

GIT User Guide

Internal Only

February 16th, 2016

Revision 1.3

THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE. Except for a limited copyright license to copy this specification for internal use only, no license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted herein.

Intel disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information in this specification. Intel does not warrant or represent that such implementation(s) will not infringe such rights.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

This document is an intermediate draft for comment only and is subject to change without notice. Readers should not design products based on this document.

Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

\*Other names and brands may be claimed as the property of others.

Copyright © 2015 Intel Corporation.

1. Contents

[1 Document Description 1](#_Toc456943824)

[1.1 Target Audience 1](#_Toc456943825)

[1.2 Terms 1](#_Toc456943826)

[1.3 Related Information 3](#_Toc456943827)

[1.3.1 Useful publications and sources of information 3](#_Toc456943828)

[1.4 Conventions Used in this Document 3](#_Toc456943829)

[2 Installation and Configuration 5](#_Toc456943830)

[2.1 Installation 5](#_Toc456943831)

[2.1.1 Git 5](#_Toc456943832)

[2.1.2 TortoiseGit 6](#_Toc456943833)

[2.2 Configuration 6](#_Toc456943834)

[2.2.1 Git 6](#_Toc456943835)

[2.2.2 TortoiseGit 10](#_Toc456943836)

[2.2.3 Outlook 12](#_Toc456943837)

[3 Setup EDKII Git Environment 15](#_Toc456943838)

[3.1 GitHub Access 15](#_Toc456943839)

[3.1.1 Setup SSH Key 15](#_Toc456943840)

[3.2 TeamForge Access 17](#_Toc456943841)

[3.3 Git EDKII (Mirror) 19](#_Toc456943842)

[3.3.1 Step by step to setup EDKII tree 19](#_Toc456943843)

[3.4 Git R9 (Mirror) 20](#_Toc456943844)

[3.4.1 Step by step to setup R9 tree 20](#_Toc456943845)

[3.5 Git-Svn EDKII (Read&Write) 21](#_Toc456943846)

[3.5.1 Step by step to setup EDKII tree 21](#_Toc456943847)

[4 Develop Process 24](#_Toc456943848)

[4.1 Developer Process 24](#_Toc456943849)

[4.1.1 Using Git 24](#_Toc456943850)

[4.1.2 Using TortoiseGit 30](#_Toc456943851)

[4.2 Maintainer Process 38](#_Toc456943852)

[4.2.1 Using Git 38](#_Toc456943853)

[4.2.2 Using TortoiseGit 41](#_Toc456943854)

[4.3 GitHub – Using “Fork” 45](#_Toc456943855)

[4.3.1 Using Git 47](#_Toc456943856)

[4.3.2 Using TortoiseGit 50](#_Toc456943857)

[4.4 Useful tips 57](#_Toc456943858)

[4.4.1 Git EDKII and SVN R9/BP mixed development 57](#_Toc456943859)

[4.4.2 EDKII and R9/BP multiple repos development 57](#_Toc456943860)

[Appendix A Git Commands 59](#_Toc456943861)

[A.1 Git Commands 59](#_Toc456943862)

[A.1.1 Am 59](#_Toc456943863)

[A.1.2 Branch & Tag 59](#_Toc456943864)

[A.1.3 Cherry-pick 61](#_Toc456943865)

[A.1.4 Clone 61](#_Toc456943866)

[A.1.5 Commit 62](#_Toc456943867)

[A.1.6 Diff & Difftool 62](#_Toc456943868)

[A.1.7 Fetch & Pull 63](#_Toc456943869)

[A.1.8 Format-patch 63](#_Toc456943870)

[A.1.9 Log 64](#_Toc456943871)

[A.1.10 Merge 64](#_Toc456943872)

[A.1.11 Push 64](#_Toc456943873)

[A.1.12 Rebase 65](#_Toc456943874)

[A.1.13 Remote 66](#_Toc456943875)

[A.1.14 Reset 67](#_Toc456943876)

[A.1.15 Rm 69](#_Toc456943877)

[A.1.16 Send-email 69](#_Toc456943878)

[A.1.17 Status 70](#_Toc456943879)

[A.2 TortoiseGit Operations 70](#_Toc456943880)

[A.2.1 Apply patch 70](#_Toc456943881)

[A.2.2 Branch & Tag 71](#_Toc456943882)

[A.2.3 Cherry-pick 73](#_Toc456943883)

[A.2.4 Clone 74](#_Toc456943884)

[A.2.5 Commit 74](#_Toc456943885)

[A.2.6 Create patch 75](#_Toc456943886)

[A.2.7 Diff 77](#_Toc456943887)

[A.2.8 Fetch & Pull 79](#_Toc456943888)

[A.2.9 Log 79](#_Toc456943889)

[A.2.10 Merge 81](#_Toc456943890)

[A.2.11 Push 82](#_Toc456943891)

[A.2.12 Rebase 83](#_Toc456943892)

[A.2.13 Remote 84](#_Toc456943893)

[A.2.14 Reset 84](#_Toc456943894)

[A.2.15 Rm 85](#_Toc456943895)

[Appendix B Git Config File(.gitconfig) 87](#_Toc456943896)

[Appendix C ssh Config File (.ssh/config) 88](#_Toc456943897)

[Appendix D subversion servers File (.subversion/servers) 89](#_Toc456943898)

[Appendix E Config File Location 90](#_Toc456943899)

[Appendix F Proxy List 91](#_Toc456943900)

[Appendix G Svn and Git Mapping 92](#_Toc456943901)

1. FIGURES

[**Figure 1. Steps for installing Git on Windows OS** 6](#_Toc456943902)

[**Figure 2. Steps for configuring Git at global level** 7](#_Toc456943903)

[**Figure 3. Steps for configuring Git at repository level** 9](#_Toc456943904)

[**Figure 4. Files to ignore at repository level** 10](#_Toc456943905)

[**Figure 5. Configure Git using TortoiseGit** 11](#_Toc456943906)

[**Figure 6. SSH client replacement for TortoiseGit** 11](#_Toc456943907)

[**Figure 7. Outlook configuration for EDKII patch review** 13](#_Toc456943908)

[**Figure 8. Outlook configuration for interleaved posting** 14](#_Toc456943909)

[**Figure 9. Add SSH key** 16](#_Toc456943910)

[**Figure 10. Add an SSH key to GitHub** 16](#_Toc456943911)

[**Figure 11. Reactivate TeamForge account** 18](#_Toc456943912)

[**Figure 11. Search Projects** 18](#_Toc456943913)

[**Figure 12. Join the Project** 18](#_Toc456943914)

[**Figure 13. Setup SSH key in TeamForge** 19](#_Toc456943915)

[**Figure 14. Steps for cloning EDKII tree from GitHub** 20](#_Toc456943916)

[**Figure 15. Proxy setting for git-svn** 21](#_Toc456943917)

[**Figure 16. Clone SVN trees via git-svn using TortoiseGit** 22](#_Toc456943918)

[**Figure 17. Get latest code via git-svn using TortoiseGit** 22](#_Toc456943919)

[**Figure 18. Push change back to SVN server via git-svn using TortoiseGit** 23](#_Toc456943920)

[**Figure 19. Conflicts occur during git-rebase** 26](#_Toc456943921)

[**Figure 20. Resolve conflicts by text editor** 26](#_Toc456943922)

[**Figure 21. Example repository history for modifying multiple commits** 28](#_Toc456943923)

[**Figure 22. Choose which commits to modify** 28](#_Toc456943924)

[**Figure 23. Clone a Git repository using TortoiseGit** 30](#_Toc456943925)

[**Figure 24. Getting changes on “master” from remote using TortoiseGit** 31](#_Toc456943926)

[**Figure 25. Integrate changes from “master” to “devel” using TortoiseGit** 31](#_Toc456943927)

[**Figure 26. Steps for creating patches for review using TortoiseGit** 32](#_Toc456943928)

[**Figure 27. Modify the message of the last local commit using TortoiseGit** 33](#_Toc456943929)

[**Figure 28. Modify the messages of several local commits using TortoiseGit** 34](#_Toc456943930)

[**Figure 29. Modify the content of several local commits using TortoiseGit** 35](#_Toc456943931)

[**Figure 30. Modify the order of several local commits using TortoiseGit** 36](#_Toc456943932)

[**Figure 31. Modify the order of several local commits using TortoiseGit** 36](#_Toc456943933)

[**Figure 32. Split local commits using TortoiseGit** 37](#_Toc456943934)

[**Figure 33. Delete local commits using TortoiseGit** 38](#_Toc456943935)

[**Figure 34. “GitPatchExtractor” Outlook plugin usage** 39](#_Toc456943936)

[**Figure 35. Add “Reviewed-by” attributions** 41](#_Toc456943937)

[**Figure 36. Add “Reviewed-by” to one commit using TortoiseGit** 42](#_Toc456943938)

[**Figure 37. Add “Reviewed-by” to multiple commits using TortoiseGit** 44](#_Toc456943939)

[**Figure 38. Integrate change to “master” for push operation using TortoiseGit** 44](#_Toc456943940)

[**Figure 39. Push local “master” branch to remote “master” branch at “origin”** 45](#_Toc456943941)

[**Figure 40. Open-source EDKII project at GitHub** 46](#_Toc456943942)

[**Figure 41. One’s own fork of the EDKII project** 46](#_Toc456943943)

[**Figure 42. Delete remote branches via the web interface** 49](#_Toc456943944)

[**Figure 43. Delete the EDK2 fork repository** 50](#_Toc456943945)

[**Figure 44. Add the fork remote using TortoiseGit** 51](#_Toc456943946)

[**Figure 45. Create the “fork\_master” branch using TortoiseGit** 52](#_Toc456943947)

[**Figure 46. Keep remotes aligned using TortoiseGit** 53](#_Toc456943948)

[**Figure 47. Push changes to the forked repository using TortoiseGit** 54](#_Toc456943949)

[**Figure 48. Create a local branch for review using TortoiseGit** 55](#_Toc456943950)

[**Figure 49. Push commits to upstream using TortoiseGit** 56](#_Toc456943951)

[**Figure 50. Delete a remote locally using TortoiseGit** 57](#_Toc456943952)

[**Figure 51. Local branches information** 60](#_Toc456943953)

[**Figure 52. Repository structure after git-cherry-pick** 61](#_Toc456943954)

[**Figure 53. Edit commit message using “vim”** 62](#_Toc456943955)

[**Figure 54. Repository structure before git-merge** 64](#_Toc456943956)

[**Figure 55. Repository structure after git-merge** 64](#_Toc456943957)

[**Figure 56. Repository structure after git-rebase onto “master”** 65](#_Toc456943958)

[**Figure 57. Repository structure that two branches share a common change** 66](#_Toc456943959)

[**Figure 58. Repository structure after git-rebase onto “master” (2)** 66](#_Toc456943960)

[**Figure 59. Remotes information after git-clone** 66](#_Toc456943961)

[**Figure 60. Three modes of git-reset command** 69](#_Toc456943962)

[**Figure 61. Difference between git-rm and rm** 69](#_Toc456943963)

[**Figure 62. A clean working tree** 70](#_Toc456943964)

[**Figure 63. Status showing differences between working tree & staging area** 70](#_Toc456943965)

[**Figure 64. Apply “.patch” files to Git repository using TortoiseGit** 71](#_Toc456943966)

[**Figure 65. Create a local branch using TortoiseGit** 71](#_Toc456943967)

[**Figure 66. Switch to a local branch using TortoiseGit** 72](#_Toc456943968)

[**Figure 67. Rename/Delete a local branch using TortoiseGit** 72](#_Toc456943969)

[**Figure 68. Create a tag using TortoiseGit** 73](#_Toc456943970)

[**Figure 69. Steps for cherry-picking commits from one branch to another using TortoiseGit** 74](#_Toc456943971)

[**Figure 70. Clone a Git repository using TortoiseGit** 74](#_Toc456943972)

[**Figure 71. Commit local changes to the staging area using TortoiseGit** 75](#_Toc456943973)

[**Figure 72. Patch subject prefix setting in TortoiseGit** 76](#_Toc456943974)

[**Figure 73. Steps for creating patches for review using TortoiseGit** 77](#_Toc456943975)

[**Figure 74. Generated patches and an additional cover letter** 77](#_Toc456943976)

[**Figure 75. Show difference between two commits using TortoiseGit** 78](#_Toc456943977)

[**Figure 76. Show difference between two branches using TortoiseGit** 78](#_Toc456943978)

[**Figure 77. Fetch changes from a Git remote using TortoiseGit** 79](#_Toc456943979)

[**Figure 78. Pull changes from a Git remote using TortoiseGit** 79](#_Toc456943980)

[**Figure 79. Log history of a Git repository using TortoiseGit** 80](#_Toc456943981)

[**Figure 80. Show remote-tracking branches using TortoiseGit** 81](#_Toc456943982)

[**Figure 81. Possible log history of a project** 81](#_Toc456943983)

[**Figure 82. Steps for merging commits from one branch to another using TortoiseGit** 82](#_Toc456943984)

[**Figure 83. Log history after the git-merge operation** 82](#_Toc456943985)

[**Figure 84. Push local commits to the remote Git repository using TortoiseGit** 83](#_Toc456943986)

[**Figure 85. Possible log history of a project** 83](#_Toc456943987)

[**Figure 86. Log history after the git-rebase operation** 84](#_Toc456943988)

[**Figure 87. Associated URL of a Git remote using TortoiseGit** 84](#_Toc456943989)

[**Figure 88. Steps for undoing changes on a local branch using TortoiseGit** 85](#_Toc456943990)

[**Figure 89. Remove files from Git repository using TortoiseGit** 86](#_Toc456943991)

1. TABLES

[Table 1 Font Conventions 3](#_Toc456943992)

1. Revision History

|  |  |  |
| --- | --- | --- |
| Revision Number | Description | Revision Date |
| 0.1 | Initial document | 02/10/2015 |
| 0.2 | Draft content | 03/10/2015 |
| 1.0 | First version | 03/31/2015 |
| 1.1 | 1. Remove SubModule usage  2. Add multiple workspace usage.  3. Give TianoAd SVN and GIT mapping | 10/30/2015 |
| 1.2 | 1. Move Git usage as appendix  2. Add more useful tips. | 11/17/2015 |
| 1.3 | 1. Refine content in Chapter 4  2. Remove Appendix H | 02/16/2016 |

# Document Description

## Target Audience

This internal document targets for developers who use GIT in EDKII and R9/BP daily development work.

## Terms

The following terms are used throughout this document to describe varying aspects of input localization:

Commit

A commit is a snapshot of your project – code, files, everything — at a particular point in time.

Branch

A branch is simply a lightweight movable pointer to a series of commits. Branching means you diverge from the main line of development and continue to do work without disturbing the main line commit history.

Git directory

The Git directory (.git) is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you clone a repository from another computer.

Git Bash

Git Bash is a Linux-style shell included in “Git for Windows”, and is a slimmed-down version of Cygwin. It provides enough of a POSIX layer to run a bash and it supports all Git commands.

HEAD

HEAD is the pointer to the current branch reference, which is in turn a pointer to the last commit made on that branch. That means HEAD will be the parent of the next commit that is created. It’s generally simplest to think of HEAD as the snapshot of your last commit.

Index

Synonym for “Staging area”.

Master branch

Like the other branches, “master” does not have any special meaning. It is the default name for a starting branch when you run git-init

Remote

Remote usually means remote repositories. Remote repositories are versions of your project that are hosted on the Internet or network somewhere. You can have several of them, each of which generally is either read-only or read/write for you.

Remote “origin”

Like the branch name “master” does not have any special meaning in Git, “origin” is the default name for a remote when you run git-clone.

Remote branches

Branches at the Git repository server.

Remote-tracking branches

Remote-tracking branches are references (pointers) to the state of branches in your remote repositories. They are local branches that you can’t move; they’re moved automatically for you whenever you do any network communication. Remote-tracking branches act as bookmarks to remind you where the branches on your remote repositories were the last time you connected to them.

Repository

A repository is a collection of your code, which you’ve created to host your code changes, from commits to pushes. A repository is the key aspect of Git and version controls.

Staging area

The staging area is a file, generally contained in your Git directory, which stores information about what will go into your next commit. It’s sometimes referred to as the “index”, but it’s also common to refer to it as the staging area.

Tag

Tag is a snapshot of a particular branch – tag is created on a branch. The tagged code contents cannot be altered afterwards – that is the main difference between branch and a tag.

Tracking branches

Checking out a local branch from a remote branch automatically creates what is called a “tracking branch” (or sometimes an “upstream branch”). Tracking branches are local branches that have a direct relationship to a remote branch. If you’re on a tracking branch and type git pull, Git automatically knows which server to fetch from and branch to merge into.

Working directory

The working directory is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.

## Related Information

### Useful publications and sources of information

*Pro Git*, a book written by Scott Chacon and Ben Straub and published by Apress, <http://git-scm.com/book/en/v2>.

*TortoiseGit*, a Windows Shell Interface to Git and based on TortoiseSVN. It's open source and can fully be built with freely available software, <https://code.google.com/p/tortoisegit/>.

*Wikipedia*, a free-access, free content Internet encyclopedia, supported and hosted by the non-profit Wikimedia Foundation. Those who can access the site and follow its rules can edit most of its articles, <https://www.wikipedia.org/>.

GitHub is a web-based Git repository hosting service, which offers all of the distributed revision control and source code management (SCM) functionality of Git as well as adding its own features, <https://github.com/>

Secure Git is an IT solution that integrates open-source tools such as Git, Gerrit, and Gitweb in a secure, enterprise-wide manner using a third-party product called TeamForge (TF), <http://goto/teamforge>

## Conventions Used in this Document

This document uses typographic and illustrative conventions described below.

Table 1 Font Conventions

| Typographic Convention | Typographic Convention Description |
| --- | --- |
| Plain Text | The normal text typeface is used for the vast majority of the descriptive text in a specification. |
| Plain Text (blue) | Any plain text that is underlined and in blue indicates an active link to the cross-reference. |
| **Bold** | In text, a **Bold** typeface can be used as a running head within a paragraph. |
| *Italic* | In text, an *Italic* typeface can be used as emphasis to introduce a new term or to indicate a manual or specification name. |
| BOLD Monospace | Computer code, example code segments, and all prototype code segments use a BOLD Monospace typeface with a dark red color. These code listings normally appear in one or more separate paragraphs, though words or segments can also be embedded in a normal text paragraph. |
| **Bold Monospace** | Words in a **Bold** Monospace typeface that is underlined and in blue indicate an active hyperlink to the code definition for that function or type definition. Click on the word to follow the hyperlink. |
| $(VAR) | This symbol VAR defined by the utility or input files. |
| Italic Monospace | In code or in text, words in Italic Monospace indicate placeholder names for variable information that must be supplied (i.e., arguments). |

1. Due to management and files size considerations, only the first occurrence of the reference on each is an active link. Subsequent references on the same page will not be actively linked to the definition and will use the standard, nonunderlined BOLD Monospace typeface. Find the first instance of the name (in the underlined BOLD Monospace typeface) on the page and click on the word to jump to the function or type definition.

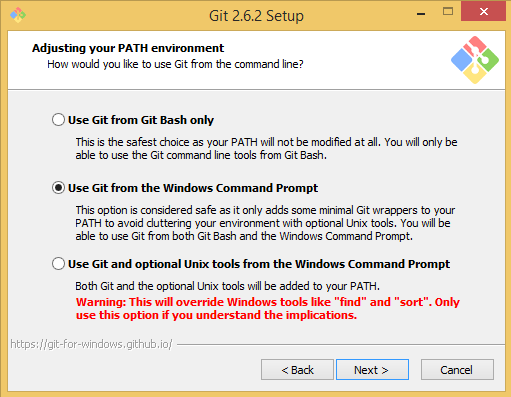
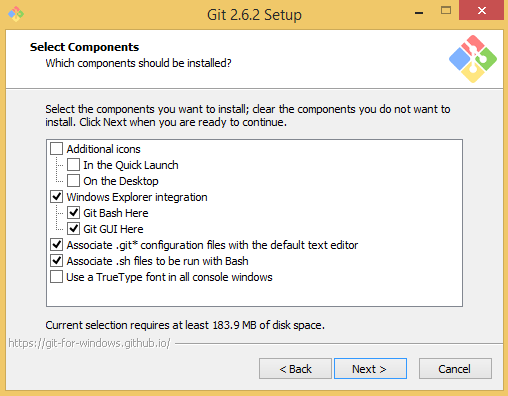
# Installation and Configuration

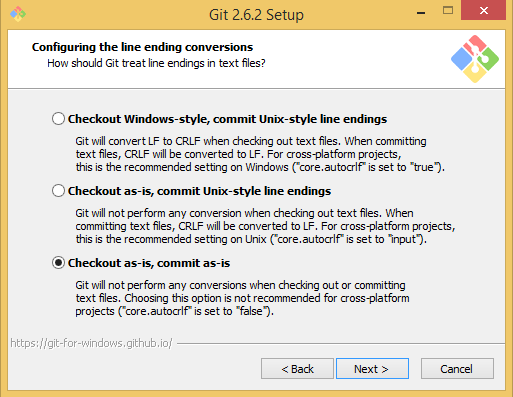
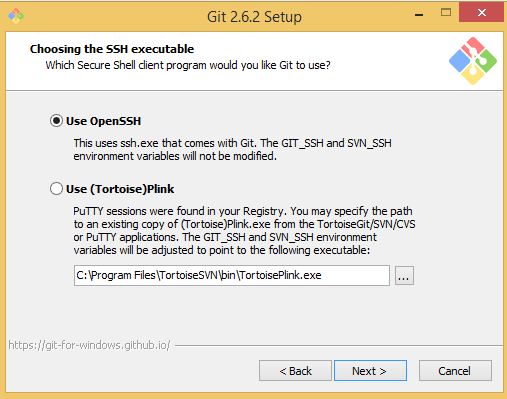
This chapter will introduce the installation of Git and TortoiseGit. Configurations for Git, TortoiseGit and Outlook will also be shown.

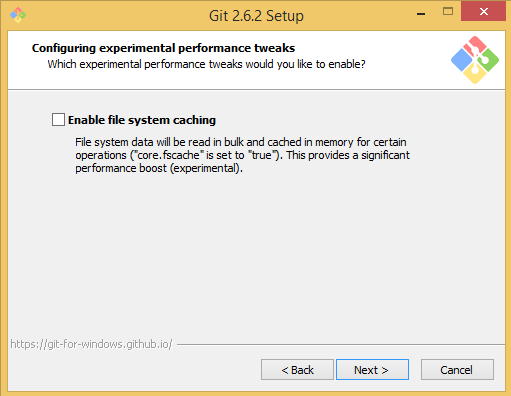
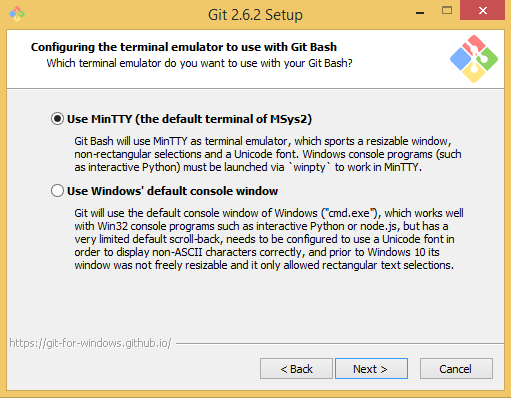
## Installation

### Git

Git for Windows OS is available at: <http://git-scm.com/download/win> (For now, please choose the 32-bit version, the 64-bit version has some issues with git-svn commands). After downloading the setup file, please following the steps shown below to install Git. Here are the installation steps of Git 2.6.2. The latest Git version may be the different step. Please be aware of the configuration in each step.







**Figure 1. Steps for installing Git on Windows OS**

Selections in each setup page can be modified according to your preference or project requirements. Pictures above displays the recommended settings for installing Git under Windows OS.

### TortoiseGit

TortoiseGit for windows OS is available at <https://tortoisegit.org/download/>. After downloading the setup file, follow the instructions and use the default settings during installation.

## Configuration

### Git

Git comes with a tool called git config that lets you get and set configuration variables that control all aspects of how Git looks and operates.

On Linux system, these variables can be stored in three different places. Each level overrides values in the previous level, so values in .git/config trump those in /etc/gitconfig.

1. /etc/gitconfig file: Contains values for every user on the system and all their repositories. If you pass the option --system to git config, it reads and writes from this file specifically.
2. ~/.gitconfig or ~/.config/git/config file: Specific to your user. You can make Git read and write to this file specifically by passing the --global option.
3. config file in the Git repository directory (that is, .git/config) of whatever repository you’re currently using: Specific to that single repository.

On Windows systems, Git looks for the .gitconfig file in the C:\Users\UserId. It also looks for /etc/gitconfig that’s relative to the git installation directory in Windows system.

In addition, we strongly suggest using SSH protocol to communicate with Git servers. The protocol use a manually generated public-private key pair to perform the authentication, allowing users or programs to log in without having to specify a password.

