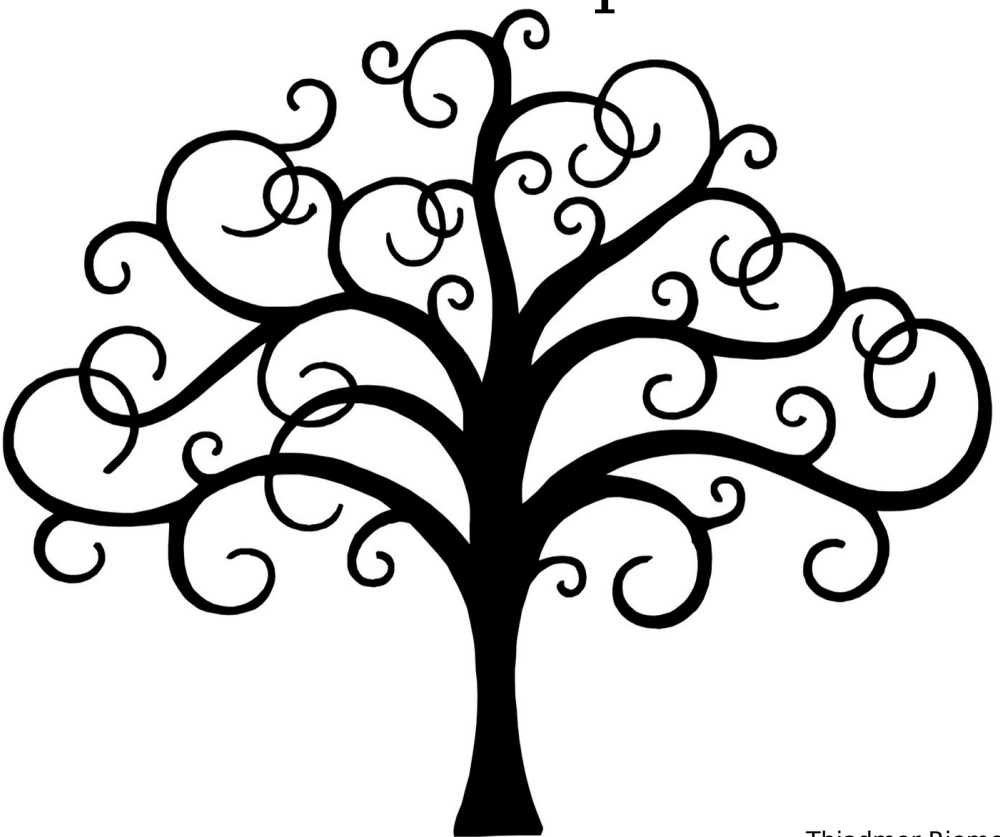
# GNU MAKE+ & makedepend



Thiadmer Riemersma

# Table of Contents

# Introduction

Make+ is a derivative of GNU Make, with modifications to make it more usable as a stand-alone utility (outside the GNU environment), and to enhance cross-platform compatibility. Although all changes are minor, they have implications on how Make behaves and how the makefile syntax subtly changes. I found it therefore more practical to write a new manual than to publish only a list of changes from the original GNU Make.

Make+ is based on GNU Make 4.2. Many of the modifications of Make+ with regards to GNU Make are in fact inspired by OPUS Make version 6.1. OPUS Make has not received updates since 1998. However, the fact that more than 20 years after its release, some of its features only now are merged into another Make implementation, is a testimony of how advanced OPUS Make was in its day.

Stuart Feldman designed and authored the first version of Make in 1976, while at Bell Labs. In 2003, he received the ACM Software System Award for it, with the attribution that "[…] there is probably no large software system in the world today that has not been processed by a version or offspring of MAKE". GNU Make was implemented by Richard Stallman and Roland McGrath in 1988, and it is currently maintained by Paul D. Smith.

By and large, Make+ is compatible with GNU Make. A warning icon in the margin, like the one pictured here, indicates a function of Make+ that is not fully backward compatible with how GNU Make performs it. (New features in Make+ that do not exist in GNU Make are not marked.)

The terms *variable* and *macro* are used interchangeably, but this manual mostly sticks to the term macro. Since it functions as (possibly parameterized) text replacement, I find the term *macro* more fitting than *variable*.

## *License*

Like GNU Make, Make+ is free software, licensed 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.

This manual is copyright Thiadmer Riemersma, and licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

# *Changes from GNU Make*

**Recipes start with an indentation of four spaces**, or a TAB character (or any combination of spaces and TAB characters). See page 19; see also page 33.

**Trailing spaces on lines read from a makefile are stripped**. Note that comments are erased from each line as well, but any spaces before the "**#**" characters are kept. See page 17.

When a comment appears on a continued line, **the comment ends at the continuation**. The following line is not part of the comment. See page 17.

**Comments are allowed inside recipes**, but only full-line comments. Trailing comments are only handled in *makefile* context, not in *rule* context. See pages 17 and 20.

**Descriptive run-time macros are added** so that makefiles are more readable. See page 28.

**There is basic support for file and path names with spaces**. The run-time macros (called "automatic variables" in GNU Make) add a backslash before each space in a filename. A core set of path handling functions support spaces that are escaped with a backslash (see page 36). A new function `$(quote…)` intelligently translates a filename with spaces to a format accepted by shell commands. A new macro modifier for run-time macros is introduced so that handling filenames with spaces is more convenient (see page 29).

**The built-in rules and macros are read from a configuration file**. The default settings of inference rules and macros can therefore be appropriately fine-tuned for a development environment. This approach also allows for project-local, or user-specific configurations (to overrule a global configuration). See page 24.

**The `.path` directive allows to locate target files in sub-directories**. This directive is similar to the `vpath` directive, but where `vpath` is only suitable to locate *source* files, `.path` is also useful to set the location of *target* files to be built. Both the `vpath` and `.path` directives have also been enhanced to match files without extension. See page 45.

**The `ifdef` directive tests whether a macro is defined**. In contrast to GNU Make, in which `ifdef` tests whether a macro is non-empty. See page 43.

**Compatibility with Microsoft Windows is improved**. Some features of GNU Make are actually features of the Unix shell (e.g. bash). For example, the option to concatenate multiple commands on a single line, separated by a semicolon (“;”), is a feature of bash and other Unix shells; GNU Make just passes the complete string to it. The default shell of Microsoft Windows (cmd.exe) does not have this feature though, and therefore Make+ emulates it. See also page 19.

# Overview of the Make Process

Make runs in two phases. In “read phase”, Make reads the makefile, evaluates the macros, applies the directives and builds the dependency tree. After that, in the “run phase”, Make runs over the targets that it may need to build, scans the dependencies of each target, compares time-stamps, and executes the shell lines to create the target from its sources.

## *Terminology*

The Make process is not without its jargon. An overview of some key terms is below.

- There are two distinct phases of Make: **read phase**, when Make reads the files that provide it with project information (called **makefiles**), and **run phase**, when Make builds the project’s targets (see page 7).
- A **target** is a “thing” that can be made. It is usually a file (such as a source file, an object file, or an executable file), and the target is said to *exist* if the file is present on disk. An alternative term used for a target is a **goal**.
- The target’s **time-stamp** is the time that the target was last changed. In practice, it is usually the time that the file was last written (if you save a file without making changes to it, this will typically update the time-stamp). Some targets are *not* represented by a file (such as phony targets) and the time-stamps of these targets are assumed to always be out-of-date.
- A **dependency** is the relationship between a target and the resources needed to make it. For example, an executable file that is linked from a set of object files would need to be re-linked if any of the object files change. In this case, we say that the executable file has a *dependency* on the object files.
- The “resources” that a target depends on are called its **sources**. Note that a “source” is not necessarily a file containing source code. For a target that is an executable file, its sources are typically a set of object files. An alternative term used for sources (to avoid the confusion between a target’s source and a source code file) is **prerequisites**.

- A source may also be a target with its own list of sources. This process repeats recursively, thus forming the **dependency tree**. Any target that does not appear among the sources of any other target, is called a **top-level target**.
- To **make** or **build** a target means: to bring it up to date. Make compares the time-stamps of each target to its sources and runs its **recipe** if any of the sources have a more recent date/time than the target. A recipe is a collection of commands (one or more) that Make runs to build the target from its sources.
- Most of the commands that Make runs to bring a target up to date are **shell lines**. A shell line is a command that is passed unaltered to the *shell* of the operating system, for execution. All invocations of compilers, linkers and other build tools launched by Make are shell lines.
- A **rule** is the combination of a **dependency line** and a recipe. It is an **explicit rule** if both the dependency line and the recipe are explicitly present. The recipe may also be inferred from **inference rules**, which are generalized rules that match on targets and sources via wildcards or file extensions.
- A **macro** associates a *name* and a *value*. In the macro expansion of a macro *name*, the *name* is replaced by the *value*. Macros are defined in *read phase*, and are used in read phase to organize names and path locations of targets, sources and tools, compiler options, etc. In *run phase*, macros also provide information about the current target being built. A macro is alternatively called a **variable**.

## *Read phase*

In read phase, Make parses the command line, and reads the configuration file and one or more makefiles. From this set, it collects macro definitions, rules, dependency lines, and recipes.

Make interprets the lines from the makefile in either of two contexts. The *normal* context (or "*makefile*" context) is used for everything except reading the shell lines. The shell lines are interpreted in "*rule*" context. The two contexts differ in their handling of special characters and directives.

Algorithmically speaking, dependency lines define a set of dependency trees, each with a target node connected by a dependency arc to each of its source nodes. Each source is itself also a target and so is connected, in turn, to its own sources. At the leaf nodes of the trees are the targets without any sources.

## Run phase

In the run phase, Make builds the targets that are specified on the command line or, if none were specified, the first *normal* target in the makefile. With “normal” is meant: not a directive, special target or inference rule. This first normal target is called the *default* target.

In a nutshell, the steps involved in making a target are:

1. **Locate the target**: Look for the target as a file on disk and, if found, get its location and time-stamp. If not found, assume it has a very old time-stamp.

2. **Make its sources**: First make the target’s explicit sources (see page 18).

3. **Locate its inferred source or inferred recipes**: If the rule does not have any shell lines, Make matches the sources and target names to patterns to look up an inferred recipe to make the target (see page 22). If the dependency line also omits an explicit source, Make uses these same inference rules to match a source to the target.

4. **Compare and update the target**: If any of a target’s sources have a newer time-stamp than the target, the target is updated by executing the shell lines. Once the shell lines have been executed, the target is “made” and Make updates its internal copy of the target’s time-stamp.

The consequence of steps 2 and 3 is that Make traverses the dependency tree recursively. Make goes as far as it can down the dependency tree, then works back up checking the time-stamp of each target against its sources, according to step 4. For targets that have at least one newer source, the target’s recipe is run to update the target.

# *Running Make*

When running Make, you can specify the targets to build on the command line, together with macro definitions and options. All of these are optional, if no target is specified on the command line, Make builds the default target; Make has sensible defaults for all options.

When no filename for the makefile is specified on the command line (the `-f` option), Make tries the default filenames `GNUmakefile`, `makefile` and `Makefile`, in that order (on operating systems with case-insensitive filenames, the `makefile` and `Makefile` refer to the same file).

## Setting targets

To set the target to build on Make, you need only to add its name on the command line.

```
make blinky
```

The example above runs Make and instructs it to build the target `blinky`. This overrules the default target. There may be multiple targets on the command line, which are then built in sequential order.

No target names are predefined or mandated, but a convention has formed for a set of standard target names.

