Source Control Method & Approach

Jmr Reference

Summary Source control is a central element in secure file storage, transfer, sharing and use. It is also really damn easy and quick when practiced right, cross platform too. The following doc illustrates this with the intent of promoting wide & uniform use.

Author Justin Reina

Date 10/31/17

Platform Coverage
- Windows
- Linux
- Macintosh

Repository Location
- Local, stored only on your computer
  - e.g. on 'C:\' drive
- Remote, stored on an external server
  - e.g. GitHub

Software Selection
- terminal (Win: Cygwin/Linux:tty/ Mac:Terminal)
- git
- gitk

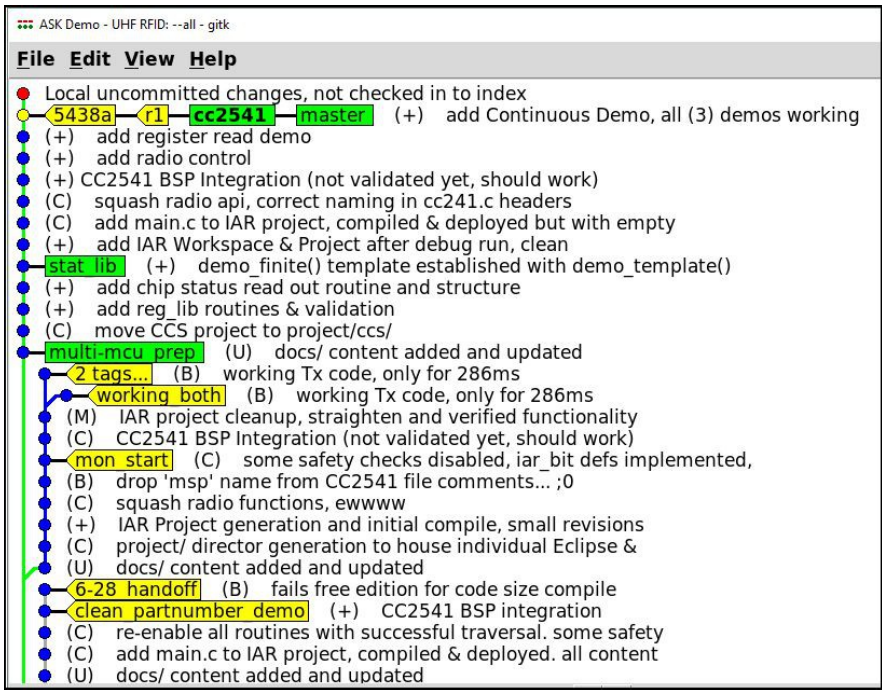
Auxillary Interface
- GitHub Desktop
- GitHub Web
- iOS CodeHub

<u>Useful & Recommended</u>
- Cygwin (Windows Bash Terminal)
- Eclipse (Git & Team Viewer Views)

Base Intent
- Track all revisions & changes
- Tag & track releases
- Tag & track development
- Dev ideas (e.g. a new feature, debugging an issue, etc.)

Example (Jmr, ASK Ref Project)

The example shown in Figure 1 illustrates:
- Tags(yellow) – tracking
  - e.g. 'r1' for rev 1, released to team
  - e.g. '6-28_handoff' for last handoff
- Branches (green) – development
  - e.g. 'stat_lib' for the statistics dev
- Form & Structure – type of commit
  - '(+)' – "Addition"
  - '(C)' – "Change"
  - '(B)' – "Bug"
  - '(M)' – "Misc."
  - '(U)' – "Update"
  - '(*)' – "Unknown"



```
Local uncommitted changes, not checked in to index
5438a r1 cc2541 master  (+)  add Continuous Demo, all (3) demos working
(+)  add register read demo
(+)  add radio control
(+) CC2541 BSP Integration (not validated yet, should work)
(C)  squash radio api, correct naming in cc241.c headers
(C)  add main.c to IAR project, compiled & deployed but with empty
(+)  add IAR Workspace & Project after debug run, clean
stat_lib  (+)  demo_finite() template established with demo_template()
(+)  add chip status read out routine and structure
(+)  add reg_lib routines & validation
(C)  move CCS project to project/ccs/
multi-mcu_prep  (U)  docs/ content added and updated
  2 tags...  (B)  working Tx code, only for 286ms
    working_both  (B)  working Tx code, only for 286ms
  (M)  IAR project cleanup, straighten and verified functionality
  (C)  CC2541 BSP Integration (not validated yet, should work)
  mon_start  (C)  some safety checks disabled, iar_bit defs implemented,
  (B)  drop 'msp' name from CC2541 file comments... ;0
  (C)  squash radio functions, ewwww
  (+)  IAR Project generation and initial compile, small revisions
  (C)  project/ director generation to house individual Eclipse &
  (U)  docs/ content added and updated
  6-28_handoff  (B)  fails free edition for code size compile
  clean_partnumber_demo  (+)  CC2541 BSP integration
  (C)  re-enable all routines with successful traversal. some safety
  (C)  add main.c to IAR project, compiled & deployed. all content
  (U)  docs/ content added and updated
```

