Python Makefile
Version 3.0

Pierre Molinaro

30 mai 2016

Contents

1 Versions

Version	**Date**	**Commentaire**
1.0	March 4, 2015	Initial version
2.0	October 2, 2015	Definition of more than one target for a rule (exemple at page 14)
2.1	October 5, 2015	Added a dynamic checking of arguments type
		Added a simulation of the `clean` target (page 13)
2.2	October 24, 2015	Replaced `subprocess.Popen` by `subprocess.call` in `runCommand`
		Checks external tools exist by using the `find_executable` function (see note on page 4)
		Added an optional boolean argument in `Make` class constructor to display the complete path of external tools (see notes on pages 3 and 9)
2.3	April 16, 2016	Added a progress bar display (see page 7)
3.0	May 30, 2016	Python 3 compatibility: the script can be used with Python 2 and Python 3

Version 2.0 breaks previous versions scripts. The first argument of `makefile.Rule` must be a list. The lines modified to be compatible with version 2.0 and higher are commented with "`# Release 2`".

2 Introduction

The `makefile` module is designed to build easily Python scripts which are a replacement for traditional makefiles. The `makefile` class have many assets:

- Dependencies rules description is simpler;
- Blanks in paths can be used freely;
- A full featured programming langage.

In the following, we will give an example of usage of this class.

3 An example

The example consists of two C files and a C header file.

File `main.c`:

```c
#include "myRoutine.h"

int main (int argc, char * argv []) {
  myRoutine () ;
  return 0 ;
}
```

File `myRoutine.h`:

```c
#ifndef MYROUTINE_DEFINED
#define MYROUTINE_DEFINED
```

```
void myRoutine (void) ;

#endif
```

File `myRoutine.c`:

```
#include "myRoutine.h"
#include <stdio.h>

void myRoutine (void) {
  printf ("Hello␣world!\n") ;
}
```

Separate compilation is done by the commands:

```
mkdir -p objects
gcc -c main.c -o objects/main.c.o
gcc -c myRoutine.c -o objects/myRoutine.c.o
gcc objects/main.c.o objects/myRoutine.c.o -o myRoutine
```

The object files are stored in a directory `objects` that must be created if it does not exist.

4 Construction of the Python script

This script is contained in an executable file `build.py`, located in the same directory as the C source files. Its content is described step by step.

The first step is to import the module `makefile`, and other useful modules.

```
import sys, os, makefile
```

To work with relative paths, it sets the current directory, obtained from the argument 0.

```
scriptDir = os.path.dirname (os.path.abspath (sys.argv[0]))
os.chdir (scriptDir)
```

Then we create an object `make`, which is the object that centralizes implementation of construction. A string argument must be provided, it is the goal to be built. The precise definition of this goal is made later.

```
make = makefile.Make ("all")
```

Note. From version 2.2, the creation of the `make` has an optional second Boolean argument. If it is true, calling each command displays the full path of the utility called. By default, this optional argument is false.

```
make = makefile.Make ("all", True) # Displays the path of each command called
```

To present a general solution, we build two lists, one containing the list of C files to compile, the other the list of object files to be linked. The second is initially empty, it is built by looping through the list of source files.

```
sourceList = ["main.c", "myRoutine.c"]
objectList = []
for source in sourceList:
  ...
```

In the body of the loop, we construct each rule corresponding to the compilation of a source file. We begin by defining the *target* of the rule, which is here the object file built by compiling the source C. We also want to collect in `ObjectList` the list of object files.

```
  object = "objects/" + source + ".o"
  objectList.append (object)
```

Then we instantiate an object `rule` specifying as first argument the list of target files of the rule. The second argument is optional, it is the title that is displayed when running the rule. By default, without the second argument, the text `Building` followed by the name of the target is displayed.

```
  rule = makefile.Rule ([object], "Compiling " + source)  # Release 2
```

Please note that the target (here "`[object]`") must be a list of files. You can not put a name such as `all` or `clean` here: these names are *goals* and their definition is described later in this document.

Then dependencies are added. Here there is only one dependency, the source file. The variable `rule.mDependences` is a list of strings, initially empty.

```
  rule.mDependences.append (source)
```

Caution, do not add here the dependencies to the header files. We will see how to take them into account later.