`all` Build all top-level targets in the makefile. By convention, `all` is also the default target.

`clean` Delete all intermediate and output files that a normal of Make creates. The common method to rebuild a project completely, is to run `make clean`, and then to run `make all` (but see also the `-B` option on page 10 to rebuild all).

`depend` Update the dependencies of C and C++ files, by scanning those files for the header files in `#include` directives.

`distclean`
The `clean` target does usually not delete platform configuration files, because this would impair a following `make all` from successfully building the project. This target deletes those files as well.

`install`
: Copy the executable files, configuration files and documentation files into the appropriate directories. This target is common for Unix-like platforms, where executables, configurations, documentation and other files have specific locations in the directory tree.

`test`
: Build and run unit tests on the project.

## Defining macros

Macros can be defined on the command line.* One common purpose is that these macros are tested for in the makefile, for example to make a “debug” or a “release” build of the project. Macros set on the command line overrule the ones set in the makefile, which you can use to (temporarily) rebuild the project with different options. For example:

```
make CFLAGS='-g -O'
```

To define a macro on the command line, type the macro name, followed by the “`=`” symbol and optionally followed by the value. If the value is omitted, the macro is defined as empty.

When defining macros on the command line, note that there should be no white-space between the macro name and the “`=`”, and neither should there be white-space between the “`=`” and the value. If the value has embedded white-space, the entire value should be between quotes. These should be single quotes on Unix-like platforms and double quotes on Microsoft Windows.

## Command line switches

`-B, --always-make`
: Build the entire dependency tree for the target(s) to make, skipping the process of comparing time-stamps.

`-C` *`directory`*`, --directory=`*`directory`*
: Run Make from the given directory (let Make set the specified directory as the current directory before continuing to read the makefile).

`-d, --debug, --debug=`*`flags`*
: Prints out the steps that Make performs and the actions that it takes, for the purpose of tracing the execution of a makefile. The `-d` option emits all

* For macros defined in the makefile, see page 26.

output; the `--debug` option (without flags) activates “basic” output. The option with flags allows the flags to be set to `all`, `basic`, `implicit`, `jobs`, `makefile`, `none` and `verbose`.

`-e, --environment-overrides`
: When set, environment variables override the macros set in a makefile (reversing the default precedence order).

`--eval=`*`expression`*
: Run the expression through the `eval` function, before reading any makefiles.

`-f` *`file`*`, --file=`*`file`*`, --makefile=`*`file`*
: Set the name of the makefile to read. A default extension of “.make” or “.mak” is tried on the file if it is not found.

`-h, --help`
: Print brief usage information.

`-i, --ignore-errors`
: Ignore all errors in the recipes that are executed in the process of building the targets.

`-I` *`directory`*`, --include-dir=`*`directory`*
: Set the specified directory where the include directive locates the files. You add this option multiple times to specify multiple include directories.

`-j` *`number`*`, --jobs=`*`number`*
: Set the maximum number of jobs to run in parallel; when omitting the number, the maximum is set to infinite.

`-k, --keep-going`
: Continue making other targets if some target cannot be made.

`-l` *`value`*`, --load-average=`*`value`*`, --max-load=`*`value`*
: Restrict the number of jobs so that the system load stays below the given value.

`-L, --check-symlink-times`
: Take the most recent time of the file versus the symbolic link to that file as the modification time-stamp. Only available on operating systems that support symbolic links.

`-n`, `--just-print`, `--dry-run`, `--recon`
: Print the recipes that would be executed, but do not actually execute them.

`-o` *file*, `--old-file=`*file*, `--assume-old=`*file*
: Assume that the specified file is very old, but yet to not rebuilt it from its sources.

`-O`*type*, `--output-sync=`*type*
: Synchronize output of parallel jobs.

`-p`, `--print-data-base`
: Print the “data base” of rules and macros that Make has built from the makefile (and all included files).

`-q`, `--question`
: Do not execute recipes (like -n), but do not print output either. Instead, set the exit code of Make according to whether targets are out of date.

`-r`, `--no-builtin-rules`
: Purge all built-in rules from the Make “data base”.

`-R`, `--no-builtin-macros`, `--no-builtin-variables`
: Purge all built-in macros from the Make “data base”. This option implies `-r` as well (because the built-in rules contain many references to the built-in macros, these rules are of no much use when the macros are disabled).

`-s`, `--silent`, `--quiet`
: Do not print the shell lines that Make executes on the display.

`-S`, `--no-keep-going`, `--stop`
: Overrule the `-k` option. Since the default of Make is also to *not* keep going, this option is only needed to cancel an inherited `-k` option.

`-t`, `--touch`
: Instead if building the targets, adjust their time-stamps to the current system time.

`--trace`
: Show information on recipes that Make executes, including the line number in the makefile and the reason that the recipe executed.

`-v, --version`
: Print version information of Make and quit.

`-w, --print-directory`
: Print the active working directory at the start and the end of executing the makefile.

`--no-print-directory`
: Overrule the `--print-directory` option.

`-W` *`file`*`, --what-if=`*`file`*`, --assume-new=`*`file`*`, --new-file=`*`file`*
: Pretends that the specified filename has the current system time (ignoring the actual time-stamp of that file). This will cause any target that depends on the specified file to be rebuild. You can set multiple files as “new”, by repeating this option on the command line.

`--warn-undefined-macros, --warn-undefined-variables`
: Issue a warning when the makefile evaluates a macro that has not been set to a value.

`--no-warn-undefined-macros`, `--no-warn-undefined-variables`
: Silently ignore references to undefined macros (this is the default).

# Makefile Syntax

The makefile is a plain text file. The default filename for the makefile is literally `makefile` or `Makefile`, but you can set a different name with the command line option `-f`.

Make+ supports makefiles in the UTF8 encoding, but it actually interprets only the ASCII subset of UTF8.

At its simplest, a makefile contains a set of rules, with a general form as below:

```
target(s) : source(s)
    recipe
```

A target is typically a file, either an existing file or a file that can be built. A target may also represent a goal or action instead of a file on disk; this is referred to as a "phony" target.

A source is a dependency for the target. The source must exist (and be up to date) for the target to be built. Usually a target depends on multiple sources, but a target need depend on any sources at all (this is not uncommon for phony targets).

The recipe is the set of commands that Make must execute to build the target from the sources. Make executes these commands on the command shell of the operating system, which is why these commands are also called "shell lines". A recipe may contain of a single shell line in a recipe, or may consist of multiple shell lines. Each shell line must be indented by a TAB or by four (or more) space characters.

Next to rules, a typical makefile will also contain macro definitions, to avoid repeating the same set of filenames or compiler switches on various lines in the makefile (see page 26). A makefile may also have one or more generalized rules for building a class of targets from a class of sources. These are pattern rules and suffix rules; collectively called inference rules (see page 22). Finally, directives allow for pre-processing, conditionally excluding parts of a makefile, and setting options for search paths or filename matching.

## Walk-through of a simple makefile

A simple program called `blinky.hex` is built from three source files: `blinky.c`, `cmsisBoot.c` and `cmsisSystem.c`. These source files are compiled individually to

object files (`blinky.o`, `cmsisBoot.o` and `cmsisSystem.o` respectively), and then the object files are linked into an executable file. The makefile looks like:

```
1. blinky.hex : blinky.o cmsisBoot.o cmsisSystem.o
2.     armlink -o blinky.elf blinky.o cmsisBoot.o cmsisSystem.o
3.     objcopy --output-target ihex blinky.elf blinky.hex
4.
5. blinky.o : blinky.c
6.     armcc -c -mcpu=cortex-m0 blinky.c
7.
8. cmsisBoot.o : cmsisBoot.c
9.     armcc -c -mcpu=cortex-m0 cmsisBoot.c
10.
11.cmsisSystem.o : cmsisSystem.c
12.    armcc -c -mcpu=cortex-m0 cmsisSystem.c
13.
14.clean :
15.    rm blinky.elf blinky.hex
16.    rm *.o
```

The first dependency line (on line 1) says that `blinky.hex` depends on `blinky.o`, `cmsisBoot.o` and `cmsisSystem.o`. The second dependency line says that `blinky.o` depends on `blinky.c` (line 5), and the third and fourth dependency lines mark dependencies of `cmsisBoot.o` on `cmsisBoot.c`, and `cmsisSystem.o` on `cmsisSystem.c` (lines 8 and 11).

Assume that `blinky.c` is newer than `blinky.o`, that `cmsisBoot.c` is older than `cmsisBoot.o`, and that `cmsisSystem.o` does not exist.

Running `Make` causes the first target in the makefile, `blinky.hex` to be evaluated. This target depends on `blinky.o`, `cmsisBoot.o` and `cmsisSystem.o`, which are then evaluated in order. Diving deepet, `Make` finds that target `blinky.o` depends on `blinky.c`, which is subsequently evaluated. Target `blinky.c` doesn't depend on anything, but it exists and so it is implicitly "up to date". Comparing `blinky.o` to `blinky.c` shows `blinky.c` is newer, so `blinky.o` must be rebuilt. This is done by running the recipe, which is on line 6:

```
    armcc -c -mcpu=cortex-m0 blinky.c
```

At this point, `blinky.o` is up to date.

Next, `cmsisBoot.o` is evaluated. It depends on `cmsisBoot.c`, which is implicitly up to date. Comparing `cmsisBoot.o` to `cmsisBoot.c` shows that `cmsisBoot.o` is

newer, so that it doesn't need rebuilding. At this point `cmsisBoot.o` is also marked "up to date".

Next, `cmsisSystem.o` is evaluated. It depends on `cmsisSystem.c`, which is implicitly up to date. File `cmsisSystem.o` is not found, so the target `cmsisSystem.o` is assumed to be *very old*. So `cmsisSystem.o` must be rebuilt. This is done by running the recipe that is on line 12:

```
    armcc -c -mcpu=cortex-m0 cmsisSystem.c
```

At this point `cmsisSystem.o` is up to date.

Back up the dependency tree, comparing `blinky.hex` against `blinky.o`, `cmsisBoot.o` and `cmsisSystem.o` shows that `blinky.o` and `cmsisSystem.o` are newer (because these have just been rebuilt), so `blinky.hex` is updated with its recipe, consisting of two shell lines that are run in order (lines 2 and 3):

```
    armlink -o blinky.elf blinky.o cmsisBoot.o cmsisSystem.o
    objcopy --output-target ihex blinky.elf blinky.hex
```

In summary: `blinky.c` is recompiled because it was newer than its object module (meaning that the source file had been edited after it was last compiled); `cmsisBoot.c` is *not* recompiled because there is no need to, `cmsisSystem.c` is compiled because its corresponding object module did not exist, and then `blinky.hex` is relinked and converted because the compilation of `blinky.c` and `cmsisSystem.c` changes the time-stamps of their respective object files.

Finally, the `clean` target at the bottom of the makefile does not represent a file, but a "goal" to carry out some specific actions. This target is not a prerequisite of any other target, which means that it won't be build unless is it mentioned on the command line when running Make, for example:

```
make clean
```

Targets that do not refer to files are called *phony* targets. Targets that might exist as a file, but should still be considered a "goal" for some specific action, can explicitly be marked as "phony".

# General syntax

## Continued makefile lines

There is no limit on the length of a line in a makefile. However, for easier reading a long line can be broken up by putting a \ at the end of the line and the rest of the text on the next physical line of the makefile. For example:

```
the quick brown fox jumps over the lazy dog
```

is the same as:

```
the quick brown fox jumps \
over the lazy dog
```

To have a "\" as the last character on a line, escape it by doubling it. For example:

```
the quick brown fox jumps \\
over the lazy dog
```

Note that Make+ trims trailing white-space from the lines that it reads from a makefile.

