



White paper

**Manage embedded software
with Subversion**

COPYRIGHT NOTICE

© Copyright 2011 Atollic AB. All rights reserved. No part of this document may be reproduced or distributed without the prior written consent of Atollic AB.

TRADEMARK

Atollic, Atollic TrueSTUDIO, Atollic TrueINSPECTOR, Atollic TrueVERIFIER and Atollic TrueANALYZER and the Atollic logotype are trademarks or registered trademarks owned by Atollic. ECLIPSE™ is a registered trademark of the Eclipse foundation. All other product names are trademarks or registered trademarks of their respective owners.

DISCLAIMER

The information in this document is subject to change without notice and does not represent a commitment of Atollic AB. The information contained in this document is assumed to be accurate, but Atollic assumes no responsibility for any errors or omissions. In no event shall Atollic AB, its employees, its contractors, or the authors of this document be liable for any type of damage, losses, costs, charges, claims, demands, claim for lost profits, fees, or expenses of any nature or kind.

DOCUMENT IDENTIFICATION

ASW-WPSVN August 2011

REVISION

First version

Atollic AB

Science Park
Gjuterigatan 7
SE- 553 18 Jönköping
Sweden

+46 (0) 36 19 60 50

E-mail: sales@atollic.com

Web: www.atollic.com

Atollic Inc

115 Route 46
Building F, Suite 1000
Mountain Lakes, NJ 07046-1668
USA

+1 (973) 784 0047 (Voice)

+1 (877) 218 9117 (Toll Free)

+1 (973) 794 0075 (Fax)

E-mail: sales.usa@atollic.com

Web: www.atollic.com

Contents

Abstract	1
Introduction.....	2
Version control using Subversion.....	5
What is Subversion?.....	5
Basic concepts.....	6
The Subversion repository	6
working copies	6
Importing	7
Checking-out	8
Committing	9
Updating.....	10
Revisions	11
Branches & merges	11
Release branches	12
Feature branches	14
Tags	16
Keyword substitution	17
System configurations.....	17
Getting Subversion.....	18
Benefits of IDE integration	19
Connecting to Subversion	19
Accessing subversion	20
Browsing the repository.....	23
Committing code changes.....	25
Browsing file history.....	26
Browsing line history.....	27
The revision graph.....	28

Summary..... 30

Tables

No table of figures entries found.

ABSTRACT

The development of any non-trivial software application involves constant changes to source code, often by more than one developer in a team, until the application is deemed to meet specification. Unless means are employed to track changes in an orderly fashion, chaos is bound to ensue. This will extend development times and ultimately lead to poor software quality.

The way out of this dilemma is to use some kind of revision control methodology. In practice, solutions vary widely from updating comments in code and header modules, to tracking changes in a spreadsheet, to the use of software applications specifically designed to track revisions during development. Version control tools as they are known, offer the most capable solution. The tradeoff in employing this solution is the investment of time needed to learn how to use these kinds of tools effectively versus the benefit derived from their usage. Busy developers must carefully pick and choose how to spend their time in order to satisfy the diverse and competing demands of their work.

The best way to implement version control in an organization, especially when multi-developer teams are involved, is to use one of the excellent version control tools available. This approach confers maximum benefit, flexibility and long range productivity, but at the cost of learning to use the chosen tool effectively. Mastering this kind of tool is greatly simplified by first understanding the basic model upon which version control tools are built, and the terminology involved.

This whitepaper lays out the terminology and principles of version control using the Subversion tool as an example. This will give the reader a solid grounding in fundamentals which will substantially shorten the learning curve. This article also examines further productivity benefits that result from incorporation of a graphical interface to the version control database server directly within the embedded C/C++ integrated development environment.

INTRODUCTION

William Shakespeare once wrote: “many a truth is spoken in jest.” Over the years practitioners of software development have made many serious and whimsical observations on the process. Two particularly relevant quotes to this discussion are:

“Any code of your own that you haven't looked at for six or more months might as well have been written by someone else” - Eagleson's law

and

“You can't have great software without a great team, and most software teams behave like dysfunctional families” - Jim McCarthy

These maxims illustrate two essential points that make the use of version control a necessity instead of a luxury in real software projects.

The first point goes to the fact that sooner or later, developers must go back to old code for maintenance, bug fixing or feature enhancement. The difficulty inherent in understanding legacy code is familiar to anyone who has gone through the process.

