Using Git for Project Management with µVision

MDK Version 5 Tutorial

AN279, Spring 2015, V 1.0

Abstract

Teamwork is the basis of many modern microcontroller development projects. Often teams are distributed all over the world and over various time zones. Synchronizing the development work is the main task for project managers. As this work cannot be done by sending files via email from one location to another, Software Version Control Systems (SVCS) are used. In the past, Concurrent Versions System (CVS) and Subversion (SVN) have been popular for workflows with a centralized server infrastructure. Nowadays, more and more decentralized workflows have been established, making use of a new SVCS called Git.

This application note describes how to integrate Git into µVision and how to establish a robust workflow for version control of projects using Software Packs.

Contents

Introduction

Revision control is being used for a couple of years now in software development. In the past, centralized server infrastructures have been used to track the changes in the source code of a microcontroller development project. With the release of Git, decentralized VCS have become more and more popular.

Git, which is free software distributed under the terms of the GNU General Public License, was developed for maintaining the Linux kernel. It became more and more popular and is now the de-facto standard in source control management (SCM), another term for VCS.

This application note assumes that you have installed Git on your PC (refer to Appendix A: Software) and do know how to use the Windows command shell (cmd.exe).

Workflows

Using Git, you can make use of various workflows when maintaining a software project:

- Centralized workflow as known to users of CVS and SVN
- Feature Branch workflow that adds a branch for every feature that is to be developed to the master branch
- Gitflow workflow using specific roles to different branches such as develop, feature, and release branch
- Forking workflow that is a truly distributed workflow in the sense that every developer has his own fork of the master branch and only one project maintainer is accepting commits to the master branch

This application note explains how to use µVision in a centralized workflow environment. Other workflows can be realized as well using a mix of Git commands from within µVision and using the command line/GUI. For more information on workflows, see Appendix B: Links.

Centralized Workflow

In a centralized work environment, there is usually a single collaboration model: the centralized workflow. A central repository accepts code changes, and every developer synchronizes his work to it. This is just like using SVN for example. However, using Git has a few advantages over the traditional SVN workflow.

First, it gives every developer an own local copy of the entire project. In this local environment, each developer can work independently of all other colleagues. The local repository is used for commits and they can completely forget about the original one until it's convenient for them.

Second, all developers can use Git's robust branching and merging model. Git branches are designed to be fail-safe when adding code and sharing changes between repositories.

The centralized workflow uses a central repository serving as the entry point for all project changes. The default development branch is called 'master' and all changes are committed into this branch. No other branches are required by this workflow:

1. Initialize the central repository
2. Clone a Repository from a Server
3. Feature work (stage/commit/push)
4. Manage Conflicts

Using Git with µVision

Project Files under Version Control

Before setting up the workflow, the project manager should be absolutely clear about the files that need to be version controlled. Of course all source code files need to be versioned, but there are a couple of files that are special to µVision that need to be monitored as well.

- All user generated source files (*.c, *.cpp, *.h, *.inc, *.s)
- Project file: *Project*.uvprojx (is used to build the project from scratch)
- Project options file: *Project*.uvoptx (contains information about the debugger and trace configuration)
- Project file for multi-project workspaces: *Project*.uvmpw
- Configuration files for the run-time environment that are copied to the project (all files below .\RTE)
- List of #includes created by software components: RTE\RTE_Components.h file
- Device configuration file: for example RTE\Device\LPC1857\RTE_Device.h
- Linker control file (*Project*.sct) if created manually
- All relevant Pack files (for example ARM::CMSIS, Keil::Middleware, Device Family Packs, etc.)

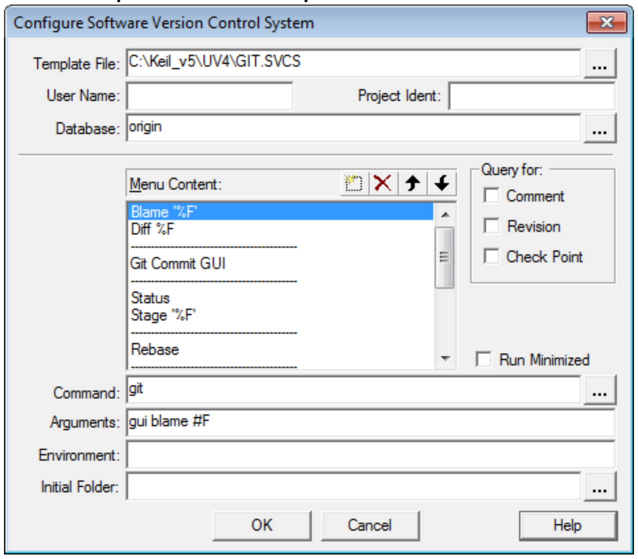
Files that do not need to be monitored

- Project screen layout file: *Project*.uvguix.*username*
- All files that are part of a Pack (the complete Pack will be revision controlled and is available to every user as soon as he is installing it using Pack Installer)
- Generated output files in the sub-directories .\Listings and .\Objects
- INI files for debug adapters