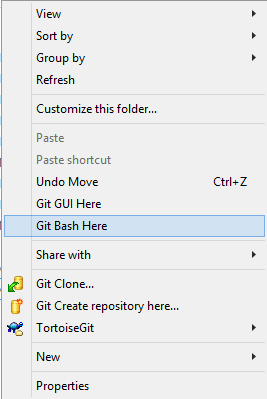
#### System Level

Since different users may have different preferences for configuring Git, so no system level settings for Git is configured. If one feels like an option is necessary for all the users of a PC, one can move the corresponding configuration variables from global/repository level to here.

#### Global Level

One can configure the global level settings via opening the Windows file explorer and right click in any directory. Then select “Git Bash Here”, a command line windows will show up. Type the below command:

vim ~/.gitconfig



**Figure 2. Steps for configuring Git at global level**

In the editor, input the following mandatory configurations:

**[user]**

**name = Your Name**

**email = your.name@intel.com**

**[format]**

**coverLetter = auto**

**[sendemail]**

**smtpserver = smtp.intel.com**

**confirm = always**

**suppresscc = self**

**[core]**

**autocrlf = false**

You can also add some global level configurations to improve development efficiency.

If one feels like to use external text editor (i.e. Notepad++) instead of the default vim editor, one can add the below configuration under “[core]” field in ~/.gitconfig:

**autocrlf = false**

Or use the below command:

git config --global core.editor "'C:/Program Files (x86)/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin"

If one feels like to use external diff utility/3-way merge tool (i.e. BeyondCompare3) instead of using git-diff command, one can add the below configuration in ~/.gitconfig:

**[diff]**

**tool = bc3**

**[difftool "bc3"]**

**path = c:/Program Files (x86)/Beyond Compare 3/bcomp.exe**

**[merge]**

**tool = bc3**

**[mergetool "bc3"]**

**path = c:/Program Files (x86)/Beyond Compare 3/bcomp.exe**

Or use the following commands:

git config --global diff.tool bc3

git config --global difftool.bc3.path "c:/Program Files (x86)/Beyond Compare 3/bcomp.exe"

git config --global merge.tool bc3

git config --global mergetool.bc3.path "c:/Program Files (x86)/Beyond Compare 3/bcomp.exe"

To use external diff utility tools, one can use git-difftool command instead of git-diff with the same parameters. To use external 3-way merge tool, one can type

git mergetool <*Filename*>

If one needs to use proxy to access Git repositories by http/https protocol (Git URLs starting with http(s)://), one can add below configuration in ~/.gitconfig:

**[url "https://"]**

**insteadOf = git://**

**[https]**

**proxy =** <*proxyname*>**.intel.com:**<*port*>

**[http]**

**proxy =** <*proxyname*>**.intel.com:**<*port*>

Or use the following commands:

git config --global https.proxy <*proxyname*>.intel.com:<*port*>

git config --global http.proxy <*proxyname*>.intel.com:<*port*>

Common proxy list:

proxy = proxy-prc.intel.com:911

proxy = child-prc.intel.com:913

proxy = proxy-us.intel.com:911

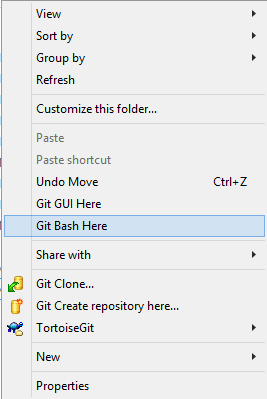
proxy = proxy-sc.intel.com:911

#### Repository Level

None of the following configurations at repository level is mandatory, however, one may find some of them will help during develop process.

One can configure the repository level settings via right click in a specific Git repository. Then select “Git Bash Here” and type the below command:

vim .git/config



**Figure 3. Steps for configuring Git at repository level**

If one feels like having a commit message template for a Git repository, one can add the below configuration in .git/config inside a Git repository:

**[commit]**

**template = path/to/msgtemplate.txt**

Or use the below command:

git config commit.template path/to/msgtemplate.txt

The content of file msgtemplate.txt may differ among repositories. Here’s an example for the open-source EDKII project:

**PkgName-ModuleName: Brief-single-line-summary**

**Full-commit-message**

**Cc: Package Maintainer Name <Package Maintainer Email Address>**

**Contributed-under: TianoCore Contribution Agreement 1.0**

**Signed-off-by: Contributor Name <contributor@email.server>**

**Reviewed-by: Reviewer Name <reviewer@reviewer-email.server>**

If one put the template file directly in the Git repository, then this file will need to be listed in the repository .gitignore file (will be introduced later). Otherwise, one should give an absolute/relative path of the template file.

A .gitignore file specifies intentionally untracked files that Git should ignore. Files already tracked by Git are not affected, if one do want to ignore a tracked file, one need to remove the file using git-remove command first and then add the file to .gitignore.

If one feels like having untracked files for a Git repository, one can add a .gitignore file inside a Git repository:



**Figure 4. Files to ignore at repository level**

And adding the following lines (an example) to the file:

**Build/**

**tags/**

**\*.bak**

**\*.patch**

Please be careful when editing the .gitignore file under the root for open-source EDKII repository. Since the .gitignore file has already be tracked by the repository, if .gitignore is modified for local usage, one must make sure this change should not be included in any commits unless having the permission.

There is a more simple method of having a local intentionally untracked files list for EDKII repository (or any repository with .gitignore already tracked by Git). One can create a file outside of the repository (e.g. ../myignore.txt) and add the below configuration under “[core]” field in .git/config inside the EDKII repository:

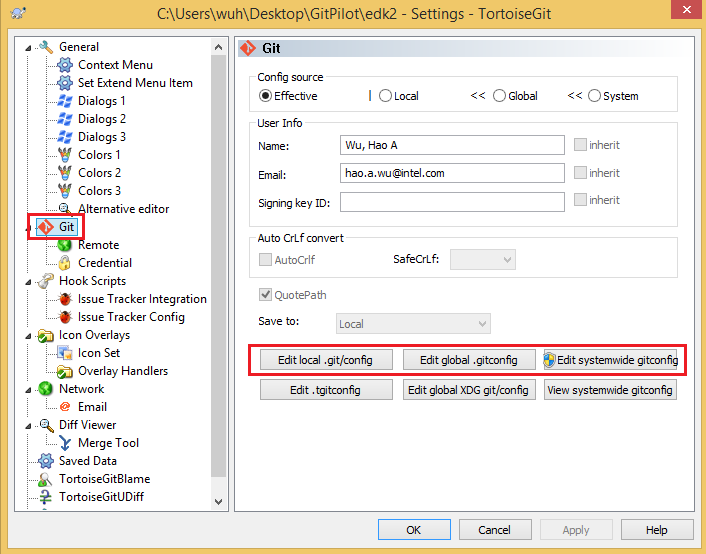
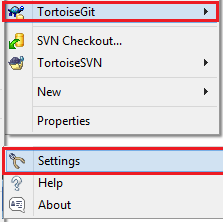
**excludesfile = ../myignore.txt**

Or use the below command:

git config core.excludesfile ../myignore.txt

### TortoiseGit

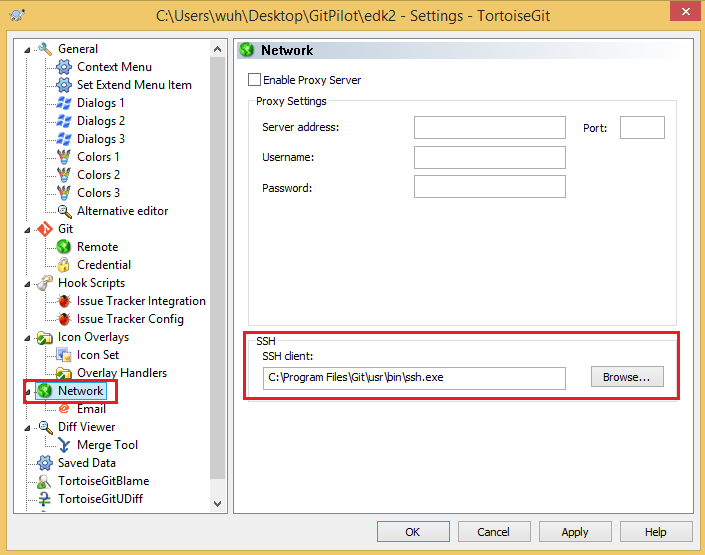
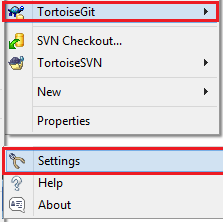
TortoiseGit is also capable of configuring Git at three levels. One can open the Windows file explorer and right click in any directory (repository level configuration will require right click inside a Git repository). Then follow the steps shown in **Figure 5**.



**Figure 5. Configure Git using TortoiseGit**

After selecting one of the setting files, an editor will pop out, users can following the instructions given in Section 2.2.1 to configure Git.

Besides, if one will use proxy to access Git repositories by ssh protocol (Git URLs starting with ssh://), one can open the Windows file explorer and right click in any directory and follow the steps shown in **Figure 6**.



**Figure 6. SSH client replacement for TortoiseGit**

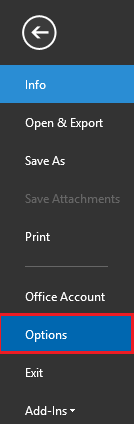
It’s worth pointing out that the default installation folder for Git is changing between versions. Therefore, the “ssh.exe” location might be different from the one shown in **Figure 6**. One can find the location of “ssh.exe” by opening the Windows file explorer and right click in any directory. Then select “Git Bash Here”, type the following command:

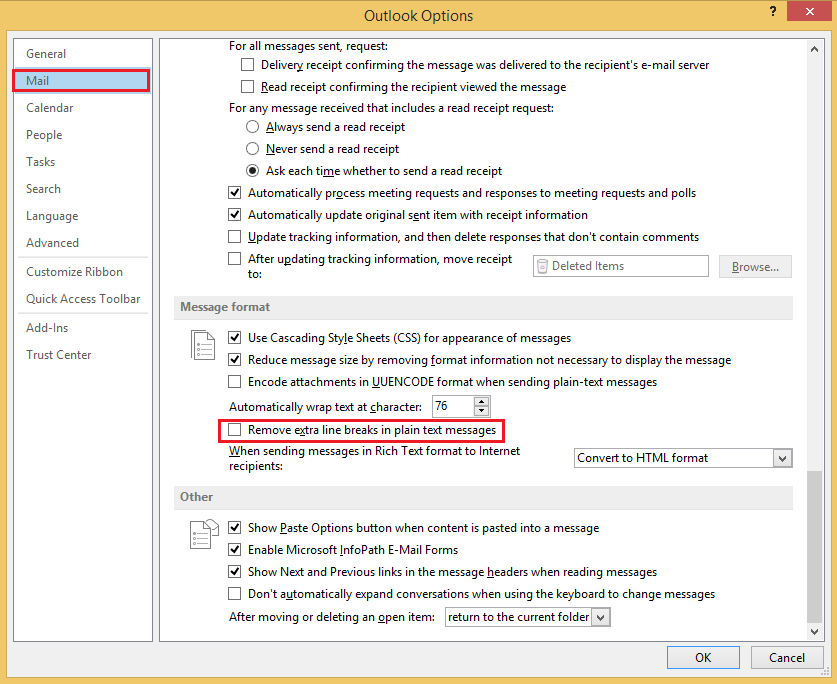
where ssh

### Outlook

Some Outlook configurations are necessary during the review process of the open-source EDKII project. If one does not involve in the patch review process, this section can be skipped.

Normally, the patch itself is in the mail content. Therefore, the reviewers need to configure Outlook to make sure that it will NOT remove extra line breaks which might corrupt the patch:

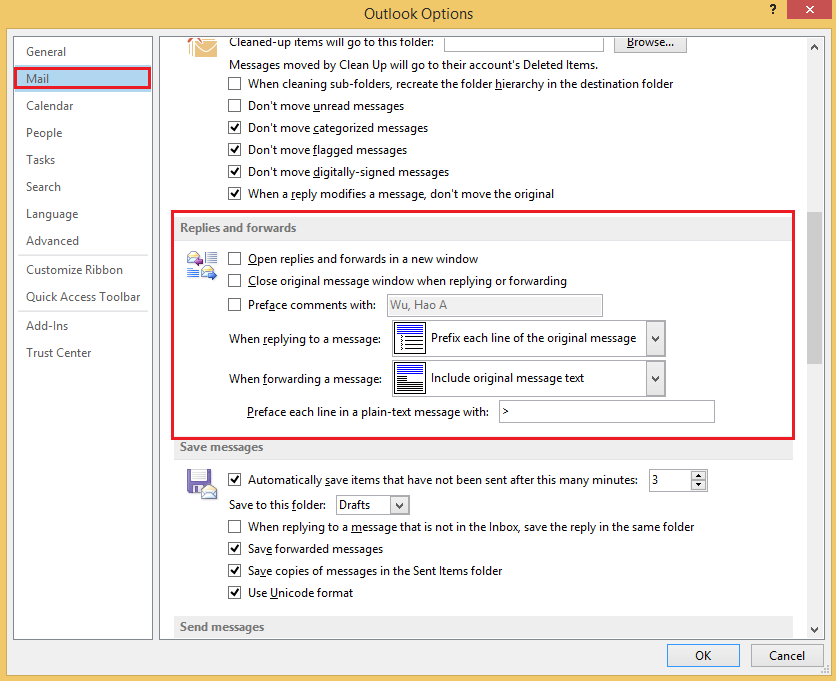




**Figure 7. Outlook configuration for EDKII patch review**

When replying patch mails, open-source contributors seem to have a preference for the posting styles. The main options are interleaved posting (also called inline replying, in which the different parts of the reply follow the relevant parts of the original post), bottom-posting (in which the reply follows the quote) or top-posting (in which the reply precedes the quoted original message).

In open-source community, they prefer to use interleaved posting. One can use this kind of posting manner by making the following changes in Outlook:



**Figure 8. Outlook configuration for interleaved posting**

# Setup EDKII Git Environment

## GitHub Access

When you go to <https://github.com/> for the first time, pick an email and set a password to create an account. Then follow the instructions to finish the registration.

### Setup SSH Key

SSH keys are a way to identify trusted computers, without involving passwords.

Firstly, we need to check for existing SSH keys on your computer. Open the Windows file explorer and right click in any directory. Then select “Git Bash Here” and type:

ls -al ~/.ssh

Check the directory listing to see if you already have a public SSH key. The default public key file names are:

id\_dsa

id\_rsa.pub

Secondly, to generate a new SSH key, copy and paste the text below, making sure to substitute in your email address. The default settings are preferred, so when you're prompted to "Enter a file in which to save the key", just press Enter to continue.

ssh-keygen -t rsa -C "your\_email@example.com"

Then, you'll be asked to enter a passphrase (which is optional). To move on, add your new key to the ssh-agent:

eval `ssh-agent`

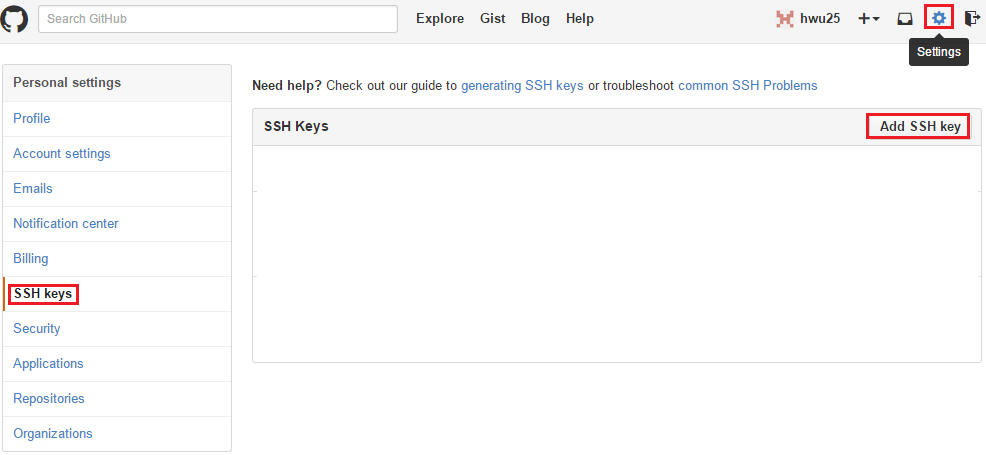
ssh-add ~/.ssh/id\_rsa

Thirdly, run the following command to copy the key to your clipboard.

clip < ~/.ssh/id\_rsa.pub

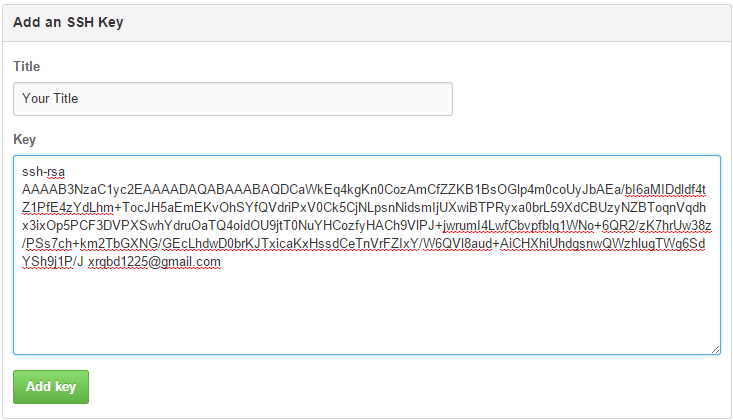
Now that you have the key copied, it's time to add it to GitHub:

1. Go to <https://github.com/>
2. In the top right corner of any page, click your user profile icon and select the “Settings” option on the top-right corner shown on the below:
3. In the user settings sidebar, click SSH keys.
4. Click Add SSH key.



**Figure 9. Add SSH key**

1. In the Title field, add a descriptive label for the new key. For example, you might call this key “edk2 github”.
2. Paste your key into the "Key" field.
3. Click Add key.



**Figure 10. Add an SSH key to GitHub**

1. Confirm the action by entering your GitHub password.

Finally, to make sure everything is working, you'll now try SSHing to GitHub. When you do this, you will be asked to authenticate this action using your password, which was the passphrase you created earlier.

Open up your Git Bash and type:

ssh -T git@github.com

You may see this warning:

The authenticity of host 'github.com (207.97.227.239)' can't be established.

RSA key fingerprint is ...

Are you sure you want to continue connecting (yes/no)?

Verify that the fingerprint in your terminal matches the one we've provided up above, and then type “yes.” If that username is yours, you've successfully set up your SSH key.However, if you get the following error message:

ssh: connect to host github.com port 22: Bad file number

For Windows OS users, you need to create a new file named config inside your ~/.ssh/ folder. This is usually located at C:\Users\USERNAME\.ssh\. Paste the following code and enter a workable proxy server into your config file. And, make sure the directory that includes connect.exe be set in PATH env so that connect.exe can be found. The directory that includes connect.exe can be found by where connect.exe command in git Bash.

Host github.com

User git

Hostname ssh.github.com

PreferredAuthentications publickey

IdentityFile ~/.ssh/id\_rsa

Port 443

ProxyCommand connect.exe -H <proxyaddr>.intel.com:<port> %h %p

For Linux OS users, you need to install “corkscrew” first via the following command:

sudo apt-get install corkscrew

Then a file named config should be created under directory ~/.ssh/. Paste the following code and enter a workable proxy server into your config file.

Host github.com

User git

Hostname ssh.github.com

PreferredAuthentications publickey

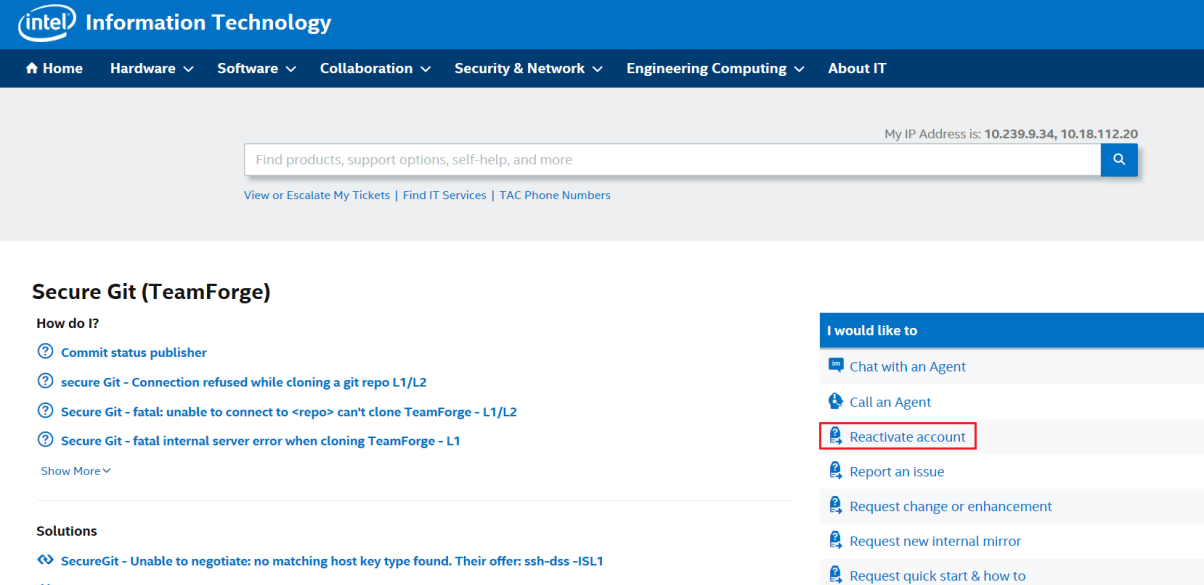
IdentityFile ~/.ssh/id\_rsa

Port 443

ProxyCommand corkscrew <proxyaddr>.intel.com <port> %h %p

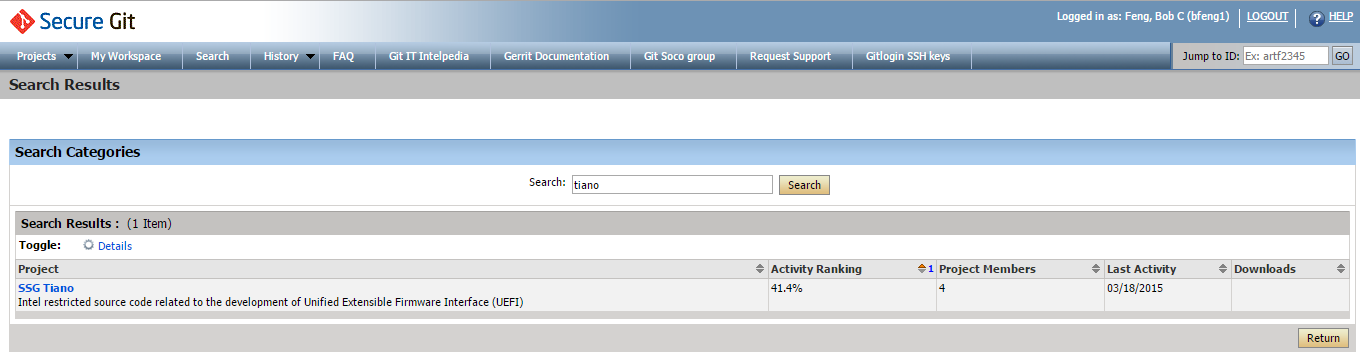
## TeamForge Access

Go to <http://goto/teamforge>, create an account, log in with your corporate user name and password. If an error shown as “Your account is deleted” occurred during the log in process, one can go to this link <http://it.intel.com/#/topic/826> to reactivate the TeamForge account.

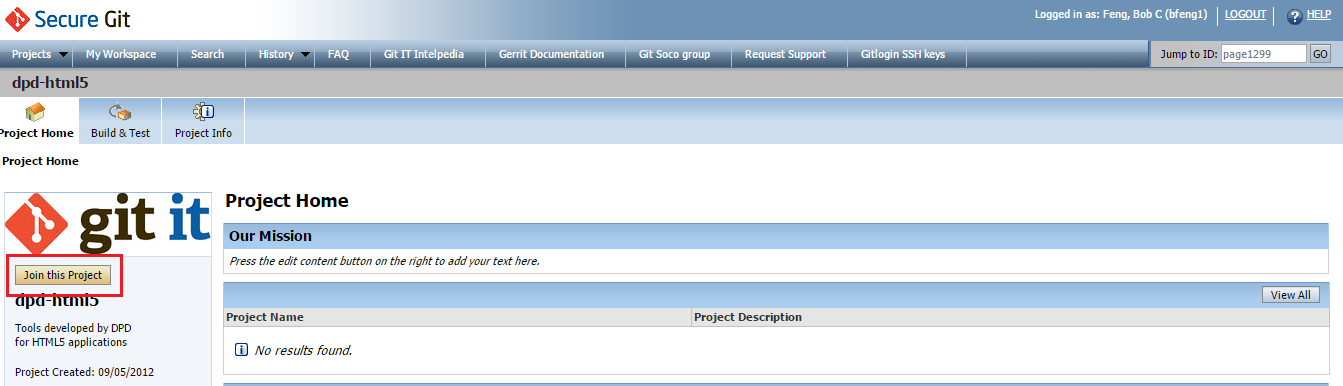


**Figure 11. Reactivate TeamForge account**

Find the specific project by clicking **Project->Project Categories** and inputting the project name to the search input. Clicking the project name and in the project page, clicking Join this Project button. The project administrator needs to approve that request.



