# More on GNUstep Makefiles

Nicola Pero nicola.pero@meta-innovation.com

First Edition: November 2000; Last Updated: March 2010

## 1 Introduction

In this tutorial we will learn something more about the GNUstep makefile package. We will cover libraries, and aggregate projects.

## 2 Libraries

### 2.1 Compiling A Library

We start with a very simple example of a library. Our tiny library will contain a single class, called `HelloWorld`, which has a method to print out a nice string.

The library has only one header file (called `HelloWorld.h`), which is the following:

```
#ifndef _HELLOWORLD_H_
#define _HELLOWORLD_H_

#include <Foundation/Foundation.h>

@interface HelloWorld : NSObject
+ (void) printMessage;
@end

#endif /* _HELLOWORLD_H_ */
```

(This header file quite simply says that `HelloWorld` is a subclass of `NSObject`, and implements a single class method `printMessage` [a *class* method is what in java would be called a *static* method]; the `#ifdef`s are the standard way of protecting a C header file from multiple inclusions).

The source code of our class is in the file `HelloWorld.m`, and is the following:

```
#include "HelloWorld.h"

@implementation HelloWorld
+ (void) printMessage
{
  printf ("Hello World!\n");
}
@end
```

(This implements the `printMessage` class method for the class `HelloWorld`, and all what this method does is printing out `Hello World!`.)

To compile our library, we create a GNUmakefile as follows:

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

LIBRARY_NAME = libHelloWorld
libHelloWorld_HEADER_FILES = HelloWorld.h
libHelloWorld_HEADER_FILES_INSTALL_DIR = HelloWorld
libHelloWorld_OBJC_FILES = HelloWorld.m

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

The main differences with the `GNUmakefile` for a tool or an application are that we include `library.make` instead of `tool.make` or `application.make`, and that we set the `xxx_HEADER_FILES` variable to tell the make system which are the library header files. This is quite important because the header files will be installed with the library when the library is installed.

In order to do things cleanly, each library should install its headers in a different directory, so headers from different libraries don't get mixed and confused; this is why we specify that our header file has to be installed in a `HelloWorld` directory:

```
libHelloWorld_HEADER_FILES_INSTALL_DIR = HelloWorld
```

As a consequence, an application or a tool which needs to use the library will include the header file by using

```
#include "HelloWorld/HelloWorld.h"
```

because we have installed it into the `HelloWorld` directory.

As usual, to compile type `make` and to install type `make install`.

## 2.2 Linking your app or tool against a GNUstep library

In our first example, we want to write a tiny tool which uses our `libHelloWorld`. The tool source code is in the file `main.m`, which is the following:

```
#include <Foundation/Foundation.h>
#include <HelloWorld/HelloWorld.h>

int main (void)
{
  [HelloWorld printMessage];

  return 0;
}
```

(We invoke the `printMessage` method of the `HelloWorld` class, then exit.).

We write our usual GNUmakefile (but including GNUmakefile.preamble):

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

```
TOOL_NAME = HelloWorldTest
HelloWorldTest_OBJC_FILES = main.m

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

Then, here is the GNUmakefile.preamble, in which we tell the make package about the library we want to link against:

```
HelloWorldTest_TOOL_LIBS += -lHelloWorld
```

If you have correctly installed the library `HelloWorld`, this is all you need to do. If you needed to link against more than one library, you would simply put them on the same line, as in:

```
HelloWorldTest_TOOL_LIBS += -lHelloWorld -lHelloMoon
```

which links against the two libraries `HelloWorld` and `HelloMoon`.

If `HelloWorld` were an application, you would need to use

```
HelloWorldTest_GUI_LIBS += -lHelloWorld
```

(the difference is `GUI` instead of `TOOL`).

## 2.3 Linking against an external library

If the library you want to link against is not a GNUstep library (ie, not managed by the GNUstep make package), for example a C library you get from an external source, you need to tell the GNUstep make package where the library can be found. In this case, your GNUmakefile.preamble would look something like the following:

```
HelloWorldTest_TOOL_LIBS    += -lNicola
HelloWorldTest_INCLUDE_DIRS += -I/opt/nicola/include/
HelloWorldTest_LIB_DIRS     += -L/opt/nicola/libs/
```

where I am linking against the library `libNicola`, which is in the directory `/opt/nicola/libs/` and whose headers are in `/opt/nicola/include/`.

## 2.4 Linking a library against another library

You might need to build a shared library (for example called libNicola) which depends on another library (for example on libHelloWorld), and requiring the other library to be loaded automatically whenever your library is. We say that your library (libNicola) depends on the other one (libHelloWorld).

This case is quite simple - you write a usual GNUmakefile for your library:

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

LIBRARY_NAME = libNicola
libNicola_OBJC_FILES = two.m

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

and add a GNUmakefile.preamble in which you tell the make package that this library depends on the library libHelloWorld:

```
libNicola_LIBRARIES_DEPEND_UPON += -lHelloWorld
```

# 3 Building more than one thing in the same GNUmakefile

Up to now, we discussed how to use a GNUmakefile to build a single thing. For example, a tool or a library. In this section we will discuss how to build more than one thing; for example, a number of tools, or a tool and a library. It is very common in practice.

## 3.1 Building many things of the same type

If you want to build more than one tool in the same GNUmakefile, you simply need to list all the tools you want to build in the TOOL_NAME variable. For example:

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

TOOL_NAME = Client Server
Client_OBJC_FILES = ClientAPI.m FrontEnd.m
Server_OBJC_FILES = ServerAPI.m BackEnd.m

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

In this case, two tools, Client and Server, will be built. The tool Client will be built by compiling and linking the files ClientAPI.m and FrontEnd.m, while the Server will be built compiling and linking the files ServerAPI.m and BackEnd.m.

The same applies to applications, or libraries, or any other type of projects. For example, for applications you would list all the applications you want to build in the APP_NAME variable, and similarly for libraries.

Please note that when you specify multiple tools, apps or libraries to be built in the same GNUmakefile, they may be built in any order; starting with gnustep-make 2.4.0, they will even be built and linked in parallel if you are doing a parallel build (please check the forthcoming 'Parallel Building' Tutorial for more info).

## 3.2 Building things of different type

If you want to build a tool and an application at the same time, you include both tool.make and application.make at the same time. For example:

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

TOOL_NAME = CommandLineClient
CommandLineClient_OBJC_FILES = ClientAPI.m CommandLineFrontEnd.m

APP_NAME = GUIClient
```

```
GUIClient_OBJC_FILES = ClientAPI.m GUIClientFrontEnd.m

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

Note that in this case the order in which you include tool.make and application.make is important: the tool and the application are built in the specified order. In the example above tool.make is included before application.make and hence the tool will be built before the application.

# 4 Aggregate projects

Up to now, all examples consisted of projects in a single directory, using a single GNUmakefile. Aggregate projects allow you to have a number of projects in different directories.

As an example, suppose that you are writing a networked game. Your source code might contain two main projects to build: a gui application (the game client) and a command line tool (the server). Naturally enough, you want to develop and distribute the two subprojects together, but they are big enough that you want to keep them in separate directories. This is where GNUstep aggregate projects come handy.

Imagine that your game is called `MyGame`. You will have a top-level directory

```
MyGame
```

and two subdirectories

```
MyGame/Server
MyGame/Client
```

In `MyGame/Server` you keep the source code of your server tool, with its own `GNUmakefile`. In `MyGame/Client` you keep the source code of your client application, with its own `GNUmakefile`.

You can now you add the following `GNUmakefile` in the top-level directory:

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

PACKAGE_NAME = MyGame

SUBPROJECTS = Server Client

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

This `GNUmakefile` simply tells to the make package that your project consists of two aggregate subprojects, `Server`, and `Client`. Please note that the make package follows the order you specify, so in this case `Server` is always compiled before `Client` (this could be important if one of your subprojects is a library, and another subproject is an application which needs to be linked against that library: then, you always want the library to be compiled before the application, so the library should come before the application in the list of subprojects).

In this example, we have two aggregate subprojects, but you can have any number of aggregate subprojects.

At this point you are ready. Typing

```
make
```

in the top-level directory will cause the make package to step into the `Server` subdirectory, and run `make` there, and then step into the `Client` subdirectory, and run `make` there.

The same will work with all the standard make commands, such as `make clean`, `make distclean`, `make install` etc.

Aggregate subprojects can be nested, so that for example the `Server` project could be itself composed of subprojects.

## 4.1 Serial and parallel subdirectories

Starting with version 2.4.0, gnustep-make supports two new ways of organizing multi-directory projects. From 2.4.0 you can use serial-subdirectories.make and parallel-subdirectories.make instead of aggregate.make to better control how you want the build to be done when using parallel building. Please check the forthcoming "Parallel Building" Tutorial for more information.

At the moment, for maximum compatibility with older versions of gnustep-make, using aggregate.make is still the recommended way of controlling subdirectories. We mentioned the existence of serial-subdirectories.make and parallel-subdirectories.make so that you know what they are in case you stumble upon them.