New version management tools

Version labelling using git

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 than I require, 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 new version documented below, is simpler and quicker than my previous approach.

Auto generated version string

For directories, usually associated with applications, the auto generated version string is of the form

```
[branch-]year.month.day.(release number | sha1)[+][?]
```

For individual files, to avoid the need for many tags, the format is similar

```
[branch-]year.month.day.commits[+]      '[' sha1 ']'
```

Where

Component	Notes
branch	This is the branch the commit is on, excluded if the branch is main or master. For detached branches the name (detached) is used.
year.month.day	The GMT date of the associated git commit, leading zeros are omitted
release number	This is used when the commit has an associated tag. It is the end numeric value of the commit tag matching {dir}-rnnn, where {dir} is the name of the directory containing the source files,
sha1	If commit does not have an associated release tag then the sha1 value of the commit is displayed
commits	This is the number of times the individual file has been changed during git commits
'+'	A plus sign is appended if the build includes uncommitted files, other than version file itself.
'?'	A question mark is added if the files are untracked, i.e. using files outside of git

Generating version from Git

There are several utilities that support the new version format

getVersion - getVersion.cmd (windows batch) & getVersion.pl (perl)

This utility is the primary utility used to generate a version string for an application. It should be run from within the directory for which the version string is required.

```
usage: getVersion -v | -h | [-q] [-s] [-w|-W]
-v    shows the version of the script
-h    shows simple help
-q    supresses displaying the version on the console if file written
-s    simplified console output version, also no warning messages
-w    writes the header file if the version changed
-W    writes the header file even if the version was unchanged

The default generated file is _version.h as a C/C++ header file
This can be overriden using a configuration file version.in, see below.

Because the -W option always writes the header file, it can be used for force a
rebuild of files that dependend on it. For C and C++ by using __DATE__ and
__TIME__ macros the build date/time can be captured.
The -s option is to support build tool capture of the version string from stdout
```

How it works

The tool uses 2 git commands

```
git status  used to get the branch and whether there are uncommited files
            other than the version header file
git log     used to get the commit date, sha1 and any associated tag
```

1. Using the above information the commit time is converted into the year.month.day component.
2. The branch- prefix is added if branch is not main or master.
3. if the commit has an associated tag of the form {dir}-rnnn, where {dir} is the directory containing the source, then the nnn is added, else the sha1 is added.
4. Finally if there are uncommitted files the plus sign is added.

If git isn't present then the version number uses the information stored in the last generated version file, appending a question mark to the end, if it doesn't already have one. If there is no version file it uses xxxx.xx.xx.xx?.

This approach allows some support for builds where a snapshot is taken from GitHub, rather than using a repository clone. To achieve this, generated version file is committed in the repository. The release tool **mkrelease** updates this file before the commit with a new release number, but unfortunately a normal commit, uses the existing release number or shar1, most likely with a plus sign suffix. It is therefore not recommended to create builds using snapshots of informally released commits. Also be aware that all snapshot builds that use **getVersion** will have a '?' appended to the version string, to indicate that the version is not being tracked.

Using version.in

If **version.in** is not present the **gitVersion** -w/-W options write a simple #define to the file _version.h of the form

```
#define GIT_VERSION "version string"
```

However to support other languages **getVersion** if **version.in** exists it will be used to override the generated file name or content, or both. The **version.in** file is processed as follows

If the first none blank line is of the format [filename], then filename will be used instead of _version.h.

If there is any other content this is treated as a template and is copied into the target file, replacing @V@ with the GIT_VERSION string and @D@ with the current UTC date and time in yyyy-mm-dd hh:mm:ss format.

The @D@ option is of use when the date and time is needed and no alternative solution exists.

```
Some examples
To include some static information, still writing to _version.h
#define APP_NAME    "superprog"
#define APP_CONTRIBUTOR "A N Other"
#define GIT_VERSION "@V@"

To create an assembler version string, writing to ver.inc
[ver.inc]
myver:  db '@V@', 0     ; git_version (see note below)
build:  db '@D@', 0     ; the build date
```

Note a line containing the text GIT_VERSION (case insensitive) is assumed to contain the previous file version, which is used to generate a default version string if Git is not in use. To be detected the version string should be enclosed in single or double quotes and the first quoted string on the line.

mkRelease - mkrelease.cmd (windows batch)

This utility generates the next release number, creates a version string from it, commits it to git and associates a git tag . The tag is of the format

`{dir}-r{nnn}`

where {dir} is the current directory name and {nnn} is the release number.

```
usage: mkRelease -v | -h | [-f] [-r nnn] [message]
-v      shows the version of the script
-h      shows simple help
-f      assumes 'Y' response to any confirmations
-r nnn  uses the release number nnn, rather than the next sequential number
The optional message is included in the commit message, otherwise an editor is
opened to allow entry of the commit message
The tool supports version.in in the same way as gitVersion

Note the -r option allows nnn to be any text, however no checks are made on
clashes with other tags. It was introduced to allow an indication of historic
releases done under other version management schemes. For numeric values, leading
0s should be omitted.
```

Unlike getVersion, mkRelease will only work if git is installed and the directory being processed is in a repository and not headless.