Configure µVision's SVCS

Before using Git for SCM, you need to configure µVision's software version control menu. µVision 5.15 is providing a SVCS template file for Git. It resides in the installation directory (for example C:\Keil_v5\UV4) and is called GIT.SVCS. For versions previous to 5.15, please use the file as it is provided in this application note's ZIP file and copy it to the µVision installation directory.

1. Open µVision and go to **SVCS → Configure Software Version Control…**
2. Enter the path to the template file:



Template File: C:\Keil_v5\UV4\GIT.SVCS
User Name:
Project Ident:
Database: origin

Menu Content:
Blame '%F'
Diff %F
Git Commit GUI
Status
Stage '%F'
Rebase

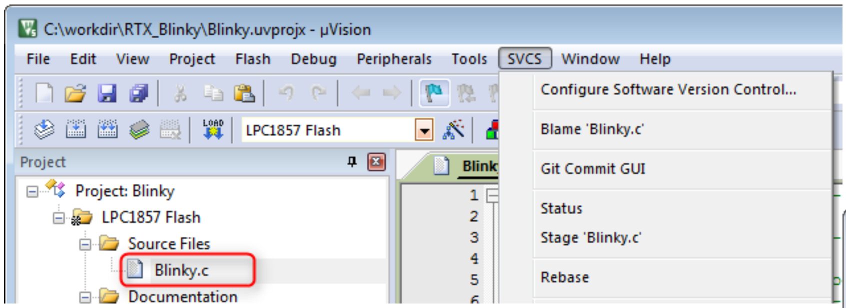
Query for: ☐ Comment ☐ Revision ☐ Check Point
☐ Run Minimized

Command: git
Arguments: gui blame #F
Environment:
Initial Folder:

When done, press **OK**. Now, µVision is ready to use Git as the version control system.

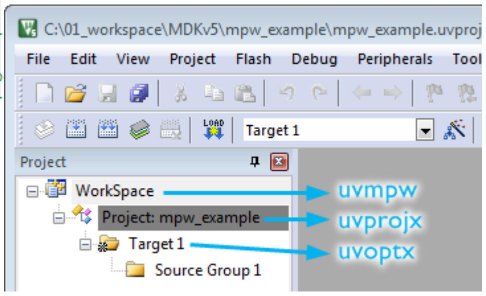
Accessing µVision Project Files

When working with the SVCS menu, you can usually access files that are highlighted in the Project window of µVision. For example, if you are clicking on the File Blinky.c in the Project window and you then go to the **SVCS** menu, you will see the SVCS options for the Blinky.c file:


C:\workdir\RTX_Blinky\Blinky.uvprojx - µVision
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
LPC1857 Flash
Project
Project: Blinky
LPC1857 Flash
Source Files
Blinky.c
Documentation
Configure Software Version Control...
Blame 'Blinky.c'
Git Commit GUI
Status
Stage 'Blinky.c'
Rebase