## Comments

Everything on a makefile line after the comment character # is ignored. Use \# for a literal "#" character.

If a \ appears at the end of a comment line, the comment runs to the end of the line and the line is continued into the next line. For example:

```
the quick brown fox \
ran over # foxes don't jump \
the lazy dog
```

is the same as:

```
the quick brown fox ran over the lazy dog
```

In other words, the comment is limited to "foxes don't jump".

The above snipped shows a trailing comment. Trailing comments are only supported in normal context. In rule context, comments must be on a line of their own ("full line comment").

# *Rules*

A rule in a makefile has two parts: a dependency line and a recipe. The dependency line tells Make what to build and when to build it. The recipe tells Make how to build it. The recipe is a list of shell lines and each shell line is a command, such as a compile or link, that you run to update the target. A recipe may be omitted, in which cases it is inferred from inference rules. If not omitted, it is an *explicit* rule.

## dependency lines

A dependency line indictes a relationship between one or more targets and the resources they depend on. The general form of a dependency line is:

```
test.elf : main.o sub.o
```

This line declares that `test.elf` depends on `main.o` and `sub.o`. When Make runs this line, it compares the time-stamp of `test.elf` against its sources. If `test.elf` is older than either source, it must be updated. Note that an absent target (one that does not exist as a file) is regarded as *very* old.

A dependency line can declare that several targets depend on several sources, such as:

```
main.o sub.o : system.h io.h
```

This means that both `main.o` and `sub.o` depend both on `system.h` and `io.h`. It is equivalent to:

```
main.o : system.h io.h
sub.o  : system.h io.h
```

## Dependency lines can use wild-cards

The sources side of a dependency line can use a wild-card specification, such as:

```
main.o : *.h
```

Although Make provides this feature, its use is discouraged because of the potential for mishap. For example:

```
test.elf : *.o
    ld -o$(.TARGET) $(.SOURCES)
```

This rule will work fine until a required "`.o`" file is accidentally deleted. Then, on the next run, Make won't rebuilt that object module (because it isn't collected on

the dependency line) and the link command will probably fail because of unresolved external symbols.

### The dependency line separator ":"

The ":" between targets and sources on a dependency line is also the character used for the drive separator in DOS, OS/2 and Microsoft Windows. To distinguish this colon from the drive separator, single-letter target names must have a space or tab between the name and the colon.

Although multi-letter target names need no special handling, for consistency it is suggested to always add white-space before and after the ":".

### Recipes

All lines directly following a dependency line and indented with a TAB or four (or more) spaces, form the recipe. The recipe is a list of commands that are executed to update the target, if any of the target's sources are newer than the target. Typically, the shell lines run programs that compile source files, link object files, create libraries, etc.

Typically, there is a single command on each line. You may, however, separate multiple commands on a single line with semicolons ("`;`"). This is useful when the recipe line is generated from a macro expansion. To pass a semicolon to the command line, you have to escape it by prefixing it with a backslash ("`\;`").

A single-line recipe, or the first shell line of a multi-line recipe, may follow on the dependency line after a semicolon ("`;`"). For example, the following two are equivalent:

```
test.elf : test.c ; gcc -o test.elf test.c
```

```
test.elf : test.c
	gcc -o test.elf test.c
```

Make parses the recipe in "rule context", which differs from its normal context. Make supports macro substitution and line continuation in rule context, but other special characters are passed to the shell unmodified. Specifically, trailing comments (behind a command) are *not supported*. A full-line comment (where the first non-white-space character is a "#") is still supported in rule context.

Conditional directives (see page 43) are supported in rule context, but other directives cause an implicit switch back to normal context.

Macro definitions are *not* supported in rule context. However, when a macro definition starts in the first column, the context automatically switches back to normal context. In other words, if macro definitions appear below a recipe, the first of them must start in the first column of the line.

## Empty recipes

An empty recipe is a recipe that does nothing. There are two ways to specify an empty recipe:

```
target : sources ;
```

and

```
target : sources
    # empty
```

In the second variant, the comment character is significant (the text of the comment is not significant). Without a comment, the target would have *no recipe*, and a recipe could then be inferred by matching inference rules to the sources and the target. With the comment, it forms an *explicit rule*, but one that happens to have an empty recipe. Stopping Make from applying an inference rule is a typical reason for an empty recipe.

## Shell-line prefixes

Prefixes control some aspects of a shell line's execution. There may be multiple prefixes on each line, and they prefix the program/command name (hence the name). The first non-prefix character ends the prefixes list. White-space is allowed between the prefixes and the program/command name.

`@` Don't echo shell line (silent operation).
Usually Make echoes (displays) the shell line to the screen before executing it. The "@" prefix prevents display of the line. See also the `.SILENT` directive on a target on page 48.

`-` Ignore the exit status of the shell command.
Normally a shell command that returns a non-zero exit status causes Make to terminate with an error message. The "`-`" causes Make to ignore the exit status and continue its processing.

`+` Ignore "no-execute" command line options, and execute anyway.
This prefix causes the command line options `-t` (`--touch`), `-n` (`--just-`

`print`), or `-q` (`--question`) to be ignored for the shell lines that it occurs on. These command line options normally skip the execution of shell commands.

# *Inference rules*

Inference rules generalize the build process, so that the recipe to build each target, or even its sources, do not need to be specified explicitly each time. As an example, compiling C code (`.c` files) into object modules is a common occurrence. Rather than requiring each rule for an object module to state that it depends on a `.c` file with the same base name, Make uses an inference rule to infer that dependency. The source determined by an inference rule is called the *inferred source*.

Besides providing the inferred dependency of a target on a source, inference rules also provide the recipe for updating the target from the source, and this recipes is inherited by the target if it does not have its own shell lines.

A set of inference rules is predefined (the "built-in" rules, *see page 24*), and you can change their definitions or define your own rules. If you wish that all makefiles use your tailor made inference rules, the best place to put them in is the configuration file.

There are two forms of inference rules:  pattern rules and suffix rules. Pattern rules are the more generalized and flexible form. Suffix rules are present for backward compatibility, but they offer no advantage over pattern rules.

## Defining pattern rules

Pattern rules look like normal explicit rules, but they have the *rule character* "**%**" in the dependency line. The rule character is a wild-card, matching zero or more characters.

As an example, below is a pattern rule that builds a "`.o`" file from a "`.c`" file with the same filename stem:

```
%.o : %.c
    cc -c -mcpu=cortex-m0 -o$(.TARGET) $(.SOURCE)
```

The recipe updates the object module by running the C compiler. The special tokens `$(.TARGET)` and `$(.SOURCE)` on the shell line are run-time macros that contain the filenames of the target and the source respectively, after Make has matched `%.o` to a target and `%.c` to a source (see page 28).

Pattern rules are defined with the following general form, with the additional requirement that both the target and the source specifications must contain a single `%`:

```
target_pattern : source_pattern
    recipe
```

The `%` character need not be at the start of the target and source specifications. In particular, there may be a directory specification before the `%`. A target filename or source filename matches the rule if the characters before and/or after the `%` match the filename, and if the text gobbled up by the `%` is identical for the target and the source side.

If more than one pattern rule matches a target, Make will use the one that where the `%` gobbles up the lowest number of characters (the most specific match); if there are still multiple candidates, Make takes the one that was defined first.

### Suffix rules

A suffix rule that is the equivalent to the pattern rule above is:

```
.c.o :
    cc -c -mcpu=cortex-m0 -o$(.TARGET) $(.SOURCE)
```

The suffix rule lists only the extensions of the source and target (with the source extension first).

### Static pattern rules

Static pattern rules limit the scope of a pattern rule to a list of specific targets. The syntax of a static is:

```
targets : target_pattern : source_pattern
    recipe
```

The *`targets`* list contains one or more target filenames. For example, for a project that is built for a dual Cortex-M4/M0 micro-controller:

```
APPLICATION_OBJECTS := app.o database.o loader.o
SUPERVISOR_OBJECTS  := watchdog.o lowpower.o

$(APPLICATION_OBJECTS) : %.o : %.c
    cc -c -mcpu=cortex-m4 -o$(.TARGET) $(.SOURCE)

$(SUPERVISOR_OBJECTS) : %.o : %.c
    cc -c -mcpu=cortex-m0 -o$(.TARGET) $(.SOURCE)
```

In the above snippet, the first pattern rule will apply to the targets `app.o`, `database.o` and `loader.o`, whereas the targets `watchdog.o` and `lowpower.o` will be

matched only by the second pattern rule. This allows you to set different recipes on pattern rules that have the same patterns for targets and sources.

### Inference rules without a source

If a pattern rule is defined with only a target and not a source, the rule is effectively a general wild-card. It provides a way to associate a recipe on a target, when the stem of the target filename would not match that of its source(s). This happens, for example, in rules that create libraries from a collection of object modules.

### Using inference rules

Inference rules are searched automatically when building a target *that has no recipe*. Make first builds the target's explicit sources (those listed on the dependency line), then uses its inference rules to search for an inferred source. The search proceeds by finding all rules that match the target, determining each possible inferred source name in turn, then checking whether the inferred source exists as a file.

If the inferred source exists as a file:

1. If the target has no recipe, the target inherits the recipe of the inference rule.
2. The inferred source is added to the target as a source, and then it is subsequently built.

## *Built-in rules*

The built-in pattern rules, suffix rules and macro definitions are stored in a configuration file. The file is called `make.conf` (or `make.cfg` in DOS). Make+ locates it, in this order, from the following locations:

- The current directory, from which Make is run.
- The user's "home" directory, or (under Microsoft Windows) the "application data" directory.
- The `/etc` directory for Unix-like operating systems, and in the same directory that the `make` executable is in under Microsoft Windows and DOS.

This hierarchy allows you to create per-project and per-user configurations of built-in rules and macros. To verify which configuration Make+ is using, run Make with

the `--debug` option. The full path to the configuration file is printed below the version information.

The configuration file uses a subset of the syntax for makefiles. Essentially, it supports definitions of pattern rules, suffix rules and macros that are set with the "`=`" or "`:=`" operators.

Each platform or development environment typically has its own set of applicable rules. Make+ comes with configuration files for typical environments. To use one of these configurations, rename the respective file to `make.conf` and place it in the appropriate location. You can use these example files as a basis for new configurations as well.

You can set default command line switches in the configuration file by defining the macro `GNUMAKEFLAGS` with the switches as its value. Any switches given on the command line overrule the ones in the `GNUMAKEFLAGS` macro. For example, to have Make warn you by default for referencing undefined macros (or "variables"), add the following line to the `make.conf`:

```
GNUMAKEFLAGS = --warn-undefined-macros
```

Without a `make.conf` file, Make runs without any built-in rules or macros. This is essentially the same as running Make with options "`-r -R`" (see page 10). The makefile must then define all rules and macros explicitly (or explicitly include a file with the appropriate configuration).

# *Macros*

A macro is the association of a *name* and a text *value*. A macro *reference* is an expression of the form `$(name)` or `${name}` where the `$` is the macro character. The reference is *expanded* to produce its value.

When the macro name is a single character, the parentheses or braces around the name are optional. For example, `$a` and `$(a)` are equivalent. Macro names are case-sensitive.

To put a literal dollar sign in the makefile, use two dollar signs: `$$`.

