Gerrit Development Workflow

Recent Upgrades

- A new deployment ready branch “rdk-next” has been created with higher standards of test qualification
- **rdk-next** is the new CMF branch that the community will push changes to for review.
- CMF has initially synchronized the rdk-next branch with Comcast production branch code. Automated scripts running daily will identify new Comcast commits and push them for review to rdk-next branches on CMF Gerrit.
- **rdk-dev-yymm** is a new CMF integration branch, created monthly and baselined off rdk-next. This branch will be hosted per repository in conjunction with rdk-next, with the goal of incorporating community changes at the earliest juncture.
- Community changes, once approved, will be cherrypicked to rdk-dev-yymm and and will thus be available prior to the completion of Comcast down-streaming/ round-trip process.
- Approved contributions will be down-streamed to Comcast for predeployment validation using their test process
    - Defects will be planned in monthly sprints
    - Features will be presented for Architecture Review to be scheduled to an upcoming sprint. Sprint timelines to be published to contributor.
    - Contributions pending validation will be available in monthly development iteration branches

- Downstreamed Community changes, successfully merged to Comcast Sprint branches, will be cherrypicked to Comcast production branch.
- When downstreamed Community changes are merged to the Comcast production branch they can be merged immediately on the CMF Gerrit rdk-next branch.
- Comcast changesets, imported or contributed daily to rdk-next , will also be cherry picked to the rdk-dev-yymm branch throughout the monthly cycle (once they have passed the necessary gating criteria of scan, build). In this way, the rdk-dev-yymm branch will also keep sync with Comcast production branch.

rdk-next-branch	rdk-dev-yymm branch
💡 Quarterly releases	💡 Monthly Iteration releases
💡 Deployment ready code	💡 Reviewed by component owners
💡 Pre-Deployment tests	💡 Tested with Reference platforms

Upgraded work-flow

1. User will do code contribution to rdk-next branch. This will undergo:
    a. Code reviews
    b. Build verification
    c. License compliance scan
    d. Test validation
2. Once successful, the change will get cherry-picked to rdk-dev-yymm (monthly dev) branch
3. This code is then down-streamed to Comcast branch where pre-deployment test validation are done
4. Once Comcast accepts, the change is merged to rdk-next.
5. Thus the change gets merged to rdk-next

Code Contribution Process

- For features - open a RDKDEV (for video or generic contributions) or RDKBDEV (for Broadband contributions)
- Clone the component code from rdk-next branch
- Apply changes
- Push the patch to Gerrit for review
- Trigger reviews and handle review comments

- Change set trigger: Use TDK for validation and BlckDuck scan for license complicance

Feature Contribution - JIRA Guidelines

A feature contribution should follow after creating an appropriate JIRA project. This will present a clear picture about the architecture, testing details and other information which will be helpful during the acceptance process of the contribution.

Mandatory information

1. **Project**
   For Video & build system (Yocto) related contributions, the ticket should be created under RDKDEV. For broadband, the ticket should be created under RDKBDEV project.
2. **Issue Type**
   Issue type corresponds to the type of contributions we are making. The following issue types can be possible
   Incident - Build failure incident issues with the code verification steps such as Black duck scan, Jenkins verification etc.
   Bug - Bugs in existing component code.
   Task - An individual task which may be part of a new feature or improvement.
   Improvement - Improvements such as code refactoring or enhancements in current code.
   New feature - New feature contributions.
3. **Ticket Status**
   Status should be initially Open, and transitioned to the appropriate value while the contribution is being worked on.
4. **Summary**

   A brief summary about what we are trying to contribute
5. **Description**

   A descriptive information about the contribution should be present so that component developers & architecture team can do assessment of the feature. Below details are desirable if the contribution is a new feature or having an significant impact on the current architecture.

Solution Overview
Brief introduction on what the current system lacks & what needs to be done: 1. Individual task/highlighted point #1 brief description. 2. Individual task/highlighted point #2 brief description.