Documentation in the form of comments, state chart diagrams, graphics etc. may or may not exist, or be adequate to make the picture clear. Revision control history that tells who made changes, when they made them, and why they were made, is invaluable in gaining the vital insight necessary to effect modifications to an existing code base in a timely fashion.

Jim McCarthy's comment reflects the sad but often true fact that communication among members of a development team may not always be the best. Using version control is a way to naturally enforce good communication and discipline among team members.

When team members use version control, they are communicating with one another every time they perform fundamental operations such as checking in/out, merging, committing and tagging code, creating branches etc. These practices not only help individual team members orient themselves within the landscape of a complex project, they leave behind an irrefutable trail of history and documentation that can be mined for information at any subsequent time.

In the past, many embedded applications consisted of only a few source code modules with a single author. Although this describes many current applications, it is also true that contemporary projects often require hundreds of modules. Some of these are developed in house, some are bought in, some are contributed by contractors, and some are Open Source. Application development nowadays is more software engineering, which involves more a demanding design, integration and testing effort than just writing and debugging code.

As the development progresses over time, it is typical for thousands of code changes to be made. If version control methodology is not used, it very quickly becomes unclear who made what changes, when and why. A bug fix may inadvertently introduce new bugs. As time goes on, valuable information about what the original code base looked like can be lost forever, making it impossible to revert to a previous code state of known working

behavior. In this scenario, teams lose time, unnecessarily duplicate effort, and possibly work at cross purposes. Version control methodology is the key to bringing order to a potentially chaotic situation.

Further complicating the issue is that today's development teams often consist of many developers spread out over different buildings, cities, countries or even continents. Communication may be hindered by time zone differences or business trips that inhibit asking simple questions like "is this bug fixed yet," or "have these changes been made." Version control is good news to such teams, because of they make it easier to obtain this kind of information even when colleagues are unavailable for comment.

Single developer situations are also excellent candidates for using version control. Besides Eagleson's Law, the single developer frequently is a jack-of-all-trades whose responsibilities may include not only software development, but hardware design, parts specification and ordering, purchasing etc. In a small company, the same individual may also be doing the marketing, sales and the financial books. Time is at a premium! Version control methodology helps developers return their focus to a project after the inevitable interruptions. This pays dividends in the long run, preventing procedural errors like forgetting to fix important bugs or unnecessary re-engineering of old code.

Most developers already use version control systems, while some haven't started yet. Arguments against adopting version control methodology frequently involve questioning whether there will be payback on the investment of time to learn such tools. Others contend that less rigorous methods such as regular backups, and/or tracking revisions on spreadsheets are just as effective. Single developers often argue that since they are the only authors and modifiers of the code that version control is unnecessary, because they have it "all in their head."

All of these arguments are of questionable merit. Most professional software development organizations insist on version control methodology supported by tools specifically designed for that purpose as a matter of policy. Using version control methodology is also considered as good software development practice and is usually a required protocol of development for mission critical software such as industrial control, avionics or medical devices. In the past, there have been examples of companies which have faced legal liabilities from their lack of implementation of version control when product defects occurred. These facts are persuasive evidence of the positive cost/benefit ratio of employing a sound version control methodology.

Another factor that inhibits the adoption of version control methodology is the fact that interacting with a version control database server is thought to require a separate application to acquire, install, support, and learn how to use. All of these are true, and the effort to do these things varies considerably.

There is a wide variety of choices of both command line and GUI driven version control clients. Some integrated development tools on the market have limited interfaces to version control database servers. Using some of them gives you the idea that they are "bolted on" rather than designed in from the start. In some cases, these solutions limit access to full version control capabilities, and are less than ideal.

This white paper will describe the benefits of using the Subversion version control system in the context of embedded systems software development. Subversion is an Open Source

program. This document will also discuss the usability and efficiency benefits of deeply integrated Subversion GUI clients, such as that found in **Atollic TrueSTUDIO®**.

The Subversion client that is integrated into **Atollic TrueSTUDIO®** is a state-of-the-art graphical user interface. Its placement within the IDE lends itself to substantial productivity increases over having to leave the development environment, perform routine version control functions with a separate client, and then re-enter the development environment to carry on work.