**Figure 12. Search Projects**

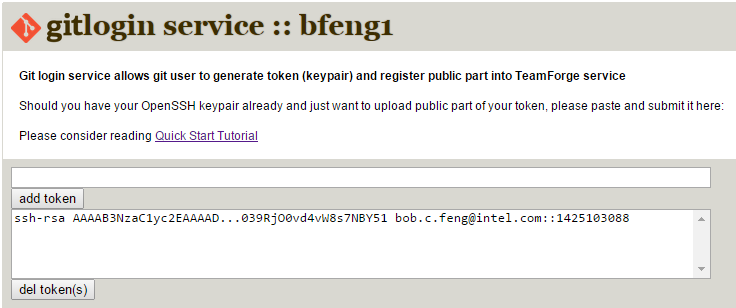


**Figure 13. Join the Project**

Similar to the above section, copy the SSH Key by running the following command in “Git Bash”:

clip < ~/.ssh/id\_rsa.pub

Then, login to <https://gitlogin.devtools.intel.com/> with your TeamForge credentials, paste the key string and click "add token"



**Figure 14. Setup SSH key in TeamForge**

Modify config file to list intel.com address and ssh-dss Algorithms

Host \*intel.com

HostKeyAlgorithms +ssh-dss

The command below is a quick user test to see what key/transport problems may/may not exist. If you get the “Welcome to Gerrit” text, your SSH configuration is correct.

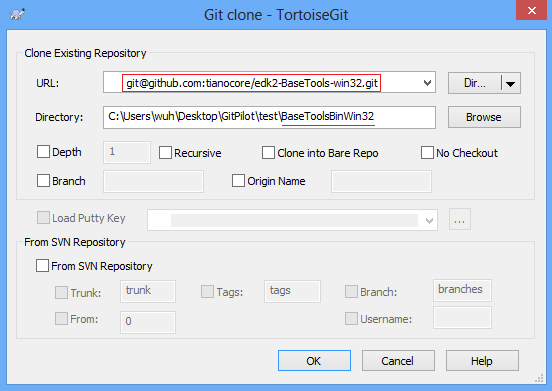
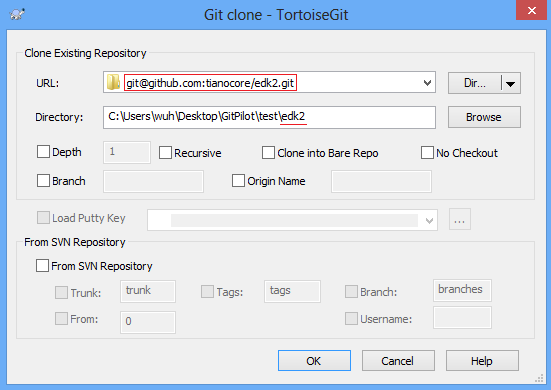
ssh -p 29418 git-amr-1.devtools.intel.com

## Git EDKII (Mirror)

### Step by step to setup EDKII tree

To setup an EDKII tree, two Git repositories should be cloned from GitHub, the edk2 ([git@github.com:tianocore/edk2.git](mailto:git@github.com:tianocore/edk2.git)) and the edk2-BaseTools-win32 ([git@github.com:tianocore/edk2-BaseTools-win32.git](mailto:git@github.com:tianocore/edk2-BaseTools-win32.git)).





**Figure 15. Steps for cloning EDKII tree from GitHub**

After the two Git repositories are cloned, run cmd.exe and configure EDK\_TOOLS\_BIN environment. Then, enter into edk2 directory, type edksetup.bat.

set EDK\_TOOLS\_BIN=%CD%\BaseToolsBinWin32

This command will let edksetup.bat know where to find the binary BaseTools.

Please keep in mind that “edk2” and “BaseToolsBinWin32” are two independent Git repositories, one needs to use Git Pull command in both directories to get the newest codes for the EDKII tree.

## Git R9 (Mirror)

Apply TeamForge Access can refer to Git Migration Guide document Chapter 4.2.1

### Step by step to setup R9 tree

The Git R9 repository consists of four parts.

|  |  |
| --- | --- |
| edk2 | git@github.com:tianocore/edk2.git |
| edk2-BaseTools-win32 | git@github.com:tianocore/edk2-BaseTools-win32.git |
| FatPkg | git@github.com:tianocore/edk2-FatPkg.git |
| ssg\_tiano-r9 | ssh://git-amr-1.devtools.intel.com:29418/ssg\_tiano-r9 |

Create root directory, then clone them one by one. After all of them are cloned, run cmd.exe and type ssg\_tiano-r9/R9Setup.bat to configure multiple workspaces and the binary tool path, then call platform scripts to build platform.

Please keep in mind that they are four independent Git repositories. To pull the latest code, Git Pull command is required to run in each Git repository. To switch to remote branch, Git Checkout command is required to run in each Git repository.

## Git-Svn EDKII (Read&Write)

Command “git svn” is a bidirectional operation between a Subversion repository and Git. It is a simple conduit for change sets between Subversion and Git. It provides a bidirectional flow of changes between a Subversion and a Git repository.

Developers can use this tool to manage local codes via Git while communicating with a remote SVN repository.

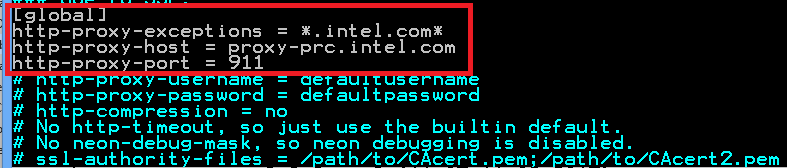
### Step by step to setup EDKII tree

Because git-svn will not clone the external part of an SVN repository, one has to clone two SVN repositories to setup an EDKII tree: the SVN EDKII tree (<https://svn.code.sf.net/p/edk2/code/trunk/edk2>) and SVN EDKII BaseTools Win32 tree ([https://svn.code.sf.net/p/edk2-toolbinaries/code/trunk/Win32](https://svn.code.sf.net/p/edk2-toolbinaries/code/trunk/Win32%20)).

Before cloning the two SVN trees using git-svn, one need to open git bash command line tool and run the following command.

vim ~/.subversion/servers

Modify or add those lines according to your local proxy settings.



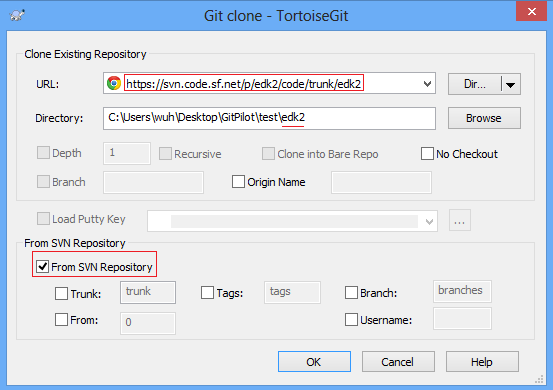
**Figure 16. Proxy setting for git-svn**

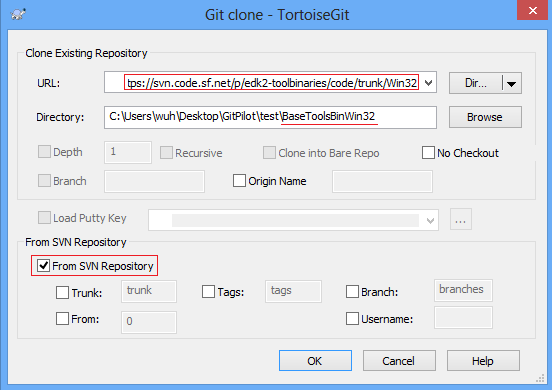
One can clone the SVN EDKII tree and SVN EDKII BaseTools Win32 tree using Git commands in “Git Bash” by:

git svn clone https://svn.code.sf.net/p/edk2/code/trunk/edk2

git svn clone https://svn.code.sf.net/p/edk2-toolbinaries/code/trunk/Win32

Also, one can use the TortoiseGit clone operation as well:





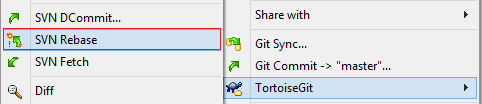
**Figure 17. Clone SVN trees via git-svn using TortoiseGit**

After the two Git repositories are cloned, run cmd.exe and configure EDK\_TOOLS\_BIN environment. Then, enter into edk2 directory, type edksetup.bat.

set EDK\_TOOLS\_BIN=%CD%\BaseToolsBinWin32

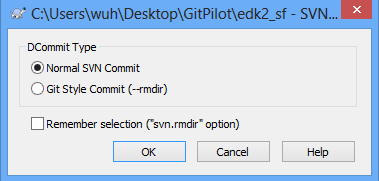
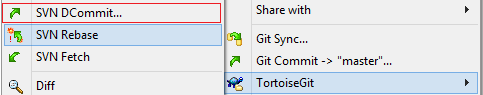
This command will let edksetup.bat know where to find the binary BaseTools.

Please keep in mind that “edk2” and “BaseToolsBinWin32” are two independent repositories, one needs to use git svn rebase command in “Git Bash” (or TortoiseGit “SVN Rebase” operation) for both directories (despite the soft link created before) to get the newest codes for the EDKII tree.



**Figure 18. Get latest code via git-svn using TortoiseGit**

One can push local commits back to SVN server by git svn dcommit or use the “SVN Dcommit...” operation via TortoiseGit.



**Figure 19. Push change back to SVN server via git-svn using TortoiseGit**

This operation will push all the commits that are not on the SVN server.

# Develop Process

For now, only open-source EDKII project gets migrated from SVN to Git, while closed-source BP and R9 projects remain using SVN. Therefore, this chapter mainly focus on the develop process for the EDKII project. The mixed developing scenario of EDKII and R9/BP is briefly covered in Section 4.4.

## Developer Process

In this section, the developer process that is applicable to the EDKII project is given in detail.

### Using Git

#### Clone the Repository

One should first select a directory to put EDKII codes. Once the decision is made, one can open the “Git Bash” command line and type one of the following commands to clone the EDKII Git repository:

git clone https://github.com/tianocore/edk2.git

git clone git@github.com:tianocore/edk2.git

The difference of the two git-clone command is that the former uses HTTPS protocol and the latter uses SSH protocol when communicating with the GitHub server. Moreover, using the SSH protocol requires setting up GitHub SSH key, which is introduced in Section 3.1.1.

When the clone operation completes, a remote called “origin” will appear in the repository.

#### Create Branch for Development

After the clone operation is done, the active local branch is called “master”. This branch works similarly as the “trunk” for Subversion.

Now one can start his/her code development by creating a new local branch (In this section, branch “devel” will be used as an example for the new local branch) based on the “master”:

git checkout –b devel master

At this point, branch “devel” is the same as “master”, but after the developers make some local code changes and commit them to “devel”, the two branches will apparently diverge.

It is strongly recommended that developers should create a new branch based on “master” when implementing a new feature or fixing a new bug. The branch name is supposed to reflect the summary of the work. After the works on the new branch pass the code review and ready for check-in, they will be integrated back to the “master” branch. Therefore, “master” should always stay in a clean state during the develop process.

If one needs to work based on branch rather than “master” (e.g. branch “UDK2014.SP1”), one can checkout to that branch before creating a new branch for development:

git checkout UDK2014.SP1

#### Commit Local Changes

After changes have been made in the working tree, one can commit them to the staging area by:

git add <file1> <file2> ...

git commit

The detailed commands are listed at Section A.1.5.

The EDKII project has requirements for the format of commit message. Its commit message format can be found in Contributions.txt of every package. One can create a template commit message file and use it at the global or repository level. Please refer to Section 2.2.1.3 for more details on configuring a template for Git commit message.

#### Integrate Remote Changes

Before creating patches for local commits, it is better to integrate the latest changes from remote to see whether there is any conflict between one’s local changes and latest remote changes.

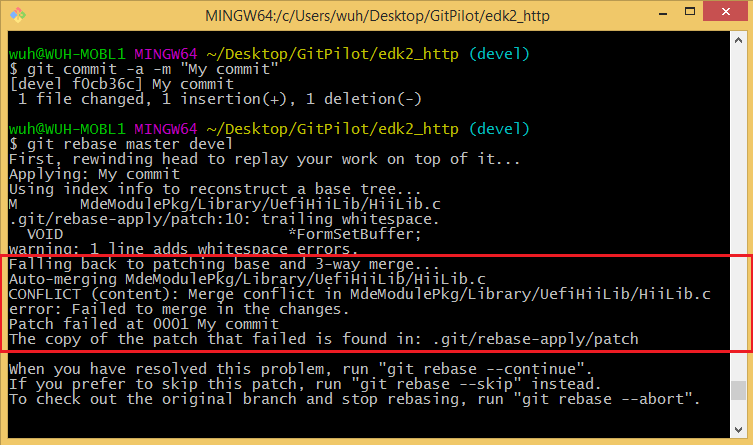
Once again, if our “devel” branch is based on the “master” branch, one need to execute the following commands:

git checkout master

git pull (or git pull origin)

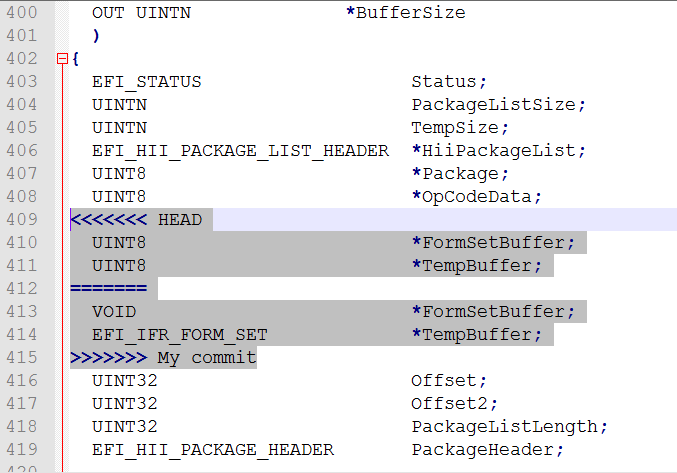
git rebase master devel

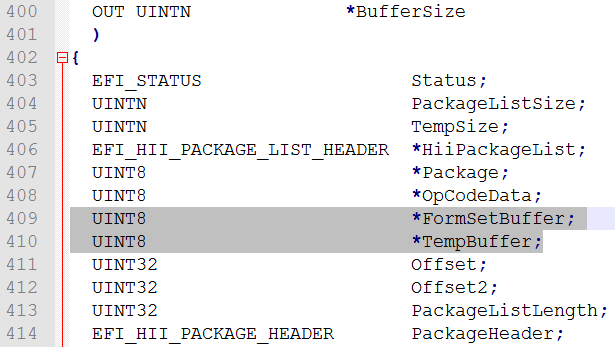
If there are conflicts during the rebase process, it is likely that one of your local commits modified the same place of a file with a remote commit. And Git cannot decide which one should be taken:



**Figure 20. Conflicts occur during git-rebase**

One should resolve the conflict manually by open the conflict file via an editor, found the text “<<<<<<< HEAD” which marks the starting point of the conflict:





**Figure 21. Resolve conflicts by text editor**

The content between “<<<<<<< HEAD” and “=======” is the changes made by remote (i.e. changes on local “master” branch). While the content between “=======” and “>>>>>>> <Local commit message title>” is the changes made by developers at their local “devel” branch. In the example shown by the upper part of **Figure 21**, both commits modify the type of pointer “FormSetBuffer”, and Git does not know which one to take. A possible solution for the conflicts is shown by the lower part of **Figure 21**, which simply takes the changes made by remote and abandon the local changes.

One can also setup using an external merge tool (refer Section 2.2.1.2 for details) to resolve conflicts during git-rebase process:

git mergetool

After all the conflicts have been resolved, one can run:

git add -A

git rebase --continue

If more conflicts occur during the git-rebase process, please follow the above steps to resolve until the rebase command succeeds.

#### Create and Send Patch

When latest remote changes have been applied to one’s “devel” branch, it is better to re-test the changes brought by local commits at “devel”. After that, one can create patches for local commits for reviewing.

The detailed steps for creating patches are mentioned in Section A.1.8. If patches are created for multiple commits, a cover-letter named “0000-cover-letter.patch” will be generated automatically. It is highly recommended that one should edit this file to write a summary for the general purpose of the patch series.

After patch files are generated, one should use the “PatchCheck.py” script under “edk2\BaseTools\Scripts” directory to verify the commits are correctly formatted. To check the latest *N* commits, one can use the following command:

python BaseTools/Scripts/PatchCheck.py –N (replace “N” with any number)

If any patch fails the check, one can follow the steps in Section 4.1.1.6 to modify the local commits and generate the patch files again.

If all the patches pass the check, one can then use the git-send-email command introduced in Section A.1.16 for mail review. A common git-send-email command for EDKII project may look like:

git send-email \*.patch --to <edk2\_mailing\_list> --to <reviewer\_email> --cc <package\_maintainer\_email> --cc <others>

#### Modify Local Commits

Normally, one will get feedbacks from patch reviewers and need to modify local commits for further review. This section will introduce some common scenarios when editing local commits.

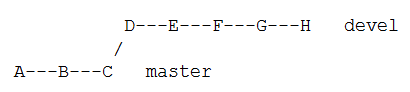
If the latest commit on current branch is going to be modified, one can directly make file changes at the working tree. Once files changes are done, type:

git add –A

git commit --am

The “vim” editor will pop out, showing the commit message of the latest commit, one can modify the message or just leave it unchanged.

If multiple commits (E, G) are going to be changed for the following history:



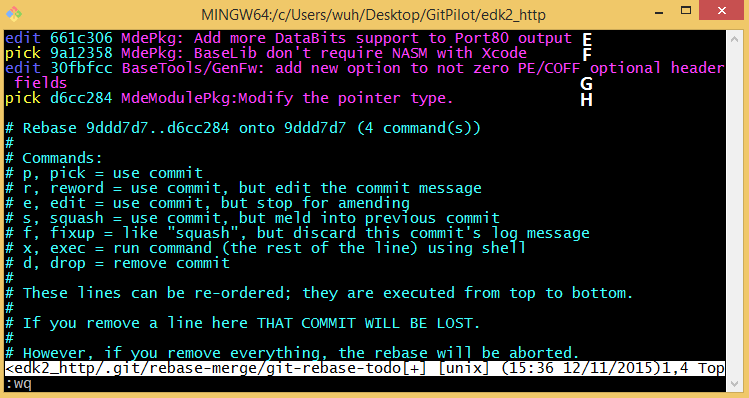
**Figure 22. Example repository history for modifying multiple commits**

One can use one of the below commands:

git rebase –i <sha1\_for\_E>~1

git rebase –i <sha1\_for\_D>

Next, the “vim” editor will pop out:



**Figure 23. Choose which commits to modify**

Replace the “pick” keyword with “edit” for commits that are going to be modified. Then the git-rebase command will stop the rebase process at the commits whose option is “edit”. In this example, the rebase process will stop at commit E and G twice. When stopping at commit E, Git will treat E as the latest commit of a temporary branch, one can go through the process for changing the latest commit to modify commit E. Once done, use the command:

git rebase –-continue

The rebase process will be restarted and then it will stop at commit G. After all the commits marked with “edit” is processed, the rebase process is done.

If one wants to change the order of some commits, command git rebase –i can also be used. In the pop out “vim” editor, one can re-order the commits and leave the “pick” keyword untouched. Please make sure that, commits which have dependencies (e.g. commit G changes the lines added in commit F) cannot be re-ordered, otherwise, the rebase process fails.

If one wants to split a commit (e.g. commit E), git rebase –i <sha1\_for\_E>~1 is used and select “edit” for commit E in “vim”. When the rebase process stops at commit E, follow the below commands:

git reset --mixed HEAD~1 (or git reset HEAD~1)

git add <file1> <file2> ...

git commit (edit commit message for split commit #1)

git add <more\_files>

git commit (edit commit message for split commit #2)

...

git add –A (add all the remaining files to the staging area)

git commit (edit commit message for the last split commit)

git rebase –-contine

If one wants to combine multiple adjacent commits together (e.g. combine commit E, F and G), git rebase –i <sha1\_for\_E>~1 is used and select keyword “squash” for commit F and G. The rebase process will stop at commit G and automatically pop out the “vim” editor containing a combination of three commits. One can work out a new commit message for the new commit based on original three commit messages. Once the new message is finalized, the rebase process will restart automatically.

If one wants to delete a commit from the log history (e.g. commit F), git rebase –i <sha1\_for\_F>~1 is used and choose keyword “drop” for commit F. Please make sure that commits G and H are not dependent on F, otherwise, the rebase process fails.

After changes have been made according to the review feedback, one can go through steps described in Section 4.1.1.5 to send out review mails again. Normally, a new subject prefix indicating the version of the patch series is needed, the steps can be found in Section A.1.8.

#### Push Changes

When local changes pass review process, module/package owners will help developers to push the changes to GitHub. Therefore, developers do not need to push their commits to the EDKII GitHub remote.

#### Delete Local Branch

After local changes have been pushed to the Git server, one may delete the “devel” branch since it has already been integrated to the “master” branch.

One should make sure that “devel” is not the current active branch, and then it can be deleted via:

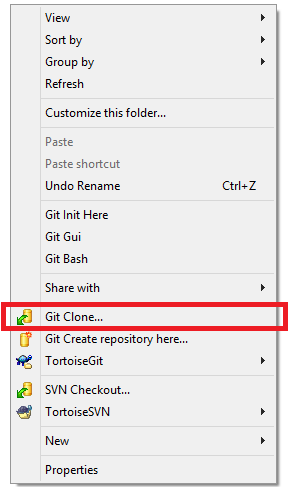
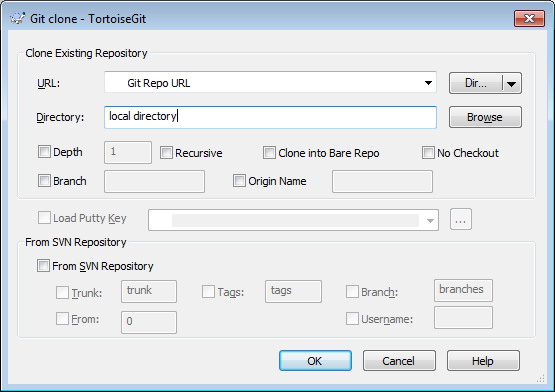
git branch –D devel

### Using TortoiseGit

In the previous section, Git commands used during the develop process for a general project have been discussed in detail. This chapter will introduce the corresponding TortoiseGit operations. It is highly recommended that one should go through Section A.2 and Section 4.1.1 before continuing.

#### Clone the Repository

One can clone a Git repository to your local computer by: Choose a directory to put the code – Right click at the directory – Git Clone… – Input the URL of the Git repository.

**Figure 24. Clone a Git repository using TortoiseGit**

#### Create Branch for Development

Creating and switching branches via TortoiseGit is introduced in Section A.2.2. Please refer to those sections for more details.

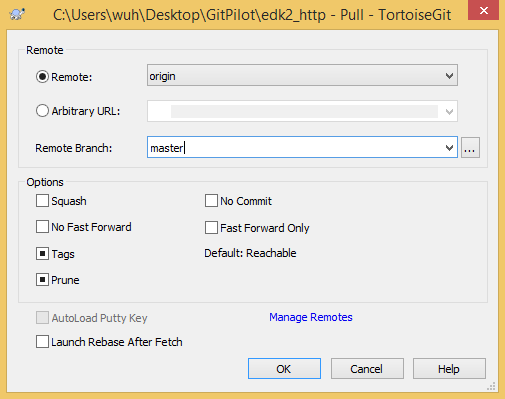
Local branch “devel” will be used in this section as a demonstration local branch name during develop process.

#### Commit Local Changes

Commit local changes via TortoiseGit is introduced in Section A.2.5. Please refer to those sections for more details.

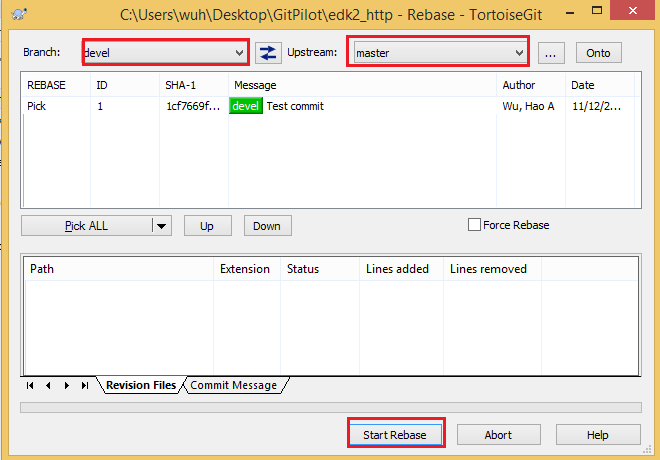
#### Integrate Remote Changes

One should first switch to the “master” branch via method introduced in Section A.2.2, then one should use the “Pull…” to get the latest code changes for “master” branch from the Git server:



**Figure 25. Getting changes on “master” from remote using TortoiseGit**

Next, one needs to integrate the latest changes on “master” to branch “devel”. One can do this via: Right click at the directory – TortoiseGit – Rebase... and choose the options shown by **Figure 26**:

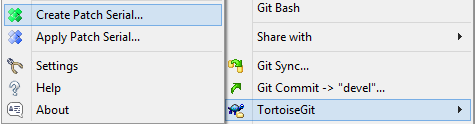


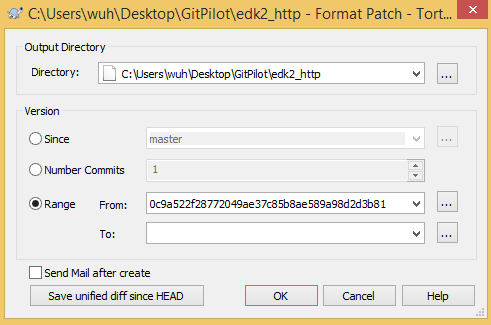
**Figure 26. Integrate changes from “master” to “devel” using TortoiseGit**

If one encounters conflicts during the integration process, please refer to Section 4.1.1.4 to resolve them.