Architecture Checklist

```
The following items should be considered/addressed in the documentation for any RDK design initiative
JIRA Update Checklist
----------------------

The following JIRA fields MUST be filled in to be considered "Definition Complete":

RDK SoC, RDK OEM - populate these fields for any user story where we have dependency on OEM and/or SoC

OEM/SoC Impact Details - description of impact (or "see Solution Overview" if included in the architecture
specification)
* Platforms - ensure correct list of devices
* Validation - type of testing
* Regression - is regression required?
* Impacted Party - Comcast Only, Syndication, etc.
* Dependency - Internal/External
* Description - Solution Overview and Architecture Checklist

Testing impact/Guidance

------------------------
* Impacted modules
* Test process

Automated Testing
------------------

* Automation test procedure.

Diagnostics, Telemetry and Logging
-----------------------------------

* N/A

Outbound Network Connections
-----------------------------

* Does this component make outbound connection requests?
* If yes, do the connection requests retry in the case of failure?
** Do the repeated requests use an exponential back-off?
** If a maximum back-off has been defined, is it greater than 10 minutes?

Security
----------

* For Security Review - Do feature elements:
** make any changes to network port assignments?
** change iptables rules?
** require credentials, passwords, secret data?
** make any changes to our network connections?
** connect to new services or servers?
** use data input from users or external tools?
** use any cryptographic functions?
** create or disclose proprietary or sensitive Co. or device data?
** properly log operational and configuration changes?
** If possible describe what could happen if feature elements are:
*** spoofed?
*** tampered with?
*** used by an unauthorized actor?
** Advanced questions (optional)
*** what happens if a record of actions taken is destroyed?
*** what happens if an attacker attempts to DOS with the feature?

SI Concerns
-------------

* Yes/No/Any

Performance expectations
-------------------------

* Yes/No/Any


Timing consideration
----------------------

* If Any.
```

Supplementary Information

1. Impacted component(s) - Fill in list of impacted RDK components
2. RDK SI Impact - System Integration impacts
3. CPE SW Components - Component names.
4. Test Notes - Describe what tests are performed to validate this contribution and the procedure.
5. Unit Test Result - Description.

Gerrit Review Process

In order to contribute code, first-time users are requested to agree to the license at https://wiki.rdkcentral.com/signup.action.

The rdk-next source is hosted at code.rdkcentral.com. Earlier, we were using master source code, now everything is migrated to rdk-next. You can submit your changes for review via that site using the workflow outlined below.

1. Clone the component repository from the Gerrit server https://code.rdkcentral.com/r/ into a local workspace

Clone with commit-msg hook (to add Change-ID footer to commit messages)

```
git clone https://code.rdkcentral.com/r/rdk/component1 -b rdk-next && (cd component1 && gitdir=$(git rev-parse
--git-dir); curl -o ${gitdir}/hooks/commit-msg https://code.rdkcentral.com/r/tools/hooks/commit-msg ; chmod +x
${gitdir}/hooks/commit-msg)
```

Note: The commit-msg hook is installed in the local Git repository and is a prerequisite for Gerrit to accept commits. The inclusion of the unique Change-ID in the commit message allows Gerrit to automatically associate a new version of a change back to its original review.

Note: You may need to configure your Git identity on the cloned repository. The email address that your local Git uses should match the email address listed in Gerrit.

Example commands to run are as follows:

```
$ git config user.name "John Doe"

$ git config user.email "john.doe@example.org"
```

2. Work on the change, commit to local clone

Each commit constitutes a change in Gerrit and must be approved separately. It is recommended to squash several commits into one that represents a change to the project.

If necessary, it is possible to squash a series of commits into a single commit before publishing them, using interactive rebase:

```
$ git rebase --interactive
```

It is important to preserve the *Change-Id* line when editing and there should only be one "pick" entry at the end of this process. The end result is to submit one change to Gerrit.

3. Push the new change to Gerrit for review

Commits will be BLOCKED if the format of the commit message does not comply with the standard. You will see a warning as to why the commit was blocked.

Mandatory Information in Commit Message

1. Associated JIRA ticket
2. Reason for change information
3. Test procedure by which change can be verified
4. Possible risks of failure

```
<JIRA TICKET #1>, <JIRA TICKET #2>, <JIRA TICKET #n> : <one line summary of change>
<empty line>
Reason for change: <explanation of change>
Test Procedure: < test procedure>
Risks: <side effects and other considerations> [Note: state None if there are no other considerations]
<empty line>
Signed-off-by: Your Name <your_name@email.com>
```

```
$ git push origin HEAD:refs/for/<branch>
```

When interfacing with Gerrit you push to a virtual branch /refs/for/<branch>, representing "code review before submission to branch". Gerrit will subsequently assign a unique URL for the change, to facilitate access and review via the web UI.

Notes:

- *HEAD* is a Git symbolic reference to the most recent commit on the current branch. When you change branches, *HEAD* is updated to refer to the new branch's latest commit.
- The *refspec* in the git push operation takes the form ***source:destination*** (***source*** is the local ref being pushed, ***destination*** is the remote ref being updated).