This rest of this document is divided in two parts:

- Version control using Subversion
- Benefits of IDE integration

The first part provides an overview of benefits using a version control system in general, and how Subversion works in particular. This part is good introductory reading for developers who are new to the concepts of version control systems. Experienced Subversion users may wish to skip the first part, and go directly to the second part and read more about the benefits of deeply integrated Subversion GUI clients in embedded C/C++ IDE's.

VERSION CONTROL USING SUBVERSION

Subversion (also called SVN) is probably the most widely used version control system in the software industry, due to its stability, availability on almost all platforms and rich feature-set. Subversion is Open Source code. This is another factor in its wide acceptance and use in the software industry. Owing to its large user base, it has a noteworthy history of stability and utility. Subversion scales very well and at least one aircraft manufacturer has used Subversion in a 100+ developer mission-critical software project.

Subversion was designed to replace the now aging CVS, which previously was one of the more popular version control systems in the software industry.

WHAT IS SUBVERSION?

Subversion is a version control system, which manages files and folders, and their changes over time. Subversion provides quantitative traceability of all changes in all files, throughout the complete lifetime of the project.

Subversion tracks changes not only in files, but in folders as well. Furthermore, version control methodology is not limited to ASCII-text files such as source code and header files. Any file including binary files like images, audio or even office documents can be put under version control. This is especially important as many of these kinds of files are important to the structure of the application, as well as forming the backbone of application specification and documentation.

Subversion allows you to retrieve older versions of your files for reuse, or to examine the changes made between any two versions of the same file. Subversion traces all changes made to a file, and so you can see how a file changed over time. It is also possible to see who made a change, when and why. If a code change introduced a problem, just revert to the previous stable version to undo the change and continue from there. In essence, version control systems are time machines that let you move backward and forward in time, and see what files and folders looked like at certain times during the project lifecycle.

The Subversion database server is best used by accessing it over a network using a client/server model. This has important advantages. It allows developers on the team to keep all relevant files in a central location that is accessible to everyone. The developers can work on the same source code files simultaneously. More than one developer can make changes to the same file at the same time, and the changes can be subsequently merged. In the rare event that the same line has been modified by two or more developers at the same time, a conflict manager detects the conflict and provides tools to resolve the conflicting code changes.

All the versioned files are stored in a centralized place called the repository. The Subversion repository can be accessed from client computers anywhere on the network. Many developers can work on the same source code files from various geographical locations, as long as the Subversion server repository can be accessed using a LAN or WAN connection. Consequently, it doesn't matter if a development team is in the same place or

spread out over various cities, countries or continents. All developers can have equal access the shared Subversion repository regardless of physical location.

BASIC CONCEPTS

This section provides an overview of how developers typically work with Subversion, and covers important concepts like the Subversion repository, local working copies, checking out and committing code, etc.

THE SUBVERSION REPOSITORY

Subversion stores all its data which consists of files and their change histories in a centralized place called the repository. The repository is typically stored on a network server devoted to the needs of the software development team. With suitable security credentials, any number of developers can connect to the repository on the server and access the files and folders stored there using a Subversion client.

The main repository concepts are:

- The repository contains a file tree with all the files and folders in your project. As such, the repository is your centralized safe deposit box where the master copy of all your source code files and possibly supporting files and documents, is stored.
- By writing to a file in the repository, Subversion remembers the previous state of the affected file, and then updates the file, causing your changes to immediately become available to other developers in the team.
- Reading from a file in the repository delivers the latest version of the file, including all the latest changes made by other developers.

The critical difference between Subversion and an ordinary file server which also stores a file tree, is that Subversion remembers every change to every file that has ever been made. It remembers every change to folders as well, such as creating, renaming or deleting files, or just moving them around in the folder structure.

When reading files from the repository, the default behavior is to receive the latest version of the file or files you extracted. Subversion clients also have the ability to retrieve any earlier version as well. This enables Subversion to answer questions like “What did this file look like last week?” or “Who made the last change to this file, and what changes were made?” The Subversion repository with its full version tracking capability provides traceability of every change in every file and folder that has ever been made.

WORKING COPIES

A Subversion working copy is a file tree from the repository that has been replicated on the local computer. It contains all or parts of the files in the project, and can be regarded as the private, temporary workspace of the developer who owns the drive where it exists.

