`gb-starter-kit` build system breakdown

Eldred Habert[*]

February 22, 2025

Summary and Intended Audience

This document is useful for people who wish to understand why gb-starter-kit's Makefile (build script) is written in the way that it is, why and how it does what it does, and perhaps also gain some insight into Makefile good practices.

It does contain a fairly quick primer on what a Makefile *is*, so it is intended to be suitable to people who have never touched a Makefile before.

Contents

[*]Contributions by JL2210, Damian Yerrick, Evie, and Starleaf.

1 Build rules

If you already know how a Makefile is structured, then you can skip ahead to subsection 1.1.

A Makefile's primary function is to describe how to build files from other files. Since it is *descriptive* in nature (like HTML), and not *imperative* like a programming language, Make (the program) is able to decide by itself how to perform the actual building optimally.

In their purest form, build **rules** look a little something like this:

```
hello.o: hello.c hello.h
    gcc -c -o hello.o hello.c
```

Here's how to read the above snippet:

The file **hello.o** is built from the files **hello.c** and **hello.h**.

To *update* **hello.o**, the following commands should be run:

- `gcc -c -o hello.o hello.c`

There's a lot to unpack here. First, a bit of terminology:

- `hello.o` is called a **target**, since it's what the rule aims to describe.
- `hello.c` and `hello.h` are called **prerequisites**, since they are necessary to build the "target". You can also think of them as *ingredients*.
- And finally, the list of commands is called the **recipe**.

Also, Make is designed to avoid running commands that are superfluous; this helps keep build times low. Make determines that a file is "out of date" by *comparing the time of the last modification* of itself, versus that of each of its dependencies. When Make determines that one of the **prerequisites** is "newer" than the **target** (or that the target doesn't exist at all), then the **recipe** is run.

Now, let's tweak the rule a little, for some extra conveniences.

For example, we might want to modify the C compiler we use, without having to search-and-replace in a lot of places in the Makefile (assuming there are many rules like this one). We can use **variables** for that.[1]

```
CC := gcc

hello.o: hello.c hello.h
    ${CC} -c -o hello.o hello.c
```

[1]It is possible to use parentheses instead of braces, but I've found it helpful to reserve braces for variables, since function calls (more on them later) require parentheses.

Variables in Make work essentially as if you copy-pasted their contents wherever you reference them. You could write `CC := gcc -g` and it would work.[2]

Another irk with that rule is that there is some repetition—the file names appear both in the first line and in the recipe! Thankfully, Make defines some ✨magic✨ variables that let us do away with that. They just... have *weird* names.[3]

```
hello.o: hello.c hello.h
	${CC} -c -o $@ $<
```

Here are the only four of these "automatic variables" you will probably ever need to know about:

$@	The rule's *current* target
$<	The first (leftmost) prerequisite
$^	*All* of the prerequisites
$*	The part of the file names matched by the `%` character

Ah, now's a good time to explain the `%` character. See, what we've examined thus far is an **explicit** rule: it explicitly specifies how to build one[4] file, and that's it. By contrast, an **implicit** rule describes how to make *any* file matching some criteria. The most common type of implicit rule is what's known as a **pattern rule**:

```
%.o: %.c %.h
	${CC} -c -o $@ $<
```

This starts getting more complicated: instead of defining a rule that clearly states how to make specific file(s), we now have a rule that, broadly, states how to make a *kind* of file.

When trying to look up info on a file, Make first looks for an explicit rule stating how to make it. If there are none, then it tries all pattern rules one by one: for each, it tries to replace the `%` in the target portion so that said portion matches the file's path. (There are some rules for choosing a specific one when multiple match at the same time, but if you want to get that technical, you should definitely start reading Make's documentation.)

Using automatic variables is largely a nicety when using explicit rules, but they become pretty much required when you start using implicit rules.

