Version management using Git

Version labelling

There are many approaches that have been used to identify versions, for examples read the Wikipedia page Software versioning.

Whilst each has its merits, most are more complex that I required, so I have created a simple version labelling approach for my own developments, for which I have written scripts that support auto generation of the label using git information.

The label has the form

Major.Minor.CommitCount[qualifier][-branch]

Component	Notes
Major.Minor	This comes from a Git tag. See below for additional information
CommitCount	This is the number of git commits for the current application since the Git tag. **Warning:** Comparing versions across branches is not reliable because the commit count reflects commits relevant to the current branch. Merging will account for the additional commits from the branch. **version.cmd** has a commented out code option to make CommitCount take into account commits on all branches. Although fine for local use, when using a remote repository, unless all branches are shared, this will give different results
qualifier	Is only present in two cases + - if the version contains pre-committed files x - this is used when git is not available i.e. untracked files Note this changed from .P and .X on 22-Aug-2021
branch	This is present if the version is not on the master/main branch. Note headless versions have the branch name HEAD

Example	Meaning
1.2.0	Built using a tagged Git version 1.2 on the master/main branch
1.2.4	Built using source which is 4 commits after 1.2.0 on the master/main branch
1.2.4+	Build using same base sources as 1.2.4 but has some modified / additional files, not yet committed to Git - on the master/main branch
1.2.5-dev	Built using source which is 5 commits after 1.2.0 on the dev branch
1.2.5x-dev	Same as 1.2.5-dev above but has some modified / additional files, not yet uncommitted to Git - on the dev branch
1.2.0x	Built with files outside Git management, originally based on 1.2.0 sources

Note within windows resource files the file version is only displayed for numeric values. In this case the qualifier and branch are collapsed into a single digit.

0 -> normal build

1 -> build with uncommitted files

untracked and branch builds have the file version set to 0.0.0.0.

Generating version from Git

In many of my projects I use a Visual Studio solution file that contains multiple project files. A challenge on many of the approaches I considered and partially implemented, was how to track different versions of each of the projects. Some of the options I considered are:

- Using submodules in git. Possible but makes management of the Visual Studio sub projects challenging and adds many git repositories and complexity to support standalone and Visual Studio solution builds.
- Using subtrees. Considered but has additional overhead to manage
- Using subgit. See subgit repository for details. I did implement this in a windows batch file and although it works, it has similar problems to the submodules approach.

The approach I chose was to keep the files in a solution together but to add a prefix to tags relating to a specific project. This allows me to track separate version numbers for each project.

Git tag

The tags used in Git have the form

[appid-]Major.Minor

Component	Description
appid	This is the project prefix. It can be omitted if the feature isn't required/applicable
Major	This is a number reflecting major changes in the project. e.g. a major change to the way an application works internally
Minor	This is a number representing smaller but significant changes in the project. e.g. a new functional feature c.f. cosmetic changes and fixes

To help manage the creation of tags, I have written a utility **release.cmd** (a windows batch file) that will generate the next minor or major tag. In addition it will create a version.in file that contains default version information to allow builds outside Git management. If there is an existing version.in file it is used to automatically pick up the appid.

Note there is an implied Git Tag of [app-id-]0.0 for the start of the project development.

version.cmd

The batch file **verison.cmd** managed the generation of the version information from Git. It can be invoked in two modes

1. version [options]
2. version [options] cacheDir versionFile

options are

-h shows a simple summary of how to use the script

-q don't show the version information to the console

-f override the use of the cached information

-a appid sets the appid to use. An appid of . is replaced by the current directory name. The value overrides any existing name in version.in

Mode 1 will calculate the version and show the information on the console. This can also be used to get version information for any directory tree, which is potentially useful for none build situations; the **-a appid** option may also be useful in this case.

Mode 2 will generate a version file for use during the application build. The cacheDir has a file which records the previous version generated. When the script is invoked, if version hasn't changed the version file is not written. This helps optimise the build process if the version hasn't changed.