#### Create and Send Patch

To create patches for local commits: right-click in the working directory – TortoiseGit – Prepare Patch serial… – Select the local commits for generating patches.





**Figure 27. Steps for creating patches for review using TortoiseGit**

One can choose the location of the generated patch file(s) in the “Output Directory” field.

Meanwhile, the “Version” field is used to specify the local commits that will be generated to patch files.

If “Since” is selected, one needs to choose a branch (e.g. “master” here). Then TortoiseGit will find the first common commit (e.g. commit A) on both branches (“master” and current local branch “devel”) and generate patches for each commit starting from A to the latest commit on “devel”.

If “Number Commits” is selected, a positive number “x” is then required. TortoiseGit will generate patches for the latest “x” commits on branch “devel”.

If “Range” is selected, the SHA1 value of two commits should be given. Then TortoiseGit will find the first common ancestor of both commits (e.g. commit B) and generate patches for each commit starting from B to commit given in the “To:” field.

After patch files are generated, one should use the “PatchCheck.py” script under “edk2\BaseTools\Scripts” directory to verify the commits are correctly formatted. To check the latest *N* commits, one should open a command line at the root directory of the EDKII project and then use the following command:

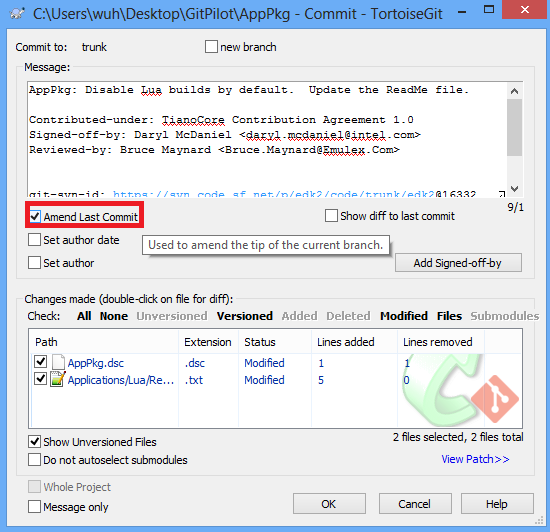
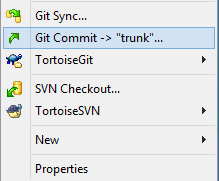
python BaseTools/Scripts/PatchCheck.py –N (replace “N” with any number)

If any patch fails the check, one can follow the steps in Section 4.1.2.6 to modify the local commits and generate the patch files again.

When it comes to sending patches for mail review, the functions TortoiseGit provide cannot meet the requirements for the EDKII mailing list, thus one should use git-send-email command in “Git Bash” instead. Please refer to Section 4.1.1.5 for more details.

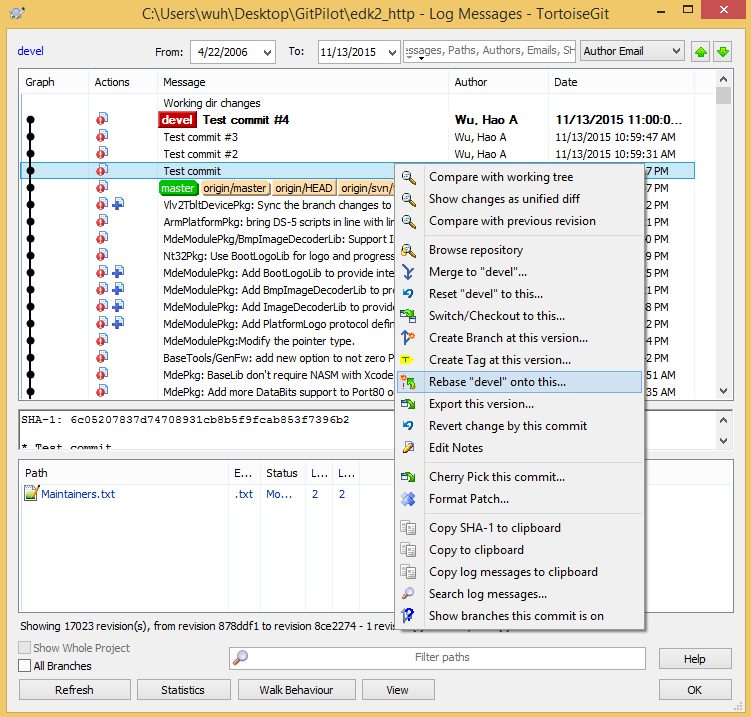
#### Modify Local Commits

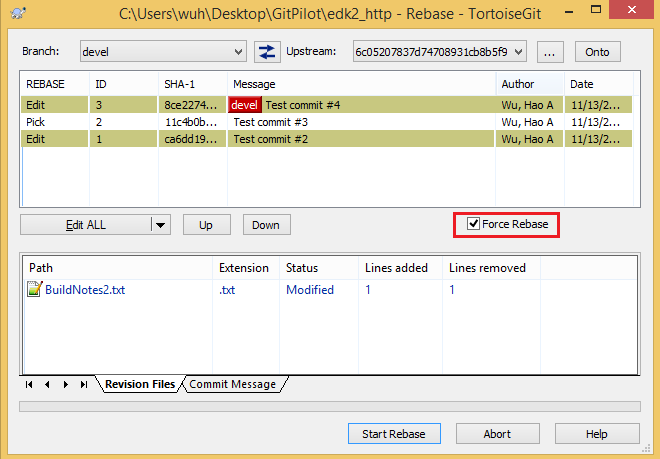
If only the last commit’s message is about to be modified, you can right-click in your repository – Git Commit -> “devel”… – Check the “Amend Last Commit” checkbox – Modify the commit message – OK.



**Figure 28. Modify the message of the last local commit using TortoiseGit**

If multiple commits are to be changed, right-click in your repository – TortoiseGit – Show log – Right-click at the earliest commit you want to change – Rebase “devel” on to this – Check the “Force Rebase” checkbox

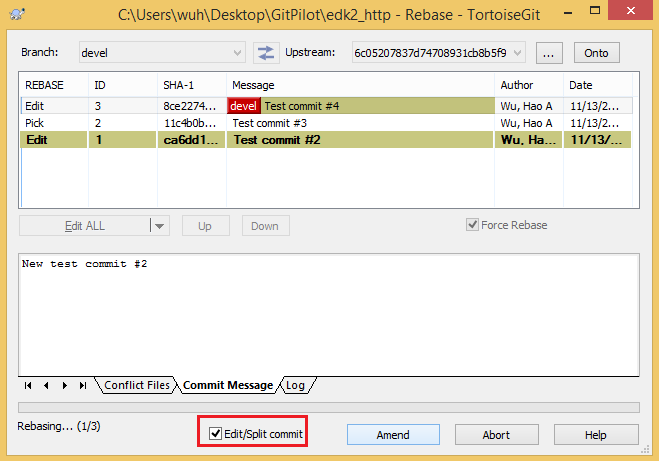


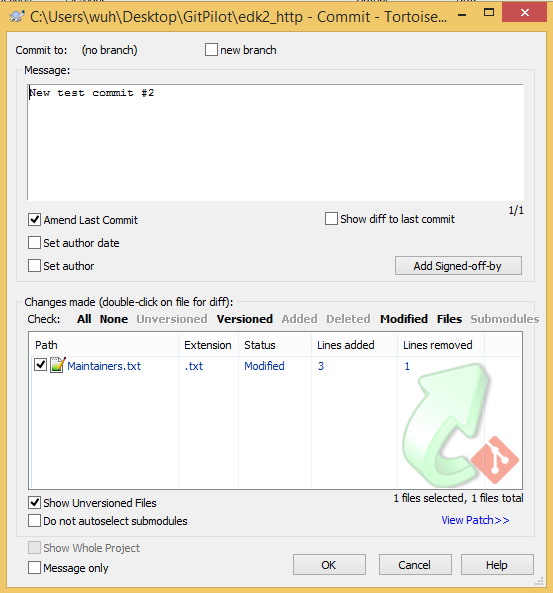


**Figure 29. Modify the messages of several local commits using TortoiseGit**

From the popup dialog box, select “Edit” for commits that need to be modified and click the “Start Rebase” button.

The rebase process will stop at each commit that keyword “Edit” is chosen for it, and wait for the user to modify the commit message. If the content of the commit needs to be changed as, one can just modify at the working tree during the rebase process. Once changes have been made, select the “Edit/Split commit” checkbox shown in **Figure 30** and then click “Amend” button. One can modify the commit message if needed and click “OK” to continue the rebase process.

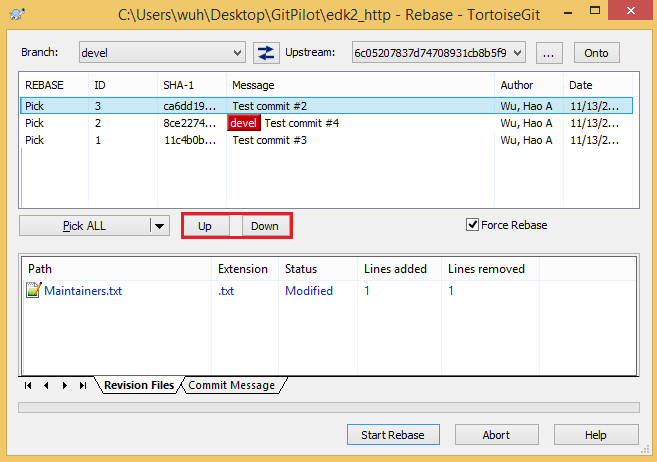




**Figure 30. Modify the content of several local commits using TortoiseGit**

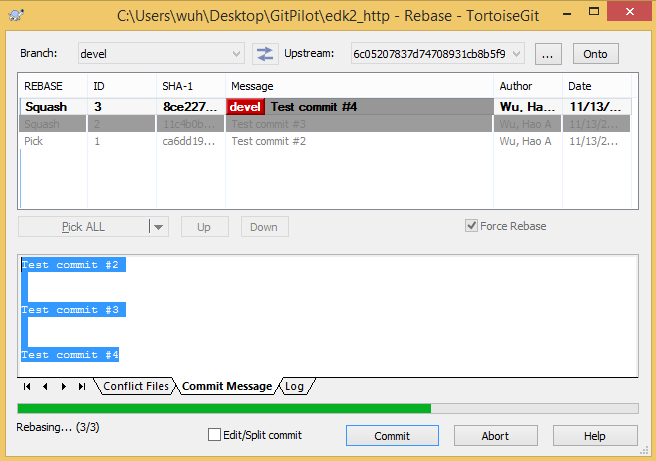
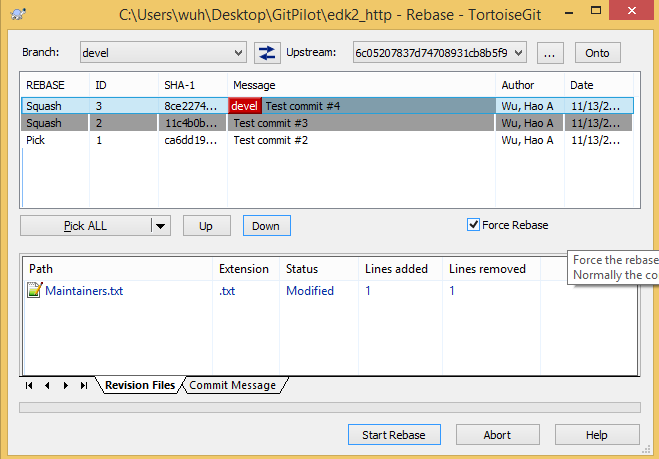
The whole rebase process is done after all the commits that marked with keyword “Edit” is amended.

If one wants change the order of some commits, one can follow the steps shown in **Figure 29** first. Then in the following pop out window, one can choose a commit press “Up” or “Down” to change the order.



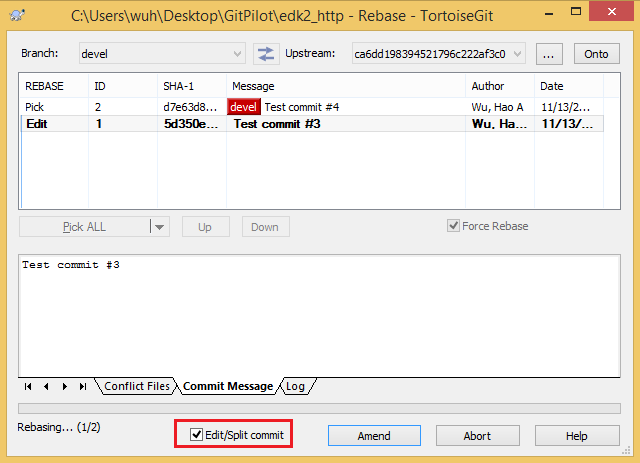
**Figure 31. Modify the order of several local commits using TortoiseGit**

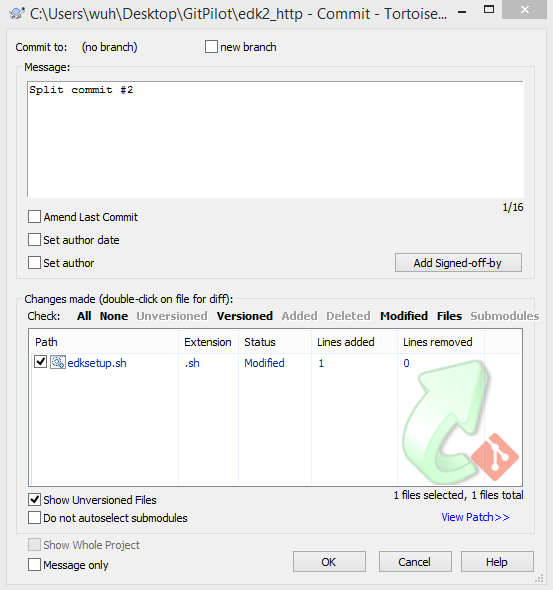
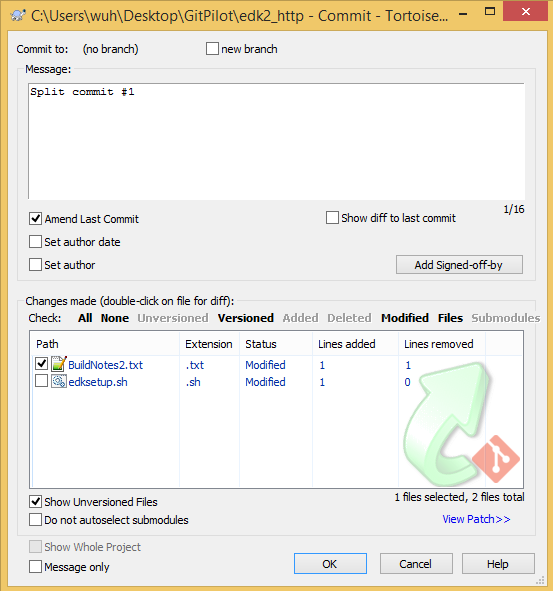
If one wants to combine several adjacent commits, one can select the “Squash” keyword for the commit. Choosing “Squash” for one commit will combine the commit itself with the previous one. Combining more than two commits requires choosing “Squash” for all the commits except the oldest one. The rebase process will stop at the last commit that will be combined and all the commit messages will be appended after each other for user’s further modification.



**Figure 32. Modify the order of several local commits using TortoiseGit**

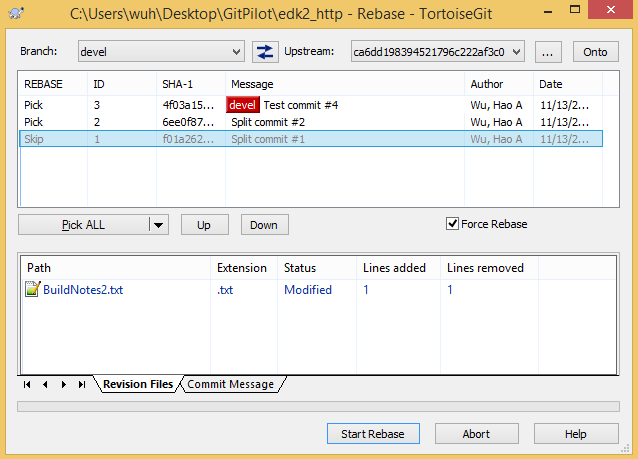
If one wants to split one local commit into smaller ones, one can select the “Edit” keyword for the commit. When the rebase process stops at the commit, check “Edit/Split commit” and press “Amend”. Then one can choose part of the files originally contained in this commit and write a new commit message for the first split. After the first split one is committed, a new window containing the remaining file will pop out automatically, one can repeat the process until all the files have been included in a commit.





**Figure 33. Split local commits using TortoiseGit**

If one wants to delete a local commit, one can choose the “Skip” keyword for the commit during rebase process.



**Figure 34. Delete local commits using TortoiseGit**

#### Push Changes

When local changes pass review process, module/package owners will help developers to push the changes to GitHub. Therefore, developers do not need to push their commits to the EDKII GitHub remote.

#### Delete Local Branch

After local changes have been pushed to the Git server, one may delete the “devel” branch since it has already been integrated to the “master” branch.

One should make sure that “devel” is not the current active branch, then it can be deleted via steps introduced in Section A.2.2.

## Maintainer Process

### Using Git

#### Clone/Update the Repository

If the maintainer does not have an EDKII repository, one can open the “Git Bash” command line and type one of the following commands to clone the EDKII Git repository:

git clone https://github.com/tianocore/edk2.git

git clone git@github.com:tianocore/edk2.git (require GitHub SSH key setup)

When the clone operation completes, a remote called “origin” will appear in the repository.

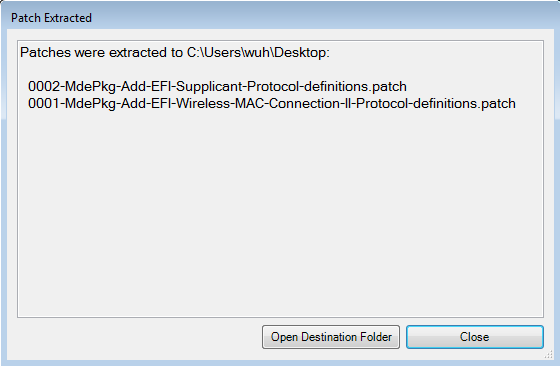
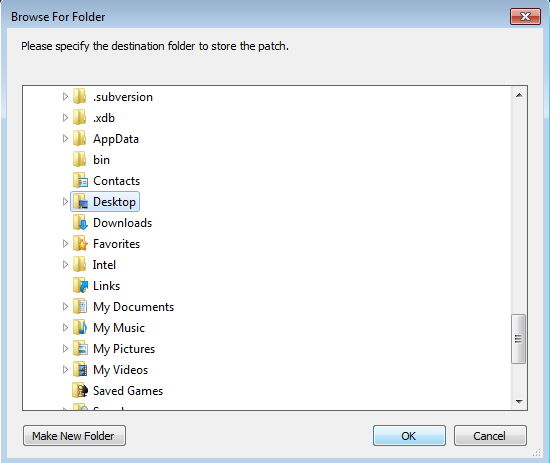
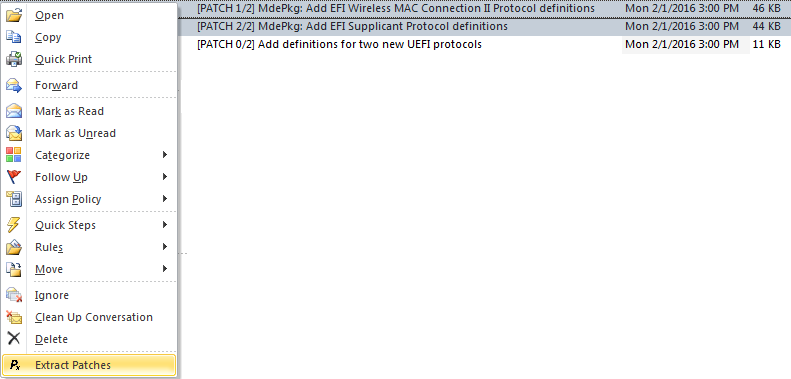
If the EDKII repository has been cloned before, maintainers should update the tree to the latest version via:

git checkout master

git pull (or git pull origin)

#### Generate Patch Files from Outlook

To review mail patches received by Outlook, maintainers should first get the “GitPatchExtractor” plugin setup file from: <https://ssvn.intel.com:80/ssg/csd/tiano/tianoad/Research/Developer/wuh/MailPatch/setup.exe>. Then one should close Outlook, run “setup.exe” to install the plugin. After the plugin is installed, one can open Outlook and right click any single or multiple patch mail(s), choose “Extract Patch” or “Extract Patches” (when multiple are selected).



**Figure 35. “GitPatchExtractor” Outlook plugin usage**

#### Create Branch for Review

After patch files are generated from review mails, maintainers can create local branch to review code changes. One can start the review process by creating a new local branch (In this section, branch “review” will be used as an example for the new local branch) based on the “master”:

git checkout –b review master

#### Apply Patches

Maintainers can copy all the patch files to the EDKII project root directory and use the following command to apply them to the local “review” branch:

git am --3way --ignore-space-change --keep-cr \*.patch

If the git-am command fails with conflicts, maintainers can optionally try to resolve them or they can ask the patch supplier to resolve the conflicts and send out review mails again.

If the git-am command succeeds, maintainers can start review the patches in detail. Please remember to delete the .patch files in the EDKII project root directory when the patches have been applied to the review branch.

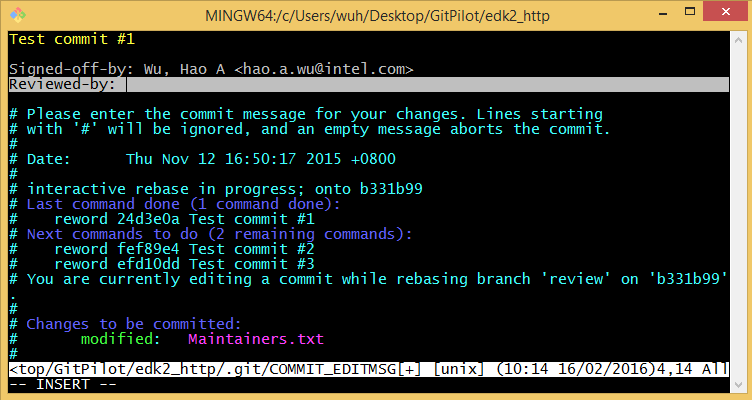
#### Add Reviewed-by Attributions

When the review process completes, maintainers should first add the “Reviewed-by” tag in each reviewed commit by using the command:

git rebase -i master

Next, a “vim” editor will pop out, showing the title of each reviewed commit, one should replace the keyword “pick” at the beginning of each line with “reword” (or simply “r”) to allow one to add the Reviewed-by attributions.





**Figure 36. Add “Reviewed-by” attributions**

#### Push changes

Now, reviewed patches can be pushed to remote. Before using the git-push command, one should go through the steps listed in Section 4.1.1.4 again to make sure the local “master” branch is align with the latest code on the server.

Next, one can integrate the commits at branch “review” into “master” by:

git checkout master

git merge --ff-only review

Once the merge process is done without any conflict, one can first use the git-push command introduced in Section A.1.11 with a “--dry-run” option to show exactly what is going to be pushed to remote server:

git push origin master --dry-run

After making sure the contents to be pushed are correct, one can then use the following command to upload changes to the server:

git push origin master

### Using TortoiseGit

#### Clone/Update the Repository

If the maintainer does not have an EDKII repository, one can follow the steps in Section 4.1.2.1 to download the repository. If the repository already exists, one can update it by first switching to the “master” branch via method introduced in Section A.2.2, then using the “Pull…” to get the latest code changes for “master” branch from the Git server shown in **Figure 25**.

#### Generate Patch Files from Outlook

The steps are the same as Section 4.2.1.2.

#### Create Branch for Review

Creating and switching branches via TortoiseGit is introduced in Section A.2.2. Please refer to those sections for more details.

Local branch “review” will be used in this section as a demonstration local branch name during develop process.