## Defining macros

Macros are defined at read time in a makefile with macro definition lines of the form:

| | | | |
|---|---|---|---|
| *name* | **=** | *text* | standard definition |
| *name* | **?=** | *text* | conditional definition |
| *name* | **:=** | *text* | expanded definition |
| *name* | **+=** | *text* | appended definition |
| *name* | **!=** | *text* | shell definition |

The name can contain any character except “`=`”, “`:`” and white-space. Any macro references in the name are expanded, so if you want a literal “`$`” in the name, you must use “`$$`”.

White space before and after the operator (“`=`”, “`?=`”, “`:=`”, “`+=`” and “`!=`”) is ignored. A macro typically starts in the first column, but it may be indented. Note, though, that macro definitions are only allowed in normal context (as opposed to rule context). A macro definition that starts in the first column implicitly switches back to normal context, and so is always valid (see page 20).

The text value is arbitrary text and can reference other macros with expressions of the form `$(macro_name)`.

A **standard macro** definition (`=`) sets the text to the name. The text value is not interpreted at the time of definition. When no text is given, the name is defined to an empty string.

```
MODEL   = L
CFLAGS  = -A$(MODEL)
```

A **conditional macro** definition (`?=`) is like the standard macro definition, but the macro is defined only if it isn't already defined.

```
MODEL   ?= S
```

An **expanded macro** definition (`:=`) defines the macro and sets its value to the expansion of the text parameter. That is, any macro references in the text parameter are replaced with the macro value. Expanded macros are useful when the text contains macro references that are expensive to calculate.

```
OBJS  := $(wildcard *.o)
```

An **appended macro** definition (`+=`) appends text to the current value of the macro. A single space is inserted between the original text and the appended text.

```
ifdef DEBUG
  CFLAGS += -g
endif
```

If the macro was not yet defined, this definition is the same as a standard macro definition. Alternatively, if the macro was defined as an *expanded* macro, the "`+=`" operator expands the text to be appended, so that the macro stays an expanded macro.

To append text to a macro *without* space separator, redefine the macro with a reference to the previous macro. Note that "`:=`" must be used instead of a plain "`=`", because otherwise it would create a recursive definition.

```
PATH := $(PATH);../source
```

A **shell macro** definition (`!=`) passes the text to the shell and sets the output of the shell command as the value of the macro. See also the `$(shell…)` function on page 39.

## Macro precedence

Macros can be defined in makefiles, on the command line, be inherited from environment variables, or can be predefined by Make.

A macro's precedence is determined by the location that the macro is defined. To redefine an existing macro, the new definition must have a precedence at least as high, or the macro definition must use the `override` directive, as shown below:

```
override CDEBUG = -Wall -Wextra
```

From highest precedence to lowest precedence, the order is:

1. Run-time macros (see the next section).
2. Macros defined with the `override` directive.
3. Macros defined on the command line.
4. Macros defined in the makefile.
5. Environment variables (if the `-e` command line option is used, environment variables take precedence over macros defined in the makefile, see page 11).
6. Built-in macros (read from the configuration file).

## Run-time macros

Run-time macros are set dynamically by Make according to the current target being built. Run-time macros can only be used in a *recipe* (although *secondary expansion* allows run-time macros in the dependency list of a rule as well). The GNU Make manual refers to run-time macros as “automatic variables”.

| Macro | Alias | Value |
|---|---|---|
| `$@` | `$(.TARGET)` | The path to the target of a rule. If a rule has multiple targets, it holds the target that must be built. |
| `$<` | `$(.SOURCE)` | The path name of the inferred source, or the path of the first explicit source. |
| `$?` | `$(.NEWSOURCES)` | A space-separated list of all prerequisites that are newer than the target. |
| `$^` | `$(.SOURCES)` | A space-separated list of all prerequisites, where duplicate names have been removed, and excluding any *order-only* prerequisites. |
| `$+` | `$(.SOURCES+)` | A space-separated list of all prerequisites, without removing duplicate names (but still excluding any *order-only* prerequisites). |
| `$*` | `$(.STEM)` | The stem with which an implicit rule matches (or in an explicit rule, `$@` without its suffix), see also the `.SUFFIXES` directive on page 48. |
| `$%` | | The name of the member in the target, in the case that the target is a library. |
| `$\|` | | A space-separated list of order-only prerequisites. |

For each run-time macro, its short-hand and its alias refer to the same value. The short-hands are compatible with most other Make implementations, whereas the

aliases are probably unique to Opus Make and this implementation of Make+. Two macros do not have an alias, because these are rarely used.

## Macro modifiers

The directory or filename parts can be split off the run-time macros by adding a suffix. For the short-hand macro names (`$@`, `$<`, etc.), the suffix follows the name immediately; for the alias names, there is a comma between the macro name and the suffix (see also the table below).

The suffix `D` returns the directory part of the run-time macro, so `$(@D)` returns the directory part of the target (without trailing slash). If the target name has no directory part, the `D` suffix returns a ".". The `F` suffix returns the filename part of the target name (including the extension). The `Q` suffix encloses the target name in double quotes if that name contains (escaped) spaces.

| Macro | Alias | Value |
|---|---|---|
| `$(@D)`<br>`$(@F)` | `$(.TARGET,D)`<br>`$(.TARGET,F)` | The directory or the filename of the target. |
| `$(@Q)` | `$(.TARGET,Q)` | The target path name optionally enclosed in double quotes. |
| `$(<D)`<br>`$(<F)` | `$(.SOURCE,D)`<br>`$(.SOURCE,F)` | The directory or the filename of the inferred source, or the first explicit source. |
| `$(<Q)` | `$(.SOURCE,Q)` | The inferred or first explicit source suitably enclosed in double quotes. |
| `$(?D)`<br>`$(?F)` | `$(.NEWSOURCES,D)`<br>`$(.NEWSOURCES,F)` | A space-separated list of directories/filenames of all prerequisites that are newer than the target. |
| `$(?Q)` | `$(.NEWSOURCES,Q)` | A space-separated list of all prerequisites newer than the target, with each item suitably enclosed in double quotes. |
| `$(^D)`<br>`$(^F)` | `$(.SOURCES,D)`<br>`$(.SOURCES,F)` | A space-separated list of directories/filenames of all prerequisites, without duplicates or *order-only* prerequisites. |
| `$(^Q)` | | A space-separated list of all prerequisites, without duplicates and suitably enclosed in double quotes. |
| `$(+D)`<br>`$(+F)` | `$(.SOURCES+,D)`<br>`$(.SOURCES+,F)` | A space-separated list of directories/filenames of all prerequisites including duplicates, excluding *order-only* prerequisites. |
| `$(+Q)` | `$(.SOURCES+,Q)` | A space-separated list of all prerequisites including duplicates, suitably enclosed in double quotes. |
| `$(*D)`<br>`$(*F)` | `$(.STEM,D)`<br>`$(.STEM,F)` | The directory or the filename of the stem of the target. |
| `$(*Q)` | `$(.STEM,Q)` | The stem of the target, suitably enclosed in double quotes. |
| `$(\|D)`<br>`$(\|F)` | | A space-separated list of directories/filenames of order-only prerequisites. |

| | |
|---|---|
| `$(\|Q)` | A space-separated list of order-only prerequisites, where each item is suitably enclosed in double quotes. |

For general purpose path manipulation, see functions `$(dir…)` and `$(notdir…)` on page 37. The `D` and `F` macro modifiers are defined in terms of these functions. The `Q` suffix is defined in terms of the `$(quote…)` function (see page 37).

## Suffix substitution

A special syntax exists to match and replace the last part of each word in a macro value. In practice, it is often used to replace file extensions while expanding a macro. For example:

```
CFILES  := blinky.c cmsisBoot.c cmsisSystem.c
OBJECTS := $(CFILES:.c=.o)
```

In the above snippet, `OBJECTS` is assigned the string "`blinky.o cmsisBoot.o cmsisSystem.o`".

The match text ("`.c`" in the snippet above) is only matched at the end of each word in the macro value, in it will only be swapped for the replacement text on a match.

For more general pattern substitution, see the `$(patsubst…)` function on page 36.

## Multi-line macros

Macros spanning multiple lines can be defined with the `define` directive, according to the following syntax:

```
define name =
text
text
endef
```

A macro defined with this construct allows for multiple lines of text in its value, but it is equivalent to single-line macros all other respects. The "`=`" assignment operator can also be "`?=`", "`:=`" or "`+=`", with the same meanings as for the single-line macros. The assignment operator may be omitted, in which case it is assumed to be a standard definition ("`=`"). The text lines in the macro may contain references to other macros.

Macro definitions with the define directive may be nested.

## Undefining a macro

Macros can be undefined with the `undefine` directive.

```
undefine CFLAGS
```

To undefine a command-line macro, this line should be preceded with the `override` directive.

Run-time macros cannot be undefined.

## Target-local and pattern-local macros

Standard macros are global, and valid from the point that they are defined. Target-local macros, on the other hand, are local to the target that is currently built. The macro is valid only in the target's recipe, and in the recipes of all sources of that target. That is, sources of a target inherit the target-local variables that are set on the target.

A target-local macro is defined with the syntax:

```
target(s) : name = text
```

The same set of assignment operators as for standard macro definitions can be used ("`=`", "`?=`", "`:=`" and "`+=`").

A typical use for target-local macros is to set additional options for the build of a specific source file, particularly in the case that source files are built with an inference rule.

In the example makefile below, all `.c` files are an inference rule that references the macro `$(CFLAGS)`. On line 11, an additional compiler option is added to `$(CFLAGS)` for the target `cmsisBoot.o` (with the effect that `cmsisBoot.c` is built with that additional option).

```
1. CFLAGS = -mcpu=cortex-m0
2.
3. %.o : %.c
4.     armcc -c $(CFLAGS) -o$(.TARGET) $(.SOURCE)
5.
6. blinky.elf : blinky.o cmsisBoot.o cmsisSystem.o
7.     armlink -o blinky.elf blinky.o cmsisBoot.o cmsisSystem.o
8.
9. blinky.o : blinky.c
10.
```

```
11.cmsisBoot.o : CFLAGS += -Od
12.cmsisBoot.o : cmsisBoot.c
13.
14.cmsisSystem.o : cmsisSystem.c
```

Target-local variables have the same precedence as other macros defined in a makefile. Specifically, macros set on the command line have higher precedence than target-local variables, and the override directive is needed on the target-local definition to override a command-line macro.

Pattern-local macros are similar to target-local macros, but instead apply to any target that matches a particular pattern. A pattern is a string with a "%" wild-card character. For example, the definition below sets an additional option for building object modules in a particular sub-directory:

```
lib/%.o : CFLAGS += -GS
```

## Built-in macros

Only a few macros are truly built-in, these include the run-time macros (see page 28) and the ones listed in this section. Apart from these, Make considers all macros defined in the configuration file as "built-in" as well.

`CURDIR`
: The current directory that Make operates in (setting this macro to a different value has no effect; however, see section Changing the current directory on page 56).

`.DEFAULT_GOAL`
: Expands to a string containing the default target. You can change the default target by assigning a new value to this macro.

`.FEATURES`
: Expands to the list of features supported by the version of Make. The following tokens can appear in this list:

| | |
|---|---|
| `archives` | Make can access object modules inside libraries. |
| `check-symlink` | Support for the `-L` option; see page 11. |
| `else-if` | Support for the "else if" construct, see page 44. |
| `ifset` | Support for the `ifset` and `ifclear` directives, see page 44. |
| `guile` | Support for the Guile scripting language. |
| `jobserver` | Support for parallel builds ("job server"). |
| `load` | Support for calling into shared libraries. |
| `oneshell` | Support for the `.ONESHELL` directive, see page 47. |
| `order-only` | Support for "order only" prerequisits. |

| | |
|---|---|
| `runtime-macros` | Support for alias names for run-time macros, see page 28 |
| `second-expansion` | Support for the `.SECONDEXPANSION` directive. |
| `shortest-stem` | The pattern matching method for dependency lines. |
| `target-specific` | Support for target-local and pattern-local macros, see page 31. |
| `undefine` | Support for the `undefine` directive, see page 31. |
| `.path` | Support for the `.path` directive, see page 45. |

To check whether an implementation of Make supports a particular feature, you can use function `$(findword…)`, see page 35.