Note if the versionFile ends in .cs, then a C# VersionAssembly file is created rather than a C/C++ include file.

How it works

A number of ideas for the tool came from GIT-VS-VERSION-GEN.bat, however there are significant differences in the implementation.

The following are the key steps in deriving the version information

1. If the file version.in exists in the current directory, parse it to load in default values, but in particular get the appid (**GIT_APPID**) if there is one.
   Note the command line **-a appid** option will override this.
2. Get the current branch (**GIT_BRANCH**) and test if there are any uncommitted files for the current directory tree and set **GIT_QUALIFIER** to **.P** if there are. **GIT_BUILDTYPE** is set to **2** if uncommitted files else if **GIT_BRANCH** is not 'master' or 'main' set to 1 else set to 0
3. If there is no branch, then git cannot be used.
   - if file version.in does not then report an error and exit
   - In mode 2, copy file version.in to versionFile and delete the cache file
   - in mode 1 or mode 2 without the --quiet option copy, populate version information using the defaults from step 1 and continue at step 13
4. if **GIT_BRANCH** not equal to 'master' or 'main' add **-GIT_BRANCH** to the **GIT_QUALIFIER**
5. The commit hash (**GIT_SHA1**) and commit time (**GIT_CTIME**) for the most recent commit in relevant to the current directory are obtained from Git
6. Git is used to obtain the highest versioned tag (**strTAG**) matching **GIT_APPID-**[0-9]*.*[0-9] which is on the path to the commit **GIT_SHA1**. Note the **GIT_APPID-** is omitted if there is no **GIT_APPID**. See note at end on the implications of this.
7. The count of commits (**GIT_COMMITS**) from this tag to the commit identified in step 4 are obtained from Git. If no tag was found then the count is from the project's first commit
   Note this count is limited to those impacting the current directory tree. This prevents commits on other projects incrementing the commit number. The count does however take into account changes on other branches
8. Any **GIT_APPID-** prefix is removed from **strTAG** and if **strTAG** was not found it is set to **0.0**
   This is the Major.Minor part of the version

9. From the above the **GIT_VERSION** is calculated as **strTAG.GIT_COMMITS** with **GIT_QUALIFIER** added if not blank
10. For mode 2, unless --force is used, then the contents of the cacheDir\GIT_VERSION_INFO file are compared to **GIT_VERSION-GIT_SHA1**. If they are the same a message is shown assuming no change and the script terminates.
11. In mode 2 the cacheDir\GIT_VERSION_INFO file is updated with the new value **GIT_VERSION-GIT_SHA1**
12. In mode 2 the following information is written to the versionFile using #define statements

Label	Type	Value
GIT_APPID	String	GIT_APPID from step 1. Note omitted if GIT_APPID is blank. Also not used in C# generated file
GIT_APPNAME	String	The application name - note preferred replacement for GIT_APPDIR. Taken from version.in or defaults to parent directory
GIT_PORT	String	Optional. Used to record version and copyright for ported code e.g. "2.2 (C) Whitesmiths"
GIT_VERSION	String	GIT_VERSION from step 9 or from version.in if no Git support
GIT_VERSION_RC	CSV or String	strTAG,GIT_COMMITS,GIT_BUILDTYPE with dot in strTAG replaced with a comma. Value from version.in used if no Git support. For C# commas are replaced by dot and the value is a String rather than CSV. For untracked and branch builds this is set to 0,0,0,0 to avoid confusion.
GIT_SHA1	String	GIT_SHA1 value from step 5, or set to "untracked" if no Git support
GIT_BUILDTYPE	Number	GIT_BUILDTYPE from step 2 or set to 3 if no Git support
GIT_APPDIR	String	**Depreciated**. The current application directory taken from version.in or defaulting to the immediate parent directory.
GIT_CTIME	String	GIT_CTIME in format YYYY-MM-DD HH:MM:SS. All times are in GMT so are reproducible globally. This is found in step 5 or from version.in if no Git support
GIT_YEAR	String	First 4 characters of GIT_CTIME. Useful for copyright year