There is, however, a *very important* requirement for Make to accept an implicit rule as applicable: **all of its prerequisitites** must either already exist, or be able to be made (by explicit or implicit rules). This leads to a common source of confusion This causes, in particular, a commonly confusing error message, which we will illustrate using a little feline example.

Let's pretend that we are in a directory containing the following Makefile:

[2]This is part of why Make *really* can't deal with whitespace in file names.

[3]Oh, and, braces are not necessary around a variable's name when it's a single character. So you can write `$Q` as a shorthand for `${Q}` / `$(Q)`, for example.

[4]One or more, actually; but a finite number regardless.

```
%.meow: %.cat
    printf '%s goes "meow!"\n' "$$(cat $<)" >$@

%.gz: %
    gzip <$< >$@
```

...and that we run the following commands:

```
$ echo Pachatte >my.cat
$ make my.meow.gz # Ignore the `rm my.meow' line for now.
printf '%s goes "meow!"\n' "$(cat my.cat)" >my.meow
gzip <my.meow >my.meow.gz
rm my.meow
$ zcat <my.meow.gz # `zcat' prints the contents of gzipped files.
Pachatte goes "meow!"
$ ls # Notice the lack of `no.cat'...
Makefile  my.cat  my.meow.gz
$ make no.meow.gz
make: *** No rule to make target 'no.meow.gz'.  Stop.
```

You might expect Make to complain that `no.cat` does not exist, rather than that it doesn't know how to make `no.meow.gz`. *Clearly*, making a `.gz` file requires the corresponding file (`no.meow`), and that in turn `no.cat`!

Unfortunately, this is not how things go. Since `no.cat` doesn't exist, Make rejects the first pattern rule, and decides that it doesn't know how to make `no.meow`. In turn, this causes it to reject the second pattern rule, and decide that it doesn't know how to make `no.meow.gz` either!

> **tl;dr**: If Make complains that it doesn't know how to make a file when it looks like there is a pattern rule that *should* work: check that rule's prerequisites.

You can try passing the `-d` / `--debug` option to ask Make to "show its work", but note that Make has a **ton** of built-in implicit rules that will create a lot of noise, so I recommend also passing the `-r` / `--no-builtin-rules`.

Also, the `rm my.meow` command was inserted by Make itself, since it decided that `my.meow` is an intermediate file that doesn't need to be kept.

Now, let's start looking at actual rules from the Makefile.

1.1 Asset rules

Although these are probably not what you came here for, they are simpler than the assembling rules explained later, and so will serve as a more gentle introduction.

1.1.1 Graphics conversion

Both of these rules delegate the task to RGBGFX.

5 ⟨*Asset rules* 5⟩≡ (15c) 6a ▷

```
assets/%.2bpp: assets/%.png
	@mkdir -p "${@D}"
	${RGBGFX} -o $@ $<

assets/%.1bpp: assets/%.png
	@mkdir -p "${@D}"
	${RGBGFX} -d 1 -o $@ $<
```

These are almost identical, with the single difference that `.1bpp` files are converted with an additional `-d 1` option.

What's new, is the `@mkdir -p "${@D}"` line. It creates the directory that the target will be created into; but how exactly does it do that?

Let's begin with the `@` sign: it *suppresses command echo*. If you have used Make, you probably noticed that it prints every command right before executing it.[5] But this `mkdir` command is not really interesting, and you will see later that it's present *everywhere*. Having it echoed all the time would pretty much drown out the commands we *really* care about; so, we suppress it.

On to the command itself: `mkdir` stands for "make directory", and it pretty much does what it says on the tin. Now, `mkdir` has some interesting behaviour: by default, `mkdir a/b` errors out if **a doesn't exist**, and also if **a/b already exists**. The `-p` option, which can also be written as `--parents`, is designed to suppress the former behaviour ("create the directory I'm interested in, but also all of its parents as necessary"), but interestingly it *also* suppresses the latter behaviour.