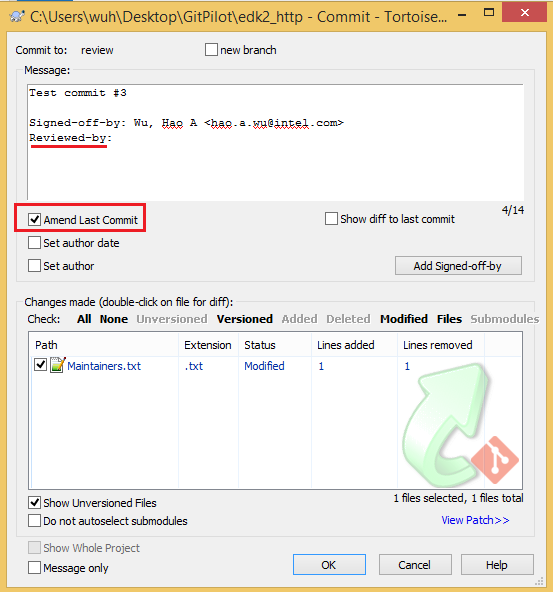
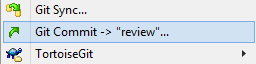
#### Apply Patches

One can follow the instructions in Section A.2.1 to apply patches to the “review” branch.

#### Add Reviewed-by Attributions

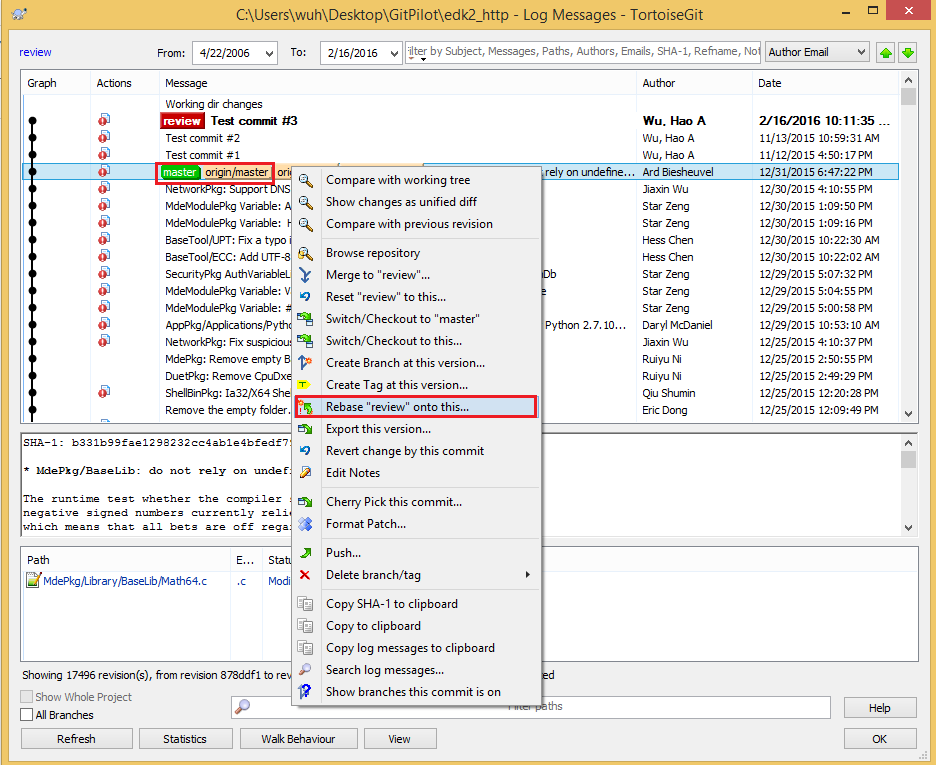
When the review process completes, maintainers should first add the “Reviewed-by” tag in each reviewed commit.

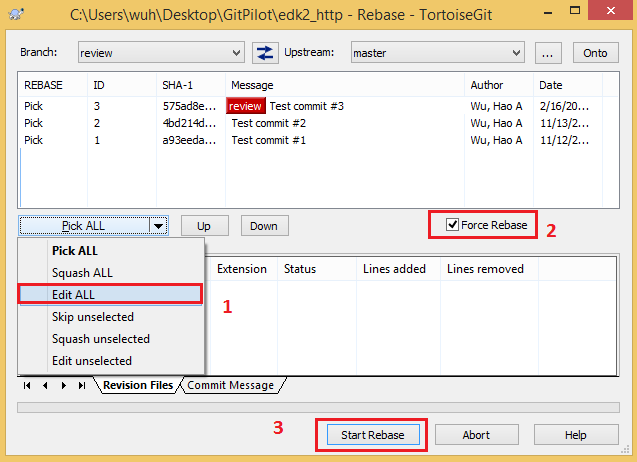
If only there is only one commit gets reviewed, one can right-click in your repository – Git Commit -> “review”… – Check the “Amend Last Commit” checkbox – Add the “Reviewed-by” tag – OK.

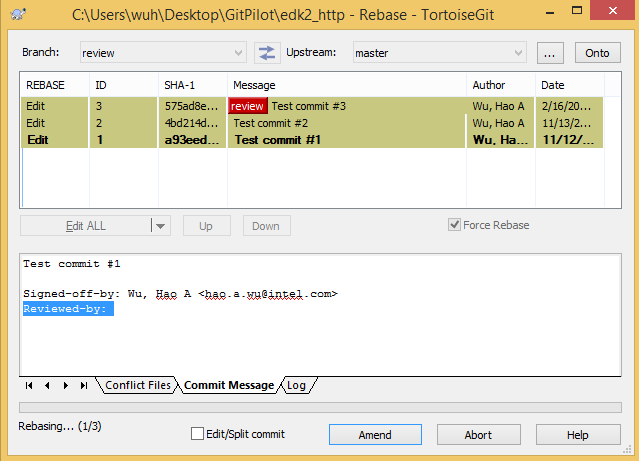


**Figure 37. Add “Reviewed-by” to one commit using TortoiseGit**

If the “Reviewed-by” tag is going to be added to multiple commits, right-click in your repository – TortoiseGit – Show log – Right-click at the commit where “master” is at – Rebase “review” on to this – Check the “Force Rebase” checkbox







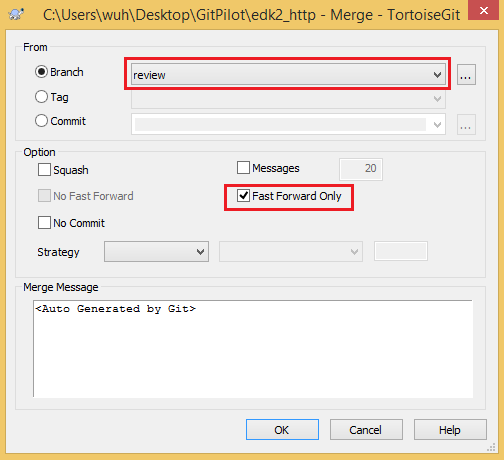
**Figure 38. Add “Reviewed-by” to multiple commits using TortoiseGit**

From the popup dialog box, select “Edit All” and check the “Force Rebase” checkbox. Then click the “Start Rebase” button. The rebase process will stop at each commit, thus, one can add the “Reviewed-by” tag. When finished, click “Amend” to process the next commit until all the reviewed commits are added with the “Reviewed-by” tags.

#### Push changes

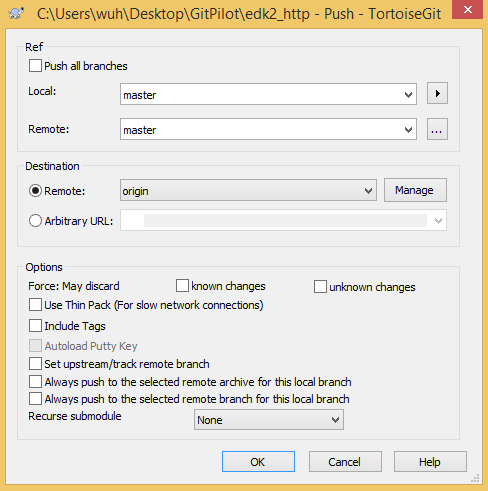
When local changes pass review process, they can be pushed to remote. Before using the push operation, one should go through the steps listed in Section 4.1.2.4 again to make sure the local changes are based upon latest code on remote.

Next, one can integrate the commits at “review” into master. One needs to checkout to master branch first, and then run the “Merge” command introduced in Section A.2.10. In the pop out window, choose “review” branch and check the “Fast Forward Only” option.



**Figure 39. Integrate change to “master” for push operation using TortoiseGit**

Once the merge process is done without any conflict, one can use the push operation introduced in Section A.2.11 to upload changes to the remote server.



**Figure 40. Push local “master” branch to remote “master” branch at “origin”**

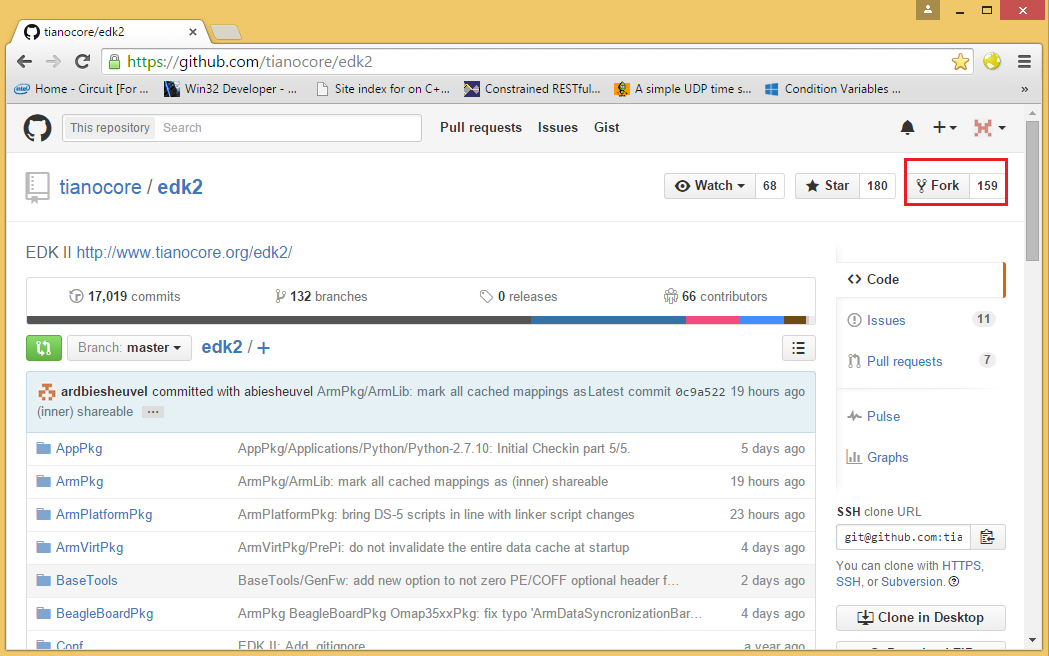
## GitHub – Using “Fork”

“Fork” is a feature supported by GitHub. A fork is a copy of a repository. Forking a repository allows one to freely experiment with changes without affecting the original project. Therefore, one can do all the works at one’s own copy and give reviewers a URL of his/her own repository plus a branch name for code review.

GitHub provides some useful articles on the “Fork” feature. One can visit <https://help.github.com/articles/fork-a-repo/> and <https://help.github.com/articles/syncing-a-fork/> for more details.

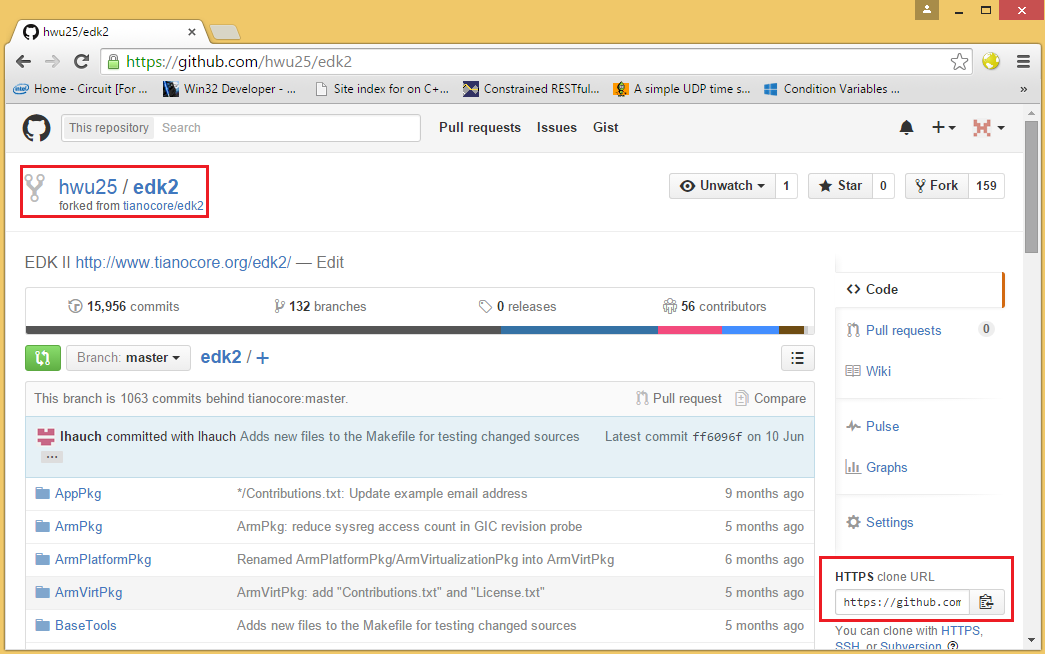
In this section, develop process based on the fork feature will be introduced. Since the open-source EDKII project locates at GitHub, the following content will use the EDKII project as demonstration repository.

To fork the EDKII project at GitHub, one need to follow the steps introduced in Section 3.1. When the preparations are done, one can go to the EDKII project at GitHub at <https://github.com/tianocore/edk2>. Usually, this original repository is called the upstream repository. Next, click the “Fork” button at the top-right corner.



**Figure 41. Open-source EDKII project at GitHub**

Once done, one will have a copy of the EDKII repository. Also a new URL for one’s own repository will be given:



**Figure 42. One’s own fork of the EDKII project**

### Using Git

#### Add Remote for the Fork

After forking one’s own copy of the EDKII repository at GitHub, a new remote pointing to the forked repository can be added to the local Git repository (cloned from the upstream repository). In this section, the new remote will be called “myfork”, one can add remote “myfork” by:

git remote add myfork https://github.com/<username@GitHub>/edk2.git

Once the new remote has been added, one can get the codes from the forked repository by:

git fetch myfork

It worth pointing out that the git-fetch command will not create/change any local branch, it will only create/change remote-tracking branches (local branches that cannot be moved).

#### Keep Remotes Aligned

By default, our local “master” branch is linked with the remote “master” branch on the remote “origin”. Similarly, such a local branch should be created for remote “master” branch for remote “myfork”:

git checkout –b fork\_master myfork/master

After executing the above command, a local branch called “fork\_master” will be created.

Before one can make changes based on “fork\_master”, one may update it to keep align with the latest changes on the upstream remote (orgin). One can do so by (please make sure that “fork\_master” do not have any local commits):

git checkout master

git pull origin

git checkout fork\_master

git merge master --ff-only

git push myfork HEAD:master

Now, one can do some development duties based on “fork\_master”.

#### Create Branch for Development

It is nearly the same process as described in Section 4.1.1.2, the only difference is that the starting point is from “fork\_master”.

#### Commit Local Changes

It is nearly the same process as described in Section 4.1.1.3, the only difference is that the starting point is from “fork\_master”.

#### Push to Fork

Assume that one has made changes at local branch “fork\_devel” and ready for review, he/she should push this local branch to remote “myfork” by:

git push myfork

Or if a different remote branch name is desired:

git push myfork HEAD:<specified\_remote\_branch\_name>

When the git-push command completes, a new remote branch called “fork\_devel” (if no new specified name is given) will be created at remote “myfork”.

Next, one can send out a review request email to the reviewers including a URL of the forked repository and a remote branch name where the changes are pushed.

#### Review the Fork

For reviewers, they will receive a remote URL and a remote branch name to review. They can integrate the changes into their local repository by:

git remote add fork\_review <URL\_given>

git fetch fork\_review

git checkout –b <new\_branch\_name> fork\_review/<remote\_branch\_name\_given>

At this point, the reviewer will have a new local branch which is exactly the same as the “fork\_devel” branch in the developer’s repository. Further integrations might be done by the reviewer to test/review the commits.

#### Refine the Fork

Normally, changes made by developers need be refined based on review feedbacks. When it comes with fork, a good way to provide a new version of changes is to create a new remote branch (e.g. “fork\_devel\_V2”) at the forked repository, which requires repeating the steps in Section 4.3.1.3 to Section 4.3.1.6.

#### Push to Upstream

One should integrate the latest changes from the upstream remote before check-in local commits. First, the two masters “master” and “fork\_master” should get aligned by steps in Section 4.3.1.2. Then one can follow the below steps to push changes to the upstream remote:

git rebase master fork\_devel

git checkout master

git merge --ff-only fork-devel

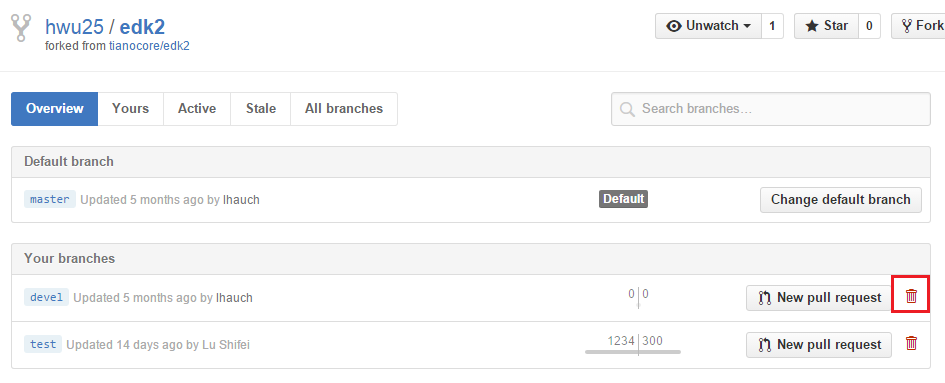
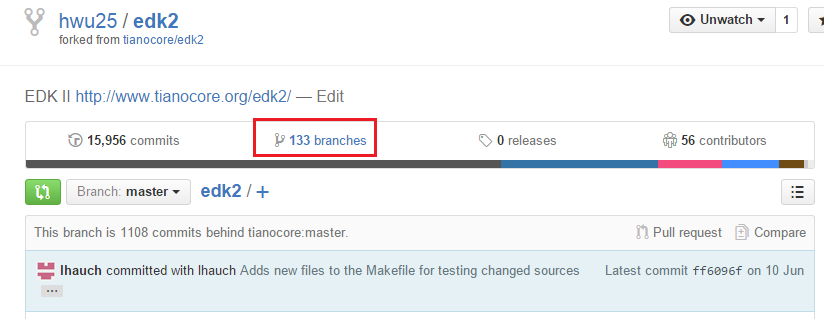
git push origin

#### Delete Branch at Fork Remote

After changes have been pushed to the upstream repository, the remote branch at forked remote can be deleted by:

git push myfork --delete <remote\_branch\_name>

One can also delete a remote branch via web interface:



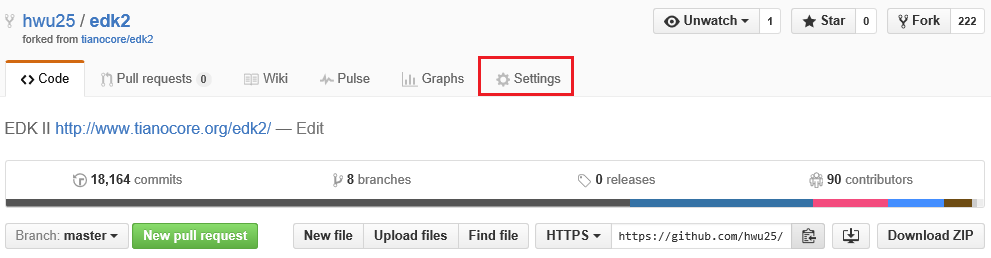
**Figure 43. Delete remote branches via the web interface**

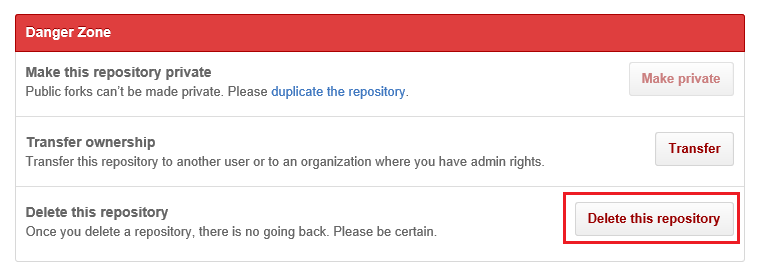
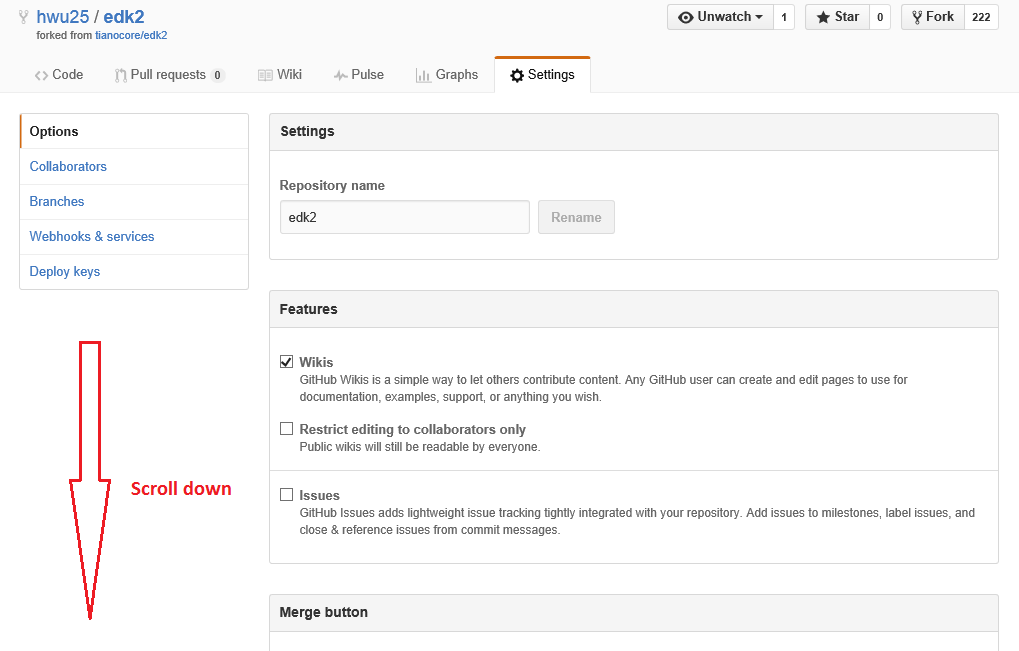
#### Delete the Fork

The EDK2 fork repository under one’s GitHub account can be deleted by first running:

git remote remove myfork

Then one should go to the EDK2 fork repository page at GitHub and follow the steps shown in **Figure 44** to delete the fork repository.



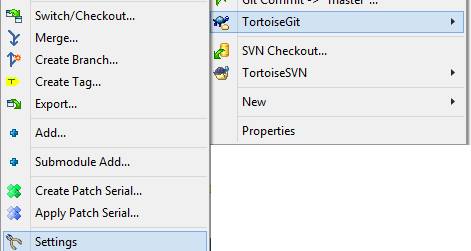


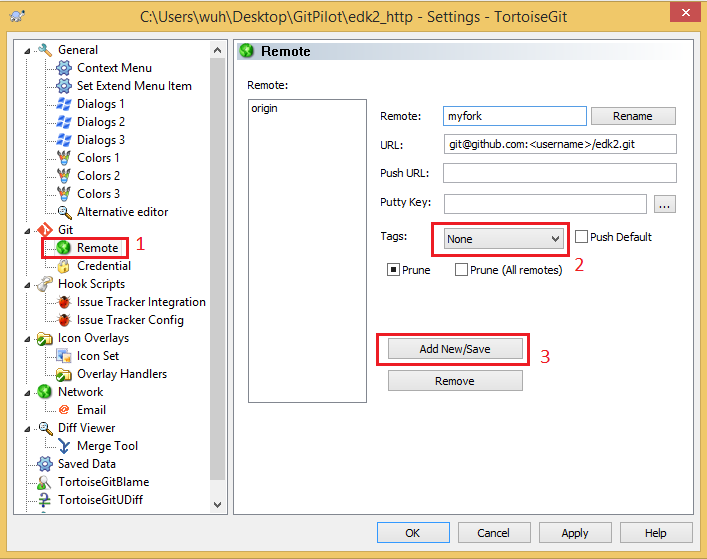
**Figure 44. Delete the EDK2 fork repository**

### Using TortoiseGit

#### Add Remote for the Fork

After forking one’s own copy of the EDKII repository at GitHub, a new remote pointing to the forked repository can be added to the local Git repository (cloned from the upstream repository). In this section, the new remote will be called “myfork”, one can right-click in your repository – TortoiseGit – Settings – Select the “Remote” tab on the left-hand side to fill in the information of the forked repository.