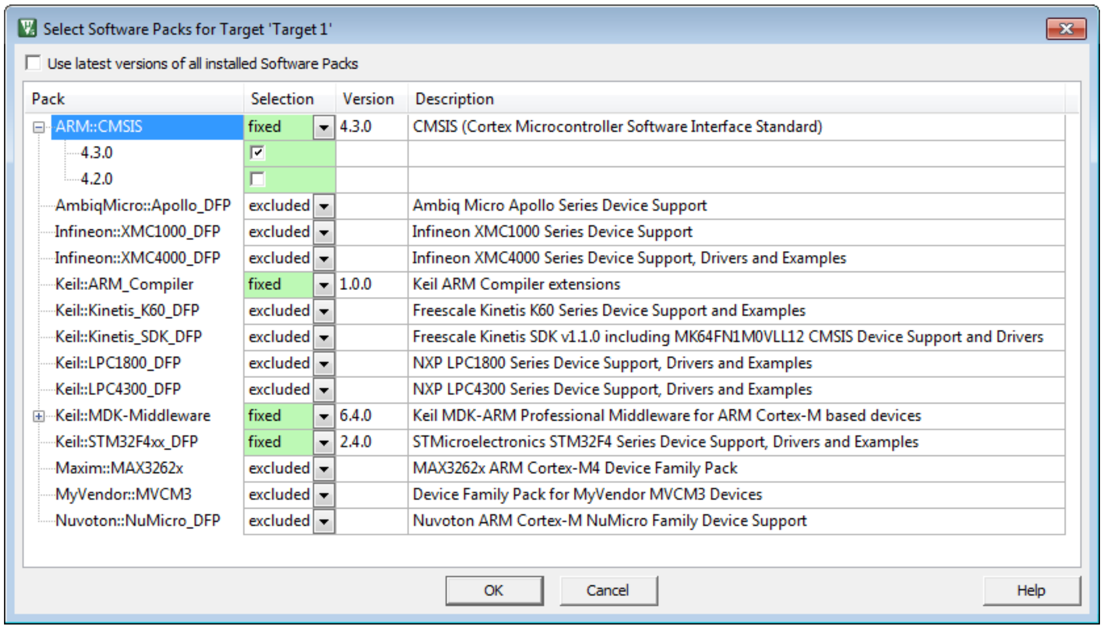
To catch µVision's project related files (uvmpw, uvprojx, uvoptx), click on the following (or equivalent):

- WorkSpace → uvmpw
- Project → uvprojx
- Target 1 → uvoptx


C:\01_workspace\MDKv5\mpw_example\mpw_example.uvproj
File Edit View Project Flash Debug Peripherals Tool
Target 1
Project
WorkSpace
Project: mpw_example
Target 1
Source Group 1
uvmpw
uvprojx
uvoptx

Version Control of Software Packs

To ensure that only a certain version of a Software Pack is used in a project, use the **Select Software Packs…** icon: This will open a new window that enables you to set the versions of all Software Packs:



☐ Use latest versions of all installed Software Packs

Pack	Selection	Version	Description
ARM::CMSIS	fixed	4.3.0	CMSIS (Cortex Microcontroller Software Interface Standard)
4.3.0	✓		
4.2.0	☐		
AmbiqMicro::Apollo_DFP	excluded		Ambiq Micro Apollo Series Device Support
Infineon::XMC1000_DFP	excluded		Infineon XMC1000 Series Device Support
Infineon::XMC4000_DFP	excluded		Infineon XMC4000 Series Device Support, Drivers and Examples
Keil::ARM_Compiler	fixed	1.0.0	Keil ARM Compiler extensions
Keil::Kinetis_K60_DFP	excluded		Freescale Kinetis K60 Series Device Support and Examples
Keil::Kinetis_SDK_DFP	excluded		Freescale Kinetis SDK v1.1.0 including MK64FN1M0VLL12 CMSIS Device Support and Drivers
Keil::LPC1800_DFP	excluded		NXP LPC1800 Series Device Support, Drivers and Examples
Keil::LPC4300_DFP	excluded		NXP LPC4300 Series Device Support, Drivers and Examples
Keil::MDK-Middleware	fixed	6.4.0	Keil MDK-ARM Professional Middleware for ARM Cortex-M based devices
Keil::STM32F4xx_DFP	fixed	2.4.0	STMicroelectronics STM32F4 Series Device Support, Drivers and Examples
Maxim::MAX3262x	excluded		MAX3262x ARM Cortex-M4 Device Family Pack
MyVendor::MVCM3	excluded		Device Family Pack for MyVendor MVCM3 Devices
Nuvoton::NuMicro_DFP	excluded		Nuvoton ARM Cortex-M NuMicro Family Device Support

Hint: Uncheck **Use latest versions of all installed Software Packs**.

The information about the selected Software Pack versions is stored in the *Project*.uvprojx file that is also under version control using this workflow.

Centralized Workflow Example

The following sections show an example for the centralized workflow. Other workflows can be derived from the example. If the SVCS menu does not provide an entry for a required Git command, use either the GUI or the command line instead.

1. Initialize the Central Repository

Not all steps in the workflow can be accomplished using µVision. Some steps (especially the one for initializing the repository) need to be made using additional software. Throughout this application note, Git for Windows is used. You can either use the tool's GUI or type in the required commands in a Windows command shell as described here.