Unless -f is specified, it prompts to proceed, when not on the main/master branch or if there are uncommitted files, other than the version file.

The tool uses the invoke time in GMT to ensure consistency of the date and version, including avoiding risks around midnight. It finds the next release number to use by adding one to the previous numeric release number.

It then tries the commit, if message is specified, the commit message is
`year.month.day.release: message`
otherwise it invokes the editor, pre-populated with the text
`year.month.day release:`

If the commit is succeeds, the annotated tag is created, with the annotation message i
`Release year.month.day.release`

If the commit fails, then the version file is rolled back to its previous content.

Additional support scripts

revisions.pl

revisions.pl identifies the file version of a specific file and attempts to determine how many revisions there have been to the file

```
usage: usage: revisions.pl -v | -h | [-a|-i|-u|-n]* [--] [file | dir]*
where -v shows version info
      -h show usage and exit
      -a assume directories contain apps
      -n no expansion of directory
      -i include ignored files
      -u include untracked files
      -- forces end of option processing, to allow file with - prefix
if no file or dir specified then the default is .
for directories unless -n is specified, the versions of the directory content is
shown
file or dir can contain wildcard characters but in this case directories starting
with '.' are excluded
```

When the -a option is specified, directory versions are shown as per getVersion, otherwise the version string for individual files is used.

```
Examples
>revisons.pl
*Directory* .                   2023.4.23.73+  [ec77992]
...
- README.md                     2020.10.12.13  [f7a16a5]
- VERSION.html                  2021.8.25.4    [b639f33]
- VERSION.md                    2021.8.25.9    [b639f33]
...

>revisions.pl -a
*Directory* .                   2023.4.23.ec77992+
...
- README.md                     2020.10.12.13  [f7a16a5]
- VERSION.md                    2021.8.25.9    [b639f33]
- VERSION.pdf                   2021.8.25.4    [b639f33]
...

>revisons.pl VERSION.*
VERSION.html                    2021.8.25.4    [b639f33]
VERSION.md                      2021.8.25.9    [b639f33]
VERSION.pdf                     2021.8.25.4    [b639f33]
version.cmd                     2023.4.19.18   [6256cd3]
version.cs                      2021.8.26.2    [f3fbc10]
version.pl                      2023.4.19.12   [a8c8343]
```

fileVer.cmd & rev.cmd

These are simplified versions of **revisions.pl,** with less options but they do not require perl to be installed. In general using **revisions.pl** is the preferred option.

```
usage: fileVer [-v] | file
where
-v      shows version info for the script
file    this lists the file string version for the named file.

usage: rev [-v] | [-s]
where -v shows version info
      -s also shows files in immediate sub-directories
This lists the versions of files in the current directory, optionally showing
those in immediate subdirectories
```

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 version of the script. It uses the individual file version information.

```
usage: installScript.pl -v | [target file+] | [-s 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
-s file use the file to provide a list of targets & files

Note the target directory and source files must already exist

If no command line arguments are present then -s installScript.cfg is assumed.
Note for the -s option, the file contains lines of the format

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
Note files with relative and full paths are supported, however files copied to
the target directory will only use the filename part.
```

install.cmd

This command is typically run post build to copy a built file to one or more locations.

```
usage: install file_with_path installRoot [configFile]
configFile defaults to installRoot\install.cfg
```

install.cfg contains lines of the form

```
srcDir,dir[,suffix]
Where
srcDir  is the name compared with the immediate parent directory name of
file_with_path.
        If this matches the line is processed
dir     is the name of directory to install to, with a leading + replaced by
installRoot.
suffix  is inserted into the installed filename just before the .exe extension
with
        $d replaced by the local date in yyyymmdd format and
        $t replaced by the local time in hhmmss format
```

Example with **install.cfg** in the current directory containing the lines

```
x86-Release,+prebuilt
x86-Release,d:\bin,_32
```

Running

```
install somepath\x86-Release\myfile.exe .
```

copies somepath\x86-Release\myfile.exe to .\prebuilt\myfile.exe and d:\bin\myfile_32.exe

install.cmd supports control lines in the install.cfg file. These determine which files the descriptor lines below it apply to. Subsequent control lines set new scope.

```
The lines are of the form
[+|-]comma separated list of files or *

The '+' enables only the named files to be processed. +* reenables processing for
all files
The '-' will exclude only the named files from processing. -* disables all
processing (not particularly useful)
```

```
Updated by Mark Ogden 11-Feb-2024
```