More on GNUstep Makefiles

Nicola Pero n.pero@mi.flashnet.it

November 2000 AD

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
```

(the `/` at the beginning is only for compatibility with older versions of gnustep-make, you can omit with newer ones). 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 Debugging version of a library

You can also make a debug version of the library, by using `make debug=yes`. The debugging version of a library has `_d` automatically appended; for example, when compiling `libHelloWorld` with debugging enabled, you would get a library called `libHelloWorld_d`. You can install both the debugging and the not-debugging versions of a library at the same time.

2.3 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`).

2.4 Debugging

If you compile your tool or application with debugging enabled, the make package will automatically search for debugging libraries. For example, if you compile our `HelloWorldTest` with debugging enabled, it will automatically add `_d` at the end of each library name, thus linking against `HelloWorld_d` and not against `HelloWorld`.

2.5 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/`.

2.6 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 Aggregate projects

A nice feature of the GNUstep make package is the support for aggregate projects.

As an example, suppose that you are writing a networked game. Your source code will probably contain two different subprojects: a gui application (the client application game) and a command line tool (the server). The server keeps the game map, and any information on the current state of the game; it allows to save, load, reset a game; to set game options. To play, the players will start a client each, and use it to connect to the server from machines on the network, and play against each other. The server is a command line tool, while the client application is a nice user-friendly gui application with lots of images and mouse actions. Naturally enough, you want to develop and distribute the two subprojects together. This is where GNUstep subprojects 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 has two 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 subprojects, but you can have any number of subprojects.

At this point you are ready. For example, typing

```
make debug=yes
```

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

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

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