A project has been added to this application note's ZIP file. It is contained in the sub-directory RTX_Blinky. Copy this folder to your hard disk, for example to C:\workdir. Open a Windows command shell by typing `cmd` in the address bar of Windows explorer and hit enter. A command shell opens up with the project directory:

At the command prompt, enter

```
C:\workdir\RTX_Blinky>git init
Initialized empty Git repository in C:/workdir/RTX_Blinky/.git/
```

Now, you have to add your remote repository to your working copy (note that this remote repository must already exist – please consult your Git server space provider for details):

```
C:\workdir\RTX_Blinky>git remote add origin https://github.com/account/rtx_blinky.git
```

Then add (stage) all files from the project:

```
C:\workdir\RTX_Blinky>git add *
warning: LF will be replaced by CRLF in Blinky.uvoptx.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in Blinky.uvprojx.
The file will have its original line endings in your working directory.
```

Note: The project files uvprojx and uvoptx have UNIX-style line endings (LF). In Windows systems, usually CRLF is used. Git automatically detects this and changes the line endings to CRLF on the server. But then the status of these two project files is always "changed". You can get around this problem using the command `git config --global core.autocrlf false`

Commit all files and add a description for the commit:

```
C:\workdir\RTX_Blinky>git commit -m "Initial version"
[master (root-commit) b71a02e] Initial version
warning: LF will be replaced by CRLF in Blinky.uvoptx.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in Blinky.uvprojx.
The file will have its original line endings in your working directory.
12 files changed, 5408 insertions(+)
create mode 100644 Blinky.c
create mode 100644 Blinky.uvoptx
create mode 100644 Blinky.uvprojx
create mode 100644 Debug_RAM.ini
create mode 100644 Packs/ARM.CMSIS.4.3.0.pack
create mode 100644 Packs/Keil.LPC1800_DFP.2.4.0.pack
create mode 100644 RTE/CMSIS/RTX_Conf_CM.c
create mode 100644 RTE/Device/LPC1857/RTE_Device.h
create mode 100644 RTE/Device/LPC1857/startup_LPC18xx.s
create mode 100644 RTE/Device/LPC1857/system_LPC18xx.c
create mode 100644 RTE/RTE_Components.h
```

Finally, push the files to the server:

```
C:\workdir\RTX_Blinky>git push -u origin master
Username/Password
Counting objects: 19, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (17/17), done.
Writing objects: 100% (19/19), 101.31 MiB | 104.00 KiB/s, done.
Total 19 (delta 0), reused 0 (delta 0)
To https://github.com/account/rtx_blinky.git
* [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

Congratulations! You have created your first Git repository. For a more detailed description of the commands that you have just used, go to section 3.

2. *Clone a Repository from a Server*

To get a copy of an existing Git repository (for example, a project that is created within your engineering group), you need to use the command `git clone`. In contrast to other version control systems such as SVN, the command is called "clone" and not "checkout". Instead of getting just a working copy, a full copy of nearly all data on the server is saved to your hard drive. For example:

```
C:\workdir> git clone https://github.com/account/rtx_blinky.git
```

You will be asked for your username and password. The command creates a sub-directory named "rtx_blinky" in C:\workdir, initializes a .git directory inside it, downloads all the data for that repository, and checks out a working copy of the latest version. If you change into the new rtx_blinky directory, you will see the µVision project files. If you want to clone the repository into a different directory, specify the name as the next command-line option:

```
C:\workdir> git clone https://github.com/account/rtx_blinky.git MyBlinky
```

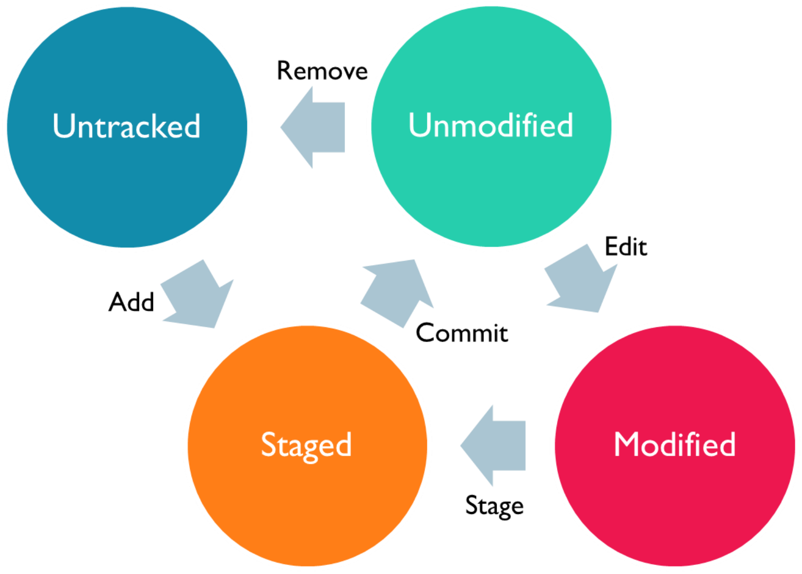
A number of different transfer protocols is available in Git. This example uses the `https://` protocol, but you may also see `git://` or `ssh://user@server:path/to/repo.git`.

3. *Feature Work*

Now, you have a Git repository and a checkout or working copy of the files for that project. Working on the project will create some changes and you will need to commit snapshots of those changes into your repository from time to time.

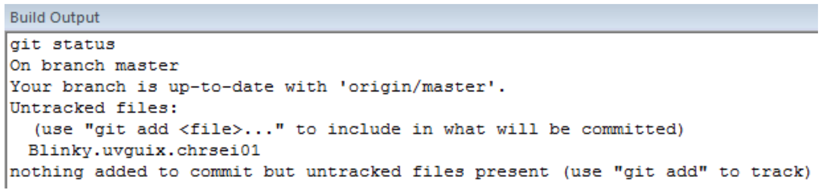
Each file in the working directory is in one of the following two states: tracked or untracked. Tracked files were already present in the last snapshot; they can be unmodified, modified, or staged. Untracked files are everything else (for example build output logs etc.) – any files in the working directory that were not present in the last snapshot and are not in the staging area. Cloning a repository for the first time, all files will be tracked and unmodified.

While editing files, Git will recognize them as modified, because they have changed since the last commit. Staging modified files and then committing all staged changes will make them be tracked/unmodified again:



Checking the File Status

To determine the status of your files use the `git status` command. It is available in the SVCS menu (Status). If you run this command directly after a clone, your **Build Output** window will show something similar:


```
git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)
 Blinky.uvguix.chrsei01
nothing added to commit but untracked files present (use "git add" to track)
```

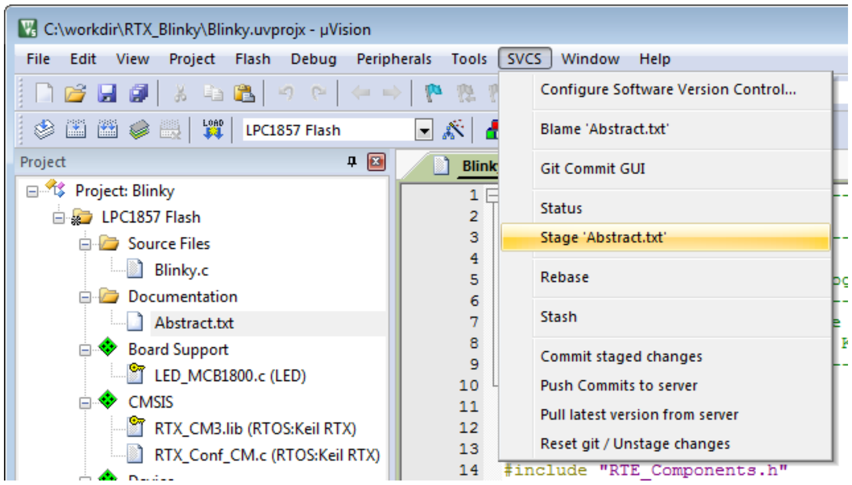
When opening the project, µVision automatically generates a uvguix.*username* file. This is marked as not tracked (because it is not required).

Staging (Adding) an Untracked File

For this example, we will create a text file to will explain the usage of the project. The file will be added to the working copy of the project and later committed to the origin.

In µVision, right-click on **Target 1** and choose **Add Group**. **New Group** will be added in the **Project** window. Click on **New Group** and a cursor should appear. Change the group name to **Documentation**. Right-click on **Documentation** and choose **Add New Item to Group 'Documentation'**. In the next window, click on Text File (.txt) and enter **Abstract** as the name. The file will be added and opened in the editor. Enter any text that might seem fit. A `git status` will show the file as untracked.

In order to begin tracking (or adding) Abstract.txt, use `git add`. Make sure that the file is highlighted in the **Project** window and then go to **SVCS → Stage 'Abstract.txt'**:



The **Build Output** echoes the command: `git add C:\workdir\RTX_Blinky\Abstract.txt`

Rerun the `git status` command to see the file now being tracked and staged for the next commit:

```
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
  new file:   Abstract.txt
```

Before committing the file, let's change another file so that it changes from unmodified to modified.

Staging Modified Files

Double-click Blinky.c to open it. After line 26 there are two empty lines. Delete line 27 and save the file. Another git status will output the following:

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
  modified:   Blinky.c
```

To stage the changed file, use the **Stage** item on the SVCS menu as shown in the previous step. Check the successful staging using the `git status` command.