We now construct the command to be executed to build the target. The "`rule.mCommand`" variable is a list of strings, initially empty.

```
  rule.mCommand.append ("gcc")
  rule.mCommand += ["-c", source]
  rule.mCommand += ["-o", object]
```

It is essential to define a string for each argument. If for example we want to add the "`-O2`" option, writing `rule.mCommand.append("gcc -O2")` is a mistake, it is mandatory to write `rule.mCommand += ["gcc", "-O2"]`.

Note. From version 2.2, the existence of the tool called is verified. An error message is displayed if the tool is not found; For example, if you write `rule.mCommand.append("gccc")`, then the following error message is displayed:

```
*** Cannot find 'gccc' executable ***
```

The construction of the rule is complete; we can now add it to the Makefile.

```
  make.addRule (rule)
```

To summarize. The construction of the rules governing the building from the C sources is:

```
sourceList = ["main.c", "myRoutine.c"]
objectList = []
for source in sourceList:
  object = "objects/" + source + ".o"
  objectList.append (object)
  rule = makefile.Rule ([object], "Compiling " + source) # Release 2
  rule.mDependences.append (source)
  rule.mCommand.append ("gcc")
  rule.mCommand += ["-c", source]
  rule.mCommand += ["-o", object]
  make.addRule (rule)
```

Note that the object files are stored in the `objects`. Maybe it does not exist, and `gcc` will not create it, triggering an error if it does not exist. In fact, there is no need to worry about it. Indeed, when a rule is to be enforced, our makefile implicitly creates the target file directory, if this directory does not exist.

We will now build the rule related to the linking. The name of the executable file is defined by the variable `product`, and dependancies are object files listed in the variable `ObjectList`.

```
product = "myRoutine"
rule = makefile.Rule ([product], "Linking " + product) # Release 2
rule.mDependences += objectList
rule.mCommand += ["gcc"]
rule.mCommand += objectList
rule.mCommand += ["-o", product]
make.addRule (rule)
```

As a check, it is possible to display the rules.

```
make.printRules ()
```

The three rules we have set are displayed:

```
--- Print 3 rules ---
Target: "objects/main.c.o"
  Dependence:  "main.c"
  Command:  "gcc" "-c" "main.c" "-o" "objects/main.c.o"
  Title:  "Compiling main.c"
Target: "objects/myRoutine.c.o"
  Dependence:  "myRoutine.c"
  Command:  "gcc" "-c" "myRoutine.c" "-o" "objects/myRoutine.c.o"
  Title:  "Compiling myRoutine.c"
Target: "myRoutine"
  Dependence:  "objects/main.c.o"
  Dependence:  "objects/myRoutine.c.o"
  Command:  "gcc" "objects/main.c.o" "objects/myRoutine.c.o" "-o" "myRoutine"
  Title:  "Linking myRoutine"
--- End of print rule ---
```

We now define goals ("*goals*"). By default, there is a goal named `clean`, which has no effect (later we will see how to configure it). In our example, let's define two goals:

- `all`, that control the building of the executable (the one that was mentioned in the instantiation of the class `Make`);

- `compile`, which controls the compilation of sources, without linking (not used for now).

```
make.addGoal ("all", [product], "Building all")
make.addGoal ("compile", objectList, "Compile C files")
```

Each goal is defined by three arguments:

- its name, which is later used to refer;
- the list of files it builds;
- its description as a character string.

As a check, we can print a list of goals.

```
make.printGoals ()
```

We get:

```
--- Print 2 goals ---
Goal: "compile"
  Target:  "objects/main.c.o"
  Target:  "objects/myRoutine.c.o"
  Message:  "Compile C files"
Goal: "all"
  Target:  "myRoutine"
  Message:  "Building all"
--- End of print goal ---
```

Finally, we will launch the construction of the goal that has been given when instantiating the class `Make`, that is to say, the aim `all` (the construction of goal `compile` is presented below).

```
make.runGoal (0, False)
```

This call contains two arguments:

- an integer value that specifies how many rules can be executed in parallel;
  - a positive value defines the number of rules that can be executed in parallel; in particular, the value 1 requires a sequential execution rules;
  - the value 0 tells the Makefile to take the number of processors on the machine where the script runs;
  - a strictly negative value indicates the Makefile to take the number processors on the machine where the script runs, to which is added the opposite of the value; for example, -2 on a quad-core defines the execution of $4-(-2)=6$ rules in parallel;
- a Boolean value that indicates whether the command regarding each rule should be displayed or not; Title of each rule is always displayed.

