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 tools that support auto generation of the version label using git information.

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

Note with the port of the versioning tools to C, there have been some changes to the version string

- The branch is now no longer included.
- The release number is now more general revision string and supports qualifiers such as beta1, rc2 etc.
- Individual files now follow the same versioning model and no longer use counts of commits.
- To avoid confusion between sha1 and the new revision string all sha1 are prefixed with the letter 'g'.

Auto generated version string

The general format of the automatically generated version string is

```
year.month.day.(release| 'g' sha1)[+][?]
```

Component	Notes
year.month.day	The GMT date of the associated git commit in numeric form, with leading zeros omitted
release	This is used when the application commit has an associated tag It has format `revision ['-' qualfier qualifier_revision]` a release has a corresponding git tag, {dir}-r{release} where {dir} is the name of the containing directory and {release} is the release tag without any '-' See makeRelease for more details.
sha1	This is the sha1 of the associated commit and is used if there is no associated tag
'+'	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

Note the previous windows batch, perl and powershell utilities have not been updated and should no longer be used.

getVersion

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 [options] [directory | file]*
Where options are:
  -w      write new version file if version has updated
  -f      force write new version file
  -c file set alternative configuration file instead of version.in
  -d      show debugging information
If no directory or file is specified '.' is assumed

Also supports single arguments -v, -V for version info and -h for help

Because the -f 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.
See also using version.in below
```

How it works

The tool uses 3 git commands

```
git log           used to get the commit date, sha1 and any associated tag
git diff-index    used to see if files have changed. The version file itself is
ignored
git check-ignore  used to flag ignored files which may previously been in the
repository
```

1. Using the above information the commit time is converted into the year.month.day component.
2. For a directory if the commit has an associated tag of the form {dir}-r{release, where {dir} is the directory containing the source, then the {release} is used, else the sha1 is added.
3. Finally if there are uncommitted files the plus sign is added.

If git isn't present, then for directories the version number uses the information stored in the last generated version file with a question mark appended if not already present or `xxxx.xx.xx.x?` if there is no previous version. For files the version string is set to `Untracked`.

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 **makeRelease** 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 the -w or -f options are specified for each directory, the tool looks for a configuration file; **version.in** unless overridden by the -c option. This file can contain two items

```
'[' [versionFile] [ '|' crlf_override ] ']'
one or more lines of template text

The items should appear in order but are both optional. White blank lines
preceeding each are skipped.
```

verisonFile defines the file to generate
crlf_override controls the crlf format for the generated file if required, specifically

```
lf  -> unix style new lines             e.g. perl requires this
crlf -> windows style new lines         e.g. for use with old software
development tools
```

if omitted it defaults to the native text format of the os being used

template text can be any text with the strings
@v@ or @V@ being replaced with the version string and
@d@ or @D@ being replaced the current date time in the format yyyy-mm-dd hh:mm:ss
Note, to enable the tool to pick out an old version string, one of the lines containing @v@ should also contain the text string git_version (case insensitive). Unlike previous implementations the version string no longer needs to be included in quotes

template examples

```
PL/M writing to file ver.pex
[ver.pex]
DECLARE version(*) byte data('@v@', 0); /* git_version */
        built(*) byte data('@d@', 0);
Text file writing to file Package
[Package]
Git_Version     @v@
Last_Checked    @d@

Assembler writing to file ver.inc
[ver.inc]
myver:  db '@V@', 0     ; git_version
build:  db '@D@', 0     ; the build date
```

If the template file cannot be found or either of the items are missing the defaults used are

```
versionFile     _version.h
template        // Autogenerated version file
                #define GIT_VERSION "@v@"
```

makeRelease

This utility supersedes the previous script based tools and is use to generate a new release tag and consistent version file in git.