And, finally, we should talk about that little `${@D}` nugget. (I will not go into detail about the quotes, because they are passed as-is by Make to the shell, and shell quoting is an entire rabbit hole that doesn't really belong here.) See, for each automatic variable (here, `$@`, the short form of `${@}`), Make defines two extra variables: the one with an extra `D` contains the **d**irectory portion of its contents, and the one with an extra `F` contains the **f**ile portion.

All this might have been a little too theoretical and confusing, so let's try making it more concrete. Say Make is trying to build a file located at `assets/player/running.2bpp`. Substituting the "pattern" (`%`), we get:

```
assets/player/running.2bpp: assets/player/running.png
	${RGBGFX} -o $@ $<
```

And, replacing the automatic variables:

```
assets/player/running.2bpp: assets/player/running.png
	${RGBGFX} -o assets/player/running.2bpp assets/player/running.png
```

Now, picture a project that contains hundreds of `.png` files. Instead of having hundreds of bespoke rules, this one little rule can serve all of them!

[5] If you find this annoying, you can pass the `-s` "silent" option to Make.

1.1.2 Compression

Compression can be important, as one can run out of ROM size quicker than you'd expect. Smaller data also means it's easier to fit all of it in the same bank, and any bankswitch removed from code is always a win!

Compression methods are a complex topic, so gb-starter-kit provides one that's known to be relatively easy to use *and* compresses tile data reasonably well.

6a ⟨*Asset rules* 5⟩+≡ (15c) ◁5 6b▷

```
assets/%.pb16: src/tools/pb16.py assets/%
	@mkdir -p "${@D}"
	$^ $@$
```

This rule contains a cheeky little trick: its use of `$^`. See, the first prerequisite is the script that needs to be executed to perform the compression, so after `$^` is expanded, it looks like `src/tools/pb16.py assets/player/running.2bpp $@`—and this looks a lot like a command, doesn't it?

Anyhow, this rule is a good occasion to talk about what I like to call “rule chaining”. For example, if you ask Make to `make assets/player/running.2bpp.pb16`, then the following happens:

1. Make checks if `assets/player/running.2bpp.pb16` exists. (Let's pretend it does not.)
2. Make checks if it knows how to make that file. It finds the `assets/%.2bpp` rule!
   (a) Make checks if `assets/player/running.2bpp` exists. (Let's pretend that one does not exist either.)
   (b) Make checks if it knows how to make that file. It finds the `assets/%.png` rule!
      i. Make checks if `assets/player/running.png` exists. It finds that file!
   (c) Make now converts the `.png` file into `.2bpp`.
3. Make now converts the `.2bpp` file into `.2bpp.pb16`.

It's arguably a bit of an involved process, but that way, Make is able to run the last two steps in sequence, and creates `assets/player/running.2bpp.pb16` just from `assets/player/running.png`. I've seen people getting confused by the two steps involved in what can otherwise seem like a single operation (running `make` just once), so hopefully this can clear it up for you.

6b ⟨*Asset rules* 5⟩+≡ (15c) ◁6a 7a▷

```
assets/%.pb16.size: assets/%
	@mkdir -p "${@D}"
	printf 'def NB_PB16_BLOCKS equ ((%u) + 15) / 16\n' \
	    "$$(wc -c <$<)" >assets/$*.pb16.size
```

✓ The lone backslash (\) is simply a line continuation character, because this line is *really* long, and would overflow the box above otherwise!

Notice the `$$`: because we want that dollar sign to be interpreted by the shell, not by Make, we have to escape it... and that's done by doubling it[6].

Then, we have pretty much a copy-paste of the above two, but for the PackBits**8** compression scheme. Where PB16 performs well on 2bpp tile data, PB8 does better on 1bpp tile data.