Figure 1: Example Repository

Reference

- Git – About Version Control
- Wiki – Git

About Version Control (VCS)

Version control is a system that records changes over time so that you can them later. This allows easy retrieval of previous versions, enabling the following features -

1. Revert the entire project back to a previous state
2. Revert selected files back to a previous state
3. Compare changes over time
4. See who last modified something that might be causing a problem
5. Find who introduced an issue and when
6. and much more!

Using a VCS also generally means that if you screw things up or lose files you can easily recover them. And perhaps the best part, you get all this for very little overhead. It's easy!

Local Version Control Systems

Often times version-control is implemented by simple copy and paste, sometimes nestled within a ZIP file (e.g. "MyProject_v3.zip"). If clever they may even timestamp it, achieving version-control in principle. This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to.

To deal with this issue, programmers long ago developed local VCSs that had a simple database that kept all the changes to files under revision control. With Git, this is quite simply all of the contents within your *.git/* dir, plain and simple!

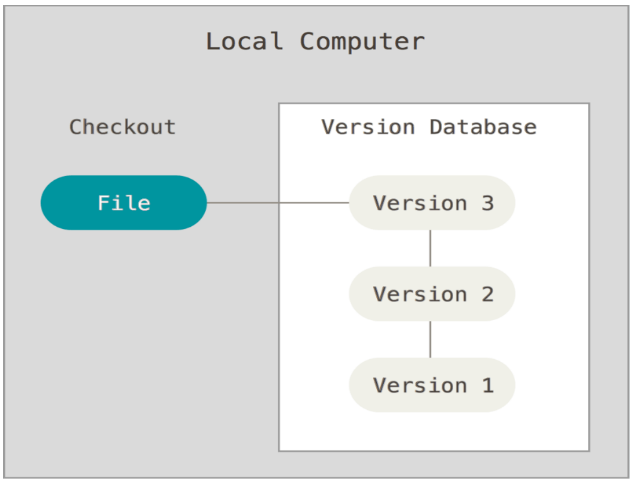


Figure 2: Version Control Example

Remote Version Control Systems

Take your local *.git/* and store its master version on a remote server, with a clean and snappy interface. Most commonly this is encountered with GitHub, where *.git/* is stored (*pushed*) and retrieved (*pulled*).

That's it!

Vocabulary

git

primary software used to implement version control. Created by Linus Torvalds to enhance & empower the linux movement, in which it has become central and primary in all aspects of development and use. Makes nodes of file contents, strung together into branches, forming a large tree of progress & activity, with the output of releases and products along the way

repository

A file archive where a large amount of source **code** is kept. Includes changes (commits), versions (branches) and releases (tags)

commit ('node')

A point along the path within a repository, a snapshot along a branch with full description of the repository at that moment along a branch

*e.g. at commit #ABCD file *sarah.txt* read "is happy", etc., for all contents in the repo at #ABC

branch

a path through a repository from one point to another

*e.g. for the new idea of popscile headlights we created the popscicle branch. On the first commit here we defined them, 2-5 prototyped, 5-10 created them and 6-20 debugged & finalized them, completing the idea and PoC

merge

any time two branches come together and join

*e.g. branch 'new_feature' was merged into 'master', updating the headlights

head ("HEAD")

Node sitting at the end of a branch

*e.g. "master(HEAD:#ABCD)" means the last commit on the master branch, with a SHA starting with 'ABCD' (for example, '#ABCD123456789...0')

SHA-ID ("SHA")

An ID number in a repository

*'SHA' is the algorithm, which generates a 40 character stream given an input blob of text (e.g. your commit's contents)

tag

An sticker or name applied to a node in a repository

*e.g. "working_new_vers", "friday_handoff", "r1"

master

Primary branch of a repository, from which most content is generated. The trunk, and if no branches all content in the repository lives here!

rebase

Revising Repo Contents

*edit, remove, squash, etc.

Example Use *Cheat Sheet*

Create a New Repository

git init

Grab a Local Copy of GitHub Repo

git clone repo-url	(e.g. 'https://github.com/ergsense/DTECTS_hw.git')

Check Repo Status (check for changes)

git status

Commit new content

*git add **	('*' for all new content, specific names otherwise)
git commit	

Reset Your Repo (i.e. reset back to HEAD)

git reset --hard

Switch to a Previous Commit

git checkout *sha-id*	(if lazy or pressed for time, use '~N' to offset from other content. e.g. "git checkout HEAD~1" to access the last commit before HEAD. You can use this for sha-id's, tags and branches too!)

