|  |
| --- |
| Accellera VIP Technical Sub Committee |
| UVM Community Development(CDEV) Workflow |
| How to contribute to UVM |
| Version 0.1 |
| **Dr. Ambar Sarkar Paradigm Works Inc.** |
| **4/2/2010** |

|  |
| --- |
| This document describes how you can get started with the UVM repository on Sourceforge as developed by Accellera VIP-TSC committee. |

Introduction

This document describes how you can quickly get started with the UVM repository on Sourceforge as developed by the Accellera VIP-TSC committee. ***Git*** is used as the version control system, and it is assumed that the user is responsible for learning git. However, there are enough pointers here to get started.

Getting Started

# First Steps

## Get git Install git from

[*http://git-scm.com/*](http://git-scm.com/)

I expect you will want to install it on your linux station. But cygwin works fine as well.

I am using version 1.6.4.2. Use this or later version. The latest (as on April 1, 2010) version is v1.7.0.4.

Here are a couple of good links to get started with git:

<http://www.sourcemage.org/Git_Guide>

<http://book.git-scm.com/book.pdf>

## Get your own clone

# Get a local snapshot of the repository

* 1. If you have committer(read/write) access to this repository, you will need your SF account and password.

***git clone ssh://<your sourceforge name>@uvm.git.sourceforge.net/gitroot/uvm/uvm***

* 1. You have read-only access

***git clone git://uvm.git.sourceforge.net/gitroot/uvm/uvm***

**Identify self to git**

***git config --global user.name "FirstName LastName"***

***git config --global user.email "user@example.com"***

This will create a directory uvm.

Workflows

Git is a very flexible and scalable version controlling system and supports a wide variety of workflows and environments. The challenge for us is to adopt a flow that will allow one to be productive without requiring a deep understanding of git, while maintaining integrity of the public repository.

The workflows presented here are suitable for the VIP-TSC needs, and strike a good balance between simplicity, flexibility, robustness, and maintainability.

There will be the following three kinds of branches:

* master: This branch is the mainline. All approved changes are merged back into this main branch.
* <feature> : Each non-trivial change must be first created in a separate branch. The committee will review the code and any subsequent modifications must be made on this branch accordingly. Once approved, this branch will be merged back into the mainline. If not approved, the branch is abandoned, and mothballed for tracking purposes.
* <release>: For every approved release of UVM, we will create a separate branch, with the name of the branch reflecting the release.

The public repository will stay on Sourceforge. Developers are expected to clone the public repository into their private workspaces, and follow the workflows as described in the following pages.

While anyone can clone the workarea and develop their own code, only the designated committers can push their changes to the central repository.

Note that any change that is pushed to the central server will be automatically notified to the mailing list, but the committer must send an email to the group with details.

## Workflow for committing features to the repository:

Each non-trivial change is implemented in a separate branch. The changes will merged into mainline only after committee approval. The following steps ***must*** be followed. For the current example, assume the implementation branch is named PW\_FEAT\_1:

Create dedicated branch locally

Add feature and any subsequent mods

Review

Merge with mainline

Changes needed

Approved?

Mothball the branch

Yes

No

**$ cd uvm  
$ git checkout –b PW\_FEAT\_1 origin/master**

**$ git checkout PW\_FEAT\_1** *// Switch to branch***$ git push origin PW\_FEAT\_1** *// Publish* **<send email >**

**$ git checkout PW\_FEAT\_1** *// Switch to branch*< Make sure all changes are committed to the local branch>  
**$ git commit …** *// Please add meaningful comments*

< See reviewer workflow on page 6>

< Keep around for record keeping, unless committee wants to delete the branch>

**$ git checkout master** *// Make sure in master branch***$ git pull origin master** *// Pull in latest from mainline*  
**$ git merge origin/PW\_FEAT\_1**  *// Merge latest master   
 // with latest PW\_FEAT\_1* **$ git push –dry-run origin master** *// Do a dry run***$git push origin master** *// Publish* **$ <send email!>**

No more changes

Publish branch

Publish to mainline

## Workflow for reviewing feature branch:

Assuming the branch is named PW\_FEAT\_1, the reviewer needs to do the following

Get the review branch locally

Fetch latest

Done review?