7a ⟨*Asset rules* 5⟩+≡ (15c) ◁6b

```
assets/%.pb8: src/tools/pb8.py assets/%
	@mkdir -p "${@D}"
	$^ $@

assets/%.pb8.size: assets/%
	@mkdir -p "${@D}"
	printf 'def NB_PB8_BLOCKS equ ((%u) + 7) / 8\n' \
		"$$(wc -c <$<)" >assets/$*.pb8.size
```

1.1.3 VPATH

Now, as we will see later, the `assets/` directory gets entirely removed when running `make clean` (the "restart from scratch" command). Yet, as we've seen above, asset rules convert files in `assets/` into other files in `assets/`. Making two versions of each rule (one expecting files from `assets/`, and the other from `src/assets/`) is a possibility, but it would very annoying.

7b ⟨*VPATH* 7b⟩≡ (15c)

```
VPATH := src
```

This line sets the `VPATH` variable, which is special to Make: when it fails to find a file, it tries again by prepending `src/` to the path. So, assets not found in `assets/` are also looked for in `src/assets/`, which is *not* cleared by `make clean`!

1.2 Assembly rules

Let's get to business!

Building a ROM with RGBDS has three steps to it, which I will now sample from its manual:

1. Assemble source files into one object file each;

2. Link all of the object files into a "raw" ROM;

[6]Internally, this is done by having a variable called `$`, and whose contents are a single dollar sign. Yes, `${$}` is valid, but why would you do that!?

3. “Fix” the ROM so the console will accept it

> If you are wondering why there is more than one step, the GB ASM tutorial has you covered.

1.2.1 Assembling source files

Here is how we create an object file:

8a ⟨*Assembly rules* 8a⟩≡ (15c) 8b ▷

```
obj/%.o: obj/%.mk
	@touch -c $@
```

Wait... what? `touch` only pretends to modify the file, so that Make doesn't re-run the rule[7]. And what's a `.mk` file, anyway?

Well, we need to talk about the ✨dependency auto-discovery✨.

1.2.2 Dependency auto-discovery

For RGBASM to assemble a file, it must also be able to read every file that gets `INCLUDE`d or `INCBIN`'d. But, we don't know what files need to be made ahead of time! (Unless you want to make an exhaustive list in the Makefile, but believe me, that gets tiring *fast*.)

So, we instead make use of RGBASM's ability to inform Make of the files it needs to assemble. This is achieved using its `-M` option, specifying the files that are generated using `-MQ` (both the `.mk` file *and* the `.o` one!). The `-MP` option is explained in the manual, but `-MG` is worth extra attention.

`-MG` tells RGBASM that some of the files it it told to `INCLUDE` and/or `INCBIN` may be missing; and that if this were to happen, instead of erroring out like usual, it should note them in the `.mk` file, and exit *normally*. When this happens, as we'll discuss more below, this will cause Make to “reload” its dependency information from the modified `.mk` file, (try to) make the newly discovered dependencies, and then re-run RGBASM, until the `.o` file is successfully created!

8b ⟨*Assembly rules* 8a⟩+≡ (15c) ◁ 8a 9a ▷

```
obj/%.mk: src/%.asm
	@mkdir -p "${@D}"
	${RGBASM} ${ASFLAGS} -o ${@:.mk=.o} $< \
	    -M $@ -MG -MP -MQ ${@:.mk=.o} -MQ $@
```

The `${@:.mk=.o}` syntax means “expand to the contents of `$@`, but replace the `.mk` file extension with `.o`”.

[7] `-c` ensures that the file isn't created if it doesn't already exist.

✖ Savvy users of Make might suggest merging the `touch` rule above with this one. But that would be incorrect! Because, then, Make assumes that the rule *will* create **both** files, even though this rule needs to be executed many times for the `.o` one to end up being created; this causes, in particular, spurious build failures, or *some* versions of Make running the same command over and over forever.

We also need to inform Make of the dependency files that it should read; but, we don't do so when running `make clean`, as we don't need dependency info when we're trying to wipe the slate clean!

