GNUstep Makefile Package Internals

Nicola Pero n.pero@mi.flashnet.it

last days of June 2001 - revised end of July 2001

Contents

1 Introduction

This short document attempts to explain simply how the gnustep makefile package works internally. When I first wrote this document, the mechanism used to be extremely complex, involving many recursive make invocations; I have now simplified it so that it involves only a single recursive make invocation per target/type/operation. As a result, I hope that the mechanism is now so simple that you can figure out how it works without reading this document, by just reading the gnustep-make source code. Anyway, the thing might still not be still totally trivial at a first approach, so this document might help you to get familiar with the gnustep-make source code in a shorter time and with less pain.

2 From 'make' to the internal-all rule

Imagine for example that in your `GNUmakefile` you include both `tool.make` and `library.make`, as in the following example:

```
include $(GNUSTEP_MAKEFILES)/common.make

TOOL_NAME = decrypt
decrypt_OBJC_FILES = decrypt.m

LIBRARY_NAME = libDvd
libDvd_OBJC_FILES = decss.m
```

```
include $(GNUSTEP_MAKEFILES)/tool.make
include $(GNUSTEP_MAKEFILES)/library.make
```

Then you type 'make' on the command line. We want to understand what happens.

Make will process your `GNUmakefile`, which includes `tool.make`, and that will include `rules.make`. In `rules.make` make finds the first rule (the one which is executed), which is

```
all:: before-all internal-all after-all
```

This means that make will build by default that target `all`, and that building that target requires building the `before-all`, `internal-all` and `after-all` targets. We ignore the `before-all` and `after-all` targets for now, and only concentrate on the core target, which is `internal-all`.

3 From the internal-all rule to the %.variables rule

Make needs to build this target `internal-all`. In `rules.make` this target appears as

```
internal-all::
```

because of the double colons (that is, because it is `internal-all::` rather than `internal-all:`) this target can have multiple totally separated rules. Each rule must be a double colon rule, and is processed separately from the other rules (even if they refer to the same target).

The real rules for `internal-all` are included by the specific makefiles; in our example, `tool.make` includes

```
internal-all:: $(TOOL_NAME:=.all.tool.variables)
```

now - in our case - because `TOOL_NAME` is `decrypt`, then this rule actually means

```
internal-all:: decrypt.all.tool.variables
```

This means that to build `internal-all`, make has to build (at least) the `decrypt.all.tool.variables` target. `library.make` includes the completely analogous rule

```
internal-all:: $(LIBRARY_NAME:=.all.library.variables)
```

which in our case means

```
internal-all:: libDvd.all.library.variables
```

This rule is completely separated from the other one; to build `internal-all`, make has to build the two different targets:

```
decrypt.all.tool.variables
libDvd.all.library.variables
```

4 The %.variables rule - dependencies

The rule for building these targets is in the `rules.make` file, it is the `%.variables` rule:

```
%.variables: %.tools %.subprojects
@ \
target=$(basename $(basename $*)); \
operation=$(subst .,,$(suffix $(basename $*))); \
type=$(subst -,_,$(subst .,,$(suffix $*))); \
echo Making $$operation for $$type $$target...; \
$(MAKE) -f $(MAKEFILE_NAME) --no-print-directory --no-keep-going \
  internal-$${type}-$$operation \
  INTERNAL_$${type}_NAME=$$target \
  TARGET=$$target \
  _SUBPROJECTS="$($(basename $(basename $*))_SUBPROJECTS)" \
  ...
```

This rule matches all targets ending in `.variables`. First of all, the rule depends on the corresponding `%.tools` and `%.subprojects` rules. This is because before processing the target itself, gnustep-make needs to process the related subprojects and (only for frameworks) the framework tools. We ignore this complication of subprojects and framework tools for now; if you look at the `%.subprojects` and `%.tools` rules you see that they do nothing if you are not actually using subprojects or framework tools in your makefile.

5 The %.variables rule - second make invocation

The rule body parses the `%.variables` string - for example when the rule is applied to

```
decrypt.all.tool.variables
```

then (remember that `$*` is the stem of the rule, `decrypt.all.tool` in this case) it extracts

```
target=decrypt
operation=all
type=tool
```

and then it runs a make subprocess, specific to that target, type and operation. In our case, the `%.variables` rules is executed twice, once to build

```
decrypt.all.tool.variables
```

and once to build

```
libDvd.all.tool.variables
```

so the result is to run two separate make processes:

```
make internal-tool-all INTERNAL_tool_NAME=decrypt TARGET=decrypt \
  _SUBPROJECTS="$(decrypt_SUBPROJECTS)" \
  OBJC_FILES="$(decrypt_OBJC_FILES)" \
  ...
make internal-library-all INTERNAL_library_NAME=libDvd TARGET=libDvd \
  _SUBPROJECTS="$(libDvd_SUBPROJECTS)" \
  OBJC_FILES="$(libDvs_OBJC_FILES)" \
  ...
```

where `...` stands for a lot of other variables, including all variables needed to perform the final stage: `OBJC_FILES`, `C_FILES`, `JAVA_FILES`, `ADDITIONAL_INCLUDE_DIRS` etc. Note that each make subprocess will get passed different, specific, variables. If gnustep-make wants to modify these variables in some way, it does it at this stage, before passing them to the submake process. For example, some library flags are filtered through the `WHICH_LIB_SCRIPT`.

What this means is that for each target/type/operation, a separate make process is run. For example, if you have two tools, `decrypt` and `crypt`, and you want to both compile and install them, it will run four make subprocesses:

```
make internal-tool-all INTERNAL_tool_NAME=decrypt ...
make internal-tool-all INTERNAL_tool_NAME=crypt ...
make internal-tool-install INTERNAL_tool_NAME=decrypt ...
make internal-tool-install INTERNAL_tool_NAME=crypt ...
```

This is the 'second make invocation'. The make package knows that it is being invoked for the second time, because of the `INTERNAL_tool_NAME` being non-empty.

6 Second make invocation

Because of the `INTERNAL_tool_NAME` variable being a non-empty string (while it was empty in the previous top-level invocation), `tool.make` will include the actual rules to build the tool; in particular, the `internal-tool-all` rule, which is then executed and builds the tool. All variables such as `OBJC_FILES` or the library flags are now available directly in the makefiles, they have already been prepared and preprocessed, so that the rules in `tool.make` can just plainly use these variables naively to perform their job (compiling, installing, or whatever).