`.INCLUDE_DIRS`
: Expands to a space separated list with directories for included files.

`MAKE`
: Expands to the path of the Make executable, for use in recursive Make (see page 55).

`MAKE_HOST`
: Expands to a string with the host that the Make executable was built for. For example, it is "`Windows`" when built for both 32-bit and 64-bit versions of Microsoft Windows. See Operating System detection on page 54 for more information.

`MAKE_VERSION`
: Expands to a string with the version of the Make executable. For example "4.2.2".

`MAKECMDGOALS`
: Expands to the target (or targets) that were mentioned on the command line.

`MAKEFLAGS`
: Assigning flags to this macro passes them on to recursively called Make (see page 55).

`MAKELEVEL`
: The nesting level of recursive Make (see page 55).

`.RECIPEINDENT`
: The minimal number of leading spaces that a recipe line must be indented with. The default value is four. See page 19.

`SHELL`

Expands to the name (possibly including a path) of the shell that runs the recipes. For example "`/bin/sh`" in Unix or Linux environments.

`.SHELLSTATUS`

Expands to the exit code (called the "`errorlevel`" value in Microsoft Windows) of the most recently run shell line.

`.space`

Expands to a single space character. It is predefined as a variable because it is sometimes necessary to insert a space in a macro or function parameter, where leading and trailing literal spaces are ignored.

`SUFFIXES`

Expands to the space-separated list of all known suffixes. The suffixes are set with the `.SUFFIXES` directive, see page 48.

`.VARIABLES`

Expands to a space-separated list with all global variables that are defined up to that point in the makefile.

# Functions

Functions, like macros, expand to a text value, but unlike macros a function creates its value by manipulating its parameters or by carrying out a system operation.

The general syntax of a function is:

```
$(name parameter,parameter)
```

Like is the case for macros, you may use braces instead of parentheses to surround the function name and parameters. The number of parameters that a function has depends on the function, a few have no parameters at all.

Parameters are separated by commas and the first parameter is separated from the function name by white-space. This leading white-space in front of the first parameter is ignored, but white-space around the commas (between the parameters) is part of the relevant parameter. The text of a parameter can contain parentheses, but only in matching pairs. The parameter text *cannot* contain a comma, however, you can achieve the same thing via variable expansion. For example, the snippet below uses the `$(subst…)` function to convert a space separated list to a comma-separated list:

```
comma := ,
FILES := blinky.o cmsisBoot.o cmsisSystem.o
LIST  := $(subst $(.space),$(comma),$(FILES))
```

## General string manipulation

`$(filter` *`patterns`*`,`*`text`*`)`
: Matches each word in *`text`* against the each of the patterns in *`patterns`*, and returns a space separated list of the matching words.

`$(filter-out` *`patterns`*`,`*`text`*`)`
: Matches each word in *`text`* against the each of the patterns in *`patterns`*, and returns a space separated list of the words that do *not* match.

`$(findstring` *`find`*`,`*`text`*`)`
: Returns *`find`* if it occurs in *`text`*, and an empty string if it does not occur in text. See also `$(findword…)`.

`$(findword` *`find`*`,`*`text`*`)`
: Returns *`find`* if *`text`* contains this word, and an empty string if it does not

occur in text. The difference between this function and `$(findstring…)` is that this function requires *`find`* to be a complete word in *`text`*, whereas function `$(findstring…)` also finds parts of a word.

`$(firstword` *`text`*`)`
: Returns the first word in *`text`*. See also `$(word…)`.

`$(lastword` *`text`*`)`
: Returns the last word in *`text`*. See also `$(word…)`.

`$(patsubst` *`pattern`*`,`*`replace`*`,`*`text`*`)`
: Matches each word in *`text`* against *`pattern`* and substitutes it by *`replace`* on a match. There may be a single "`%`" (wildcard character) in *`text`*. See also `$(subst…)`.

`$(sort` *`text`*`)`
: Returns a space-separated list of words in *`text`*, alphabetically sorted and with duplicates removed.

`$(strip` *`text`*`)`
: Strips of leading and trailing white-space from each (space-separated) word in *`text`*.

`$(subst` *`find`*`,`*`replace`*`,`*`text`*`)`
: Replaces all occurrences of *`find`* to *`replace`* in *`text`*.
See also `$(patsubst…)`.

`$(word` *`n`*`,`*`text`*`)`
: Returns word *`n`* from *`text`*, where *`n`* ranges from 1 to the number of words in *`text`*. See also `$(words…)`.

`$(wordlist` *`first`*`,`*`last`*`,`*`text`*`)`
: Returns the range of words between *`first`* and *`last`* from *`text`* in a space-separated list.

`$(words` *`text`*`)`
: Returns the number of words in *`text`*. See also `$(word…)`.

## Path and filename handling

Names of paths and filenames are strings, and hence, path/filename handling functions are also string handling functions. However, the group of path/filename han-

dling functions treats an escaped space as part of a path name, whereas the string functions treat each space as a list separator.

`$(abspath` *`pathlist`*`)`
: Returns a space-separated list with the path names in *`pathlist`*, where relative sub-directory specifications like "`./`" and "`../`" are resolved, and double "`/`" are replaced by a single one. In contrast to `$(realpath…)`, this function does not verify whether the file or path exists.

`$(addprefix` *`prefix`*`,`*`pathlist`*`)`
: Inserts *`prefix`* before each of the path names in *`pathlist`*, and returns this new list. See also `$(notdir…)`.

`$(addsuffix` *`suffix`*`,`*`pathlist`*`)`
: Appends *`suffix`* to each of the path names in *`pathlist`*, and returns this new list. See also `$(basename…)`.

`$(basename` *`pathlist`*`)`
: Returns the path names in *`pathlist`* with their suffix (extension) removed, and as a space-separated list. See also `$(suffix…)`.

`$(dir` *`pathlist`*`)`
: Returns the directory-parts of all path names in *`pathlist`*. in a space-separated list. The trailing slash (or backslash in Microsoft Windows) is *not* included in each directory-part. If a path name in *`pathlist`* does not have a directory, its directory part is set to "`.`". See also `$(notdir…)`.

`$(join` *`list1`*`,`*`list2`*`,`*`separator`*`)`
: Returns a space-separated list of words from *`list1`* joined to the words from *`list2`*. The resulting list has the words from both lists merged pairwise. The *`separator`* argument is optional. If present, it is put between each pair of words from *`list1`* and *`list2`*.

`$(notdir` *`pathlist`*`)`
: Returns the filenames of all path names in *`pathlist`*. in a space-separated list. In other words, the directory-parts of all path names in *`pathlist`* is removed. See also `$(addprefix…)`.

`$(quote` *`pathlist`*`)`
: Returns the paths in *`pathlist`* where each path that contains (escaped) spaces is surrounded by double quotes and the escape characters are

removed. Paths that have no spaces are left unchanged. See page 51 for details.

`$(realpath` *`pathlist`*`)`
: Returns a space-separated list with the path names in *`pathlist`*, where relative sub-directory specifications like "`./`" and "`../`" are resolved. On platforms that support "symlinks", these are resolved too. If a path does not exist, it is removed from the result. See also `$(abspath…)`.

`$(suffix` *`pathlist`*`)`
: Returns the suffixes (extensions) of all path names in *`pathlist`*, in a space-separated list. See also `$(basename…)` and `$(addsuffix…)`.

The table below illustrates the various functions for paths, where the macro `PATH` is set to "`./startup/../cmsis/cmsisBoot.c`"

| Function | Result |
|---|---|
| `$(abspath $(PATH))` | `cmsis/cmsisBoot.c` |
| `$(addprefix src/,$(PATH))` | `src/./startup/../cmsis/cmsisBoot.c` |
| `$(addsuffix .bak,$(PATH))` | `./startup/../cmsis/cmsisBoot.c.bak` |
| `$(basename $(PATH))` | `./startup/../cmsis/cmsisBoot` |
| `$(dir $(PATH))` | `./startup/../cmsis` |
| `$(notdir $(PATH))` | `cmsisBoot.c` |
| `$(realpath $(PATH))` | `cmsis/cmsisBoot.c` |
| `$(suffix $(PATH))` | `.c` |

## Logic functions

The logic functions consider an empty value as "false", and any non-empty value as "true". This applies to the conditions and to the return values of these functions.

`$(and` *`condition`*`,`*`condition`*`,…)`
: Returns the value of the last *`condition`* from the list of comma-separated conditions, if all these conditions are non-empty. If one *`condition`* in the list is empty, the function returns an empty string.

`$(if` *`condition`*`,`*`then`*`,`*`else`*`)`
: If *`condition`* expands to a non-empty string, the *`then`* part is evaluated; otherwise the *`else`* part is evaluated. The *`else`* part may be omitted.

`$(not condition)`

Returns an empty value if the *condition* evaluates to a non-empty string, or the fixed string “true” if the *condition* is empty.

`$(or condition,condition,…)`

Returns the value of the first *condition* from the list of comma-separated conditions that is non-empty. If every *condition* in the list is empty, the function returns an empty string.

## File and process I/O

`$(file < filename)`
`$(file > filename,text)`
`$(file >> filename,text)`

Reads from a file or writes to a file. The operation code “<” stands for read mode, and returns the contents of the file. Write mode uses operation codes “>” for overwrite and “>>” for appending to a file (in both cases, the function returns an empty string).

`$(shell command)`

Returns the output from running *command* on the shell, where newline characters are replaced by a single space. The variable `.SHELLSTATUS` (see page 34) is set to the exit value of the shell.

`$(wildcard pattern)`

Returns a space-separated list of path names that match *pattern*. Usually, the pattern contains wild-card characters (such as “?” or “*”).

## Macro control and information

`$(call macro,param,param,…)`

In a first step, the name *macro* and the values of the parameters are temporarily assigned to `$0`, `$1`, `$2` etc. Then the function expands *macro*, which will typically reference `$1` and the other parameters.

The snippet below shows a parameterized macro that swaps its two parameters:

```
SWAP = $(2) $(1)
REVERSE = $(call SWAP,fox,dog)
```

`$(defined macro)`

Returns the macro name or an empty string, depending on whether the

macro is defined. Due to the empty result on an undefined macro, this function is practical for use with logic functions, see page 38. See also `$(flavor…)` and `$(origin…)`.

`$(flavor` *`macro`*`)`

Returns a string that indicates how the macro was defined; it is one of:

`recursive`
: The macro is a standard macro definition that is expanded when it it referenced, see page 26. The term “recursive” refers the iterative expansion when the value of the macro references other macros; true recursive macros are not supported by Make.

`simple`
: The macro is an expanded definition, see page 26. The macro value therefore contains no references to other macros.

`undefined`
: The macro is not defined.

See also `$(defined…)` and `$(origin…)`.

`$(foreach` *`macro`*`,`*`list`*`,`*`text`*`)`

For each word in *`list`*, *`macro`* is temporarily defined to that word, and then *`text`* is expanded; this *`text`* will typically reference *`macro`*. Therefore, text is expanded as many times as there are (space-separated) words in list. Note that macro needs to be a macro name, not a macro reference.