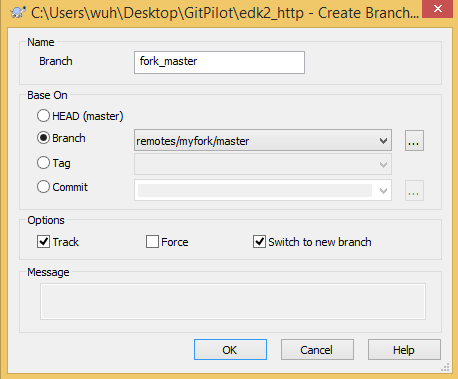
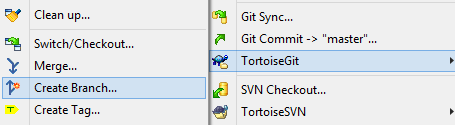
**Figure 45. Add the fork remote using TortoiseGit**

Then click “Yes” in the pop out windows to fetch the codes from the forked repository.

It worth pointing out that the fetch operation will not create/change any local branch, it will only create/change remote-tracking branches (local branches that cannot be moved).

#### Keep Remotes Aligned

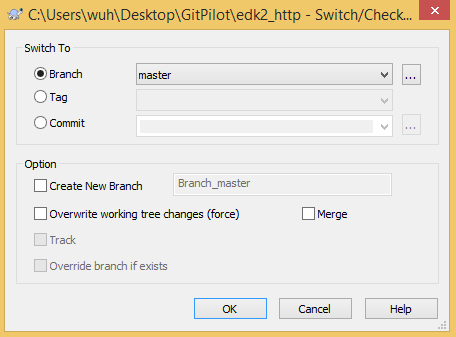
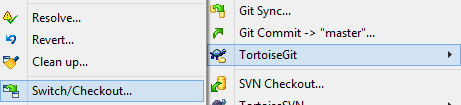
By default, our local “master” branch is linked with the remote “master” branch on the remote “origin”. Similarly, such a local branch should be created for remote “master” branch for remote “myfork”:

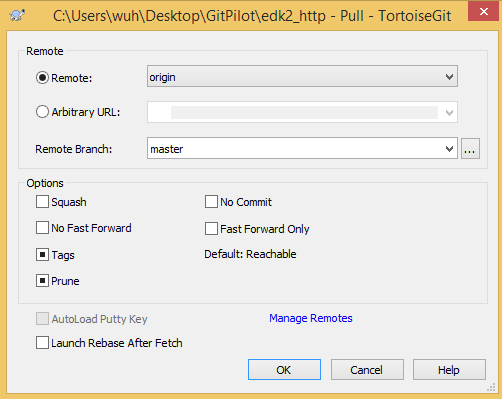
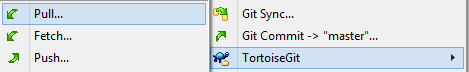


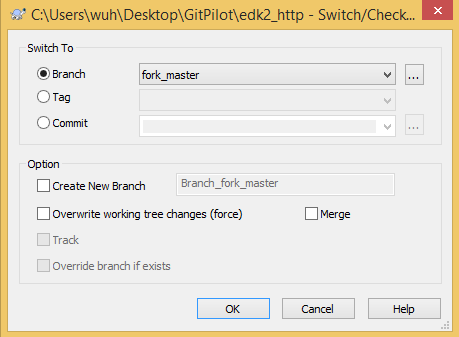
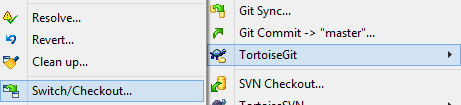
**Figure 46. Create the “fork\_master” branch using TortoiseGit**

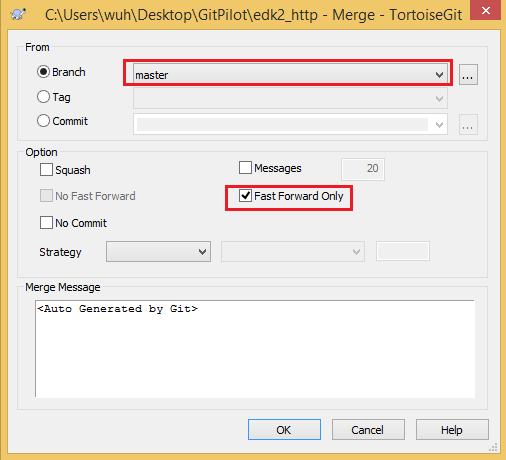
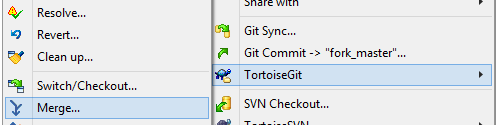
After executing the above operations, a local branch called “fork\_master” will be created.

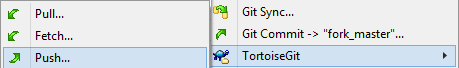
Before one can make changes based on “fork\_master”, one may update it to keep align with the latest changes on the upstream remote (orgin). One can do so by (please make sure that “fork\_master” do not have any local commits):











**Figure 47. Keep remotes aligned using TortoiseGit**

Now, one can do some development duties based on “fork\_master”.

#### Create Branch for Development

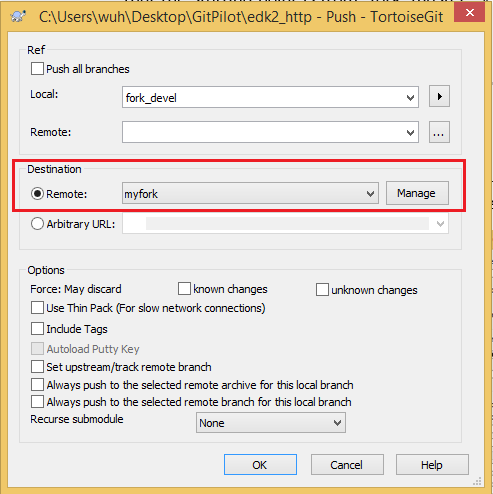
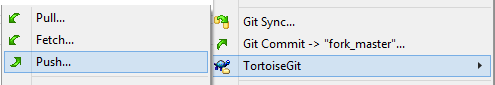
It is nearly the same process as described in Section 4.1.2.2, the only difference is that the starting point is from “fork\_master”.

#### Commit Local Changes

It is nearly the same process as described in Section 4.1.2.3, the only difference is that the starting point is from “fork\_master”.

#### Push to Fork

Assume that one has made changes at local branch “fork\_devel” and ready for review, he/she should push this local branch to remote “myfork” by:



**Figure 48. Push changes to the forked repository using TortoiseGit**

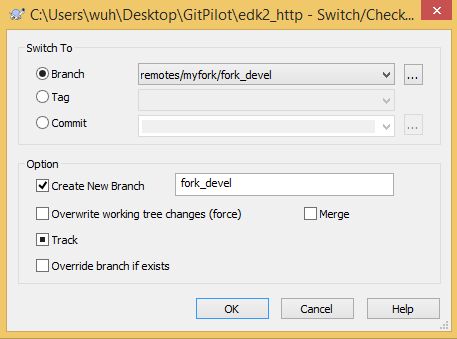
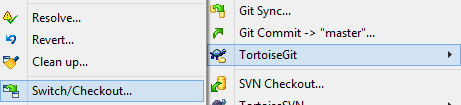
If a different remote branch name (rather than “fork\_devel”) is desired, one can fill the “Remote:” field under the “Ref” section with a specified remote branch name.

When the push operation completes, a new remote branch called “fork\_devel” (if no new specified name is given) will be created at remote “myfork”.

Next, one can send out a review request email to the reviewers including a URL of the forked repository and a remote branch name where the changes are pushed.

#### Review the Fork

For reviewers, they will receive a remote URL and a remote branch name to review. First, they need to follow the steps shown in Section 4.3.2.1 to add a new remote (the remote name can be different than “myfork”) to their local repository. Then, they can create a local branch based on the information provided by developers.



**Figure 49. Create a local branch for review using TortoiseGit**

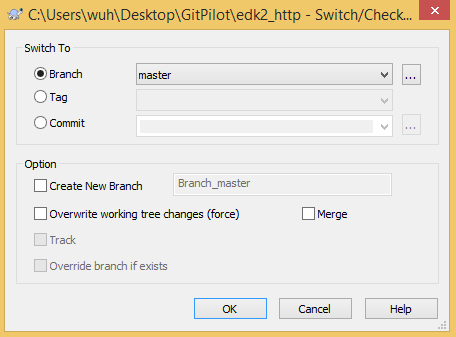
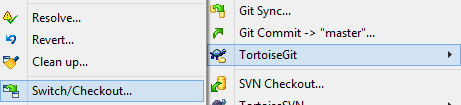
At this point, the reviewer will have a new local branch which is exactly the same as the “fork\_devel” branch in the developer’s repository. Further integrations might be done by the reviewer to test/review the commits.

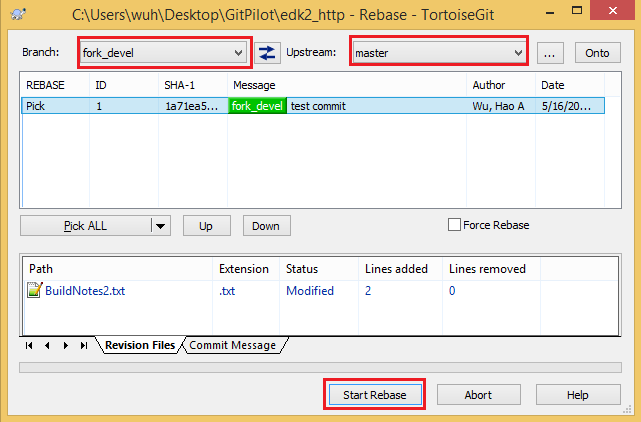
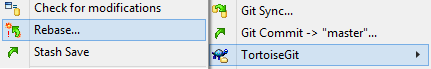
#### Refine the Fork

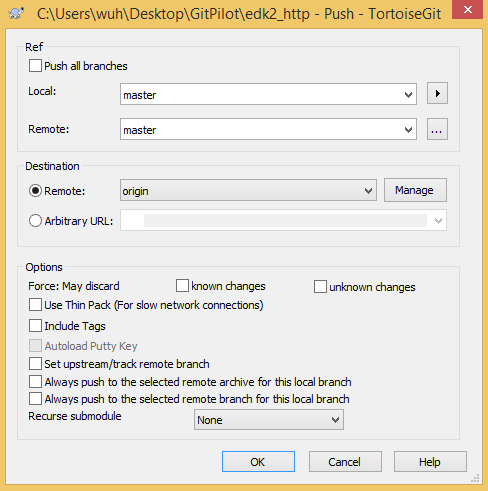
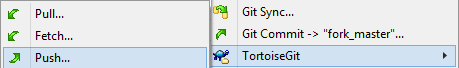
Normally, changes made by developers need be refined based on review feedbacks. When it comes with fork, a good way to provide a new version of changes is to create a new remote branch (e.g. “fork\_devel\_V2”) at the forked repository, which requires repeating the steps in Section 4.3.2.3 to Section 4.3.2.6.

#### Push to Upstream

One should integrate the latest changes from the upstream remote before check-in local commits. First, the two masters “master” and “fork\_master” should get aligned by steps in Section 4.3.2.2. Then one can follow the below steps to push changes to the upstream remote:







**Figure 50. Push commits to upstream using TortoiseGit**

#### Delete Development Branch

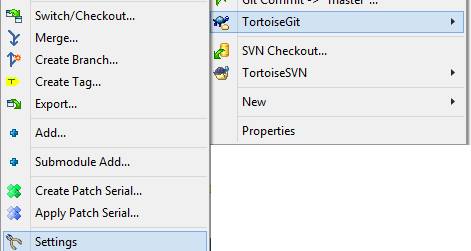
After local changes have been pushed to the upstream Git server, one may delete the local “fork\_devel” branch since it has already been integrated to the “master” branch.

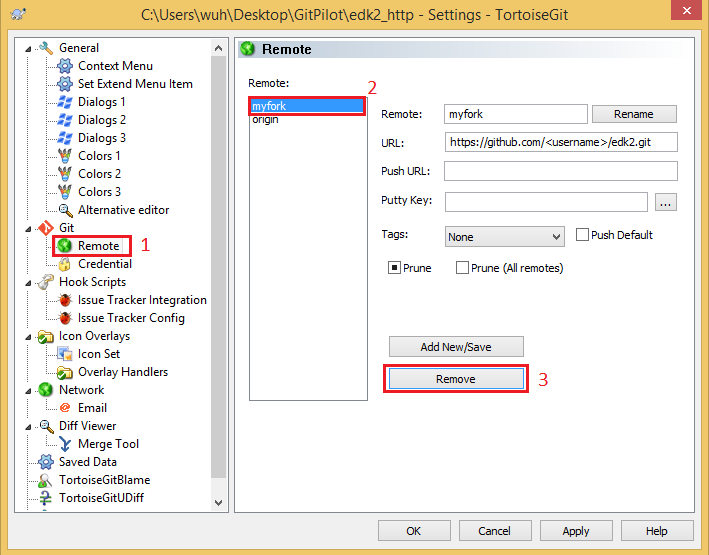
One should make sure that “fork\_devel” is not the current active branch, then it can be deleted via steps introduced in Section A.2.2.

Moreover, one can delete the “fork\_devel” branch at “myfork” remote via web interface introduced in Seciton 4.3.1.9.

#### Delete the Fork

The EDK2 fork repository under one’s GitHub account can be deleted locally first by:





**Figure 51. Delete a remote locally using TortoiseGit**

Then one should go to the EDK2 fork repository page at GitHub and follow the steps shown in **Figure 44** to delete the fork repository.

## Useful tips

### Git EDKII and SVN R9/BP mixed development

Several scripts (available at <https://ssvn.intel.com:80/ssg/csd/tiano/tianoad/Research/Developer/wuh/Scripts>) can be used to setup a develop environment. Combos supported includes: EDKII(Git), EDKII(Git)+R9(SVN) and EDKII(Git)+R9(SVN)+BP(SVN).

These scripts also help to configure the required environments to build EDKII or R9/BP platform. Developer needs to use the different way to update those source codes to the latest version.

### EDKII and R9/BP multiple repos development

Open-source EDKII, closed-source BP and R9 are separate repositories. Patches consisting changes from multiple repositories cannot be created using git-format-patch command. Therefore, when developing in a multi-repository environment, developers only need to make sure two things: 1) Sending the patches to the correct mailing list; 2) Apply the patches to the correct repository.

# Git Commands

This chapter will introduce some basic Git commands and corresponding TortoiseGit operations frequently used during development work. It uses EDK2 project as the example to explain every command. If you want to try those commands based on the instruction, you need to read Chapter 4 first to apply the access for EDK2 project. For the detail command usage during development process, please refer to Chapter 4.

* 1. Git Commands

This section will show the basic usage of some Git commands which are used frequently. Default “Git Bash” command line interface is used and the commands will be demonstrated in the open-source EDKII Git repository as an example.

For detailed usage, one can add “--help” parameter after each command.

* + 1. Am

The git-am command is used to apply a series of patch files to the Git repository.

If one wants to apply a bunch of “.patch” files to a Git repository, one can:

git am --ignote-whitespace path/to/1.patch path/to/2.patch ...

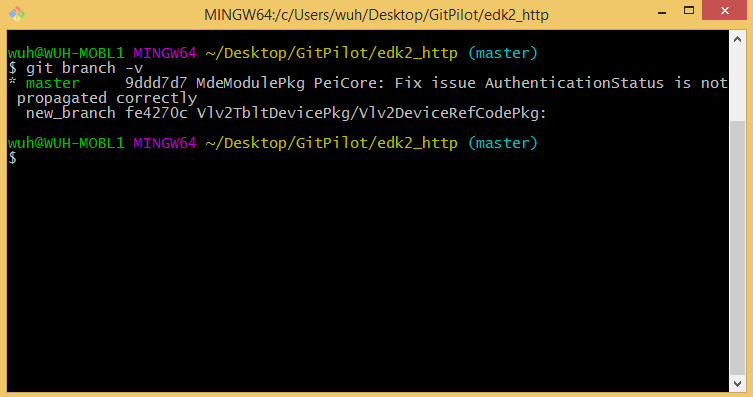
* + 1. Branch & Tag

The git-branch command allows users to list, create, or delete branches.

One can see all the local branches and which branch the user is currently at in a Git repository by:

git branch -v

One will get the following output if there are multiple local branches in the Git repository:



**Figure 52. Local branches information**

One can see all the branches (both local and remote) in a Git repository by:

git branch -av

One can rename an existing local branch by:

git branch –m <old\_name> <new\_name>

One can delete a not active (not the branch users currently at) local branch by:

git branch –d <branch\_name>

One might get the message “error: The branch 'XXX' is not fully merged.” It is caused by the deleting branch has one or more commits that not exist in the active branch. One can force Git to delete it by:

git branch –D <branch\_name>

The git-checkout command allows users to switch among different branches:

git checkout <branch\_name>

One can also create a new branch and checkout and switch to the new branch that is based on currently active branch:

git checkout –b <new\_name>

Like most version control systems, Git has the ability to tag specific points in history as being important. Typically people use this functionality to mark release points (v1.0, and so on).

The git-tag command can create, list, or delete a tag object.

One can create a tag for the current state of the codes by:

git tag <tag\_name>

One can list all the tags in the Git repository by:

git tag -l

One can delete an existing tag by:

git tag -d <tag\_name>

* + 1. Cherry-pick

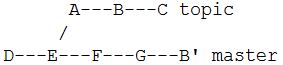
Given one or more existing commits, the git-cherry-pick command will apply the change each one introduces, recording a new commit for each on the current local branch. It is also a way to integrate changes from one branch into another.

Assume the repository history shown by **Figure 55** and the current branch is “master”.

After running the following command:

git cherry-pick <sha1\_for\_B>

The history will look like this:



**Figure 53. Repository structure after git-cherry-pick**

The new commit B’ (if no conflict during cherry-pick) will have the same content as commit B at “topic” branch. However, Their SHA1 values are different, so they are two separate commits.

One can also pick multiple commits for the git-cherry-pick command, the following commands will do the same thing that pick commit A, B and C from “topic” and create relative commit A’, B’ and C’ at “master”:

git cherry-pick A..C

git cherry-pick A B C

* + 1. Clone

To clone a Git repository, type

git clone https://github.com/tianocore/edk2.git

When the git-clone command completes, a directory called “edk2” will be in the current directory where one runs “Git Bash”. If connection failure, please check Proxy setting in .gitconfig file. The example is attached in Appendix B.

* + 1. Commit

After making some file changes in the working tree, you can commit those changes locally (to the staging area) via git-commit command.

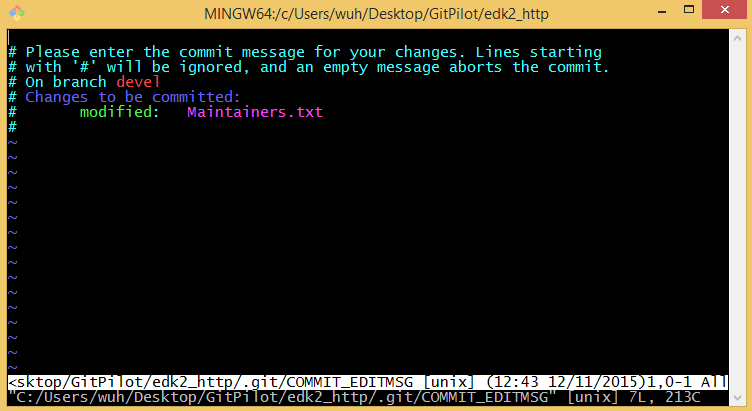
One can commit local changed files by:

git add <file1> <file2> ...

git commit

If a directory is passed to the git-add command, all the changed files under that directory is included.

After running the git-commit command, an editor (“vim” by default, one can change the default editor according to Seciton 2.2.1.2) will pop out for users to write the commit message.



**Figure 54. Edit commit message using “vim”**

* + 1. Diff & Difftool

The git-diff command is capable of showing changes between commits, commit and working tree, etc:

One can show the difference between working tree and a commit or branch by:

git diff <sha1\_of\_commit>

git diff <branch\_name>

One can show the difference between a commit/branch and a commit/branch by:

git diff <sha1/branch\_name> <sha1/branch\_name>

If an external diff utility tool is set according to Section 2.2.1.2, one can view the differences via the tool by using git-difftool command with the same parameters as the git-diff command.

* + 1. Fetch & Pull

The git-fetch command is used to fetch branches and/or tags from one or more other repositories, along with the objects necessary to complete their histories. Remote-tracking branches are updated, local branches are not touched.

git fetch origin

The git-pull command incorporates changes from a remote repository into the current local branch. More precisely, git-pull runs git-fetch with the given parameters and calls another command git-merge (which will be introduced later) to merge the retrieved branch heads into the current branch. With --rebase, it runs git-rebase (also will be introduced later) instead of git-merge.

git pull origin (--rebase)

* + 1. Format-patch

When your local commit is ready for code review, git-format-patch command allows one to create patch files for chosen commits and to send out mails to reviewers.

By default, the subject of a single patch is “[PATCH] ” followed by the concatenation of lines from the commit message up to the first blank. When multiple patches are generated, the subject prefix will instead be “[PATCH n/m] ”.

If another subject prefix is to be used instead, one can add following parameter for git-format-patch command:

git format-patch --subject-prefix='PATCH v2' ...

Then the new subject prefix for generated patch files will be “[PATCH v2]”

If one wants to generate patches for the latest 5 commits, one can type:

git format-patch HEAD~5

If one wants to generate patches for commits after a specific commit (the specific commit is not included), one can use:

git format-patch <sha1\_value\_for\_the\_specific\_commit>

If one wants to generate patches for all the commits between two specified commits, one can use:

git format-patch <sha1\_value1>..<sha2\_value2>

One should make sure that the commit represented by <sha1\_value1> is earlier in history than the commit specified by <sha2\_value2>. Also, the <sha1\_value1> commit will not be included, if one needs to include the <sha1\_value1> commit as well, one can use:

git format-patch <sha1\_value1>~1..<sha2\_value2>

* + 1. Log

The git-log command allows users to see the commit history of a branch.

The default behavior of git-log will show a detailed information of each commit including commit hash value, author of the commit, date the commit is pushed and commit messages.

The below command will summarize each commit into one line which might be useful in some cases.

git log --oneline

The below command will summarize each commit into one line and limit the number of commits listed on screen as well.

git log --oneline –n <number>

The output of the git-log can be formatted according to different requirements. One can see the user manual for git-log by:

git log --help

* + 1. Merge

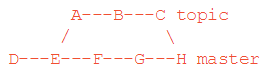
The git-merge command is used to join two or more development histories together. It is one way to integrate changes from one branch into another.

Assume the following history exists and the current branch is “master”:



**Figure 55. Repository structure before git-merge**

Then the command git merge topic will replay the changes made on the topic branch since it diverged from master (i.e., E) until its current commit (C) on top of master, and record the result in a new commit along with the names of the two parent commits and a log message from the user describing the changes.



**Figure 56. Repository structure after git-merge**

* + 1. Push

One can push local reviewed commits to the remote by the git-push command.

If no parameter is given:

git push

The branch users currently at (e.g. branch “master”) will be pushed to remote “origin”. If remote “origin” does not have a branch called “master”, it will be created remotely. Otherwise, the remote “master” will (if no conflicts) get updated and keep align with the local “master” branch.

If one wants to push to a specified remote, one should use:

git push <remote\_name>

If one wants to push the current local branch to a remote branch whose name is specified by command line, one can use:

git push <remote\_name> HEAD:<remote\_branch\_name>

If one wants to push a specified local branch to a specified remote branch, one can use:

git push <remote\_name> <local\_branch\_name>:<remote\_branch\_name>

* + 1. Rebase

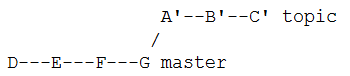
The git-rebase command is the third way to integrate changes between two branches. It works by going to the common ancestor of the two branches (the one you’re on and the one you’re rebasing onto), getting the diff introduced by each commit of the branch you’re on, saving those diffs to temporary files, resetting the current branch to the same commit as the branch you are rebasing onto, and finally applying each change in turn.

Assume the repository history shown by **Figure 55** and the current branch is “topic”.

After running one of the two following commands, where “master” is the upstream branch (the branch rebasing onto):

git rebase master

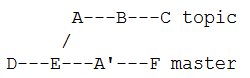
The result will be:



**Figure 57. Repository structure after git-rebase onto “master”**

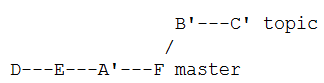
If the current branch is not “topic”, the command git rebase master topic will do the same thing.

If the upstream branch (“master” here) already contains a change you have made (e.g. because you mailed a patch which was applied upstream), then that commit will be skipped. For example, running git rebase master on the following history (in which A' and A introduce the same set of changes, but have different committer information):



**Figure 58. Repository structure that two branches share a common change**

will result in:



**Figure 59. Repository structure after git-rebase onto “master” (2)**

* + 1. Remote

The git-remote command manages the set of repositories ("remotes") whose branches you track.

One can view the information of the remotes in a Git repository by:

git remote -v

One may get the following output shown by **Figure 60**.



**Figure 60. Remotes information after git-clone**

The remote “origin” is automatically added to Git repository after the execution of git-clone command.