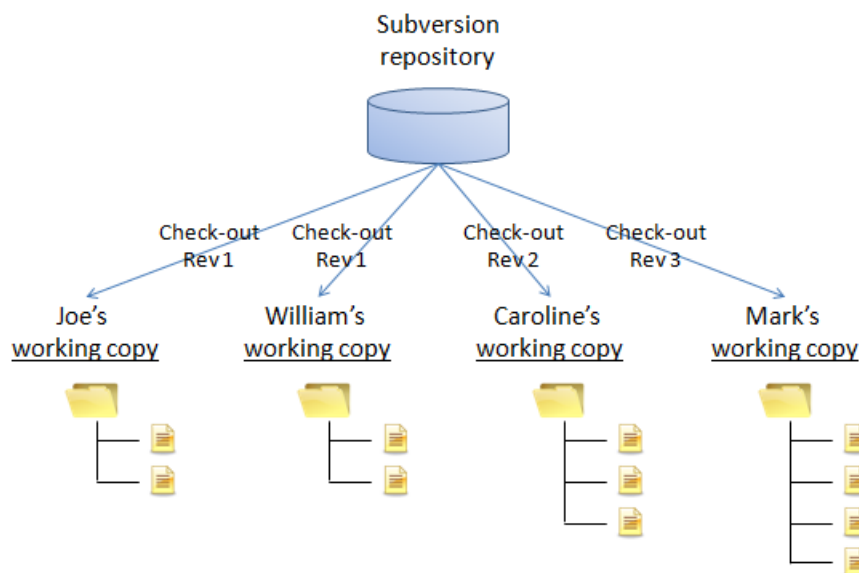


Figure 1 - Working copies in Subversion

Because the working copy is your private work area, you can edit the source code as you like, compile/debug/test, make further changes etc, without your work affecting other developers in the project. Other developers cannot see any of your changes as long as they are only stored in your local working copy.

A developer can have any number of working copies stored on various places on the local hard drive. There are several reasons why you might want to have several working copies active at the same time:

- It is possible to work on several new features or bug fixes in parallel, and not let the work on one of them interfere with the work on the other ones. The new features can then be committed to the repository individually from the various working copies and at different times. Subversion tracks everything properly even though work has been done in parallel a period of time.
- While currently working on the latest code version, there may be a reason to access an earlier version to check its source code implementation or compile and run it to examine its runtime behavior. In this case, it is desirable to check-out a new working copy of the earlier version from the repository. This has the advantage of not affecting the work being done on the latest version in the current working copy.

A working copy is thus an isolated, private area where a developer can work on the source code of a particular version without affecting other developers or versions.

IMPORTING

When a new repository has been created using the subversion administrative tools, you may want to fill it with a file tree that already exists outside of Subversion. For example,

this is the case if there is an existing project being worked on that a developer wants to move into Subversion for version control management going forward.

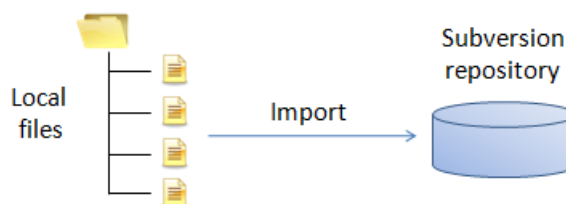


Figure 2 - Importing into Subversion

Importing an arbitrarily complex file tree into Subversion can be done using the import operation. Once the repository is filled with initial files and folders, team members can “check-out” the files into their local working copies and start to work with them under the control of Subversion.

CHECKING-OUT

A working copy is created and filled with copies of files from the Subversion repository by performing a check-out operation. When you check-out all or parts of the repository file tree, a working copy is created somewhere on your local drive.

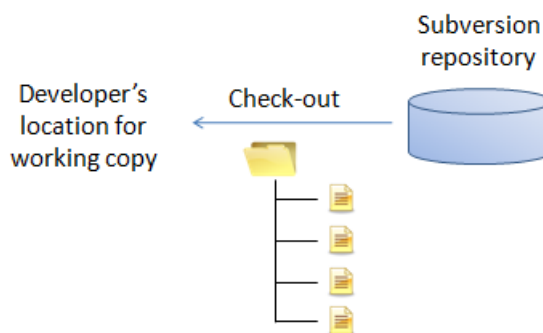


Figure 3 - Checking out from Subversion

The checked-out files are local copies stored in a local and private working copy. Any changes made to the local working copy do not affect any files in the repository until you explicitly perform a “commit” operation.

