# Generating build numbers

In addition version numbers it can be useful to individually number each build. To do this we need a mechanism for storing the last build number, incrementing it, restoring the incremented number and incorporating the number within our build.

First, we want a file we can check out of source control, change and check back in with the revised version number. This may also be a useful place to keep various other pieces of information we don't want to hard code, e.g. the version number or company name. And we would like the file to be plain text so we can change it easily.

Ideally, we want our file to look something like:

```
BuildNumber = 126
VersionNum = "0.1 alpha"
Copyright = "(c) Jiffy Software 2002"
```

As luck would have it this is actually valid Python so we can save this as `Version.py`. If we had written our file as C++ we would need a complex parsing algorithm to read the file, increment the build number and re-write the file. Since it is Python we can write another Python script which includes the file and treats these variables as, well, variables.

So, we write a script called `GenVersion.py` that looks a bit like this:

```
import Version
version.BuildNumber = version.BuildNumber +1
...
# rewite the version file
ver_output = open("version.py", "w")
ver_output.write("BuildNumber = " + str(version.BuildNumber) + "\n")
...
# write a C++ file
cpp_output = open("Version.cpp", "w")
cpp_output.write("std::string Version::BuildNumber() {\n");
cpp_output.write("\treturn \"" + str(version.BuildNumber) + "\";\n")
cpp_output.write("}\n\n")
```

Next you will want to add a rule to your makefile to execute the script:

```
Version.cpp: version.py
  bash -c "python GenVersion.py"
```

This example creates `Version.cpp` so you will need a `Version.hpp` but as this rarely changes it doesn't need regenerating. You may like to generate other files with this information, say a Windows resource script, or write the build number to the build log.

In this example I've omitted niceties such as dealing with the other variables, closing files and such. You can see the complete scripts at my web site, http://www.allankelly.net/writing. The most important nicety that is missing is an option not to bump up the build number.

Hang on you say, "Aren't we talking about incrementing the build number? Why would we want not to do it?" Actually it is important to ensure the build number is only bumped when we want it bumped. If every time a developer tried to build source code the build number would run wild and be useless.

The trick is to only bump the build number when we are doing an *official* build. Usually this means a batch build. Developers building on their local machine don't count. So, we make the default case the current build number from `Version.py`, and add a different rule to bump the build number, so we get:

```
all: Version.cpp

bumpBuild: all
  bash -c "python GenVersion.py Bump"

Version.cpp: version.py
  bash -c "python GenVersion.py NoBump"
```

By default Make will execute the `all` rule, which will only rebuild `Version.cpp` if `Version.py` is newer, in which case the build number will be unchanged. If however, we execute the `bumpBuild` rule (which we will do for an official build) then we execute the second rule, which will always generate `Version.cpp` and in the process increment the build number.

Deciding when to increment the build number can become a more vexed question still. Suppose you bump it for every over night build on Windows, suppose you now introduce a nightly Linux build. Should they use the same number? Should they even use the same number sequence? Maybe the new build should start from one.

Generated files like should not be checked into source code control. We can recreate them at any time and they only clutter up source control with frequent, minor changes. We also need to take care not to make then read-only at any time as this may cause a build error. Finally, if your build system has a "make clean" option we need to remember to delete any intermediate files we may have generated.

We can use these same principals of code generation to create other elements of our system. Examples include:

? Internationalised language resources can be built from plain text files, thus allowing translators to work with something friendlier than C++ or a proprietary file format.

? Error and other information messages can be generated from text files. Tab delimited files specifying error number, sub-system and message text can be turned into C-arrays or even C++ exception classes.

? SQL can be customised to different target databases.

? Install scripts which only install the components that are built.

Like template programming we have an option to vary what is compiled into a program.