One can add a new remote by:

git remote add <new\_remote\_name> <URL>

One can rename an existing remote by:

git remote rename <old\_name> <new\_name>

One can delete an existing remote by:

git remote remove <remote\_name>

One can change the URL of a remote by:

git remote set-url <remote\_name> <new\_URL>

* + 1. Reset

The git-reset command is a versatile command with many configurations. It can be used to remove committed snapshots, although it’s more often used to undo changes in the staging area and the working directory. In either case, it should only be used to undo local changes—you should never reset snapshots that have been shared with other developers.

If one want to discard all the changes (not committed) in the working tree, one can:

git reset --hard HEAD

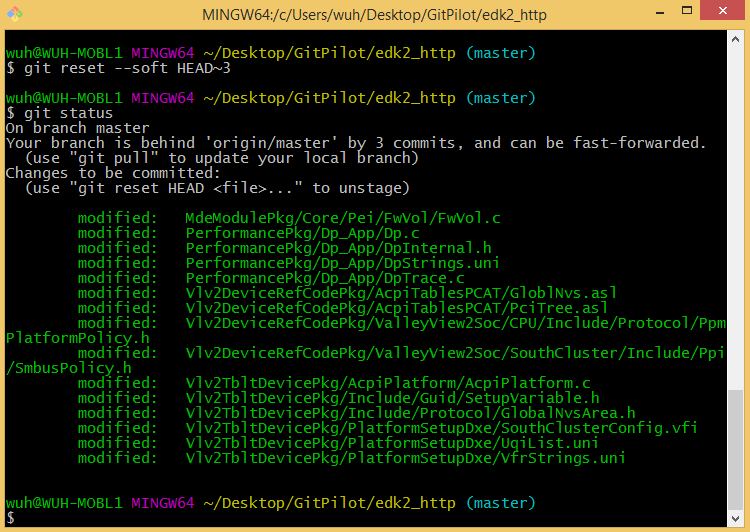
If one want to discard the latest three commits on the current branch, one can:

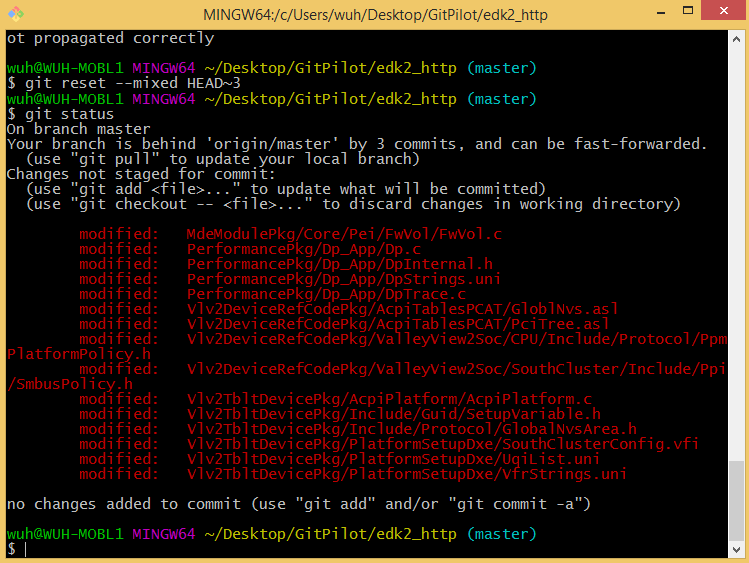
git reset --soft/--mixed/--hard HEAD~3

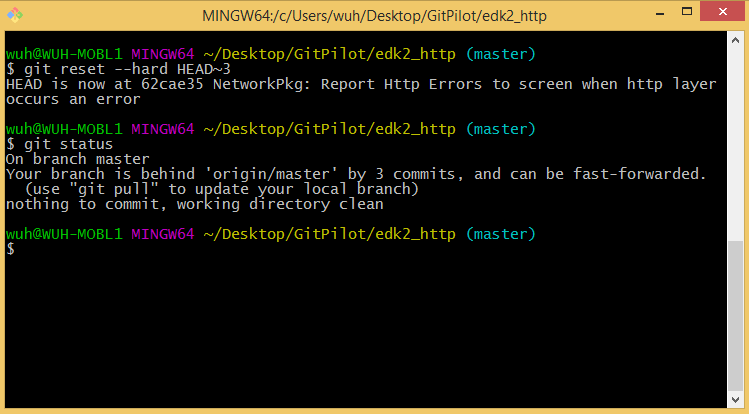
One can either select “soft”, “mixed” or “hard” to reset the current branch.

* Soft - Does not touch the index file or the working tree at all (but resets the head to <commit>, just like all modes do). This leaves all your changed files "Changes to be committed", as git status would put it.
* Mixed - Resets the index but not the working tree (i.e., the changed files are preserved but not marked for commit) and reports what has not been updated. This is the default action.
* Hard - Resets the index and working tree. Any changes to tracked files in the working tree since <commit> are discarded. (All changes introduced by commits after the selected commit are lost.)

Here’s the results of the git-reset command with the three parameters:







**Figure 61. Three modes of git-reset command**

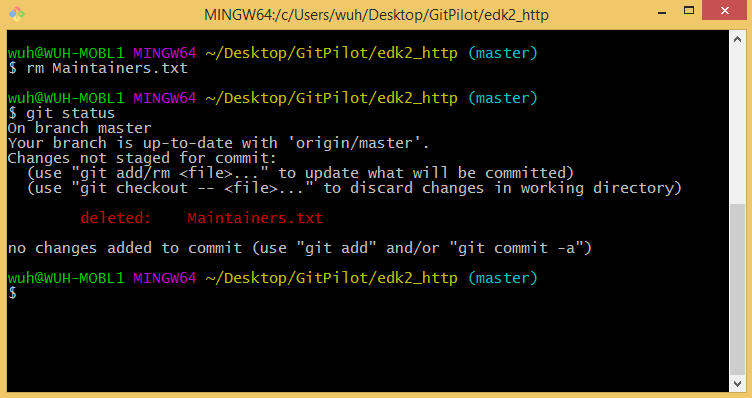
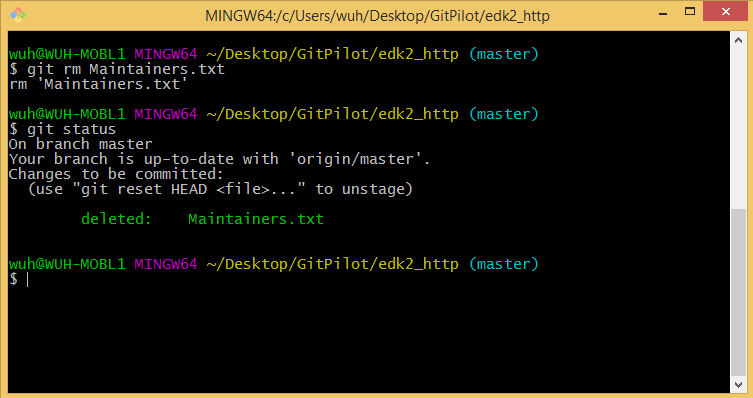
* + 1. Rm

The git-rm remove a file/directory from the staging area and from the working tree.

One can delete a file/directory from both the working tree and staging area of a Git repository by:

git rm <file\_name>

It is worth pointing out that simply deleting the files/directories in the file system only remove (“rm” command) them from the working tree, the staging area is not touched. **Figure 62** shows the difference:



**Figure 62. Difference between git-rm and rm**

* + 1. Send-email

The git-send-email command handles sending a collection of patches as emails.

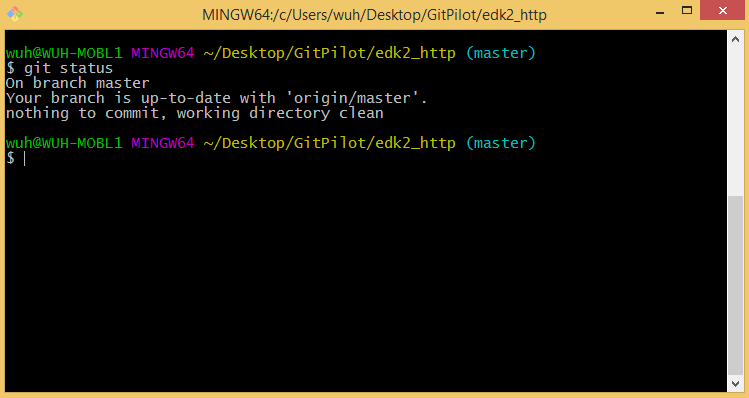
After generating a bunch of patch files one can send them out for review via:

git send-email path\to\1.patch path\to\2.patch --to person1@example.com --to person2@example.com --cc person3@example.com --cc person4@example.com

* + 1. Status

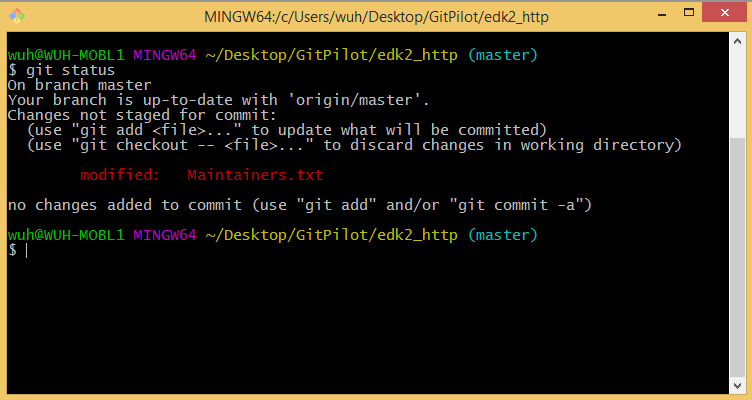
The git-status command displays paths that have differences between the index file and the current HEAD commit, paths that have differences between the working tree and the index file, and paths in the working tree that are not tracked by Git.

If the working tree is the same as the staging area (i.e. no change has been made to any file), git-status will show the following result:



**Figure 63. A clean working tree**

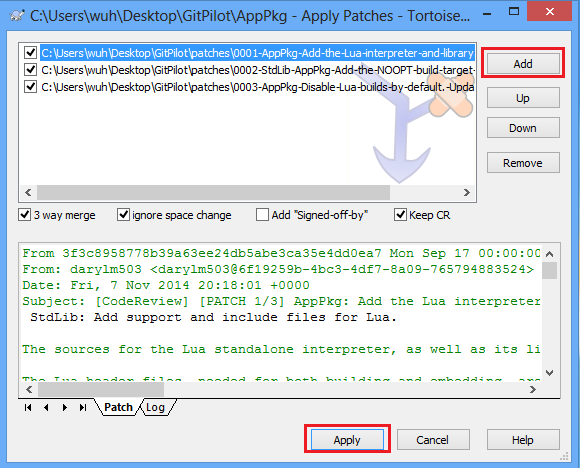
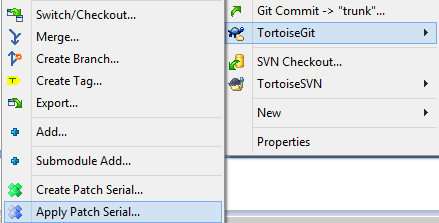
If changes have been made to files, the command will showing the below result:



**Figure 64. Status showing differences between working tree & staging area**

* 1. TortoiseGit Operations
     1. Apply patch

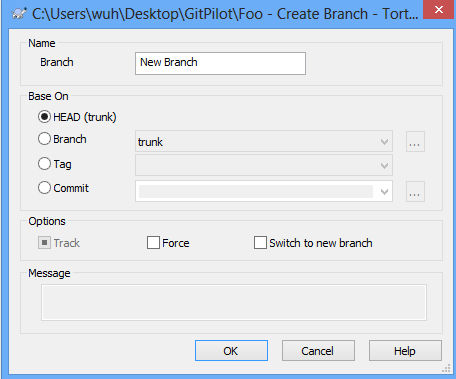
When reviewers get “.patch” files, they may want to apply the patch to their local tree for test purpose. One can apply patch files by right-click in the Git repository – TortoiseGit – Apply Patch Serial… – Add – Select the patches you want to apply – Apply.



**Figure 65. Apply “.patch” files to Git repository using TortoiseGit**

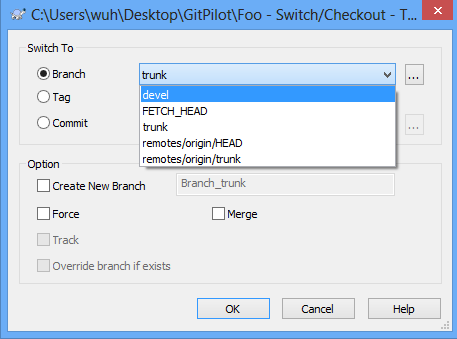
* + 1. Branch & Tag

To create a local branch, right-click at your local repository – TortoiseGit – Create Branch…. In the popup dialog, you can type a branch name, decide the base of the newly created branch and other options.



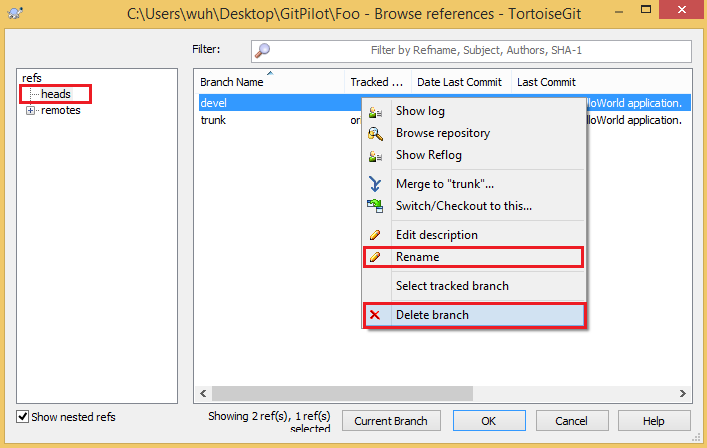
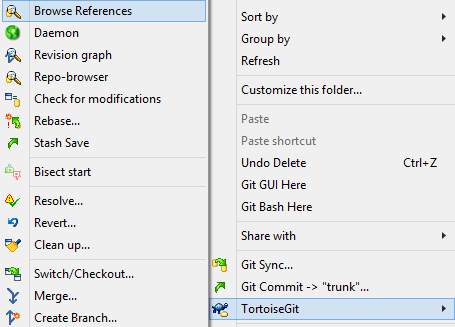
**Figure 66. Create a local branch using TortoiseGit**

To switch to an existing local branch, right-click at your local repository – TortoiseGit – Switch/Checkout…. In the popup dialog, you can select a local/remote branch in the dropbox.



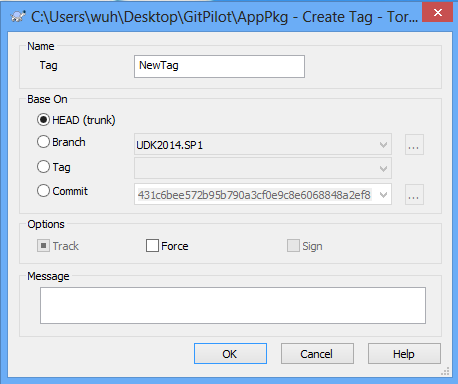
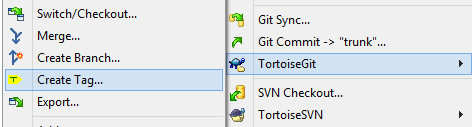
**Figure 67. Switch to a local branch using TortoiseGit**

To rename/delete a branch, right-click at your local repository – TortoiseGit – Browse References. In the popup dialog, you can select a local/remote branch in the dropbox.



**Figure 68. Rename/Delete a local branch using TortoiseGit**

To create a tag, right-click at your local repository – TortoiseGit – Create Tag…. In the popup dialog, you can choose where the tag should be based on. By selecting “HEAD”, the tag is based on the snapshot of your last commit. By selecting “Branch”, Git will create the tag basing on the tip of the chosen branch. By selecting “Tag”, new tag will be set as a copy of an existing tag. By selecting “Commit”, tag will be based on a specific commit. If the “Force” checkbox is checked, Git will replace an existing tag with the given name (instead of failing).

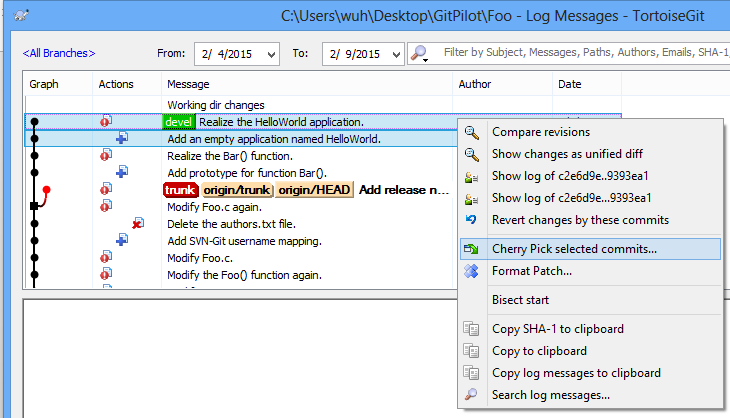


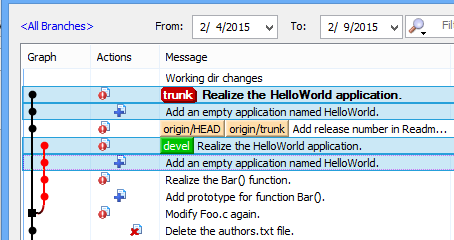
**Figure 69. Create a tag using TortoiseGit**

* + 1. Cherry-pick

Command cherry-pick is used to pick arbitrary commits in a branch to another branch. In the following example, some commits on branch “devel” are going to be picked to branch “trunk” (“devel” branch will remain unchanged).

Switch to branch “trunk” – Right-click in your repository – TortoiseGit – Show log – Check the “All Branches” checkbox – Select the commits in branch “devel” to be applied to branch “trunk” – Click the “Continue” button. One can see the selected commits are applied to “trunk”.

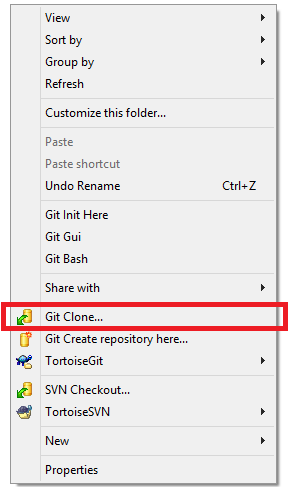
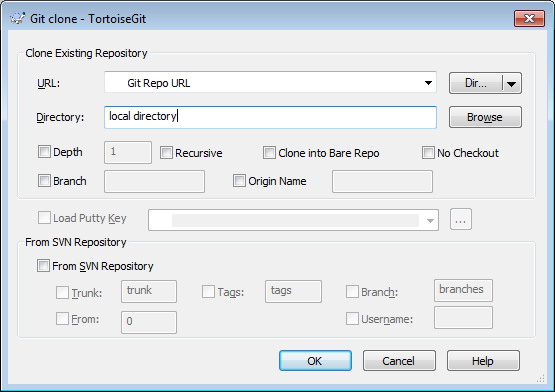




**Figure 70. Steps for cherry-picking commits from one branch to another using TortoiseGit**

* + 1. Clone

You can clone a Git repository to your local computer by: Choose a directory to put the code – Right click at the directory – Git Clone… – Input the URL of the Git repository.

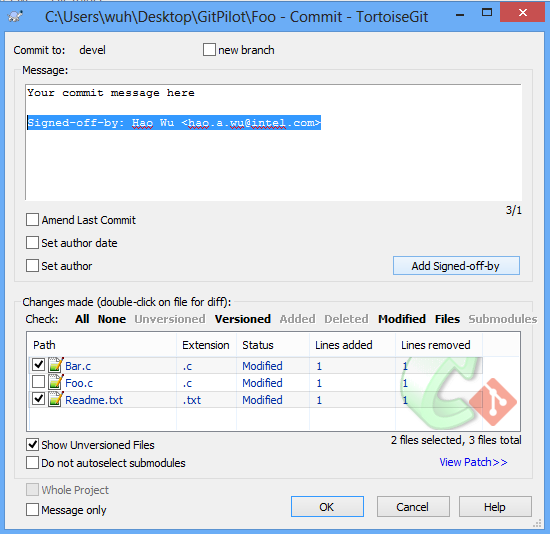
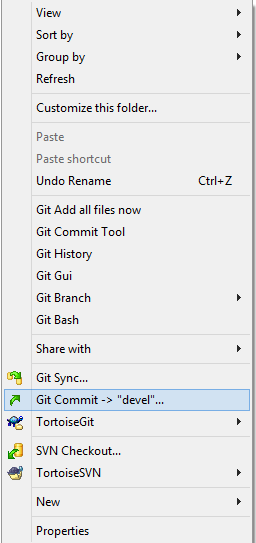
 

**Figure 71. Clone a Git repository using TortoiseGit**

* + 1. Commit

After making some changes locally, one can commit those changes locally (to the staging area).

If all your changes are going to be bundled in one patch, you can just run right-click in the working directory – Git Commit -> “devel”… (suppose “devel” is your current local working branch) – Select all files – Click “Add Sign-off-by” button – Fill in the commit message – OK.



**Figure 72. Commit local changes to the staging area using TortoiseGit**

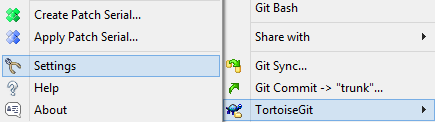
If you decide to separate your changes into multiple commits, you can repeat the above process multiple times by selecting some files to form a single commit.

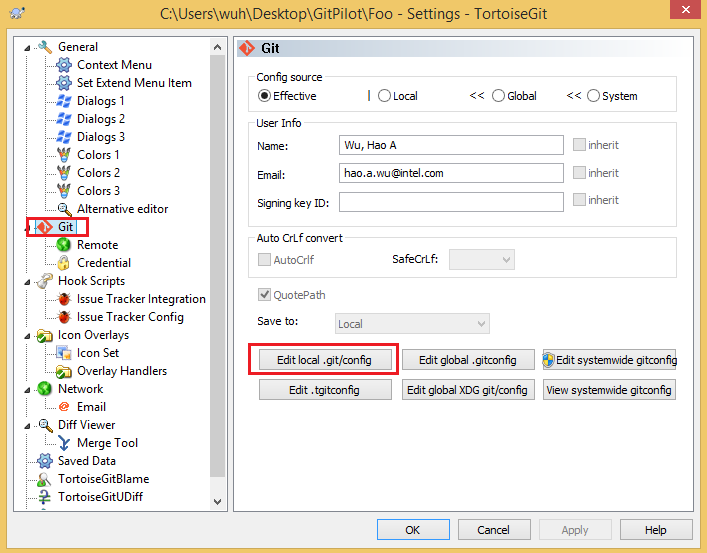
* + 1. Create patch

When the local commits are ready for code review, TortoiseGit can help to create patch files for chosen commits.

By default, the subject of a single patch is “[PATCH] ” followed by the concatenation of lines from the commit message up to the first blank. When multiple patches are generated, the subject prefix will instead be “[PATCH n/m] ”

If you want to use another subject prefix instead of the standard [PATCH] prefix in the subject line (A common case will be adding a version number of a patch series, which will be shown as an example below), you can right-click in the working directory – TortoiseGit – Settings – Select Git tab on the left –Edit local .git/config.





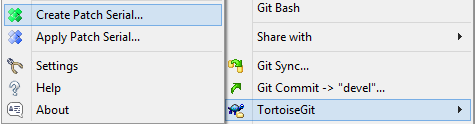
**Figure 73. Patch subject prefix setting in TortoiseGit**

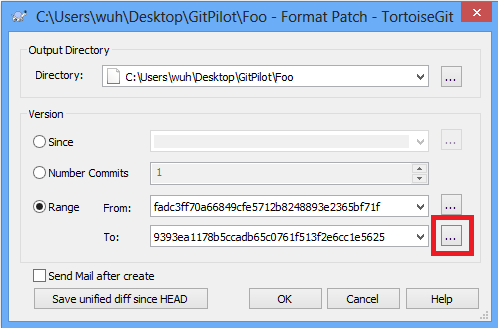
Add the following lines in the [format] section (for example):

subjectprefix = PATCH v2

Then, the subject of your review mails sent by Git will start with “[PATCH v2 n/m] ”.

To create patches for local commits: right-click in the working directory – TortoiseGit – Prepare Patch serial… – Select the local commits for generating patches.