As the illustration below shows, work done in the local working copy is done in parallel to, and separated from, the central repository.

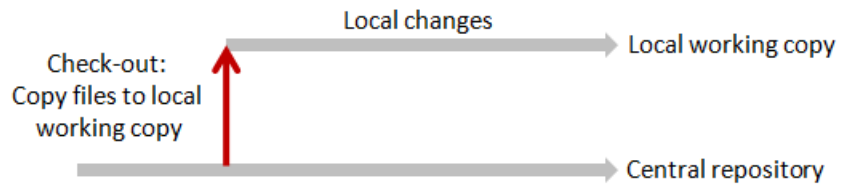


Figure 4 – The repository versus the working copy in Subversion

You can check-out different parts of the repository file tree to different working copies as you see fit. You can also check out different versions of files or folders in the repository to different working copies. This enables you to have many working copies on your computer, for the different purposes described above.

COMMITTING

After changes are made to files in the local working copy, you will want to update the repository with your changes to make them part of the latest “official” code base (often called the “mainline” or “trunk”). This enables other team members to obtain access to these changes as well. This is performed by “committing” your local working copy changes to the repository.

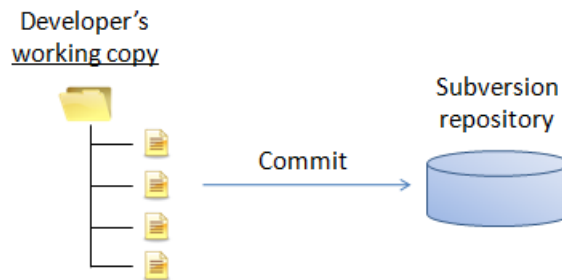


Figure 5 - Committing in Subversion

By doing so, the changes made in your local working copy are copied to the Subversion repository on the server. Technically, your local changes are automatically merged into the latest code in the repository. In effect, you have published your changes enabling other team members to access those changes by reading the most recent files from the Subversion repository to their local working copies.

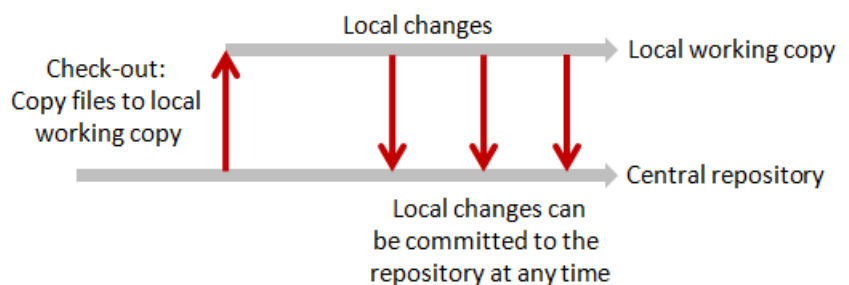


Figure 6 - Committing in Subversion

When changes are committed from your local working copy to the repository, this is done as a single atomic operation. This means that all changes are written to the repository, or in the case of problems, no change will be written to the repository. This is important, as it ensures that subsets of the revision will not be sent to the repository which would cause an inconsistent update.

UPDATING

If you have been doing work in a local working copy for some time, other developers might have done some of their own code changes in the meantime that are now available in the Subversion repository. If desired, you may merge the latest changes from the repository into your local working copy using the update operation.

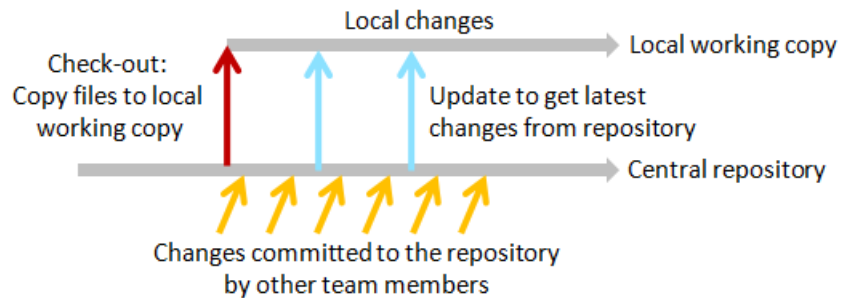


Figure 7 - Updating in Subversion

Your local working copy is thus updated with the most recent work done by other team members. It is good practice to merge the latest changes from the repository to your local working copy, and test compile it before committing your changes to the repository.