4. Review notifications and addition of new reviewers

Component owners/reviewers/approvers, defined as specific groups in Gerrit, will be added to the review by default. You may request additional feedback by specifically adding reviewers via the Gerrit web GUI.

5. Scan and build on code submission

BlackDuck, copyright scanning and build jobs will be triggered automatically from CMF Jenkins. The output of these jobs is integrated into the Gerrit voting process via custom labels and will reflect any 'red flag' in a file that has new code changes, whether introduced in the new change/patch-set or not. Scans will post any findings as comments in the Gerrit review. Build jobs also do that, but in addition will upload the build log to the corresponding JIRA ticket (if there is one) as an attachment.

6. Code review and scoring process

Reviewers can comment on and score a given change.
The default set of rules for enabling a code change for submission requires:

- a Code Review score of +2; this can only be provided by the component owner or an admin;
- +1 score on any mandatory Gerrit labels configured for the project.

The result of the scoring process and validation rules is to enable the *Submit* action on the Gerrit Web UI and subsequent merge capability to the target branch.

Label: Code Review(Highlighted in yellow color) For a change to be mergeable, the latest patch set must have a '+2' value approval in this category or label, and no '-2 Do not submit'. Thus -2 on any patch set can block a submit, while +2 on the latest patch set enables it for merging.

Labels: Blackduck/Copyright/Component-Build (Highlighted in yellow color) For a change to be mergeable, the change must have a '+1' score on these labels, and no '-1 Fails'. Thus, '-1 Fails' can block a submit, while '+1' enables a submit.

! Review input is generally referred to as labelling with a positive/negative score.

7. Submit change

Only authorised users, i.e. component owners, component approvers or admins, can submit the change allowing Gerrit to merge it to the target branch as soon as possible. A change can be submitted, having satisfied the approval conditions described earlier, by clicking the 'Submit Patch Set n' button within the Gerrit UI. When a change has been Submitted, it is automatically merged to the target branch by Gerrit.

8. Abandon change

Depending on the review outcome, it might be decided to abandon the change. The component owner or an authorised user may abandon the change by clicking the "Abandon Change" button. The abandoned changes are not removed from the Gerrit database and can be restored at a later stage.

9. Submitted, Merge Pending

If a change depends on another change that is still in review, it will enter this state. It will be merged automatically by Gerrit once all its dependencies are submitted and merged.

10. Change needs to be reworked

If you need to rework a change, you need to push another commit with the same *Change-ID* as the original in its commit message.
This is the mechanism Gerrit uses to associate or link the two items. The `--amend` option to the Git commit command prevents a new *Change-ID* being generated by the *commit-msg* hook.

The basic steps are outlined below.

First, fetch the change. If you still have the checkout that was used to push the original change, you can skip this step.

```
$ git fetch https://user@code.rdkcentral.com/r/component1 refs/changes/02/2/1 && git checkout FETCH_HEAD
```

where the numbering scheme for fetching the changes is as follows:

refs/changes/<last two digits of change number> <change number> <patch set number>

! Gerrit will specify this fetch URL via the web UI on the '*Download*' link on the review page for the change in question, you just paste it into the command line.

Next, make any necessary source changes, and do:

```
$ git commit --amend
$ git push origin HEAD:refs/for/<branch>
```

A new patch set is now appended to the Gerrit review item, and this will go through the same review process as before.

!

- The 'change number' referenced above is different to underlying Git commit ID.
- Patch-sets are numbered (starting from 1) for each review, and incremented whenever a change is amended with another Git commit.
- *FETCH_HEAD* is a Git symbolic reference and shorthand for the head of the last branch fetched and is valid only immediately after the fetch operation.

11. Gerrit merge failure as a result of a conflict

Essentially this means that the remote branch has evolved since this change was started and now software conflicts with changes in the remote branch.
The developer must resolve the merge conflicts in their local clone and then push another patch-set.
The process is resumed at step 4, with the important distinction of committing with the --amend option, once the developer pulls the latest changes. **Note:** A summary of the steps involved, assuming the local branch still exists:

Rebase the local branch to the latest state of origin/<branch>;Resolve all conflicts; Commit with the `--amend` option; Push changes to Gerrit for review. After this change a new patch set is created for the change.

Note: If the local branch no longer exists, the steps are as follows :

```
$ git fetch https://user@code.rdkcentral.com/r/rdk_component_1 refs/changes/58/58/2 && git checkout FETCH_HEAD
$ git rebase origin/<branch>
[Edit the conflicting file, cleaning up the <<<<, ==== >>> markers surrounding the conflicting lines]
$ git add <file>
$ git commit --amend
$ git push origin HEAD:refs/for/<branch>
```