9a ⟨*Assembly rules* 8a⟩+≡ (15c) ◁8b 9b▷

```
ifeq ($(filter clean,${MAKECMDGOALS}),)
include $(patsubst src/%.asm,obj/%.mk,${SRCS})
endif
```

Interestingly, Make also treats every such file as a target that needs to be made, and so it will automatically try to generate them if they don't exist yet.

Importantly also, `include` causes Make to restart from scratch after one of the included files has been modified; this is how we get the "progressively retrying" behaviour we want for dependency auto-discovery.

1.2.3 Linking and fixing the ROM

Once all of the object files are generated, we can link them all together (notice the use of `$^`!), and fix the ROM.

Notice that this also unconditionally assembles the build date file. That ensures that the build date is always refreshed.

9b ⟨*Assembly rules* 8a⟩+≡ (15c) ◁9a 10a▷

```
${ROM}: $(patsubst src/%.asm,obj/%.o,${SRCS})
	@mkdir -p "${@D}"
	${RGBASM} ${ASFLAGS} -o obj/build_date.o src/assets/build_date.asm
	${RGBLINK} ${LDFLAGS} -m bin/$*.map -n bin/$*.sym -o $@ $^ \
	&& ${RGBFIX} -v ${FIXFLAGS} $@
```

1.2.4 Submodules

A Git "submodule" is, largely, a Git repository inside of a Git repository; this has the benefit of making it easier to update the submodules, but the downside of being a little quirky.

By default, cloning the repo does not initialise submodules (so they are empty); if that happens, Make would normally fail with some "file not found" error, but this rule makes it print a more user-friendly error message instead.

Note that the real paths aren't used! Since RGBASM fails to find the files, it outputs the path(s) as passed to `include`, not where the file would actually be found (e.g. `src/hardware.inc/hardware.inc`).

10a ⟨*Assembly rules* 8a⟩+≡ (15c) ◁9b

```
hardware.inc/hardware.inc rgbds-structs/structs.asm:
    @echo '$@ is not present; have you initialized submodules?'
    @echo 'Run `git submodule update --init`,'
    @echo 'then `make clean`,'
    @echo 'then `make` again.'
    @echo 'Tip: to avoid this, use `git clone --recursive` next time!'
    @exit 1
```

You can see that there aren't any prerequisite files here—this is normal. Since there are no prerequisites, this rule can only trigger if the target file doesn't exist—precisely what we want!

2 Phony rules

Consistently with our trend of lacking consistency, let's now talk about *rules that aren't really rules*: so-called "phony rules".

These are handy as quick little aliases: typing `make all` is much easier to remember and much less tedious to type than `make bin/dinosaurs_with_lasers.gb`, for example. Since this is a commonly desirable feature, these names are common conventions.

2.1 Building the ROM

`all` is the conventional "build the things I am likely to want"—in our case, the ROM and accompanying debug information. It is also a little special, as it is the *first* rule in the file (as we will see in section 4), it is also the "default" target, i.e. what Make selects as its target if invoked without one (just `make`).

10b ⟨*Phony rules* 10b⟩≡ (15c) 11a▷

```
all: ${ROM}
.PHONY: all
```

2.2 Cleaning temporary and final files

`clean` is the conventional "forget everything prior" target; it causes everything generated by prior invocations of Make to be deleted.

(As we have seen in subsubsection 1.2.2, we also made it a little special, its presence suppressing dependency auto-discovery.)

11a ⟨*Phony rules* 10b⟩+≡ (15c) ◁10b

```
clean:
    rm -rf bin obj assets
.PHONY: clean
```

3 Configuration

Time for piles of variables!

First, this block allows one to customise where the build process will look for RGBDS. It is possible to customise the location (or even name!) of each of the programs individually, or in bulk thanks to the `RGBDS` variable.