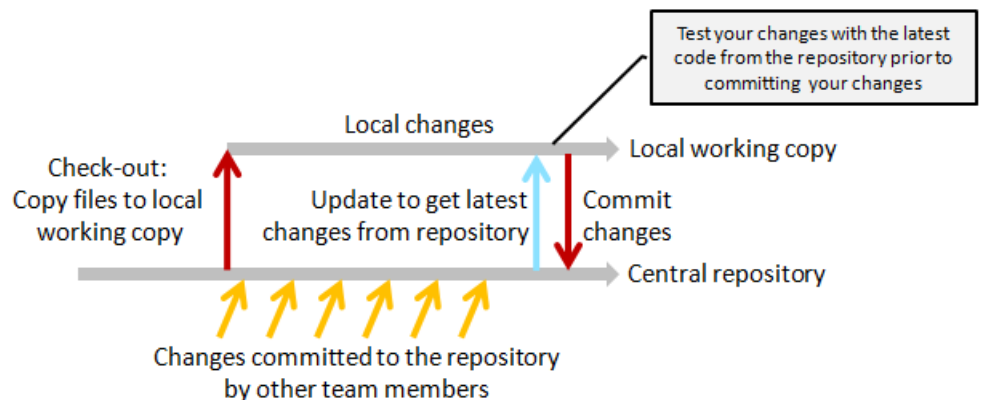


Figure 8 - Updating in Subversion

This ensures that your latest changes do not cause problems with changes other developers have recently committed to the repository.

REVISIONS

Each commit operation creates a new state of the repository file tree, which is called a revision. A newly created repository has revision number 0 (because 0 commits have been performed yet).

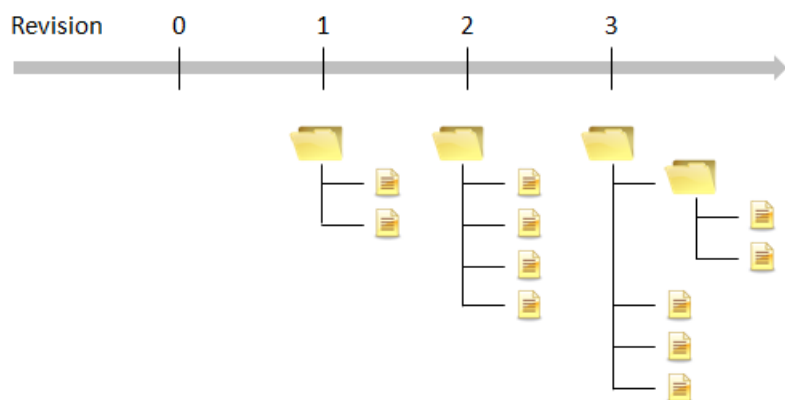


Figure 9 - Revisions in Subversion

Whenever a commit operation is performed, the revision number is increased by one, reflecting the new state of the repository file. This can be illustrated as a sequence of revision numbers, each with a repository file tree state connected to it.

BRANCHES & MERGES

Basic use of Subversion provides many benefits in terms of team collaboration, change traceability and possibilities of “going back in time.” The concept of branching and merging provides additional benefits as will be seen.

When you work with a Subversion repository, you normally start to work in the mainline or “trunk”. This is where all code changes (commits) are saved by default, as explained previously. When working only with the trunk, all committed changes are in the same place in the repository. Obtaining the latest changes from the repository using the check-out operation gives you all the code from the previous commits. This however also means that all changes are committed into the same line of development, causing problems in certain scenarios.

Branching and merging are operations that support the concept of parallel, independent, development of the same code base, i.e., development can continue in parallel lines of development on the same code base without interfering with each other.

A branch is basically a copy of the repository state in the mainline or trunk made at a specific time. Development can then be performed either in the mainline part of the repository, or in the branch part of the repository. Code changes committed to the trunk do not interfere with code changes committed in the branch and vice-versa.

When a branch is created, effectively two parallel and independent copies of the same repository file tree have been created. Development can then continue on either of the two without interference to the other.

Why would one want to copy the code in the repository into a parallel, independent, development thread? There are two common scenarios when this is useful:

- Release branches
- Feature branches

The sections below outline the merits of using branching and merging in these two scenarios.