The script displays (note the creation of the directory `objects`)

```
Making "objects" directory
[ 33%] Compiling main.c
[ 66%] Compiling myRoutine.c
[100%] Linking myRoutine
```

New in version 2.3. The percentage that is displayed represents the percentage of completed commands when the command mentioned on the same line will be completed. If N commands must be executed, for command i ($1 \leqslant i \leqslant N$) the displayed value is $100 \times i/N$. Also, the last command is always preceded by " texttt [100 %]".

If we do not want to display this information, simply call the method `doNotShowProgressString` of class `Make` before calling `runGoal`:

```
make.runGoal (0, False)
make.doNotShowProgressString ()
```

If we relaunch immediately the script, the makefile examines the dependencies and reports that there is nothing to build:

```
Nothing to make.
```

Calling `runGoal` with the last argument to `True` displays the commands that are executed:

```
Making "objects" directory
mkdir -p objects
[ 33%] Compiling main.c
gcc -c main.c -o objects/main.c.o
[ 66%] Compiling myRoutine.c
gcc -c myRoutine.c -o objects/myRoutine.c.o
[100%] Linking myRoutine
gcc objects/main.c.o objects/myRoutine.c.o -o myRoutine
```

If an error is detected, such as when executing a rule that fails because of a compilation error, the execution of `runGoal` waits for all parallel executions to finish. To view errors and drop the script by returning an error code, you can call:

```
make.printErrorCountAndExitOnError ()
```

Calling this method has no effect if there is no error.

5 Dependencies on the header files

In previous writings, we had not taken into account the dependencies on the header files for compiling C sources. In the example, these dependencies are very simple, there is only one user header file: `myRoutine.h`. One might add that file in a dependency list defined by `rule.mDependences`.

But in a real case, this requires to maintain yourself the list of dependencies for each source file. Or, depending on the evolution of the program, this list can change: the header files may be unnecessary, others can be forgotten. It is much easier and safer to entrust this task to GCC and to the Python makefile.

Automatically maintaining dependencies on header files is done in three steps.

First choose a name for the dependency file by selecting the name of the object file, to which is added the `dep`. This file will be placed in the `objects` directory.

```
  depObject = object + ".dep"
```

Then tell GCC to build it; We add the following compilation options:

```
  rule.mCommand += ["-MD", "-MP", "-MF", depObject]
```

Finally, we add to the rule the dependency to this file (you must place the instance of the class `Make` as second argument):

```
  rule.enterSecondaryDependanceFile (depObject, make)
```

Summary. The rules for compiling C sources thus become:

```
sourceList = ["main.c", "myRoutine.c"]
objectList = []
for source in sourceList:
  object = "objects/" + source + ".o"
  depObject = object + ".dep"
  objectList.append (object)
  rule = makefile.Rule ([object], "Compiling " + source) # Release 2
  rule.mDependences.append (source)
  rule.mCommand.append ("gcc")
  rule.mCommand += ["-c", source]
  rule.mCommand += ["-o", object]
  rule.mCommand += ["-MD", "-MP", "-MF", depObject]
  rule.enterSecondaryDependanceFile (depObject, make)
  make.addRule (rule)
```

Now, any modification of a header file will cause the compilation of the source files that directly or indirectly include it. This is called a *secondary* dependency because the absence of the file does not prevent the execution of the rule. Conversely, the absence of the source file causes an error, because there is no way to build it.

The execution of the `enterSecondaryDependanceFile` method can take a long time, since the date of modification of all files listed in the `depObject` file is checked. However, if the goal `clean` is invoked, this checking is not done because it is unnecessary: the execution time of the method is then negligible.