11b ⟨*Configuration* 11b⟩≡ (15b) 11c▷

```
RGBDS   ?=
RGBASM  := ${RGBDS}rgbasm
RGBLINK := ${RGBDS}rgblink
RGBFIX  := ${RGBDS}rgbfix
RGBGFX  := ${RGBDS}rgbgfx
```

The `?=` assignment may raise some eyebrows. Where `:=` sets the value of a variable, `?=` also sets it *except* if an environment variable of the same name was passed to Make, in which case that environment variable takes precedence.

This allows the following to work:

```
$ ls ~/rgbds-0.9.1
rgbasm  rgbfix  rgbgfx  rgblink
$ export RGBDS=~/rgbds-0.9.1
$ make
~/rgbds-0.9.1/rgbasm -p 0xFF (etc.)
```

Then, we have some options that will be passed to the RGBDS programs.

11c ⟨*Configuration* 11b⟩+≡ (15b) ◁11b 11d▷

```
INCDIRS  := src/ include/
WARNINGS := all extra
ASFLAGS  = -p ${PADVALUE} $(addprefix -I,${INCDIRS}) $(addprefix -W,${WARNINGS})
LDFLAGS  = -p ${PADVALUE}
FIXFLAGS = -p ${PADVALUE} -i "${GAMEID}" -k "${LICENSEE}" -l ${OLDLIC} -m ${MBC} -
```

And some file names:

11d ⟨*Configuration* 11b⟩+≡ (15b) ◁11c 12a▷

```
ROM = bin/${ROMNAME}.${ROMEXT}
SRCS := $(call rwildcard,src,*.asm)
```

And, last but not least, including the file containing project-specific configuration.

12a ⟨*Configuration* 11b⟩+≡ (15b) ◁11d

```
include project.mk
```

Rather than modifying this Makefile for each project, for example to change the ROM's file name, this can be used to override the above default configuration.

You may have noticed that some of the variable assignments above used `:=`, but others used `=`. What's the difference? A variable assigned with `:=` has its value *immediately* computed; however, `=` instead has the variable's value computed each time it it referenced. In particular, this allows the variables to reference variables that don't exist yet (`PADVALUE` is defined after `ASFLAGS`, in `project.mk`).

3.1 Project-specific configuration

Let's detail the contents of that file. The description of RGBFIX's options can be useful to understand what all of these options do, and especially the syntax that they accept.

3.1.1 ROM header

The following control the various options passed to RGBFIX, which set the fields of the ROM's header. Emulators *do* rely on some of these; but it can be cool to customise even the "useless" ones, since they show up here and there. Some people *will* notice, and find it cute or clever!

This is the ROM's version. This typically starts at 0, and is incremented for each published version.

12b ⟨*project.mk* 12b⟩≡ 12c▷

```
VERSION := 0
```

This is the game's ID, which should be 4 characters (preferably unaccented letters and/or digits).

12c ⟨*project.mk* 12b⟩+≡ ◁12b 12d▷

```
GAMEID := BOIL
```

The game's title can be up to 11 characters, again preferably unaccented letters and/or digits.

12d ⟨*project.mk* 12b⟩+≡ ◁12c 13a▷

```
TITLE := BOILERPLATE
```

These two control the licensee code, which should be two characters (like usual). You should keep the old code at 0x33, as this is required to get SGB compatibility (with no drawbacks). The default is meant to mean "**h**ome**b**rew game"! 😉

13a ⟨*project.mk* 12b⟩+≡ ◁12d 13b▷

```
LICENSEE := HB
OLDLIC := 0x33
```

The cartridge type controls the available features of the emulated cartridge, especially ROM and SRAM banking. You can get a list of valid values by running `rgbfix -m help`. (If using a no-MBC setup, consider enabling `-t` in subsubsection 3.1.2.)

13b ⟨*project.mk* 12b⟩+≡ ◁13a 13c▷

```
MBC := MBC5
```

This is the size of the on-board SRAM. It needs to be consistent with the MBC type above: this should be zero if and only if the MBC type doesn't include "RAM"[8], and vice-versa.

