[ "${PS1-}" ]
```

See also:

info line.

Show Implicit or Pattern Rules (*info rules*)

info rules [**verbose**]

Show implicit or pattern rules.

Add *verbose* if you want more info.

Show Target Name *info target*)

info target

Show current target name.

See also:

target.

Show Targets found in Makefiles (*info targets*)

info targets [**names** | **positions** | **tasks** | **all**]

Show the explicitly-named targets found in read Makefiles.

Suboptions are as follows: * *names*: shows target names, * *positions*: shows the location in the Makefile * *all*:shows names and location * *tasks*: shows target name if it has commands associated with it

Example:

```
remake<1> info targets
            .C
            .C.o
            ...
Makefile:15:
            .PHONY
            .S
            ...
```

See also:

info tasks, *target*, and *info target*.

Show Targets with Descriptions (*info tasks*)

info tasks

Show the targets that have descriptions.

A Description comment is a single line before a target that starts *#:*

Example:

```
#: This is the main target
all:
    @echo all here

#: Test things
check:
    @echo check here

#: Build distribution
dist:
    @echo dist here
```

```
remake<0> info tasks

all                  This is the main target
check                Test things
dist                 Build distribution
```

See also:

info target.

List all Variables (*info variables*)

info variables

List all POSIX-shell environment and GNU Make variables.

Each variable is shown on a single line and is preceded by a line which indicates the type of variable. This list of types is:

- automatic: variables which have values computed afresh for each rule
- default: a variable using its default value
- environment: a POSIX shell environment variable
- pattern-specific

At the end of the list, hash table statistics are shown.

Example:

```
# 'override' directive
GNUMAKEFLAGS :=
# automatic
<D = $(patsubst %/,%,$(dir $<))
# automatic
@D = $(patsubst %/,%,$(dir $@))
# default
.SHELLFLAGS := -c
# default
LD = ld
# environment
PATH = /usr/bin/:/sbin/
...
# variable set hash-table stats:
# Load=214/1024=21%, Rehash=0, Collisions=35/240=15%
```

See also:

print, *expand*.

2.4.5 Interfacing to the OS (*cd*, *pwd*, *shell*)

Set the Current Working Directory (*cd*)

cd *dir*

Set the working directory to *dir*.

Changing this changes will the working directory in any subsequent build commands that are invoked.

See also:

pwd

Print POSIX Working Directory (*pwd*)

pwd

Print working directory.

By default, this is the working directory for in any commands that run from the build.

See also:

cd

Run a POSIX Shell Command (*shell*)

shell *string*

If you need to execute occasional shell commands during your debugging session, there is no need to leave or suspend *shell*; you can just use the *shell* command or its alias *!*.

2.4.6 Run-Changing Commands (*continue*, *finish*, *next*, *quit*, *run*, *skip*, *step*)

Running, restarting, or stopping the program.

When a program is stopped there are several possibilities for further program execution. You can:

- terminate the program inside the debugger
- restart the program
- continue its execution until it would normally terminate or until a breakpoint is hit
- step exection which is runs for a limited amount of code before stopping

Continue Program execution (*continue*)

continue [*target* [**all** | **run** | **prereq** | **end**]*]

Continue executing debugged Makefile until another breakpoint or stopping point. If a target is given and valid we set a breakpoint at that target before continuing.

As with the *break* command, the place in a target is in can be specified. See *break* for a list of the meanings of the target phases.

Example:

```
continue          # Continue execution
continue dist     # Continue until the "dist" target is reached
```

See also:

next *skip*, and *step* provide other ways to progress execution.

Step out (*finish*)

finish [*count*]

Run to the completion of the target, which is also known as "step out".

With no arguments, *remake* runs to the end of the target. Any prerequisite checking and building that needs to occur is done and any shell commands that occur get run.

This is analogous to "step out" in programming-language debuggers.

When *count* is a positive number, run *finish count* additional times. The default value is 0.

See also:

skip, *continue*, and *next* provide other ways to progress execution.

Step over (*next*)

next [*count*]

Continue processing your Makefile until control reaches the next interesting target, then stop and return control to the debugger.

Argument *count* means do this *count* times or until there's another reason to stop.

See also:

If you want more fine-grained stepping use *skip*.

If you want to not stop at any of targets the current target depends on, but instead run until after this target is remade, *finish*.

skip, and *continue*, provide other ways to progress execution.

Gentle termination or *remake* (*quit*)

quit [*exit-code*]

Exit *remake*. If a numeric argument is given, it will be the exit status reported back. A status of 77 in a nested make will signals termination in the parent. So if no numeric argument is given and *MAKELEVEL* is 0, then status 0 is set; otherwise it is 77.

The program being debugged is aborted.

See also:

run restarts the debugged program.

Restart Program Execution (*run*)

run [*args*]

Run Makefile from the beginning.

You may specify arguments to give it.

With no arguments, uses arguments last specified (with *run*)

See also:

quit for terminating *remake*

Skip target (*skip*)

skip

Skip executing the remaining commands of the target you are stopped at. This may be useful if you have an action that "fixes" existing code in a Makefile.

See also:

next command. *step*, *continue*, and *finish* provide other ways to progress execution.

Step into target (*step*)

step [*count*]

Step execution until the next target is encountered.

Stepping is like *next* but it is more fine-grained. However we still don't stop at targets for which there is no rule.

Argument *count* means do this *count* times (or until there's another reason to stop.

Examples:

```
step        # step 1 event, *any* event
step 1      # same as above
step 2      # same as: step; step
```

See also:

next command. *skip*, *continue*, and *finish* provide other ways to progress execution.

set [*set-subcommand*]

Modifies parts of the debugger environment.

You can give unique prefix of the name of a subcommand to get information about just that subcommand.

Type *set* for a list of set subcommands and what they do. Type `help set` for a summary list of set subcommands.

All of the "set" commands have a corresponding *show* command.

2.4.7 Set (*basename*, *debug*, *ignore-errors*, *keep-going*, *setq*, *setqx*, *silent*)

Modifies parts of the debugger environment. You can see these environment settings with the *show* command.

Basename only in File Path (*set basename*)

set basename [**on** | **off**]

Set short filenames in debugger output.

Setting this causes the debugger output to give just the basename for filenames. This is useful in debugger testing or possibly showing examples where you don't want to hide specific filesystem and installation information.

See also:

show basename

Set GNU Make Debug Mask Debug (*set debug*)

set debug *value*

Set GNU Make debug mask (set via *–debug* or *-d*).

Set to Ignore Make Errors (*set ignore-errors*)

set ignore-errors [**on** | **off** | **toggle**]

Set value of GNU Make *–ignore-errors* (or *-i*) flag.

Set GNU Make *-k* flag (*set keep-going*)

set keep-going [**on** | **off** | **toggle**]

Set value of GNU Make *–keep-going* (or *-k*) flag.

Set GNU Make Variable without Expansion (*setq*)

setq *basename value*

Set GNU Make variable *variable* to *value*.

In contrast to *setqx*, variable definitions inside *value* are *not* expanded before assignment occurs.

See also:

setqx, *set*.

Set GNU Make Variable with Expansion (*setqx*)

setqx *variable value*

Set GNU Make variable *variable* to *value*.

In contrast to *setq*, variable definitions inside *value* are expanded before assignment occurs.

See also:

setq, *set*.

Set GNU Make *-s* Flag (*set silent*)

set silent [**on** | **off** | **toggle**]

Set value of GNU Make *–silent* (or *-s*) flags.

show [*subcommand*]

A command for showing things about the debugger. You can give unique prefix of the name of a subcommand to get information about just that subcommand. nn Type *show* for a list of show subcommands and what they do. Type `help show` for a summmary list of show subcommands. Many of the "show" commands have a corresponding *set* command.

2.4.8 Show (*args*, *basename*, *commands*, *debug*, *ignore-errors*, *keep-going*, *silent*, *version*)

Command-line Invocation (*show args*)

show args

Show command-line invocation.

Status basename-only in Path setting (*show basename*)

show basename

Show whether filename basenames or full path names are shown.

See also:

set basename

See Command History (*show commands*)

show commands

Show the history of commands you typed.

Show GNU Make *-d* flag setting (*show debug*)

show debug

Show the value of the GNU Make *–debug* (or *-d*) flag.

See also:

set debug

Show GNU Make *-i* flag (*show ignore-errors*)

show ignore-errors

Show the value of the GNU Make *–ignore-errors* (or *-i*) flag.

See also:

set ignore-errors

Show the value of the GNU Make *-k* flag (*show keep-going*)

show keep-going

Set value of GNU Make *–keep-going* (or *-k*) flag.

See also:

set keep-going

Show GNU Make *-s* flag (*show silent*)

show silent

Show the value of the GNU Make *–silent* (or *-s*) flag.

See also:

set silent

Show remake version (*show version*)

show version

Show the remake version.

The first number part of the version is the GNU Make base version. After that comes a remake-specific value which indicates its release iteration.

Example:

```
:: show version
version: 4.3+dbg-1.5
```

The above is based on GNU Make 4.3; remake's features are at level 1.5.

2.4.9 Examining the Target Stack (*backtrace*, *frame*, *up*, *down*)

The target stack is made up of targets linked by a dependency from one target on the next. The debugger assigns numbers to target frames counting from zero for the innermost or currently executing target.

At any time the debugger identifies one target as the "selected" target. When the program being debugged stops, the debugger selects the innermost targets. The commands below can be used to select other targets in the target stack by number.

Show Target Stack (*backtrace*)

backtrace [*count*]

Print target stack or Makefile target stack with the most recent frame first. An argument specifies the maximum amount of entries to show.

An arrow at the beginning of a line indicates the 'current frame'. The current frame determines the context used for many debugger commands such as expression evaluation or source-line listing.

Examples:

```
backtrace    # Print a full stack trace
backtrace 2  # Print only the top two entries
```

See also:

up, *down* and *frame*.

Absolute Target Stack Positioning (*frame*)

frame [*number*]

Change the current target to target *number* if specified, or the current target, 0, if no target number specified.

Examples:

```
frame     # Set current frame at the current stopping point
frame 0   # Same as above
frame 1   # Move to frame 1. Same as: frame 0; up
```

See also:

down, *up*, *backtrace*

Relative Target Motion towards a Less-Recent Target (*up*)

up [*count*]

Select and print the immediate child dependency target that is currently under consideration.

If *count* is the default is 1.

See also:

down and *frame*.

Relative Motion towards a More-Recent Target (*down*)

down [*count*]

Select and print the target this one caused to be examined.

If *count* is given then select that many targets down; the default is 1.

When you enter the debugger this command doesn't make a lot of sense because you are at the most-recently frame. However if you issue *down* and *frame* commands, this can change.

See also:

up and *frame*.

2.4.10 Debugger Support Commands (*help*, *source*)

Command Documentation (*help*)

help [*command* [*subcommand*]

Get help for a debugger command or subcommand.

Without an argument, print the list of available debugger commands.

When an argument is given, it is first checked to see if it is command name.

Some commands like *info*, *set*, and *show* can accept an additional subcommand to give help just about that particular subcommand. For example *help set basename* give help about the *basename* subcommand of *set*.

Read and Run Debugger Commands from a File (*source*)

source *file-glob*

Read debugger commands from the glob expansion of *file-glob*;

file-glob should resolve after glob expansion to single file.

Examples:

```
source /home/rocky/remake-dbgr.cmds  # absolute path
source ./remake-dbgr.cmds            # relative path
source remake-dbgr.cmds              # relative path - same as above
source ~/remake-dbgr.cmds            # "~" is glob expanded
source ~/[r]emake-dbgr.cmds          # Same as above
source ~/remake-dbgr.* # Includes the above, but is an error if not unique
```

CHAPTER 3

How to install

3.1 From a Package

Repology maintains a list of various bundled *remake* packages. Below are some specific distributions that contain *remake*.

At the time this documentation was built, here is status that they provide:

Check the link above for more up-to-date information.

3.1.1 Debian/Ubuntu

On Debian systems, and derivatives, *remake* can be installed by running:

```
$ sudo apt-get install remake
```

The latest version may not yet be included in the archives. If you are running a stable version of Debian or a derivative, you may need to install *remake* from the backports repository for your version to get a recent version installed.

3.1.2 MacOSX

On OSX systems, you can install from Homebrew or MacPorts.

```
$  brew install remake
```

3.2 From Source

3.2.1 SourceForge

Go to sourceforge and find the most recent version and download a tarball of that.

```
$ tar -xpf remake-xxx.tar.bz2
$ cd remake-xxx
$ ./autogen.sh
$ make && make test
$ make install # may need sudo
```

3.2.2 github

Many package managers have back-level versions of this debugger. The most recent versions is from the github.

Prerequisites

To build from sources you need:

- a previous version of remake or GNU make
- A C compiler like gcc or clang
- gettext
- GNU _Readline

Optionally you may want:

- Guile version 2.0 or greater

Additionally if installing from git you need:

- *git* (duh)
- *autoconf*
- *automake*

- *autopoint*
- *gettext* to process the language-customizaton files in the *po* director

and optionally:

- gzip and lzip (to compress the tarball)

Here is a *apt-get* command you can use to install on Debian-ish systems:

```
$ sudo apt-get install git gcc pkg-config autoconf automake autopoint gettext␣
↪libreadline-dev make guile-2.0 texinfo lzip
```

Here is a *yum*/*dnf* command for Redhat/CentOS:

```
$ sudo yum install git gcc pkgconfig autoconf automake gettext readline-devel make␣
↪guile lzip

# on CentOS 7 and later, autopoint is part of gettext-devel
$ sudo yum install git gcc pkgconfig autoconf automake gettext gettext-devel readline-
↪devel make guile lzip
```

Here is a *pkg* command for FreeBSD:

```
$ sudo pkg install git gcc pkgconf autotools automake gettext gmake readline rsync␣
↪guile2 lzip wget
```

Here is a *pkg_add* command for OpenBSD as, root:

```
$ pkg_add install git pkgconf autoconf-2.69p2 automake-1.16.1 gettext-tools ggrep␣
↪gmake readline rsync-3.1.3 guile2 lzip wget
```

To build documentation you need:

- texinfo

Add that to the *apt-get* or *yum* command above.

Simplified approach

```
$ $SHELL ./autogen.sh
$ make && make check
```

This performs the step below steps up to but not including "Building".

Creating and running configure script

```
$ autoreconf -f -i
$ patch -p0 < po/Makefile.in.in.patch # this step is optional
$ ./configure --enable-maintainer-mode "$@"
$ make po-update
$ (cd doc && make stamp-1 stamp-vti)
```

Updating language-translation text substitutions

After running *configure* run:

```
$ make po-update
```

to pull in the latest translation strings.

TeXinfo mess

```
$ (cd doc && make stamp-1 stamp-vti)
```

Building

So the full sequence is:

```
$ cd remake*
$ autoreconf -f -i
$ patch -p0 < po/Makefile.in.in.patch # this step is optional
$ ./configure
$ make po-update
$ (cd doc && make stamp-1 stamp-vti)
$ make && make check
$ make install # may need sudo
```

Unbuilding

The main targets to remove *remake* are:

- *uninstall* - removes files created via *make install* or removes installation. Since some files
- *clean* - removes files created via *make* or *make all*
- *distclean* - more aggressively removes any files that are not part of git

Therefore to remove file installed via *make install*:

```
$ make uninstall # ;-)
```

Contents

CHAPTER 4

remake manpage

4.1 Synopsis

remake [*options*] [*target*] ...

4.2 Description

`remake` remake is forked and enahanced version of GNU Make that adds improved error reporting, better tracing, profiling and a debugger.

See GNU Make for information on GNU Make and its use.

4.3 Options

Below we give options that are specific to `remake`. For the other options, please refer to the GNU Make documentation

-c | –search-parent

if a Makefile or goal target isn't found in the current directory, `remake` will search in the parent directory for a Makefile. On finding a parent the closest parent directory with a Makefile, `remake` will set its current working directory to the directory where the Makefile was found.

In this respect the short option *-c*, is like *-C* except no directory need to be specified.

-! | –post-mortem

Go into the debugger on an error. This is the Same as options: *–debugger –debugger-stop=error*

–profile

Creates callgrind profile output. Callgrind output can be used with kcachegrind, callgrind_annotate, or gprof2dot to analyze data. You can get not only timings, but a graph of the target dependencies checked

–targets

Print a list of explicitly named targets found in read-in makefiles.

–tasks

Print a list of explicitly-named targets found in read-in makefiles which have description comments. A description comment is added by putting a single comment before the target that starts with *#:*. For example, for this Makefile:

```
#: This is the main target
all:
    @echo all here

#: Test things
check:
    @echo check here

#: Build distribution
dist:
    @echo dist here
```

Running *remake –tasks* gives:

```
all                 This is the main target
check               Test things
dist                Build distribution
```

-x | –trace

Print debugging information in addition to normal processing.

If *flags* are omitted, then the behavior is the same as if *–trace=normal* was specified

flags can be one of:

- *normal*: basic tracing and shell tracing; this is the default
- *read*: for tracing all Makefiles read in,
- *noshell: which is like 'normal* but shell tracing is disabled
- *full*: for maximum tracing

-X | –debugger [=type]

Enter debugger.

If *type* is given it may be one of:

- *normal*: basic tracing and shell tracing; this is the default
- *goal*: for all tracing Makefiles read
- *preaction* like *normal* but shell tracing is disabled
- *full*: for maximum tracing.
- *fatal*: for entering the debugger on a fatal error. The *–post-mortem* option sets this
- *error*: for entering the debugger on an error.

4.4 Bugs

Since this is derived from GNU Make, it most of its bugs. See the chapter *Problems and Bugs* in "The GNU Make Manual" .

For `remake`-specific bugs see https://github.com/rocky/remake/issues.

4.5 Authors

GNU Make from which `remake` is derived, was written by Richard Stallman and Roland McGrath, and is currently maintained by Paul Smith.

However `remake` is the brainstorm of Rocky Bernstein. The help of others though has been, and is, greatly appreciated. Michael Lord Welles however thought of the name, `remake`.

4.6 Copyright

CHAPTER 5

Indices and tables

- genindex
- search

Index

T

U

W