Removing a Tracked File

The GIT.SVCS file *does not* provide the ability to use the `git rm` command. This command removes a file from the staging area and from the working directory. But the file still might be visible in µVision (especially when you have the file open in the Editor window). This might lead to unexpected behavior. To remove a file safely from a µVision project and the working repository, do the following:

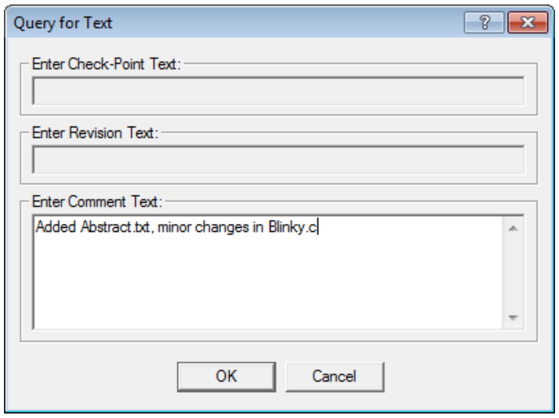
- Close the file in the **Editor** window.
- In the **Project** window, right-click on the file and select **Remove File 'MyFile.c'**
- Switch to your Windows command shell, go to the file's directory and issue the command `git rm MyFile.c`

At the next commit, the file will be gone and no longer tracked. If you modified the file in between and added it to the index already, the removal must be forces using the `–f` option. This safety feature prevents from accidentally removing data that has not yet been part of a snapshot and that cannot be recovered from Git.

Committing Changes

Before issuing the `git commit` command, check the project for unstaged files. A commit will only accept staged files. Any created or modified files which have not yet been staged won't go into this commit. They will remain modified on the hard disk.

To commit, go to **SVCS → Commit staged changes**. A window for entering the commit message will open:



Added Abstract.txt, minor changes in Blinky.c

Enter the comment text and press **OK**. The **Build Output** window will show the success of the `git commit`:

```
git commit -m C:\Users\User\AppData\Local\Temp\Sccs13
[master 1d1b0e6] C:\Users\User\AppData\Local\Temp\Sccs13
 2 files changed, 1 insertion(+), 1 deletion(-)
 create mode 100644 Abstract.txt
```

Pushing to the Repository

When the project has reached a point can be shared with others, it has to be pushed upstream (to the repository server): `git push [remote-name] [branch-name]`. The built-in command in the SVCS menu goes one step further. It uses `git push origin master` to push the master branch to the origin server.

There are two prerequisites for this command to work properly:

1. You need to have write access to the server you are pushing to *and*
2. Nobody has pushed in the meantime. If this happens, your push will be rejected:
   ```
   ! [rejected]        master -> master (fetch first)
   error: failed to push some refs to
   'https://github.com/account/rtx_blinky.git'
   hint: Updates were rejected because the remote contains work that you
   do not have locally. This is usually caused by another repository
   pushing to the same ref. You may want to first integrate the remote
   changes (e.g., 'git pull ...') before pushing again.
   See the 'Note about fast-forwards' in 'git push --help' for details.
   ```
   You'll have to pull down first and incorporate any changes or rebase before you will be allowed to push.

4. *Manage Conflicts*

Rebasing

The Git server's central repository is the one that every developer will finally commit to. If a developer's local commits diverge from this repository, Git will refuse to push the changes. Before the developer can publish on the server, all updated commits on the central repository need to be fetched and the changes need to be *rebased* on top of them. This results in a linear history on the central repository, just like in a SVN workflow.

If some of the changes are conflicting directly with upstream commits, Git will pause the rebasing process. The developer can then manually resolve the conflicts. For generating commits and merging conflicting files, Git uses the same `git status` and `git add` commands. This makes it easy to manage merges.