6 Priority between rules

By default, the rules that are ready to be executed are in the order they were added by the `addRule`. Thus, in our example, we defined the list of source files:

```
sourceList = ["main.c", "myRoutine.c"]
```

If we execute the rules sequentially (call `runGoal` with the second argument equal to 1), `main.c` will be always compiled before `myRoutine.c`.

This order can be changed by assigning a priority to a rule: just change the value of the integer field `mPriority` of the rule. By default, this field is set to zero. You can assign a positive, negative or zero value, as long as it is an integer. The more this field has higher value, the more the rule is of high priority. At equal priority the order of insertion of the rule (the texttt addRule) defines the order: the first rule inserted will be executed first.

For example, to compile the files in descending order of size, we write (just before the call to `addRule`):

```
  rule.mPriority = os.path.getsize (scriptDir + "/" + source)
```

7 Taking into account several goals

We defined the goal `compile`, but without having called it so far. We'll see how we can write a new Python script that will call this goal.

First, we modify the beginning of the script `build.py` as follows:

```
#--- Get goal as first argument
goal = "all"
if len (sys.argv) > 1 :
  goal = sys.argv [1]
#--- Get max parallel jobs
maxParallelJobs = 0 # 0 means use host processor count
if len (sys.argv) > 2 :
  maxParallelJobs = int (sys.argv [2])
#--- Build python makefile
make = makefile.Make (goal, maxParallelJobs == 1)
```

Note. From version 2.2, `makefile.Make` has an optional second Boolean argument. If it is true, calling each command displays the full path of the utility called. By default, this optional argument is false. By placing `1 == maxParallelJobs` as the second argument, it displays this path if the construction of the goals is performed sequentially, without parallelism.

Then we modify the end of the script `build.py` as follows:

```
make.runGoal (maxParallelJobs, maxParallelJobs == 1)
```

These changes allow to call this script with different arguments, allowing to set the goal and the parallelism.

For example, the script `compile.py` , which performs only a compilation is written as :

```
#! /usr/bin/env python
# -*- coding: UTF-8 -*-

import sys, os, subprocess, atexit

def cleanup():
  if childProcess.poll () == None :
    childProcess.kill ()

#--- Register a function for killing subprocess
atexit.register (cleanup)
#--- Get script absolute path
scriptDir = os.path.dirname (os.path.abspath (sys.argv [0]))
os.chdir (scriptDir)
#---
childProcess = subprocess.Popen (["python", "build.py", "compile"])
#--- Wait for subprocess termination
if childProcess.poll () == None :
  childProcess.wait ()
if childProcess.returncode != 0 :
  sys.exit (childProcess.returncode)
```

Another example is the script `verbose-build.py` which performs a compilation file by file. It clarifies the error display, in fact, parallel compilation may cause interlaced display error messages, making them difficult to understand.

```
#! /usr/bin/env python
# -*- coding: UTF-8 -*-

import sys, os, subprocess, atexit

def cleanup():
  if childProcess.poll () == None :
    childProcess.kill ()

#--- Register a function for killing subprocess
atexit.register (cleanup)
#--- Get script absolute path
scriptDir = os.path.dirname (os.path.abspath (sys.argv [0]))
os.chdir (scriptDir)
#---
childProcess = subprocess.Popen (["python", "build.py", "all", "1"])
#--- Wait for subprocess termination
if childProcess.poll () == None :
  childProcess.wait ()
if childProcess.returncode != 0 :
  sys.exit (childProcess.returncode)
```

8 Post commands

It is possible to add a rule of *post commands* that are executed one after the other after the execution of the associated command. A typical example is the application of the utility `strip` the executable produced by the linker.

We can not write a normal rule, because the dependency file is the same as the target file.

To define a *post command*, we first instantiate a new object of type `PostCommand`:

```
postCommand = makefile.PostCommand ("Stripping " + product)
```

The argument shall contain the title that will be displayed during the execution of the post command.