For example, the snippet below doubles every word in list with a dash between them:

```
LIST := the quick brown fox
TEST := $(foreach word,$(LIST),$(word)-$(word))
```

`$(origin` *`macro`*`)`

Returns a string that indicates how the macro was defined; it is one of:

`automatic`
: The macro is a run-time macro, see page 32.

`command line`
: The macro is defined on the command line.

`default`
: The macro is either defined in the configuration file (see page 24), or it is a built-in macro.

`environment`
: The macro is inherited from the environment.

`environment override`
: The macro is inherited from the environment, and it overrides a macro (with the same name) in the makefile.

`file`
: The macro is defined in the makefile.

`override`
: The macro is defined in the makefile with the `override` directive.

`undefined`
: The macro is not defined.

See also `$(flavor…)`.

`$(value` *`macro`*`)`
: Returns the value of macro without expanding any macro references inside that value. In the example below, the macro `RESULT` will be set to the string `$(MAKE_VERSION)`:

```
VER = $(MAKE_VERSION)
RESULT = $(value VER)
```

## Make and makefile control

`$(error` *`message`*`)`
: Aborts Make after printing *`message`* on the display (together with the file-name and line number in the makefile where the error occurred). The *`message`* may contain macro references (these are expanded before it is printed).

`$(eval` *`text`*`)`
: Expands *`text`* and lets Make parse the result as if it is a line in a makefile. The `$(eval…)` function can therefore dynamically insert makefile rules and directives (see page 52). This function returns an empty string.

`$(info` *`message`*`)`
: Prints *`message`* on the display, without adding filename or line number information. The *`message`* may contain macro references (these are expanded before it is printed).

`$(warning` *`message`*`)`
: Prints *`message`* on the display (together with the filename and line number in the makefile where the error occurred), but continues running. The

*`message`* may contain macro references (these are expanded before it is printed).

# Directives

Directives provide additional control over how Make processes the makefiles. There are several types of directives.

## Pre-processing directives

The `include` directive reads the contents of another file into the current makefile.

```
include settings.mak
```

An error message is issued if the file does not exist. To continue Make regardless of the missing file, use `-include` instead of `include`.

The `define` ... `endef` directives create macros, and specifically multi-line macros (see page 30).

The `undefine` directive removes a macro definition (see page 31).

The conditional directives `ifdef`, `ifndef`, `ifeq`, `ifneq`, `ifset` and `ifclear` control which lines in a makefile Make reads and runs. The form of these directives is:

```
ifdef name
  # zero or more lines for the “true” condition
else
  # zero or more lines for the “false” condition
endif
```

Conditional directives can be nested; each `if`... must be matched with an `endif`. The `else` directive is optional.

The `ifdef` and `ifndef` directives test whether or not a macro is *defined*, where the name of the macro is specified as the parameter of the directive. In the example below, additional options are added for the compiler and linker command if the macro DEBUG is defined, for example on the command line of the make command:

```
ifdef DEBUG
  CFLAGS    += -Od -g
  LINKFLAGS += -g
endif
```

The `ifeq` and `ifneq` directives compare two parameters. The parameters are separated by a comma and surrounded by parentheses. Any macros that are referenced in the parameters are expanded before the comparison.

```
ifeq ($(BUILD),Release)
  POSTLINK = strip -k $(.TARGET)
else
  POSTLINK =
endif
```

As stated, the `ifdef` and `ifndef` directives test whether a macro is defined. On the original GNU Make version, however, these directives test whether a macro is empty.* The alternative way to test for whether a macro is empty is by comparing the macro to "nothing" with `ifeq` or `ifneq`. However, this particular case is common enough that Make+ has specific directives for it: `ifset` interprets the lines below it if its parameter evaluates to anything that is not empty, and `ifclear` tests whether its parameter is empty.

```
ifset $(TOOLSDIR)
  CC = $(TOOLSDIR)/gcc
else
  CC = gcc
endif
```

The `ifeq` & `ifneq` directives can also be used to test whether a macro is empty. A difference of these directives with `ifset` & `ifclear` is that, when a macro evaluates to white-space, it is considered non-empty by `ifeq` & `ifneq`, whereas it is considered empty by `ifset` & `ifclear`,

The Logic functions (see page 38) come in handy, in combination with `ifset` & `ifclear` to test for multiple macros at the same time. The `ifdef` & `ifndef` directives test only a single macro; to test for either of two macros exist, use a syntax like the following:

```
ifset $(or $(defined ARM_THUMB2),$(defined RISCV))
  # either ARM_THUMB2 or RISCV is defined
endif
```

As a special case, a new condition may follow the `else` directive. Doing so, applies a new condition to the lines that follow below, but without increasing the nesting level. For example:

* An undefined macro evaluates to an empty value.

```
ifeq ($(BUILD),Release)
  POSTLINK = strip -k $(.TARGET)
else ifeq ($(BUILD),Debug)
  POSTLINK = dsymutil $(.TARGET)
else
  $(error Unknown build type)
endif
```

## Directives for macro attributes

Macro definitions that are prefixed with the `export` directive are passed on to the commands launched by Make, via the environment. Since the macros are stored in the environment, the macro name must adhere to the rules that the shell imposes on environment variables.

In a recursive Make scenario (see page 55), exporting macros is the way to pass macros from the parent Make to the child Make.

The variables `MAKEFLAGS` and `MAKEFILES` are exported by default (and thereby passed on), unless a makefile marks them with the `unexport` directive. For example:

```
unexport MAKEFLAGS
```

The `override` directive on a macro definition overrides the standard precedence levels of macro definitions (see page 27).

Target-local variables are by default inherited by the sources of that target. The `private` directive suppresses the inheritance, so that the target-local variable is valid *only* for the target that it is defined on.

## Directives for search paths

The `.path` and `vpath` directives specify directories where Make searches for targets that do not have a path component. The syntax is:

```
.path pattern directories
vpath pattern directories
```

The *pattern* is similar to that of the target in an inference rule. The pattern `%.c` matches all files with the extension "`.c`" for example. The *directories* parameter is a space-separated list of directories where targets that match the pattern should be searched.

```
vpath %.c  ../coreapi
```

The difference between `.path` and `vpath` is where the target is located when it is *not* found. In other words (and assuming there is a recipe for building the target from its sources), the difference is where the target will be built if it does *not yet* exist:

- If you use one or more `.path` commands, it is located in the first path listed (on the first `.path` command).
- If you only use `vpath`, it is located in the *current* directory.

Briefly, `vpath` should only be used to input files (that already exist) and `.path` can also be used to locate output files into output directories.

Executable files typically have no extension on Linux and Unix-like platforms. To set a path where the executables should be built, end the pattern with a dot.

```
.path %. ../bin
```

The above syntax will match all filenames that do not have a file extension.

## Recipe directives

These directives are only valid inside a recipe, and they run as if they are an external command or a shell line. However, these directives perform an internal operation.

The only directive in this category is `.chdir`. The `.chdir` directive switches the active directory that Make will run its commands in. It overrules a "`-C`" command line argument (see page 10). The `.chdir` directive updates the `CURDIR` macro (page 32).

A special syntax is:

```
    .chdir -
```

This command switches back to the directory that was active before the last `.chdir` command. It allows you to restore the original directory after (temporarily) switching to a different sub-directory earlier in the recipe. See section Changing the current directory on page 56 for an example and a discussion.

## Pseudo-target directives

These directives set various options. They are formatted to resemble a dependency line of a rule, as if these are "targets" that need to be built.

`.DEFAULT`

Specifies a recipe to run for any target that does not match to any rule. This directive has a recipe (but no prerequisites).

```
.DEFAULT :
    recipe
```

`.DELETE_ON_ERROR`

Sets the option to delete the target of each rule if one of the shell lines to build it returns with an error.

```
.DELETE_ON_ERROR :
```

`.EXPORT_ALL_VARIABLES`

Sets the option to export all macros by default (instead of only the macros with the `export` attribute).

```
.EXPORT_ALL_VARIABLES :
```

`.IGNORE`

Sets Make to ignore errors in building the targets that are listed as the prerequisites of the directive. If there are no prerequisites on the `.IGNORE` directive, Make ignores errors on *all* targets.

```
.IGNORE : target(s)
```

`.INTERMEDIATE`

Marks the prerequisites on the line as *intermediate* files. Intermediate files are deleted after building the targets that depend on the intermediate file.

```
.INTERMEDIATE : target(s)
```

`.LOW_RESOLUTION_TIME`

Declares which files have time-stamps that have a precision measured in seconds, but not sub-seconds. It is rarely needed.

```
.LOW_RESOLUTION_TIME : target(s)
```

`.NOTPARALLEL`

Clears the option to build targets in parallel, if such an option was set on the command line (see option `-j` on page 11).

```
.NOTPARALLEL :
```

`.ONESHELL`

Sets the option to execute all shell lines in a recipe as a compound com-

mand in a single call to the shell. This is mostly useful when using a script interpreter as a shell.

```
.ONESHELL :
```

`.PHONY`

Marks the targets on its sources list as phony. A target that has no representation as a file is implicitly phony, but it is good practice to explicitly mark phony targets that have no prerequisites as `.PHONY`.

```
.PHONY : target(s)
```

`.PRECIOUS`

Marks targets to be kept. Make normally deletes a target if it is aborted, or if the target is an *intermediate* file. This directive cancels this default operation.

```
.PRECIOUS : target(s)
```

`.SECONDARY`

Marks targets as *intermediate* files, with the exception that they are *not* deleted (as is the default for intermediate files).

```
.SECONDARY : target(s)
```

`.SECONDEXPANSION`

Lets Make run over all lines read from the makefile a second time and expand any macros again, which allows for indirectly declared macros.

```
.SECONDEXPANSION :
```

`.SILENT`

Sets the option to skip printing the shell lines that Make executes. See also the shell line prefix “@” on page 20.

```
.SILENT :
```

`.SUFFIXES`

Sets the list of file extensions that are recognized. These extensions are used by the suffix rules (see page 23), as well as to get the stem of the target filename for the `$*` and `$(.STEM)` run-time variables (see page 28).

```
.SUFFIXES : extension(s)
```

The extensions in the list should include the period that separates the filename stem from the extension. For example, below is a rule that is suitable for Unix and Linux environments:

```
.SUFFIXES .out .a .ln .o .c .cc .C .cpp .p .f .F .m .r .y .l \
          .ym .yl .s .S .mod .sym .def .h .info .dvi .tex \
          .texinfo .texi .txinfo .w .ch .web .sh .elc .el
```

# Advanced Topics

## *Rules*

### Order-only sources on a dependency line

The sources on a dependency line that appear after a pipe symbol ("|") are "order-only" sources. For example, in the line below, `builddir` is an order-only source for `test.elf`:

```
test.elf : main.o sub.o | builddir
    armlink -o test.elf main.o sub.o
```

Order-only sources do not change the order in which sources are built in the dependency list of a target. Instead, unlike a normal source, the building of an order-only source does *not* imply that the target is now out of date. After building the order-only source, Make will therefore not carry on to run the recipe for the original target.

In the above example, the act of building target `builddir` does not cause `test.elf` to be re-linked. If `main.o` and `sub.o` were updated, though, `test.elf` will be re-linked.

### Double-colon rules

When a double-colon ("`::`") separates the target and its sources, a recipe can be given after each double-colon dependency line. In this case the recipe is executed if any of the sources *on this dependency line only* is newer than the target.

Double-colon rules are the only way that multiple recipes can be associated with the same target. Double-colon rules cannot be mixed with normal (single-colon) rules.