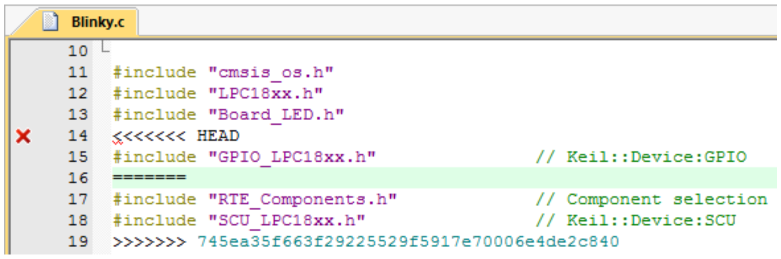
To rebase, use **SVCS → Rebase**. This will issue a `git pull --rebase origin master` that tells Git to move all of your commits to the tip of the `master` branch after synchronizing it with the changes from the central repository.

Rebasing transfers each local commit to the central repository's master branch one at a time. Thus, merge conflicts are caught on a commit-by-commit basis rather than by resolving all at once in a single merge commit.

If two developers have worked on the same features (for example `Blinky.c`), the rebasing process will generate conflicts. Git will pause the rebase (at the current commit) and the Build Output window will show the following conflict message:

```
git pull --rebase origin master
From https://github.com/account/usb2
 * branch            master     -> FETCH_HEAD
   44af614..cdfdc12  master     -> origin/master
First, rewinding head to replay your work on top of it...
Applying: C:\Users\User\AppData\Local\Temp\Sccs5
Using index info to reconstruct a base tree...
M     Blinky.c
Falling back to patching base and 3-way merge...
Auto-merging Blinky.c
CONFLICT (content): Merge conflict in Blinky.c
Failed to merge in the changes.
Patch failed at 0001 C:\Users\User\AppData\Local\Temp\Sccs5
The copy of the patch that failed is found in:
   c:/workdir/USB_test/.git/rebase-apply/patch
When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run
"git rebase --abort".
```

Also, the problem will be shown in the Editor window, for example:

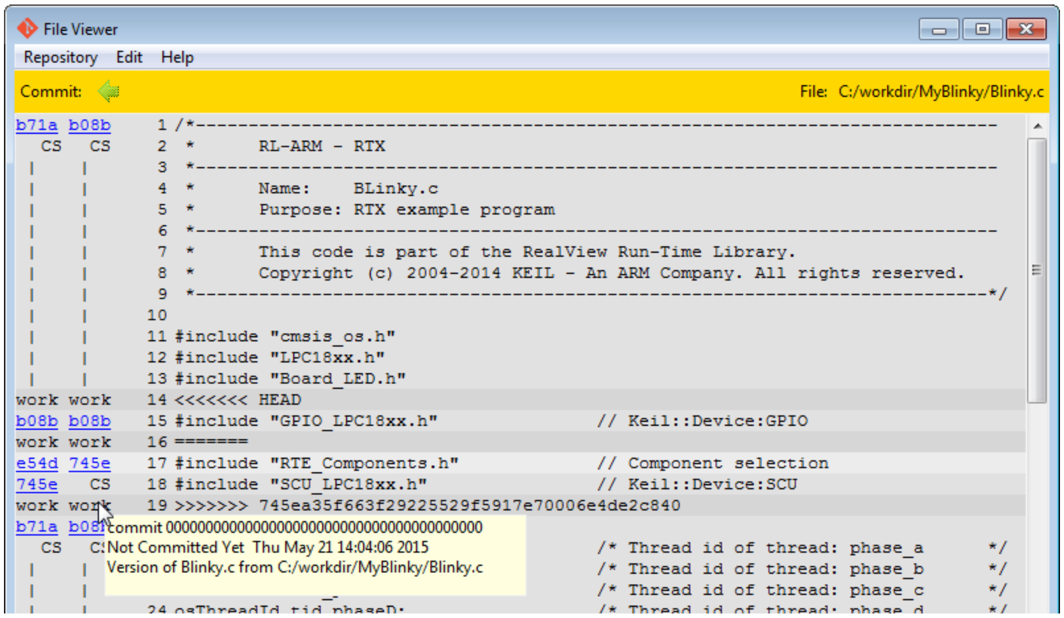

Blinky.c

```

#include "cmsis_os.h"
#include "LPC18xx.h"
#include "Board_LED.h"
<<<<<<< HEAD
#include "GPIO_LPC18xx.h"                    // Keil::Device:GPIO
=======
#include "RTE_Components.h"                  // Component selection
#include "SCU_LPC18xx.h"                     // Keil::Device:SCU
>>>>>>> 745ea35f663f29225529f5917e70006e4de2c840
```