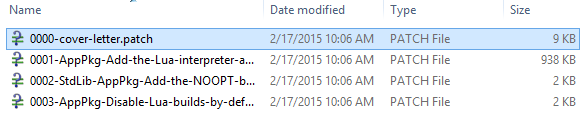


**Figure 74. Steps for creating patches for review using TortoiseGit**

Moreover, if one is creating a patch series for review, it is recommended adding a cover letter file which contains the summary description and the overall diffstat of the patch series. This feature is enabled by default in section 2.2. **Error! Reference source not found.**And one can disable this feature by: Right-click – TortoiseGit – Settings – Select Git tab on the left – Edit global.gitconfig or Edit local .git/config – remove the following line under [format] token:

coverLetter = auto

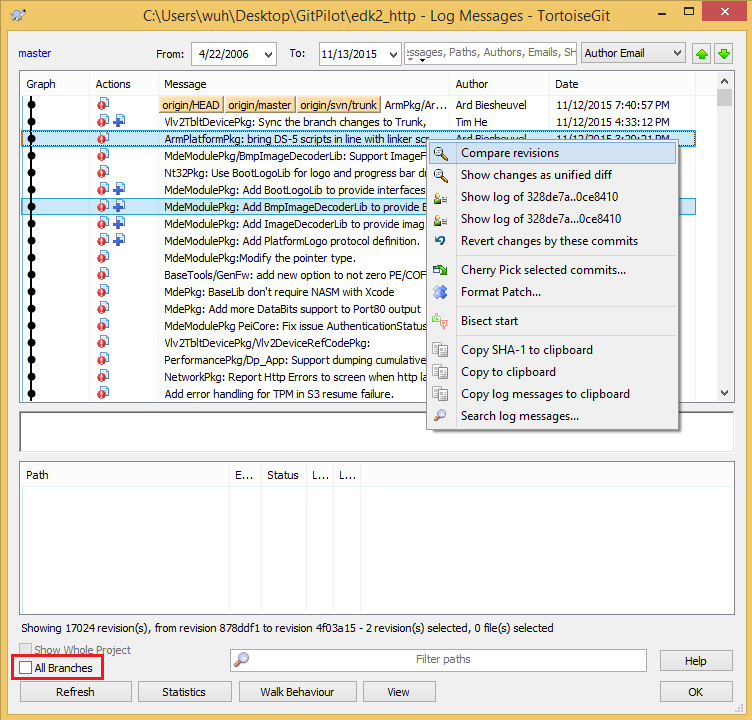
After patches are generated, you will find an additional patch named “0000-cover-letter.patch”. You need fill in a description in this file before sending it out.



**Figure 75. Generated patches and an additional cover letter**

* + 1. Diff

One can show the difference between two commits in the Log window by:



**Figure 76. Show difference between two commits using TortoiseGit**

If the two commits are not on the same branch, check the “All Branches” option.

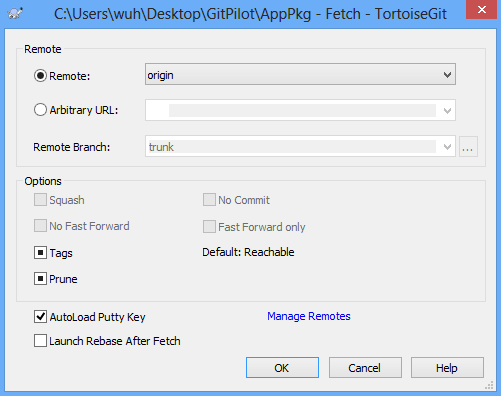
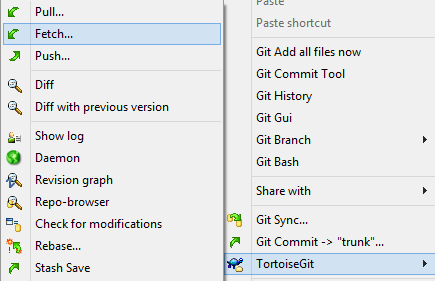
One can show the difference between two commits in the Browse References window by:



**Figure 77. Show difference between two branches using TortoiseGit**

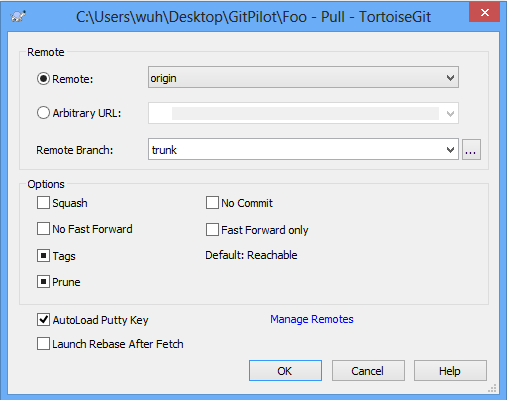
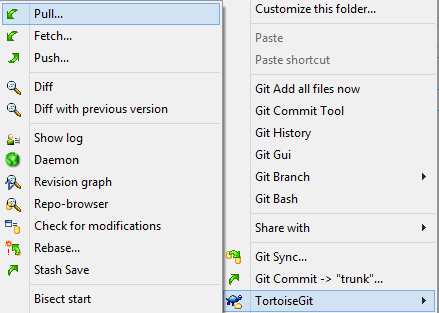
* + 1. Fetch & Pull

The git-fetch command is used to fetch branches and/or tags from one or more other repositories, along with the objects necessary to complete their histories. Remote-tracking branches are updated, local branches are not touched. To fetch changes from a remote repository: Right-click at your local repository – TortoiseGit – Fetch…



**Figure 78. Fetch changes from a Git remote using TortoiseGit**

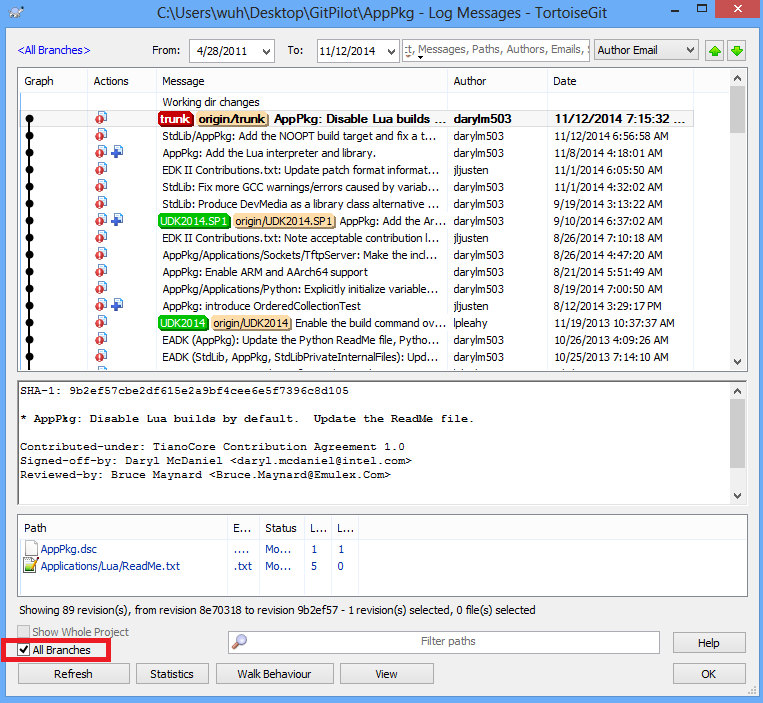
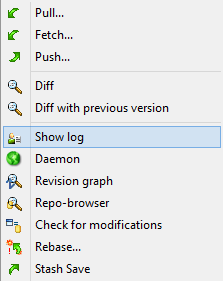
The git-pull command incorporates changes from a remote repository into the current local branch. More precisely, git-pull runs git-fetch with the given parameters and calls another command git-merge (which will be introduced later) to merge the retrieved branch heads into the current branch. With --rebase, it runs git-rebase (also will be introduced later) instead of git-merge. To update your local branch to keep up with the remote Git repository, please take the following steps: Right-click at your local repository – TortoiseGit – Pull… – Choose which remote branch to be pulled.



**Figure 79. Pull changes from a Git remote using TortoiseGit**

* + 1. Log

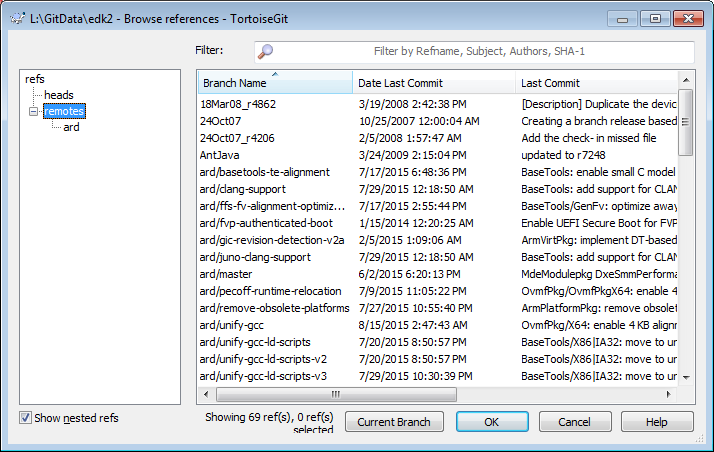
After one has created several commits, or if one has cloned a repository with an existing commit history, one will probably want to look back to see what has happened. The most basic and powerful tool to do this is the git-log command. To show the history log of a Git repository: Right-click at your local repository – TortoiseGit – Show log.



**Figure 80. Log history of a Git repository using TortoiseGit**

In the popup window, a whole list of commits of current branch will be displayed. If the “All Branches” checkbox is checked, commits from all branches (both local ones and remote ones) will be shown. For each commit, information like commit message, author of the commit and files modified is provided. Furthermore, if you double click a file of a specific commit, a detailed “diff” information will popup.

Now, all local branches will be shown above. For remote-tracking branches, they can be found in <All Branches>.



**Figure 81. Show remote-tracking branches using TortoiseGit**

* + 1. Merge

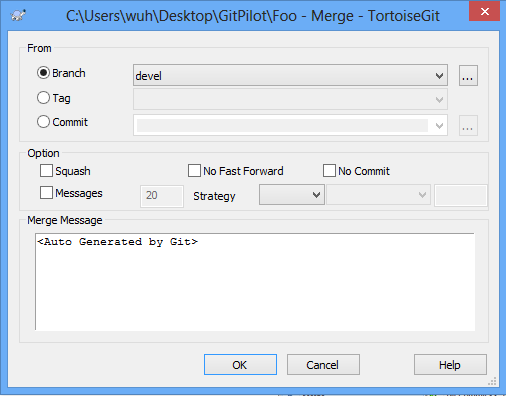
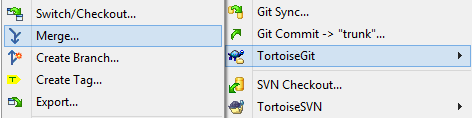
The git-merge command is used to join two or more development histories together. It is one way to integrate changes from one branch into another.

Consider the following repository structure:



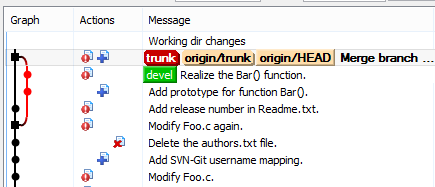
**Figure 82. Possible log history of a project**

If one wants to integrate changes made at “devel” branch to the “trunk” branch. One can right-click in your repository – TortoiseGit – Switch/Checkout… – To the branch the changes will be merged into (in this case, the “trunk”) – right-click in your repository – TortoiseGit – Merge… - In the “From” area, select branch “devel” – OK.



**Figure 83. Steps for merging commits from one branch to another using TortoiseGit**

If the merge process has no error, the result looks like the following pic:

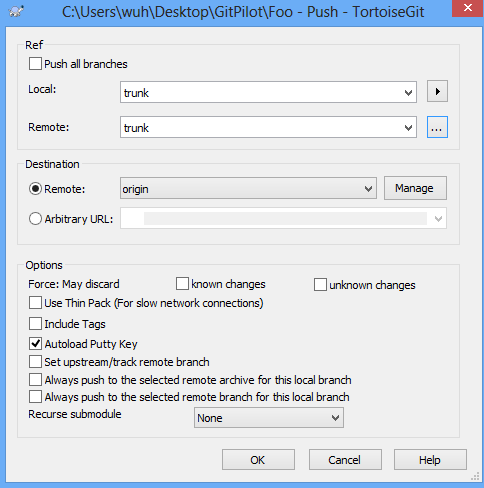
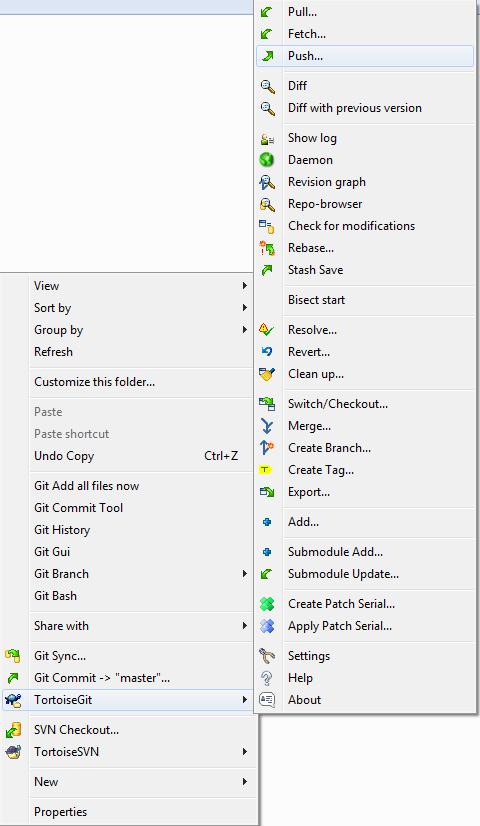


**Figure 84. Log history after the git-merge operation**

If there is any conflict during the merge process, follow the instruction provided by TortoiseGit and manually modify the file to resolve the conflict before continuing the merge process.

* + 1. Push

After your patches are reviewed, you can push it to the remote server. Right-click in your repository – TortoiseGit – Push… – OK.



**Figure 85. Push local commits to the remote Git repository using TortoiseGit**

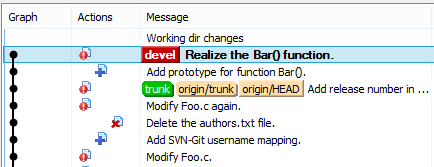
* + 1. Rebase

The git-rebase command is another way to integrate changes between branches. Consider the same situation in the git-merge command section:



**Figure 86. Possible log history of a project**

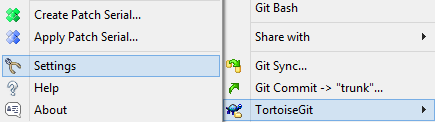
The process of rebase is: right-click in your repository – TortoiseGit – Rebase… – For “Branch:” select devel; for “Upstream:” choose trunk – Click “Start Rebase” button. The result is shown as the following pics:

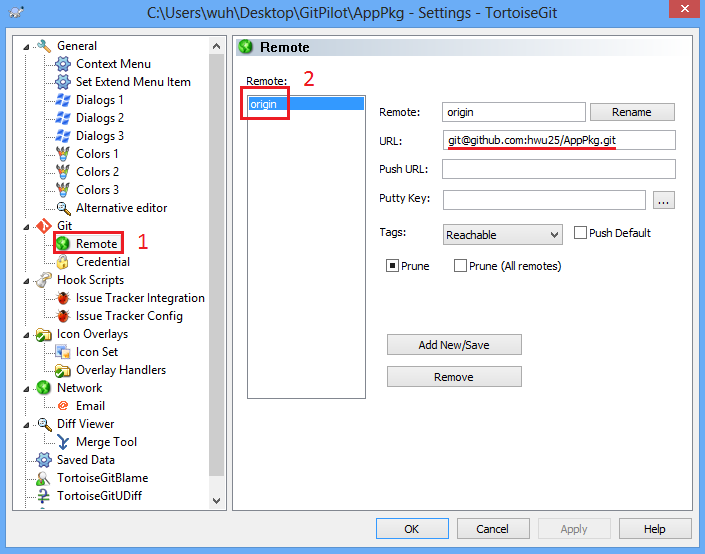


**Figure 87. Log history after the git-rebase operation**

* + 1. Remote

You can see the associated URL of a Git remote by: Right-click at your local repository – TortoiseGit – Settings – Select the “Remote” tab on the left – Select a remote. Then, you will see the related URL of the selected Git remote. You can also add more remote as here to see the change from other remote.



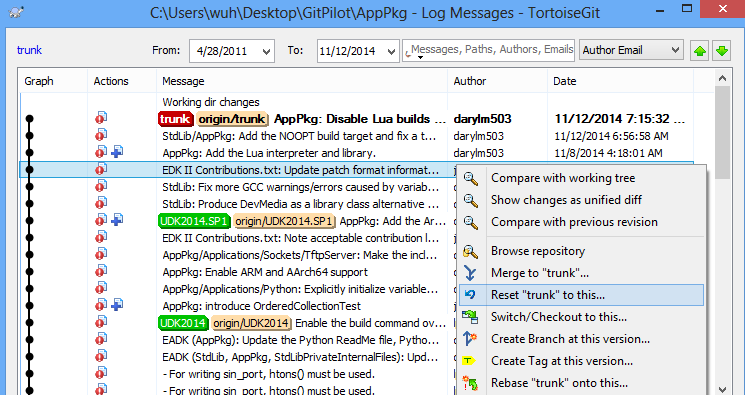


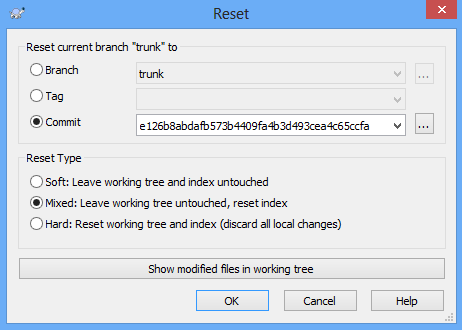
**Figure 88. Associated URL of a Git remote using TortoiseGit**

* + 1. Reset

The git-reset command is a versatile command with many configurations. It can be used to remove committed snapshots, although it’s more often used to undo changes in the staging area and the working directory. In either case, it should only be used to undo local changes—you should never reset snapshots that have been shared with other developers.

To reset the current branch back to a commit, right-click at your local repository – TortoiseGit – Show log – Select a specific commit – Reset “branch\_name” to this.



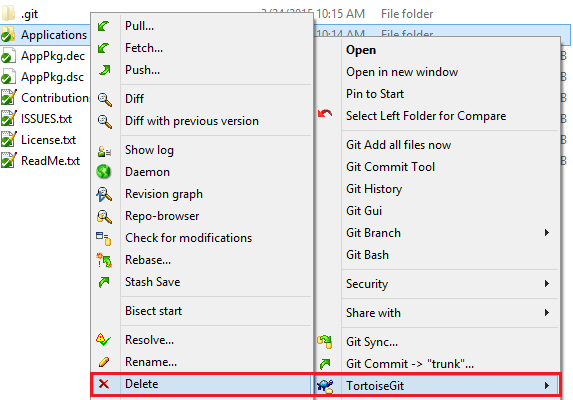


**Figure 89. Steps for undoing changes on a local branch using TortoiseGit**

In the popup windows, you can either select “Soft”, “Mixed” or “Hard” to reset the current branch. The effect of the above three modes for reset is introduced in Section A.1.14.

* + 1. Rm

If you want to delete files/directories from your repository, you can: Right click the files/directories – Delete



**Figure 90. Remove files from Git repository using TortoiseGit**

After the operation, the files/directories will be no longer in repository. It is worth pointing out that the above operation removes the files/directories from both the working tree and the index. As a comparison, simply deleting the files/directories in the file system only remove them from the working tree, the index (staging area) is not touched.

# Git Config File(.gitconfig)

**[user]**

**name = Your Name**

**email = your.name@intel.com**

**[format]**

**coverLetter = auto**

**[sendemail]**

**smtpserver = smtp.intel.com**

**confirm = always**

**suppresscc = self**

**[core]**

**autocrlf = false**

**whitespace = cr-at-eol**

**[am]**

**keepcr = true**

**[pull]**

**rebase = true**

[url "https://"]

insteadOf = git://

[https]

proxy = http://proxy-shz.intel.com:911

[http]

proxy = http://proxy-shz.intel.com:911

# ssh Config File (.ssh/config)

**Windows:**

Host \*intel.com

HostKeyAlgorithms +ssh-dss

Host github.com

User git

Hostname ssh.github.com

PreferredAuthentications publickey

IdentityFile ~/.ssh/id\_rsa

Port 443

ProxyCommand connect.exe -H proxy-prc.intel.com:911 %h %p

**Linux:**

Host \*intel.com

Host github.com

User git

Hostname ssh.github.com

PreferredAuthentications publickey

IdentityFile ~/.ssh/id\_rsa

Port 443

ProxyCommand corkscrew proxy-prc.intel.com 911 %h %p

# subversion servers File (.subversion/servers)

**[global]**

**http-proxy-exceptions = \*.intel.com**

**http-proxy-host = proxy-shz.intel.com**

**http-proxy-port = 911**

# Config File Location

**Windows:**

**C:\Users\$(UserId)\.gitconfig**

**C:\Users\$(UserId)\.ssh\config**

**C:\Users\$(UserId)\.subversion\servers**

**Linux:**

**~\.gitconfig**

**~\.ssh\config**

**~\.subversion\servers**

# Proxy List

**proxy-prc.intel.com:911**

**child-prc.intel.com:913**

**proxy-us.intel.com:911**

**proxy-sc.intel.com:911**

# Svn and Git Mapping

Table SVN URL

|  |  |
| --- | --- |
| 1 | https://svn.code.sf.net/p/edk2/code/trunk/edk2 |
| 2 | https://svn.code.sf.net/p/edk2-toolbinaries/code/trunk/Win32 |
| 3 | https://svn.code.sf.net/p/edk2-fatdriver2/code/trunk/FatPkg |
| 4 | https://ssvn.intel.com/ssg/csd/tiano/tianoad/trunk/R9 |
| 5 | https://ssvn.intel.com/ssg/csd/tiano/tianoad/trunk/BP |
| 6 | https://ssvn.intel.com:80/ssg/csd/tiano/tianoad/Documents/ |
| 7 | https://ssvn.intel.com:80/ssg/csd/tiano/tianoad/trunk/Edk2Docs/ |
| 8 | https://ssvn.intel.com:80/ssg/csd/tiano/tianoad/trunk/InternalTools/ |
| 9 | https://ssvn.intel.com:80/ssg/csd/tiano/tianoad/trunk/Platforms/ |
| 10 | https://ssvn.intel.com:80/ssg/csd/tiano/tianoad/trunk/Tools/ |
| 11 | https://ssvn.intel.com:80/ssg/csd/tiano/tianoad/trunk/UEFI\_SCTs/ |
| 12 | https://ssvn.intel.com:80/ssg/csd/tiano/tianoad/trunk/ValueAdd/ |
| 13 | https://ssvn.intel.com:80/ssg/csd/tiano/tianoad/trunk/OC/CustomerDocuments/ |
| 14 | https://ssvn.intel.com:80/ssg/csd/tiano/tianoad/trunk/OC/Documents/ |
| 15 | https://ssvn.intel.com:80/ssg/csd/tiano/tianoad/trunk/OC/Firmware/ |
| 16 | https://ssvn.intel.com:80/ssg/csd/tiano/tianoad/trunk/OC/Tools/ |
| 17 | https://ssvn.intel.com:80/ssg/csd/tiano/tianoad/branches/R9 |
| 18 | https://ssvn.intel.com:80/ssg/csd/tiano/tianoad/trunk/OC/Tools/BaseTools/Bin/Win32/ |
| 19 | https://ssvn.intel.com:80/ssg/csd/tiano/tianoad/trunk/OC/Tools/InternalTools/NewMkBinPkg/Win32/ |

Table GIT URL

|  |  |
| --- | --- |
| 1 | git@github.com:tianocore/edk2.git |
| 2 | git@github.com:tianocore/edk2-BaseTools-win32.git |
| 3 | git@github.com:tianocore/edk2-FatPkg.git |
| 4 | ssh://git-amr-1.devtools.intel.com:29418/ssg\_tiano-r9 |
| 5 | ssh://git-amr-1.devtools.intel.com:29418/ssg\_tiano-bp |
| 6 | ssh://git-amr-3.devtools.intel.com:29418/ssg\_tiano-documents |
| 7 | ssh://git-amr-3.devtools.intel.com:29418/ssg\_tiano-edk2docs |
| 8 | ssh://git-amr-3.devtools.intel.com:29418/ssg\_tiano-intenaltools |
| 9 | ssh://git-amr-3.devtools.intel.com:29418/ssg\_tiano-platforms |
| 10 | ssh://git-amr-3.devtools.intel.com:29418/ssg\_tiano-tools |
| 11 | ssh://git-amr-3.devtools.intel.com:29418/ssg\_tiano-uefi\_scts |
| 12 | ssh://git-amr-3.devtools.intel.com:29418/ssg\_tiano-valueadd |
| 13 | ssh://git-amr-3.devtools.intel.com:29418/ssg\_tiano-oc\_customerdocuments |
| 14 | ssh://git-amr-3.devtools.intel.com:29418/ssg\_tiano-oc\_documents |
| 15 | ssh://git-amr-3.devtools.intel.com:29418/ssg\_tiano-oc\_firmware |
| 16 | ssh://git-amr-3.devtools.intel.com:29418/ssg\_tiano-oc\_tools |
| 17 | ssh://git-amr-3.devtools.intel.com:29418/ssg\_tiano-uefisdv |
| 18 | ssh://git-amr-3.devtools.intel.com:29418/ssg\_tiano-oc\_basetools\_win32 |
| 19 | ssh://git-amr-3.devtools.intel.com:29418/ssg\_tiano-oc\_mkbinpkg\_win32 |