Is then added a command to be executed:

```
postCommand.mCommand += ["strip", "-A", "-n", "-r", "-u", product]
```

Finally, add the *post command* to the list of post commands for the rule:

```
rule.mPostCommands.append (postCommand)
```

Summary. The rule defines the linking becomes:

```
product = "myRoutine"
rule = makefile.Rule ([product], "Linking " + product) # Release 2
```

```
rule.mDependences += objectList
rule.mCommand += ["gcc"]
rule.mCommand += objectList
rule.mCommand += ["-o", product]
postCommand = makefile.PostCommand ("Stripping " + product)
postCommand.mCommand += ["strip", "-A", "-n", "-r", "-u", product]
rule.mPostCommands.append (postCommand)
make.addRule (rule)
```

The script now displays:

```
Making "objects" directory
[ 33%] Compiling main.c
[ 66%] Compiling myRoutine.c
[100%] Linking myRoutine
       Stripping myRoutine
```

9 Deleting targets on error

If an error occurs when executing a command (eg due to a compiler error), the executable file, if it exists, is not removed: then we have an executable that does not match the state of the program compilation..

It's easy to enforce the deletion of the executable during a command execution error. Just put the attribute `mDeleteTargetOnError` of the rule that controls the linking to `True`:

```
rule.mDeleteTargetOnError = True
```

This technique can be used for any target.

Summary. The rule that defines the linking becomes:

```
product = "myRoutine"
rule = makefile.Rule ([product], "Linking " + product)  # Release 2
rule.mDeleteTargetOnError = True
rule.mDependences += objectList
rule.mCommand += ["gcc"]
rule.mCommand += objectList
rule.mCommand += ["-o", product]
postCommand = makefile.PostCommand ("Stripping " + product)
postCommand.mCommand += ["strip", "-A", "-n", "-r", "-u", product]
rule.mPostCommands.append (postCommand)
make.addRule (rule)
```

10 The goal `clean`

By default, the goal `clean` is set but has no effect. It's easy to add deletion of files and directories when the goal `clean` is executed.

In our example, we want the execution of goal `clean` removes :

- the executable file ;
- the directory containing the generated object files.

To remove the executable when executing the goal `clean`, simply add the rule defining the linking:

```
rule.deleteTargetFileOnClean ()
```

Summary. The rule that defines the linking becomes :

```
product = "myRoutine"
rule = makefile.Rule ([product], "Linking " + product)  # Release 2
rule.deleteTargetFileOnClean ()
rule.mDeleteTargetOnError = True
rule.mDependences += objectList
rule.mCommand += ["gcc"]
rule.mCommand += objectList
rule.mCommand += ["-o", product]
postCommand = makefile.PostCommand ("Stripping " + product)
postCommand.mCommand += ["strip", "-A", "-n", "-r", "-u", product]
rule.mPostCommands.append (postCommand)
make.addRule (rule)
```

To remove the `objects` directory when executing the goal `clean`, simply add the rule defining the compilation of C sources:

```
  rule.deleteTargetDirectoryOnClean ()
```

Summary. The rules for compiling C sources thus become:

```
sourceList = ["main.c", "myRoutine.c"]
objectList = []
for source in sourceList:
  object = "objects/" + source + ".o"
  depObject = object + ".dep"
  objectList.append (object)
  rule = makefile.Rule ([object], "Compiling " + source) # Release 2
  rule.deleteTargetDirectoryOnClean ()
  rule.mDependences.append (source)
  rule.mCommand.append ("gcc")
  rule.mCommand += ["-c", source]
  rule.mCommand += ["-o", object]
  rule.mCommand += ["-MD", "-MP", "-MF", depObject]
  rule.enterSecondaryDependanceFile (depObject, make)
  rule.mPriority = os.path.getsize (scriptDir + "/" + source)
  make.addRule (rule)
```

Now just add the script `clean.py`, calling the goal `clean`:

```
#! /usr/bin/env python
# -*- coding: UTF-8 -*-

import sys, os, subprocess, atexit

def cleanup():
  if childProcess.poll () == None :
    childProcess.kill ()

#--- Register a function for killing subprocess
atexit.register (cleanup)
#--- Get script absolute path
scriptDir = os.path.dirname (os.path.abspath (sys.argv [0]))
os.chdir (scriptDir)
#---
childProcess = subprocess.Popen (["python", "build.py", "clean"])
#--- Wait for subprocess termination
if childProcess.poll () == None :
  childProcess.wait ()
if childProcess.returncode != 0 :
  sys.exit (childProcess.returncode)
```

10.1 Simulation of goal `clean`

From version 2.1, it is possible to display the commands launched by the goal `clean` without running them. To do this, simply call the method `simulateClean`.

```
  make.simulateClean ()
```

This method must be called before calling `make.runGoal` to be taken into account.

