|  |
| --- |
| Accellera VIP Technical Sub Committee |
| UVM Community Development(CDEV) Workflow |
| How to contribute to UVM |
| Version 1.0 |
| **Dr. Ambar Sarkar Paradigm Works Inc.** |
| **4/16/2011** |

|  |
| --- |
| 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***

This will create a directory uvm.

**Set some default git configurations**

*Identify self to git*

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

***git config --global user.email*** [***user@example.com***](mailto:user@example.com)

*Ensuret all branch merges are recorded consistently without losing history.*

***git config --global core.mergeoptions "--no-ff”***

Development Branches

# Terminology

**Major releases** 1.0, 1.2, 2.0 etc

**Bugfix release** 1.1.a, 1.1.b etc

**Early adopter release**. 1.0.EA, 1.2.EA etc

**Mantis branch** Where any initial coding is done. There should be a corresponding mantis filed. Branch is merged to integration branch(see below) upon code review and approval.

**Integration branch** Branch dedicated to integrating various Mantis branches for a specific release.

**Master branch** The trunk or main repository branch. This branch reflects code that is carried forward for future releases and is not tied to a specific release.

# General rules

1. **Any development done is done in a mantis branch**
   1. The only time code is directly checked into the integration branch is
      1. During release window
      2. With approval from team.
2. **Any release is always done from an integration branch**
   1. This allows us to keep release specific code in that branch.
   2. The branch is retired after public release and tagged. Becomes a candidate for deletion for future.
3. **The mainline always reflects the code that is carried forward to all future releases.** 
   1. Unless specific to a release, all commit-ids are merged back into master with team approval.
4. **Integration branches are created off the master branch**
5. **Mantis branches are created off the integration branch**

# Branches Created

The following two categories of branches will be created for any targeted release. The process is same for any kind of release, namely EA/Bugfix/Major.

1. **UVM\_<release>**

Following command will be used to create the integration branch. Typically, this will be the task of the integrator. The following commands are typically used:

git branch –no-track UVM\_<release> origin/master # Creates a local branch  
 git checkout UVM\_<release> # Switch to that branch  
 git push origin UVM\_<release> # Push this new branch to central repo

These branches will be created by branching off the main branch. This branch will act as the place for all integration area for all changes for that release. So for UVM1.2, we will create a branch called UVM\_1\_2. For UVM1.1b, we will create a branch named UVM\_1\_1\_b.

1. **Mantis\_<id>**

Any change for a release will have a mantis id associated with it. That change will be done in branch named Mantis\_<id> where <id> is the mantis id. Following command will be used to create the mantis branch by the mantis owner

git branch –no-track UVM\_<mantis> origin/UVM\_<release> # Creates a local branch  
 git checkout UVM\_<mantis> # Switch to that branch  
 git push origin UVM\_<mantis> # Push this new branch to central repo

The commands above will be done only once per branch. After that, the developer(s) will follow one of the workflows in the following pages to make further changes to the Mantis branch.

The mantis branch, upon code review and approval, will be merged into the corresponding devel (UVM\_<release> branch.

The following diagram should illustrate thebranching. UVM\_1\_1\_b and UVM\_1\_2 integration branches are created off the master. The releases are done from the corresponding integration branches, and are tagged. Once released, relevant commit ids are merged back to mainline. The selective commit is indicated using dashed lines.

Mantis A is spawned off the UVM\_1\_1\_b integration branch and merged back to the same. Same goes for Mantis C and corresponding integration branch UVM\_1\_2.

Mantis B is interesting as it represents a change for UVM\_1\_1\_b that needs to go into UVM\_1\_2 branch as well. Since UVM\_1\_2 branch was already created before the UVM\_1\_1\_b was merged back into mainline, this matntis needs to be specifically merged into the UVM\_1\_2 development branch.

**UVM\_1\_1\_b branch**

**UVM\_1\_2**

**master**

Mantis\_A

Mantis\_B

Mantis\_C

**uvm-1.1.b.tgz  
RELEASE!!!!**

**uvm-1.2.tgz  
RELEASE!!!!**

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.

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 commiter must send an email to the group with details.

The following workflows are provided

* Creating a new mantis branch
* Development on an existing mantis branch and merge after review approval
* Reviewing code on a mantis branch
* Community contribution by email

# Workflow for creating new mantis branch (Done exactly once)

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 Mantis\_x and current integration branch is UVM\_y

**$ cd uvm  
$ git checkout - - no-track -b Mantis\_x origin/INTEG\_1**

**$ git push origin Mantis\_x** *// Publish* **<send email >**

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

Publish branch

Create dedicated branch locally

# Workflow for updating mantis branch and merge after review approval:

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 Mantis\_x and current integration branch is UVM\_y:

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 Mantis\_x\_1 origin/Mantis\_x**

**$ git push origin Mantis\_x** *// Publish* **<send email >**

**$ git checkout Mantis\_x** *// 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 UVM\_y** *// Make sure in integration branch***$ git pull origin UVM\_y** *// Pull in latest from mainline*  
**$ git merge origin/Mantis\_y**  *// Merge latest master   
 // with latest FEAT\_1* **$ git push –dry-run origin UVM\_y** *// Do a dry run***$git push origin UVM\_y** *// Publish* **$ <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 FEAT\_1**

**$ git checkout FEAT\_1** *// Switch to branch* **$ git pull** *// 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/UVM\_y**

**$ git format-patch –M origin/UVM\_y***<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/UVM\_y**

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

1. *Finding out remote tracking branches*

***git remote show origin***

You will get details on how various branches are being tracked. Especially handy is what is printed towards the bottom:

…….

  Local branches configured for 'git pull':

    Mantis\_4075 merges with remote Mantis\_4075

    UVM\_1\_1\_b   merges with remote UVM\_1\_1\_b

   master      merges with remote master

  Local refs configured for 'git push':

    Mantis\_4075 pushes to Mantis\_4075 (up to date)

    UVM\_1\_1\_b   pushes to UVM\_1\_1\_b   (up to date)

    master      pushes to master      (up to date)

1. check status, check the header of the output (the second line should not have a message about “ahead”, “diverged”, or similar)

***git status | head -2***

1. in case of collisions (make sure you got a decent merge/diff tool in place, kdiff3 recommended

***git mergetool***

1. review the changes (always a good tool, others are tortoise-git, the eclipse git client,…)

***gitk***

1. delete the mantis remote branch

***git push origin :mantis\_xxxx***

1. delete the local branch

***git branch –d mantis\_xxxx***

## 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>***