No

**$ cd uvm   
$ git fetch origin  
$ git checkout PW\_FEAT\_1**

**$ git checkout PW\_FEAT\_1** *// Switch to branch* **$ git merge origin/PW\_FEAT\_1** *// Get the latest*

Yes

Send feedback

Send email

## Workflow for community contribution using email:

The typical community member will not have write access to the repository. However, we need a way for the large user community to contribute their suggested changes. This is done in two steps.

Step 1. The **community member** emails patches to a designated committer.

Step 2. Assuming the committee agrees to work on this, the **designated committer** creates a separate branch, applies the patches, and then follows the standard workflow mentioned in page 5, “Workflow for committing features to the repository:”

## Step 1. (Community member contributor):

This is the case where anyone from the community at large can contribute their changes.

The following steps should be taken by the community developer. Assume they chose the name COMM\_FEAT\_1 as the feature name.

Create dedicated branch locally

Add all mods

**$ cd uvm  
$ git checkout –b COMM\_FEAT\_1 origin/master**

**$ git checkout PW\_FEAT\_1** *// Switch to branch***$ git format-patch –M origin/master***<This will generate \*.patch files, one for each commit>*

**$ git checkout COMM\_FEAT\_1** *// Switch to branch*< Make sure all changes are committed to the local branch>  
**$ git commit …** *// Please add meaningful comments*

Create patch

Email .patch files to VIP-TSC contact (TBD)

## Step 2. (Designated committer):

Create dedicated branch locally

Merge patches in

Merge with mainline

Approved?

Mothball the branch

Yes

No

**$ cd uvm  
$ git checkout –b COMM\_FEAT\_1 origin/master**

**$ git checkout COMM\_FEAT\_1** *// Switch to branch***$ git push origin COMM\_FEAT\_1** *// Publish* **<send email >**

**$ git checkout COMM\_FEAT\_1** *// Switch to branch* **$ git am \*.patch** *// Merge user patches in***$ git commit …** *// Please add meaningful comments*

< See reviewer workflow>

< Keep around for record keeping, unless committee wants to delete the branch>

**$ git checkout master** *// Make sure in master branch***$ git pull origin master** *// Pull in latest from mainline*  
**$ git merge origin/COMM\_FEAT\_1**  *// Merge in with   
 // latest* **$ git push –dry-run origin master** *// Do a dry run***$git push origin master** *// Publish* **$ <send email!>**

Publish branch

Publish to mainline

Basic git commands

1. Help

***git help   
git help*** *<command>*

1. To add files/dirs

***git add*** *file1 file2 file3****git add*** *dirname*

1. To commit changes to local repository

***git commit***

1. To see what has changed (Including stuff that has not been added)

***git status***

or

***git difftool --tool=tkdiff <the\_file>***

1. To see what is to be commited

***git diff --cached***

1. List history

***git log***

1. To fetch changes (but not merge) from SF and see the changes

***git fetch*** *=> Get the changes from remote site and stores on local site****git log –p HEAD..FETCH\_HEAD*** *=> Tells what changed*

1. To merge with the SF changes (after calling fetch)

***git merge***

1. *You can do steps 7 and 8 together using*

***git pull***

1. *Finally, push your changes to SF using* ***[Developer only]***

***git push –dry\_run*** *=> To see what will be pushed*

***git push*** *=> push it is*

## Tagging

After commiting, use the following to create a tag from the commited version

***git tag -a -m "Newer tag" unofficial\_oct\_8\_2009 bb6e6eb9735e9bf39d2143ba9747a00000f83e22***

Push it out

***git push – tags***

To create a branch with the tagged version

***git checkout –b <name\_your\_branch> <tagname>***