Examples *Quick Reference*

Assumptions

- 'Repo' means local content here unless explicitly stated
  - e.g. "adding to the repo" means adding to your local .git/ repo copy

Notes

- Anytime work gets sticky and things won't successfully complete this gets easy, just force it
  - e.g. for the classic eggshell of a failed '*git push*', just use '*git push -f*'!
- Be sure to you are on correct branch before interacting with the remote repo.
  - If you want to interact with the whole repo, just use '*-a*' for all!

1. Create - New Local Repo

Here we will create a new repo, locally and use it, using an ammend & reset for illustration.

Establish – Create Dir & Place init content

note – for empty dirs, place an empty.txt inside for repo retention

git init	- Initialize repo
*git add **	- Add content
git status	- Confirm correct staging (new files, removed files, ignored content, etc.)
git commit	- Stores the added changes to a new commit, yeilding a sha-id for records
git status	- Confirm commit complete as intended
...	
git commit --amend	- Update commit's content, or change it's commit message
git reset --hard	- Reset the repo to it's initial state (value of commit <*sha-id*>)

2. Create – New GitHub Repo

Here we will create a new GitHub repo and add initial content, in preparation for future use.

Go to https://github.com/

Select 'New Repository' and enter a Repository name

Select 'Create Repository'

Use the created HTTPS value to checkout a local copy

e.g. '*git clone https://github.com/justinmreina/test.git*' <- The repo is now ready for use

3. Add – New Stuff to Repo

Here we update the project a bit with new content, and commit this to the repo (local).

*git add somefile1.txt somefile2.txt lib_dir/**	- add two files and an entire directory
git status	- Confirm correct staging
git commit	- Store the added changes to a new commit
git status	- Confirm commit complete as intended

4. Push - To GitHub

Here we push some new content (commits) to the GitHub remote repo.

git push - Push to remote repo. Use '-f' if tricky or painful!

5. Pull – From Github

Here we pull current content (commits) from the GitHub remote repo. This is done anytime you want to refresh or catch-up, e.g. to new content from the team, etc.

git pull - Push to remote repo. Use '-f' if tricky or painful!

6. Update – Copy of Remote GitHub Repo

Simple, '*git pull*'!

Core Components

Use the following ideas to properly achieve source control.

1. Your project is everything housed within the root directory. This includes:
   source code
   all settings (deployment & development, all!)
   description documents
   pictures
   ... quite literally everything that encompasses and contributes
   to the definition of your project/product!

2. Only one version of a project (or any file of the project!) is ever left present in the repository at a given time.
   If you want to retrieve a previous version though, it is simple. 'git checkout *sha-id*'!

3. Keep dev & experimentation local, on your PC. Only push to the server (GitHub) when ready, and complete!

4. Rebase or re-work your repository as softly and rarely as possible
   Only commit when complete. Branches otherwise!

Important Commands

Memorize these, to heart. This is 100% of what is needed to successfully implement & maintain source code control.

<u>Creation</u>

- "git init"
- "git status"

<u>Generation</u>

- "git add <*file*>"
- "git add *"
- "git rm <*file*>"

<u>Commit</u>

- "git commit"
- "git reset --hard" (reset to HEAD)
- "git reset --hard <*commit-id*>" (reset to a specific commit)

<u>Review & Correct</u>

- "gitk --all &" (see Figure 1 for example. My primary repo viewer, quick & easy)
- "git commit --amend" (update an existing commit)
- "git rebase -i HEAD~1" (where '1' is how far back you'd like to rebase)

Useful Commands

These are used often, and they promote clean & organized repository development. Establish the habit early, and often!

<u>Tagging & Tracking</u>

- "git tag *tag_name*" (tag a commit with a tag, a name for later use & reference)
- "git checkout -b *branch_name*" (checkout a new branch)

That's it!

Appendix A – Numbering Basics

The following numbering conventions are recomended, promoting uniform version numbering with clean maintenance and extensability. This format is intended for use through the entire product cycle, including conceptualization, prototype, development, productization & release.

Numbering Schema (numeric)

X.Y.Z
X - Major
Y - Minor
Z - Revision

Example

In communication the numbering schema used is decided at the time of use, e.g. -

- 'DTECTS 1' targets this market
- 'DTECTS 1.2' enables this feature
- 'DTECTS 1.2.5' fixes this bug

Notes

Product Naming is independent of the revision numbering. Revision numbering is primarily an internal tool, and may be used externally when needed.

Useful Reference

- Software Versioning - Wikipedia