```
Usage:  makerelease [options]
Where options are:
  -r rel  use the specified release for the new version file
  -m msg  use msg as the commit msg
  -c file set alternative configuration file instead of version.in
  -d      show debugging information

Also supports single arguments -v, -V for version info and -h for help
Note makeRelease does not support releasing files when in the headless state
The handling of the configuration file is as per getVersion
```

release naming

Unlike the previous release tools, this version supports more than numeric release names. Specifically the release supports names of the following format

```
revision_number [qualifier qualifier_revision_number]
where qualfier can contain one or more letters and underlines and the
two revision numbers can contain only digits and cannot start with a 0
Note the current implementation limits the qualifier length to 15 characters and
the numeric values to 65535. They can be changed if required.

In the git repository the corresponding release tag is {dir}-r{release_name}
where {dir} is the parent directory name. The corresponding version string has
the commit date prefixed and if the qualifer is used a dash is added after the
revision_number.
Examples for files assuming the directory name is test
Release name        git tag             version string
7                   test-r7             2024.11.13.7
9beta1              test-r9beta1        2024.11.18.9-beta1
```

Setting the release name (-r rel)

To make naming releases straightforward makeRelease supports simplified name specification which is described below. In the descriptions **last_revision** refers to the highest numeric value used as the revision_number in existing unqualified tags. Note # is used to separate components in the table below, it is not actually typed.

Option	**new revision set to (->)**
None	->**last_revision** + 1
-r **number** on main or master branch	if **number** is 0 ->**last_revision+ 1** if **number** is > **last_revision** or **number** = **last_revision** and the commit day is the same: ->**number** else error

Option	new revision set to (->)
-r **number** not on main or master branch	This is processed using the next rule, after adding the qualifier text string dev
-r [**number**] **qualifier** [**qualifier_revision**]	If **number** is not specified or is 0 it is replaced by **last_revision+1** **last_qualifier_revision** is set to the highest **qualifier_revision** used in any previous tags with a **number#qualifier** prefix If **qualifier_revision** == 0 or is missing it is set to **last_qualifier_revision+1** if **qualifier_revision** > **last_qualifier_revision** or **qualifier_revision** == **last_qualifier_revision** and the commit day is the same: -> **number#qualifier#qualifier_revision** else error

Examples

```
Assuming the following tags exist, with corresponding commit dates and assuming
today is 2024.11.18
test-r5         2024.11.13
test-r6         2024.11.13
test-r7         2024.11.13
test-r7dev1     2024.11.13
test-r7dev2     2024.11.13r
test-r7rc1      2024.11.13
test-r8         2024.11.18
test-r9beta1    2024.11.18
then then the above rules would be interpreted as follows
option        generated tag version string
none or -r0    test-r9       2024.11.18.9        highest revision was 8
-r 10          test-r10      2024.11.18.10       explicitly set and > 8
-r 8           test-r8       2024.11.18.8        = highest revision and same day
-r 7           error                             less than highest revision
-r rc          test-r9rc1    2024.11.18.9-rc1    highest revision was 8
-r [0]beta[0]  test-r9beta2  2024.11.18.9-beta2  auto revisions
-r [0]beta1    test-r9beta1  2024.11.18.9-beta1  auto revision, same day so ok
-r 7rc1        error                             not same day
-r 7rc3        test-r7rc3    2024.11.18.7-rc3    > previous_qualifier_revision
```

Additional notes

The tool uses the invoke time in GMT to ensure consistency of the date and version, including avoiding risks around midnight.

If no files have changed, a git commit --amend is done
otherwise It then tries the commit, if message is specified, the commit message is
`{dir} '-'year.month.day.release: message`
or it invokes the editor, pre-populated with the text
`{dir} - year.month.day.release:`

Where `{dir}` is the containing directory name

If the commit is succeeds, the annotated tag is created, with the annotation message `Release {dir} - year.month.day.release`

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

Additional support tools

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

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
Notes
srcDir, dir and suffix can each be surrounded by double quotes to allow embedded
spaces and commas.
Each field is separated by whitespace and/or a comma.
```

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 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
[+|-] whitespace and/or 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 19-Nov-2024
```