# Writing GNUstep Makefiles

Nicola Pero n.pero@mi.flashnet.it

June 2000 AD

## 1 What is it

The GNUstep make package is aimed at providing a simple and automatic way to manage compilation of GNUstep projects on different machines and environments. Your project will remain completely portable to any platform running GNUstep without the need to (explicitly) use complex packages such as autoconf or automake.

## 2 A First Tool

Let's try it out by making a little command line tool using the GNUstep make package. Let's start by creating a directory to hold our project. In this directory, type the following extremely simple program in a file called say `source.m`.

```
#import <Foundation/Foundation.h>

int
main (void)
{
  NSLog (@"Executing");
  return 0;
}
```

The function `NSLog` simply outputs the string to `stderr`, flushing the output before continuing. To compile this little program as a command line tool called `LogTest`, add in the same directory a file called `GNUmakefile`, with the following contents:

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

TOOL_NAME = LogTest
LogTest_OBJC_FILES = source.m

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

And that's it. At this point, you have all the usual standard GNU make options: typically make, make clean, make install, make distclean. For example, typing `make` in the project directory should compile our little tool. It should create a single executable `LogTest`, and put it in the subdirectory

```
shared_obj/ix86/linux-gnu/gnu-gnu-gnu-xgps
```

(or in a similar one, according to your system). To install the tool, simply type `make install`; you usually need to be root to install the tool on a system directory. If you want to have it installed in your own user GNUstep directory (eg, `/home/nicola/GNUstep`), which doesn't require you to be root and could be a better place for testing, you just need to add the line

```
GNUSTEP_INSTALLATION_DIR = $(GNUSTEP_USER_ROOT)
```

after including `common.make`, as follows:

```
include $(GNUSTEP_MAKEFILES)/common.make
GNUSTEP_INSTALLATION_DIR = $(GNUSTEP_USER_ROOT)

TOOL_NAME = LogTest
LogTest_OBJC_FILES = source.m

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

I usually do this when testing my own code and programs, and it is very handy.

# 3 Enabling Debugging

To compile this tool with debugging enabled, type in:

```
make debug=yes
```

This will create an executable with debugging symbols (i.e., compiled with the `-g` option), useful for debugging it with gdb; it will also compile the tool using the `-DDEBUG` compiler flag, which defines the preprocessor symbol `DEBUG` during the compilation. In this way, you may isolate code to be executed only when compiling with the debug option typically as follows:

```
#ifdef DEBUG
  /* Code compiled in only when debug=yes */
#endif
```

The debugging executable will be placed in:

```
shared_debug_obj/ix86/linux-gnu/gnu-gnu-gnu-xgps
```

this allows you to keep both the debugging and the not-debugging executables, since they are in different trees. To install the debugging version, type `make debug=yes install` (note: this will overwrite the not-debugging version, if any, in the installation directory; only one can be installed at a time). To clean the debugging version, type `make debug=yes clean`.

# 4 A first App

Let's try now to compile an application. Modify our source file `source.m` to read

```
#import <Foundation/Foundation.h>
#import <AppKit/AppKit.h>

int
main (void)
{
  NSAutoreleasePool *pool;

  pool = [NSAutoreleasePool new];

  [NSApplication sharedApplication];

  NSRunAlertPanel (@"Test", @"Hello from the GNUstep AppKit",
                   nil, nil, nil);

  return 0;
}
```

(Ignore the autorelease pool code for now - we'll cover autorelease pools in detail later). The line containing `sharedApplication` initializes the GNUstep GUI library; then, the following line runs an alert panel. To compile it, we rewrite the GNUmakefile as follows:

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

APP_NAME = PanelTest
PanelTest_OBJC_FILES = source.m

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

And that's it. To compile, type in `make`. The result is slightly different from a command line tool. When building an application, the application usually has a set of resources (images, text files, sound files, bundles, etc) which comes with the application. In the GNUstep framework, these resources are stored with the application executable in an 'application directory', named after the application, with `app` appended. In this case, after compilation the directory `PanelTest.app` should have been created. Our executable file is inside this directory; but the correct way to run the executable is through the `openapp` tool, in the following way:

```
openapp PanelTest.app
```

(openapp should be in your path; if it is not, you should check that GNUstep is properly installed on your system).

# 5 Debugging an Application

Debugging an application is quite simple. You compile it with debugging enabled, as in

```
make debug=yes
```

The application directory is then created with a different name: `debug` is appended instead of `app` (again, this allows you to keep both the debug and not-debug versions at the same time). In the example, the application directory would be called `PanelTest.debug`.

To debug the application, use `debugapp` instead of `openapp`:

```
debugapp PanelTest.debug
```

This will run `gdb` (the GNU debugger) on the executable setting everything ready for debugging.

# 6 Preamble and Postamble

You may happen to need to pass additional flags to the compiler (in order to link with additional libraries, for example) or to be willing to perform some additional actions after compilation or installation. The standard way of doing this is as follows: add a file called `GNUmakefile.preamble` to your project directory. An example of a `GNUmakefile.preamble` is the following:

```
ADDITIONAL_OBJCFLAGS += -Wall
```

This simply adds the `-Wall` flag when compiling (by the way, it is good practice to always use this flag). In general, you would use a `GNUmakefile.preamble` to add any additional flags you need (to tell the compiler/linker to search additional directories upon compiling/linking, to link with additional libraries, etc).

Now, you would want your `GNUmakefile` to include the contents of your `GNUmakefile.preamble` before any processing. This is usually done as follows:

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

APP_NAME = PanelTest
PanelTest_OBJC_FILES = source.m

include GNUmakefile.preamble
include $(GNUSTEP_MAKEFILES)/application.make
```

The most important thing to notice is that the `GNUmakefile.preamble` is included *before* `application.make`. That is why is called a preamble.

Sometimes you also see people using

```
-include GNUmakefile.preamble
```

(with a hyphen, `-`, prepended). The hyphen before `include` tells the make tool not to complain if the file `GNUmakefile.preamble` is not found. If you want to make sure that the `GNUmakefile.preamble` is included, you should better not use the hyphen.

If you want to perform any special operation after the GNUmakefile package has done its work, you usually put them in a `GNUmakefile.postamble` file. The `GNUmakefile.postamble` is included *after* `application.make`; that is why is called a postamble:

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

APP_NAME = PanelTest
PanelTest_OBJC_FILES = source.m

include GNUmakefile.preamble
include $(GNUSTEP_MAKEFILES)/application.make
include GNUmakefile.postamble
```

Here is a concrete example of a `GNUmakefile.postamble`:

```
after-install::
	$(MKDIRS) /home/nicola/Tools; \
	cd $(GNUSTEP_OBJ_DIR); \
	$(INSTALL) myTool /home/nicola/Tools;
```

(make sure you start each indented line with `TAB`). This will install the tool `myTool` in the directory `/home/nicola/Tools` after compilation.

You rarely need to use `GNUmakefile.postamble`s, and they were mentioned mainly to give you a complete picture.

# 7 Further Reading

For further examples and information on `GNUmakefile`s, you may want to have a look at the various test apps and tools in the GNUstep core library.