13c ⟨*project.mk* 12b⟩+≡ ◁13b 13d▷

```
SRAMSIZE := 0x00
```

If you're wondering where the ROM size parameter is—that one is automatically computed by RGBFIX.

3.1.2 Compatibility settings

Uncomment any of these to apply them, or comment them to remove them; please refer to RGBDS' documentation (offline: `man 1 rgbasm`) The defaults should be sensible for most projects, though.

These two control the Game Boy Color compatibility byte. If your game is intended to run on GBC *and* monochrome consoles (including possibly SGB), uncomment the `-c` line. If it is intended to run on GBC *only*, then you should uncomment the `-C` line **and** still present some kind of fallback screen if detecting a non-Color Game Boy: the monochrome consoles themselves **do not** check the header!

13d ⟨*project.mk* 12b⟩+≡ ◁13c 13e▷

```
# FIXFLAGS += -c
# FIXFLAGS += -C
```

This flag simply sets the SGB compatibility flag.

13e ⟨*project.mk* 12b⟩+≡ ◁13d 14a▷

```
# FIXFLAGS += -s
```

[8]This indicates the size of a separate ("discrete") RAM chip, so MBC2's built-in SRAM doesn't count and this should be set to 0.

If you only intend your game to run on monochrome systems, these two flags can be useful: they set up RGBDS' memory layout to be more convenient for you.

14a ⟨*project.mk* 12b⟩+≡ ◁13e 14b▷

```
# LDFLAGS += -d
# LDFLAGS += -w
```

And, finally, if you don't want to use a MBC, this sets RGBDS' memory layout in "tiny" mode, which is more appropriate and convenient for such projects.

14b ⟨*project.mk* 12b⟩+≡ ◁14a 14c▷

```
# LDFLAGS += -t
```

3.1.3 Miscellanea

This defines the value that the ROM will be filled with. The default value of `0xFF` is actually significant: it encodes the `rst $38` instruction, which helps catch runaway execution (say, dereferencing a bad jump table index, or forgetting a `ret`...) by making it jump to `$0038`, where a crash handler is located (by default).

14c ⟨*project.mk* 12b⟩+≡ ◁14b 14d▷

```
PADVALUE := 0xFF
```

This sets the ROM's file name.

14d ⟨*project.mk* 12b⟩+≡ ◁14c

```
ROMNAME := boilerplate
ROMEXT  := gb
```

4 Overall structure

Here is where we collect all the things we have seen thus far.

4.1 Make configuration

First, we disable a lot of Make's built-in rules, since they are at best useless to us. This also improves build time somewhat.

14e ⟨*Makefile* 14e⟩≡ 15a▷

```
.SUFFIXES:
```

Parallel builds are broken with macOS' bundled version of Make; please see this issue comment for a technical explanation. If you are using macOS, please consider installing Make from Homebrew (`brew install make`, **make sure to read the caveats it prints**). Delete the `.NOTPARALLEL` line if you want to have parallel builds regardless!

15a ⟨*Makefile* 14e⟩+≡ ◁14e 15b▷

```
ifeq (${MAKE_VERSION},3.81)
.NOTPARALLEL:
endif
```

4.2 Miscellanea

Here, we have a "recursive `$(wildcard)`" function.

15b ⟨*Makefile* 14e⟩+≡ ◁15a 15c▷

```
rwildcard = $(foreach d,$(wildcard $(1:=/*)),$(call rwildcard,$d,$2) $(filter $(su

⟨Configuration 11b⟩
```

4.3 Rules

Phony rules must come before other rules so that `all` is the default target.

15c ⟨*Makefile* 14e⟩+≡ ◁15b

```
⟨Phony rules 10b⟩

⟨VPATH 7b⟩
⟨Asset rules 5⟩
⟨Assembly rules 8a⟩
```