### Multiple targets on a dependency line

There may be multiple targets on a dependency rule, which Make handles as if there were multiple rules (one for each target) with the same list of sources. For example:

```
main.o counter.o : counter.h config.h
```

is equivalent to:

```
main.o : counter.h config.h
counter.o : counter.h config.h
```

# *Filenames with spaces*

Make uses a space as the separator for filenames on a dependency line. As a result, when a filename (or a directory name) contains a space, it must be handled in a special way. A similar situation applies to the commands in the recipes: these are passed to the shell and a shell command typically separates arguments with a space character as well.

## Dependency lines

When names of targets or sources contain spaces, these must be escaped on the dependency line. To escape a space, prefix it with a `\`. For example:

```
led\ blink.o : led\ blink.c config.h
    $CC) -c -o$(.TARGET) $(.SOURCE)
```

The filename for the target is "`led blink.o`" and it is built from the C source file "`led blink.c`". Without the backslashes, the dependency line would be interpreted as having two targets, "`led`" and "`blink.o`" and to depend on three sources, "`led`", "`blink.c`" and "`config.h`".

## Shell lines

If a shell command supports spaces that are escaped with a `\` just like Make, the same source and target names can be used in the recipes. However, a more generally accepted syntax on all platforms is to enclose a filename in double quotes.

The `$(quote…)` function (see page 37) replaces escaped spaces in a filename by a plain spaces and surrounds it with double quotes, but only if spaces were present in the filename. Filenames without a space are left unchanged. For example, the recipe in the rule in the snippet below now uses the `$(quote…)` function to translate the value of macros `$(.TARGET)` and `$(.SOURCE)` from makefile syntax to shell command syntax:

```
led\ blink.o : led\ blink.c config.h
    $CC) -c -o$(quote $(.TARGET)) $(quote $(.SOURCE))
```

The `$(quote…)` function also works on lists of filenames, where each filename gets translated from escaped spaces to double-quoted individually.

Short-hands exist for translating run-time macros to quoted format, see page 29. A shorter way to write the same recipe is below:

```
led\ blink.o : led\ blink.c config.h
    $CC) -c -o$(.TARGET,Q) $(.SOURCE,Q)
```

### Other notes

Many functions operate on lists with elements that are separated with spaces. Only a subset of the functions support escaped spaces: these are the “path and filename handling” functions, plus the `$(wildcard…)` function (see pages 36 and 39 respectively).

## *Functions*

### Dynamic generation of rules

The `$(eval…)` function (see page 41) expands its parameter and parses the result as if it were a line in the makefile. in a simplistic example, the line:

```
$(eval PROJECT = blinky)
```

has exactly the same result as typing the next line in the makefile:

```
PROJECT = blinky
```

In practice, the parameter of the `$(eval…)` function contain one or more macros, and especially parameterized macros that are invoked through the `$(call…)` function. For example:

```
1. define BUILD =
2. $1 : $2
3.     armcc -c -o$$(.TARGET) $$(.SOURCE)
4. endef
5.
6. all: blinky.hex
7.
8. $(eval $(call BUILD,blinky.o,blinky.c))
9. $(eval $(call BUILD,cmsisBoot.o,cmsisBoot.c))
10.$(eval $(call BUILD,cmsisSystem.o,cmsisSystem.c))
11.
12.blinky.hex : blinky.o cmsisBoot.o cmsisSystem.o
13.    armlink -o $(.TARGET:.hex=.elf) $(.SOURCES)
14.    objcopy --output-target ihex $(.TARGET:.hex=.elf) $(.TARGET)
```

The above snippet is a variation on the “simple makefile” on page 15. On the first four lines, it defines a multi-line macro. On line 8, the macro is expanded with `blinky.o` and `blinky.c` as the parameters `$1` and `$2`, after which `$(eval…)` then inserts the following into the makefile:

```
blinky.o : blinky.c
    armcc -c -o$(.TARGET) $(.SOURCE)
```

Note that in the definition of macro `BUILD`, the “`$`” characters have to be doubled on the shell line (line 3), so that they will be replaced by a single `$` on the generated line.

In a common scenario, the `$(eval…)` and the `$(foreach…)` functions are used together to separate the collections of files from the rules. In the next example, the three calls to `$(eval…)` are wrapped in a `$(foreach…)` call. This `$(foreach…)` runs over the three filenames in the `$(OBJECTS)` macro defined on line 6. This same macro is used on the rule for the executable, on line 8. The benefit of this technique is that when a file is added to the `$(OBJECTS)` macro, the list of sources for the `blinky.hex` target and the build rules generated by the `$(foreach…)` macro stay in sync.

```
1. define BUILD =
2. $1 : $2
3.     armcc -c -o$$(.TARGET) $$(.SOURCE)
4. endef
5.
6. OBJECTS = blinky.o cmsisBoot.o cmsisSystem.o
7.
8. blinky.hex : $(OBJECTS)
9.     armlink -o $(.TARGET:.hex=.elf) $(.SOURCES)
10.    objcopy --output-target ihex $(.TARGET:.hex=.elf) $(.TARGET)
11.
12.$(foreach FILE,$(OBJECTS),$(eval $(call BUILD,$(FILE),$(FILE:.o=.c)))
```

In practice, a source file like `blinky.c` includes header files, and therefore the rule for target `blinky.o` as generated in the above example is likely incomplete, as it fails to list the additional dependencies (apart from `blinky.c`). However, the sources for a target may be split over multiple dependency lines (as long as no more than one has a recipe). Furthermore, the additional dependencies can be generated by scanning the source files. For details, see page 57.

## Operating System detection

If you wish to create a makefile for multiple platforms, some settings will depend on the particular operating system. For example, even if you use GCC on both Linux and Microsoft Windows, to delete files (e.g. for `make clean`), you need `rm` in Linux and `erase` or `del` in Windows. Therefore, some macros (or perhaps even some sections) of the makefile are conditionally defined, depending on the operating system.

In most cases, it is sufficient to check the value of the predefined `MAKE_HOST` macro. This macro has the platform name that executable for Make was built for. Typically, on each platform, you run the platform-specific build of Make, so the value of `MAKE_HOST` then represents the platform that Make runs on.

The value of `MAKE_HOST` is:

| | |
|---|---|
| `Windows` | For any version of Microsoft Windows, both 32-bit and 64-bit |
| `Linux` | For Linux and platforms derived from Linux (including Android) |
| `Darwin` | For Apple MacOS, OSX and iOS |
| `Unix` | For Unix and Unix-like platforms (excluding Linux and MacOS) |
| `MSDOS` | Legacy Microsoft DOS (and 16-bit Windows) |
| `unknown` | Any other platform (e.g. OS/2) |

The alternatively is to query the platform information from the operating system itself. This allows you to get more detailed information, such as the processor architecture (ARM, x86, AMD64).

```
ifeq ($(OS),Windows_NT)
    PLATFORM := Windows
    ifeq ($(PROCESSOR_ARCHITECTURE),AMD64)
        PLATFORM += 64BIT
    endif
    ifeq ($(PROCESSOR_ARCHITEW6432),AMD64)
        PLATFORM += 64BIT
    endif
    ifeq ($(PROCESSOR_ARCHITECTURE),x86)
        PLATFORM += 32BIT
    endif
else
    UNAME_S := $(shell uname -s)
    UNAME_P := $(shell uname -p)
```

```
    ifeq ($(UNAME_S),Linux)
        PLATFORM := Linux
    endif
    ifeq ($(UNAME_S),Darwin)
        PLATFORM := MacOS
    endif
    ifeq ($(UNAME_P),x86_64)
        PLATFORM += 64BIT
    endif
    ifneq ($(filter %86,$(UNAME_P)),)
        PLATFORM += 32BIT
    endif
    ifneq ($(filter arm%,$(UNAME_P)),)
        PLATFORM += ARM
    endif
endif
```

The above snippet uses environment variables to detect whether it is running on Microsoft Windows or on something else. If not Microsoft Windows, it runs the `uname` command (available on all flavors of Unix as well as MacOS) to distinguish between them. Note that there are two environment variables that need to be checked on Microsoft Windows for detecting whether it is a 64-bit system; this is because Microsoft Windows virtualizes the `PROCESSOR_ARCHITECTURE` variable and presents the OS as 32-bit if the variable is looked up by a 32-bit program. The other environment variable then gives the real architecture of the OS, but that other variable is not present if it is looked up by a 64-bit program.

## *Recursive Make*

When a project consists of several self-contained modules, it is common that each of these modules resides in a separate directory, and there is a separate makefile for each module to build only that module. The makefile for the project then plunges into each of the modules' subdirectories and invokes Make recursively (in addition to building any project-specific targets that aren't part of a separate module).

There are two equivalent syntaxes for calling Make recursively:

```
module :
    chdir directory && $(MAKE)
```

```
module :
    $(MAKE) -C directory
```

The first of these actually uses a feature of the OS shell; Make passes the line unmodified to the shell (apart from expanding the `$(MAKE)` macro), and it is the shell that handles the && operator to split it into two separate commands.

The module names in the project's makefile (i.e. the "root" makefile) are typically phony targets, and it may be useful to add the `.PHONY` directive on these targets.

The `CURDIR` macro is updated in the recursive invocation. The macros `MAKEFLAGS` and `MAKEFILES` are implicitly forwarded to the "child" Make (see also page 32). Other macros can be passed to the "child" Make via environment variables. In the "parent" Make, the macro definition must be prefixed with the `export` directive (see page 45). Note that the macro name must comply with the naming rules for environment variables.

The benefit of recursive make is modularity: each sub-project can be made independently, while a single Make still builds the complete project. However, recursive make also carries risks, because the dependency tree of each sub-make may be incomplete. For example, if a first sub-make has a dependency on a file that is generated (or rebuilt) in a second sub-make, the project will only build correctly when the second sub-make runs before the first. The order in which Make builds targets is unspecified, however, and it may not be stable in case of parallel builds.

## Changing the current directory

When Make launches the commands in a recipe, either directly or via a shell, the sub-process inherits the environment and the current directory from the Make process itself. That is why the following does not have the intended effect:

```
release :
    chdir ../target
    zip -r cooltool.zip *
```

When running the `chdir` recipe line, Make creates a shell to execute the command. The current directory is correctly set in the shell, but then that shell closes. The current directory of Make does *not* change. So the `zip` command on the next line is run with a fresh copy of the environment of Make, and from the original directory.

Standard advice is to not change directories in Make, but to adorn the names of targets and sources with the desired (relative) paths. For instance:

```
release :
    zip -r cooltool.zip ../target/*
```

In many cases, this will work (and if it does, it is the preferred solution). In this particular case, however, the `zip` command will store the "`../target/`" directory prefix on every file in the archive. This is not what was intended. A possible fix might be to also add the "`-j`" option to the `zip` command, but if there are sub-directories below "`target/`" (as might be implied from the "`-r`" option, for recursing into sub-directories), these relative paths are stripped from the stored file-names too. Again, not as intended.

The snippet below has both commands for changing the directory and packing the files on a single line. The `&&` command separator runs the second command after completion of the first, and only if the first command returned success.

```
release :
    cd ../target && zip -r cooltool.zip *
```

This option actually makes use of a feature of modern shells (both Unix-like shells like `bash` and `cmd.exe` from Microsoft Windows). Make passes the entire line to the shell. In case the second command must run regardless of whether the first command reported success or failure, use a semicolon as the command separator.