13. In mode 1 or mode 2 unless --quiet option is specified, the following information is shown on the console

Label	Value
Git App Id	GIT_APPID - may be blank
Git Version	GIT_VERSION from step 9 or from version.in if no Git support
Build type	GIT_BUILDTYPE from step 2 or set to 3 if no Git support
SHA1	GIT_SHA1 value from step 5, or set to "untracked" if no Git support
App Name	The application name
Committed	GIT_CTIME in format YYYY-MM-DD HH:MM:SS. All times are in GMT so are reproducible globally. This is found in step 5 or from version.in if no Git support

Note on step 6

The method calculating the latest applicable tag may not work for everyone. Taking two cases

- branch merge into master/main. Here the commit will get the latest versioned tag from either the branch or master/main
    - If master/main is ahead of the development branch this would be correct behaviour. Explicit tag can force a new version if necessary
    - if master/main is behind the branch, then in most cases this will be valid, the exception is where a fix is done on a future development and there is a need to back port to the current master/main. For me this is not an issue as the scenario is unlikely to happen. It can also be managed by not creating a tag on a development branch, relying on the branch name to indicate future developments. This way the development will pick up the master/main versioned tag from the initial fork or most recent merge. In this case the master/main tagged version will never be less than the development one.
- branch merge from master/main. Here the branch will get the latest version from either the branch or master/main. This is typically a last step to test the development branch before merging back to the master/main. Even if the branch version is updated it will be safe for remerging, so it is unlikely to be a problem.

fileVer.cmd, revisions.cmd, revisions.pl

filever.cmd is a stripped down version of version.cmd, that identifies the file revision of a specific file. It does not rely on any appid tag but calculates the revisions since the file was first committed. It does not account for moves or renames, other than simple letter case change.

revisions.cmd will run fileVer on all files in the current directory and revisions.pl is a perl script that combines the functionality of fileVer.cmd and revisions.cmd

The primary use of these utilities is for script files that don't have a build stage where the version information could be readily imported

Note the utilities attempt to handle cases where a file has been moved. The following scenarios are handled

1. Filename is unique in the repository, all changes to filename are counted. Handles file moves
2. Directory/Filename is unique, counts all changes to the directory/filename combination. Handles directory moves.
3. All other cases only the count of the file in its current location are handled.

For all of the above the filename compare is done in a case insensitive way.

There are cases where the above approach will be wrong, particularly if the filename has been reused after one with the same name has been previously deleted. In most cases the approach is expected to provide the correct result.

usage: fileVer -v | [-q] file
usage: revisions -v | [-s] [-q]
usage: revisions.pl -v | [-q] [file | dir]*

```
where
-v  shows utility version
-q  suppresses warning about file not in a Git repository
-s  process files in immediate subdirectoies as well
for revisions.pl, if no file or directory is specified it processes the current
directory
```

Output shown on the console

```
filename    Rev: nn[+][{branch}] -- git hashval '['date']'
Where
filename    file as specified on the command line
nn          The number of commits the file has been involved in
+           added if the file is not yet committed
{branch}    shown if not on master/main branch
hashval     git SHA1
date        date the file wsa last committed in YYYY-MM-DD format and is in GMT
```

installScript.pl

This script is primarily to deploy script files to target directories, e.g. for inclusion in github repsoitories. When doing so it replaces the string _REVISION_ in the script with text showing the actual revision of the script.

usage: installScript.pl -v | [target file+]

```
where
-v      shows the utility version
target  is the directory where the files will be deployed
file+   is a list of files to deploy
Note the target directory and files must already exist

If no command line arguments are present, then the utility will use a
configuration file installScript.cfg. This file contains multiple lines of the
form
target file [ file]+
where files are separated by whitespace and can continue on to multiple lines,
with target being omitted and replaced by at least one whitespace character
To allow for embedded space characters, target and file can optionally be
enclosed in quotes.
Blank lines and lines beginning # are ignored
```

```
Updated by Mark Ogden 25-Aug-2021
```