11 Automatic opening of a source file on error

It is possible to program the automatic opening with an external editor of a source file when an error occurs. Just add the rules concerned the line:

```
  rule.mOpenSourceOnError = True
```

We will begin with causing a compilation error by inserting a character « §» at the 3rd line of file `myRoutine.h`:

```
#ifndef MYROUTINE_DEFINED
#define MYROUTINE_DEFINED
§
void myRoutine (void) ;

#endif
```

By running again the compilation, the error is detected (on a multi-heart, the two compilations occur in parallel):

```
[ 33%] Compiling myRoutine.c
[ 66%] Compiling main.c
In file included from main.c:1:
In file included from myRoutine.c:1:
./myRoutine.h:3:1:  error:  non-ASCII characters are not allowed outside of literals and iden-
tifiers
§
^
1 error generated.
```

```
./myRoutine.h:3:1:  error:  non-ASCII characters are not allowed outside of literals and iden-
tifiers
§
^
1 error generated.
Return code: 1
Wait for job termination...
1 error.
```

When the property `mOpenSourceOnError` is true and if an error occurs, the makefile research a file name at the beginning of line in the output stream of commands that caused an error . Specifically, it is based on the string that precedes the first occurrence of the character ":". This research provides four answers:

- `In file included from main.c`
- `Compiling main.c`
- `./myRoutine.h`
- `./myRoutine.h`

The first two answers do not correspond to an existing file, they are ignored. The following two causes the opening of the `./myRoutine.h` file with an external editor, which is by default:

- on Mac : `TextEdit` ;
- on Linux : `gEdit`.

It's easy to change the default text editor, simply assign the following object `make` properties :

- for Mac : `mMacTextEditor` ;
- for Linux : `mLinuxTextEditor`.

For example, to use the editor `TextWrangler` on Mac:

```
make.mMacTextEditor = "TextWrangler"
```

12 Defining multiple targets for a rule

With version 2, several target files can be build by the execution of a rule. For example, we will modify the rule linker to build a text file "map" that contains the executable memory card. Three changes are made in the listing above (denoted `note 1`, `note 2` and `note3`).

```
product = "myRoutine"
mapFile = product + ".map" # note1
rule = makefile.Rule ([product, mapFile], "Linking " + product) # Release 2, note2
rule.mDeleteTargetOnError = True
rule.deleteTargetFileOnClean ()
rule.mDependences += objectList
rule.mCommand += ["gcc"]
rule.mCommand += objectList
rule.mCommand += ["-o", product]
rule.mCommand += ["-Wl,-map," + mapFile] # note3
postCommand = makefile.PostCommand ("Stripping " + product)
postCommand.mCommand += ["strip", "-A", "-n", "-r", "-u", product]
rule.mPostCommands.append (postCommand)
make.addRule (rule)
```

note1. The new target file `mapFile` is set here.
note2. It is added to the target list constructed by the rule.
note3. The command has been completed to build the new target file.

By running the goal `all`, both files `myRoutine` and `myRoutine.map` are built. If we remove one of the two files, running the goal `all` rebuild it. If we comment out the construction of the file `my-Routine.map` (line commented by `note 3`), running the goal `all` displays an alert indicating that the file has not been built.