Blaming

To get more information on the changes, you can use Git's blame feature. While the main.c file is highlighted in the Project window, go to **SVCS → Blame 'Blinky.c'**. This will initiate the GUI of Git for Windows and show the details of the differences (hovering over the window will give more details in the yellow boxes):



```
b71a b08b    1 /*----------------------------------------------------------------------------
  CS   CS    2  *      RL-ARM - RTX
  |    |     3  *----------------------------------------------------------------------------
  |    |     4  *      Name:    BLinky.c
  |    |     5  *      Purpose: RTX example program
  |    |     6  *----------------------------------------------------------------------------
  |    |     7  *      This code is part of the RealView Run-Time Library.
  |    |     8  *      Copyright (c) 2004-2014 KEIL - An ARM Company. All rights reserved.
  |    |     9  *---------------------------------------------------------------------------*/
  |    |    10
  |    |    11 #include "cmsis_os.h"
  |    |    12 #include "LPC18xx.h"
  |    |    13 #include "Board_LED.h"
work work   14 <<<<<<< HEAD
b08b b08b   15 #include "GPIO_LPC18xx.h"                    // Keil::Device:GPIO
work work   16 =======
e54d 745e   17 #include "RTE_Components.h"                  // Component selection
745e   CS   18 #include "SCU_LPC18xx.h"                     // Keil::Device:SCU
work work   19 >>>>>>> 745ea35f663f29225529f5917e70006e4de2c840
b71a b08b                                               
  CS   C:                                               /* Thread id of thread: phase_a      */
  |    |                                                /* Thread id of thread: phase_b      */
  |    |                                                /* Thread id of thread: phase_c      */
  |    |    24 osThreadId tid_phaseD;                   /* Thread id of thread: phase_d      */
```

```
commit 0000000000000000000000000000000000000000
Not Committed Yet  Thu May 21 14:04:06 2015
Version of Blinky.c from C:/workdir/MyBlinky/Blinky.c
```

Finishing the Rebase Process

When finished merging the file(s), do not forget to stage (`git add`) them. A commit is not necessary at this time as you are rebasing. Then, you need to continue the rebase process. This is not available from within µVision. You need to switch to the Windows command shell and enter:

```
C:\workdir\rtx_blinky>git rebase --continue
```

Git will acknowledges the changes with

```
Applying: C:\Users\User\AppData\Local\Temp\Sccs5
```

Finally, do not forget to push the changes to the server (see next section).

Stashing a Project

Sometimes, things can get messed up and you do want to switch branches for a while to work on another part of the project. But you do not want to commit half-done work to get back to this point later. To come around this problem, you need to issue the `git stash` command. Stashing takes the current (messy) state of your working directory (all modified tracked files and staged changes) and saves it on a stack of unfinished changes that can be reapplied to the working directory at any time.

Pulling the Latest Version from the Server Repository

To pull the latest version from the remote server, go to **SVCS → Pull latest version from server**. This will initiate the `git pull` command:

```
git pull
From https://github.com/account/rtx_blinky
   d8418bf..6fbcd97  master     -> origin/master
Merge made by the 'recursive' strategy.
… Whatever has been changed …
```

A subsequent git push should show no errors:

```
git push origin master
To https://github.com/account/usb2.git
   6fbcd97..e0c5dd3  master -> master
```

Conclusion

This application note has shown one way of using Git with µVision. As Git offers may ways of collaboration, please consult the available literature that is stated below on how to use Git for a distributed workflow. Using a combination of the commands from the GIT.SVCS file and the command line/Git GUI, you can use any workflow supported by Git.

Appendix A: Software

To be able to use Git on a Windows PC and from within µVision, you need to install the following software:

- Git for Windows: https://msysgit.github.io/
- For MDK versions below 5.15, you need to add the µVision SVCS template file for Git which is part of this application note's ZIP file: www.keil.com/appnotes/docs/apnt_279.asp
  Please copy the file to the µVision installation directory, for example C:\Keil_v5\UV4

Appendix B: Links

Git: http://git-scm.com

Git Book: http://git-scm.com/book/en/v2

Git server space providers: https://github.com/, https://bitbucket.org, https://about.gitlab.com/

Possible Workflows: www.atlassian.com/git/tutorials/comparing-workflows

Migrate from SVN to Git: www.atlassian.com/git/tutorials/migrating-overview

More information on Git replacing LF with CRLF: http://tinyurl.com/q3y7ohx