When the recipe has more than one line that must run after switching to a different directory, another option is the `.chdir` directive (page 46). Note that in this case, it is your responsibility to switch back to the original directory upon the end of the recipe, or otherwise subsequent recipes may run from the wrong sub-directory.

```
release :
    .chdir ../target
    zip -r cooltool.zip *
    .chdir -
```

## Generating dependencies

For C and C++ source files, their dependencies on the header files that these include can be generated with a companion program to Make, called makedepend. The makedepend utility scans a list of C and C++ source files for `#include` directives and then rewrites the makefile to add these header files as "dependencies" of the target for the C/C++ files.

Originally, makedepend was distributed as part of the X Window system (copyright by the Open Group). The version presented here is an update of the original utility, with improvements for cross-platform compatibility.

## Usage

In a typical case, a makefile contains a phony target to update the dependencies. This target is typically called “depend”. Thus, when you `make depend`, the dependencies for the C/C++ sources get rebuilt:

```
SRCS   = file1.c file2.c
CFLAGS = -Wall -DDEBUG -I../include

depend :
    makedepend -- $(CFLAGS) $(SRCS)
```

In the above example, makedepend parses the options in the `CFLAGS` macro. This macro is set after a double hyphen (`--`), so that makedepend will not warn for any options in `CFLAGS` that it does not recognize. Specifically, in this example, makedepend handles the `-D` and `-I` options in `CFLAGS` but silently ignores the `-Wall` option.

By default, makedepend writes its output to a file called `makefile` (on operating systems with case-sensitive filenames, it tries both `makefile` and `Makefile`). It creates this file if it does not yet exist, but otherwise it appends the generated dependencies to that makefile.

Following the above example, the output of running `make depend` might become:

```
SRCS   = file1.c file2.c
CFLAGS = -Wall -DDEBUG -I../include

depend :
        makedepend -- $(CFLAGS) $(SRCS)

# GENERATED DEPENDENCIES. DO NOT DELETE.

file1.o : file1.h file2.h common.h
file2.o : file2.h common.h
```

Running `make depend` again will replace the lines below the comment “`# GENERATED DEPENDENCIES...`” with the freshly generated dependency list. This is why it is important to not remove or change that comment, because if makedepend does

not find that line, it appends its dependency list to the makefile again, rather than replacing the existing (and possibly outdated) dependency list.

## Keep dependencies separate from build rules

In an environment where you work in a team, and especially when the makefile is committed to version control, it may be preferable to keep machine-generated dependencies separate from the general build rules and options. Accordingly, makedepend is set to write its output to a different file, which is then included in the makefile.

For example:

```
SRCS   = file1.c file2.c
CFLAGS = -Wall -DDEBUG -I../include

depend :
        makedepend -fmakefile.dep -- $(CFLAGS) $(SRCS)

-include makefile.dep
```

The `-f` option sets the name of the output file. This output file ("`makefile.dep`") is then included in the makefile. The `include` directive (on the last line) is prefixed by a "`-`" so that Make won't complain when `makefile.dep` is initially missing.

## Automatic updating dependencies

The above examples update the dependencies when making the target depend. This step can also be automated, such that the dependencies are updated when some targets when the relevant source files change. To this end, the rule for makedepend must list the source files as the dependencies, and the targets for the object file must list the generated output file as a dependency.

```
SRCS   = file1.c file2.c
CFLAGS = -Wall -DDEBUG -I../include

makefile.dep : $(SRCS)
        makedepend -a -f$(.TARGET) -- $(CFLAGS) $(.NEWSOURCES)

$(SRCS:.c=.o) : makefile.dep

-include makefile.dep
```

Note that the shell line for makedepend uses the `$(.NEWSOURCES)` macro to list only the files from `$(SRCS)` on the command line that are newer than the target `makefile.dep`. This reduces the time needed to update the dependencies, because only the modified files are scanned.

The other important new option on the shell line for makedepend is the `-a` option. Without this option, the resulting `makefile.dep` would contain only the dependencies of the files of the most recent run. By default, makedepend removes all dependency lines from the output file. The `-a` option lets makedepend remove *only* the dependency lines of the files on the command line (which will be regenerated).

Also note how `makefile.dep` is added as a dependency to all targets that are built from the list of sources, using a macro with the list of all source files (and suffix substitution to change the extension to that of a target).

## Predefined variables

While makedepend processes the source files, the predefined macro `_makedepend` is set. Testing for this macro in the source file allows you to conditionally exclude a part of the source code from makedepend's processing.

Other predefined variables that makedepend sets are platform-specific; running `makedepend -h -v` shows the list of predefined variables. You can use the `-U` command line option to undefine a predefined variable.

When a source file on the makedepend command line has the extension ".cpp", ".cxx", ".cc" or ".c++", the variable `__cplusplus` is predefined for that file.

## Setting "system" include paths

When cross-compiling for a different platform (for example, when developing embedded software for a micro-controller on a workstation), the system include paths are likely for the workstation's main compiler setup and may not be correct for the target compiler. In this case, you can remove the system include paths with the `-I-` option and set a specific path for *system* includes.

*System* includes refer to include directives with angle brackets (`<...>`):

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

## Locate source files with .path or vpath

When source files are stored in different directories, with `.path` or `vpath` directives set to locate the files, it is practical to use run-time macros on the shell line for makedepend (since the filenames in the run-time macros include the paths that Make looked up from the `.path` and `vpath` directives). To that end, add the source files as dependencies on the depend target line.

```
CFLAGS   =-Wall -I./startup/cmsis -std=c1x

CMSIS_O  = bootLPC11Uxx.o systemLPC11Uxx.o
USBHID_O = usbhid.o iap.o tracesupport.o uart.o
COREUSB_O= usb_LPC11Uxx.o usbcore.o usbdescr.o usbuser.o hiduser.o

vpath %.c   ./startup/cmsis     # startup code (CMSIS compatible)
vpath %.c   ./coreusb           # USB support code

.PHONY: depend
depend : $(USBHID_O:.o=.c) $(CMSIS_O:.o=.c) $(COREUSB_O:.o=.c)
        makedepend -fmakefile.dep -b -I- -- $(CFLAGS) $(.SOURCES)
```

In the above example, from the top there are the definitions of `CFLAGS` and of lists of object files for various modules. It is common to list the object files, because these are the dependencies of the linker command. Following that, are `vpath` definitions, indicating where Make will search for the C files.

The depend target is declared as "phony" for good measure (as it is not a target that will exist as a file). The dependencies of the depend target are the three lists with the object files, except that the extension "`.o`" is replaced by "`.c`". This enables you to use the run-time macro `$(.SOURCES)` on the makedepend command line. When Make builds the file list for `$(.SOURCES)`, it looks through the `vpath` directories for the files, and it will prefix a file with the path where it is found. For example, if `bootLPC11Uxx.c` is found in `./startup/cmsis/`, the full path `./startup/cmsis/bootLPC11Uxx.c` is included in `$(.SOURCES)`.

## Dependencies for generated includes

When makedepend cannot access an include file, it issues a warning and does *not* add it to the dependency list for the target. In other words, an include file must exist (in the search paths), for makedepend to add it to the dependency list of the target.

When an include file is *generated* by a utility, that file would be missed by makedepend, because it does not exist *yet*. To remediate this, add the `-i` option to the makedepend command line. With this option, the warning message for a missing include file is suppressed and the name is added to the dependency list. However, this applies only to *user* includes, not *system* includes.

*User* includes refer to `#include` directives using double quotes, whereas *system* includes use angle brackets (`<...>`):

```
#include "config.h"   /* user include */
#include <stdio.h>    /* system include */
```

## Response file

For environments with limited command line lengths, makedepend can read the options for a *response file*. The response file is passed to makedepend with the following syntax (note the "@" character):

```
makedepend @options.txt
```

The response file may have all options split over multiple lines.

## Command line switches

`-D`*name*`=`*value*`, -D`*name*

Defines a symbol for the makedepend preprocessor (which works like the C/C++ preprocessor). When no value is explicitly defined, the symbol is defined as 1.

`-I`*path*

Adds a path to its list of directories that makedepend searches for files when it encounters a `#include` directive.

By default, makedepend appends the standard include directories at the end of the directory list. In Linux, Unix and OS/2, makedepend evaluates the `C_INCLUDE_PATH` environment variable for the standard includes; in Microsoft Windows, makedepend evaluates the `INCLUDE` environment variable. When the option `-I-` is set, makedepend does *not* append the standard include directories (and thus prevents makedepend from searching the standard include directories).

`-L`*path*

Adds a path for makedepend to locate *source* files. That is, if makedepend

does not find a source file that is specified on the command line in the active directory, it prepends the paths set this the `-L` option and tries again.

The `-L` option may appear multiple times. The paths are searched in the order that the `-L` options appear on the command line. The active directory is always searched first.

`-U`*`name`*

Undefines a symbol. This is mostly useful to remove a predefined symbol.

`-a` Accumulates the dependencies in the output file instead of removing the dependencies for files that are *not* listed on the command line of makedepend. With this option, you can call makedepend multiple times with different filename lists, and obtaining the accumulated dependencies of all those calls.

Note that dependencies for files that are listed on the command line are replaced in the output file. The `-a` option is not a simple “append”.

`-b` No backups. By default, makedepend copies the makefile to one with a “.bak” extension before modifying it. When this option is set, makedepend deletes the backup file.

`-c` Includes the C/C++ source file in the list of dependencies. By default, makedepend only lists all files *included* by the source file on the dependency line.

`-f`*`makefile`*

Sets the output filename, instead of the default name “makefile”. When set to “`-`” (i.e. option `-f-`), the output is sent to standard output (the console).

`-h` Help. Shows brief usage information for makedepend. Can be combined with `-v` for more verbose help.

`-i` Ignore include files that cannot be located. That is, do *not* warn about any include files that cannot be found. In addition, any missing include files are *still* added as a dependency to the target. (Without this option, include files that are not found, are omitted from the dependency list of the target; and a warning is issued.)

`-m` Enables warnings for multiple inclusion (of the same file). This option is provided to aid in debugging problems related to multiple inclusion.

`-o`*`suffix`*
: Object file suffix. The default suffix is "`.o`", as is common for Unix-like operating systems and the GNU GCC compiler suite. Om Microsoft Windows, the suffix "`.obj`" is more common, which can be set with the option `-o.obj`.

`-p`*`prefix`*
: Sets the text that is prepended to the name of each object file. This is usually used to designate a different directory for the object file.

  If the prefix pattern starts with a minus ("`-`"), that prefix is *removed* from the object filename (if the object filename starts with that prefix).

`-s`*`delimiter`*
: Sets the delimiter string below which makedepend writes the generated depedencies. The default delimiter string is:
  `# GENERATED DEPENDENCIES. DO NOT DELETE.`

`-v` Verbose operation: this option causes makedepend to show more notices and warnings during processing the files. It will also emit the list of files included by each input file.

`-w`*`width`*
: Sets the maximum line width for the lines written to the output file. The default value is 78 characters.

`-include` *`filename`*
: Processes the file includes it before processing each regular input file. This has the same affect as if the specified file were listed in an `#include` directive at the very to of each regular input file.

`--` *`options`* `--`
: Following a double hyphen (`--`), only the `-D`, `-I` and `-U` options are handled, and any others are silently ignored (source filenames are still handled). The intended purpose of this syntax is that the same `CFLAGS` macro that is passed to the C/C$^{++}$ compiler can also be passed to makedepend.

  A second double hyphen ends this special processing; it is needed if

options that are specific to makedepend follow the definitions in CFLAGS (or another makefile macro).

# Index

# D



# E



# F



# G



# I



# J

# T



# U



# V



# W