13 The complete script `build.py`

```
#! /usr/bin/env python
# -*- coding: UTF-8 -*-

import sys, os
import makefile

#--- Change dir to script absolute path
scriptDir = os.path.dirname (os.path.abspath (sys.argv [0]))
os.chdir (scriptDir)
#--- Get goal as first argument
goal = "all"
if len (sys.argv) > 1 :
  goal = sys.argv [1]
#--- Get max parallel jobs as second argument
maxParallelJobs = 0 # 0 means use host processor count
if len (sys.argv) > 2 :
  maxParallelJobs = int (sys.argv [2])
#--- Build python makefile
make = makefile.Make (goal, maxParallelJobs == 1) # Display executable if sequential build
make.mMacTextEditor = "TextWrangler"
#--- Add C files compile rule
sourceList = ["main.c", "myRoutine.c"]
objectList = []
for source in sourceList:
#--- Add compile rules
  object = "objects/" + source + ".o"
  depObject = object + ".dep"
  objectList.append (object)
  rule = makefile.Rule ([object], "Compiling " + source) # Release 2
  rule.deleteTargetDirectoryOnClean ()
  rule.mDependences.append (source)
  rule.mCommand.append ("gcc")
  rule.mCommand += ["-c", source]
  rule.mCommand += ["-o", object]
```

```
    rule.mCommand += ["-MD", "-MP", "-MF", depObject]
    rule.enterSecondaryDependanceFile (depObject, make)
    rule.mPriority = os.path.getsize (scriptDir + "/" + source)
    rule.mOpenSourceOnError = True
    make.addRule (rule)
#--- Add linker rule
product = "myRoutine"
mapFile = product + ".map"
rule = makefile.Rule ([product, mapFile], "Linking " + product) # Release 2
rule.mDeleteTargetOnError = True
rule.deleteTargetFileOnClean ()
rule.mDependences += objectList
rule.mCommand += ["gcc"]
rule.mCommand += objectList
rule.mCommand += ["-o", product]
rule.mCommand += ["-Wl,-map," + mapFile]
postCommand = makefile.PostCommand ("Stripping " + product)
postCommand.mCommand += ["strip", "-A", "-n", "-r", "-u", product]
rule.mPostCommands.append (postCommand)
make.addRule (rule)
#--- Print rules
# make.printRules ()
# make.writeRuleDependancesInDotFile ("make-deps.dot")
make.checkRules ()
#--- Add goals
make.addGoal ("all", [product, mapFile], "Building all")
make.addGoal ("compile", objectList, "Compile C files")
#make.simulateClean ()
#make.printGoals ()
#make.doNotShowProgressString ()
make.runGoal (maxParallelJobs, maxParallelJobs == 1)
#--- Build Ok ?
make.printErrorCountAndExitOnError ()
```

14 Other functions

14.1 Color printing

The module `M makefile` defines the following functions to make color prints:

- `BLACK()`, `RED()`, `GREEN()`, `YELLOW()`, `BLUE()`, `MAGENTA()`, `CYAN()`, `WHITE()`;
- `BOLD()`, `BLINK()`, `UNDERLINE()`;
- `ENDC()`, which resets the print attributes.

All of these functions returns a string that can be directly used in a `print`:

```
print makefile.RED () + makefile.BOLD () + "Erreur !" + makefile.ENDC ()
```

14.2 The method `enterError`

The method `enterError` of class `Make` will display an error message and increment the error counter maintained by the receiver. It has a string argument characters, which is printed in bold red on the terminal.

```
make.enterError ("There is an error somewhere !")
```

Recall that a non-zero error counter inhibits the execution of rules when `runGoal` is invoked.

14.3 The method `errorCount`

The method `errorCount` of class `Make` returns the current value of the error counter maintained by the receiver.

```
nombre_erreurs = make.errorCount ()
```

14.4 The method `printErrorCount`

The method `printErrorCount` of class `Make` prints the current value of the error counter maintained by the receiver if it is not zero. If it is zero, nothing is printed.

```
make.printErrorCount ()
```

14.5 Search for a file in a directory tree

The method `searchFileInDirectories` of class `Make` searches a file in a directory list:

```
path = make.searchFileInDirectories (file, list_of_directories)
```

Where :

- `file` is a character string that defines the name of the desired file ;
- `list_of_directories` is a list of strings that defines the list of directories in which the file will be searched ;

the method returns :

- if `file` is found exactly once, the file name prefixed by the directory name that contains the file ;
- the empty string if the file is not found or if it is found several times; an error message is displayed, and the error count maintained by the receiver is incremented.

Recall that a non-zero error counter inhibits the execution of rules when `runGoal` is invoked.