RELEASE BRANCHES

Release branches are used when work on multiple releases of the same software product needs to be done in parallel. Suppose for example that version 1.0 of a software product was previously released. Immediately after the version 1.0 release was made, work started on the upcoming version 2.0 release, wherein a lot of feature additions are being committed into the trunk of the repository. While work on the version 2.0 is in progress, critical bugs are reported in the previously released version 1.0. This presents a dilemma as the version 2.0 release is months away, and a bug fix release of version 1.0 is needed immediately.

Waiting for v2.0 is not an option in this case. Taking a snapshot of the half-ready version 2.0 and releasing it with the bug fix as a version 1.1 release would not be practical, as the new features being implemented for version 2.0 are not stable. The solution is to move back in time, and make a copy of the version 1.0 release and only do the needed bug fixing in the copy of version 1.0, without adding any new features. The bug fixed version 1.0 can then be released as version 1.0.1 independently on the work progressing on the upcoming version 2.0 release.

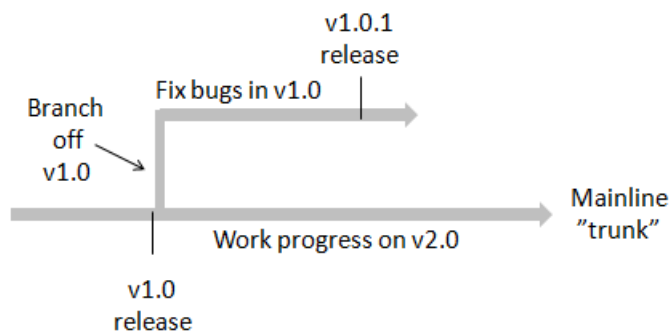


Figure 10 - Release branches in Subversion

This is an example of a release branch. Work on the upcoming version 2.0 can progress in parallel to bug fixing of older versions. Code changes being committed for either project do not interfere with each other since bug fixes are committed into the branch while version 2.0 changes are committed to the trunk.

Now, consider the fate of the bug fixes implemented for version 1.0.1. These should not be abandoned, as they will probably need to be incorporated into version 2.0. Otherwise version 2.0 will inherit unfixed bug from version 1.0. This is where merging comes into play.

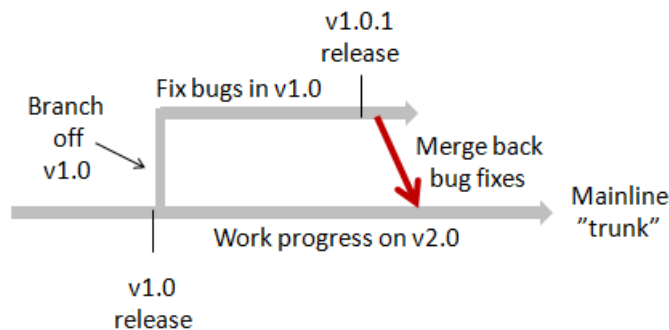


Figure 11 - Release branches in Subversion

This situation is avoided by merging the changes made in the release branch that went into version 1.0.1 into the trunk. This incorporates the bug fixes made in the v1.0.1 release into the v2.0 development thread. If it develops that additional fixes are found in v1.0.1 which necessitate the creation of a version 1.0.2, then we are already set to handle this. Continue to work in the v1.0.1 branch and fix the bugs for version 1.0.2, and then merge those changes back into the trunk so that they will be eventually incorporated into the version 2.0 release as well.

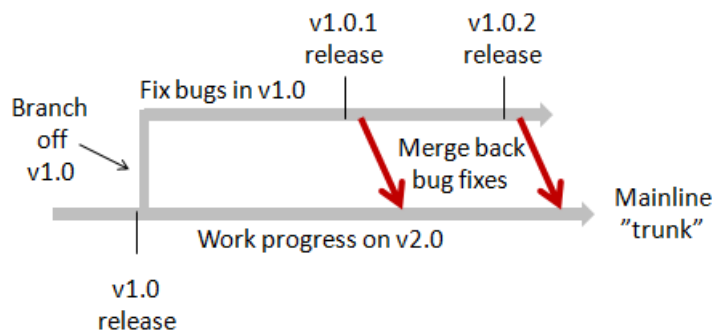


Figure 12 - Release branches in Subversion

A variant to the above scenario occurs when the next upcoming release is branched off from the mainline development when all features are completed, but before the testing and release is made.

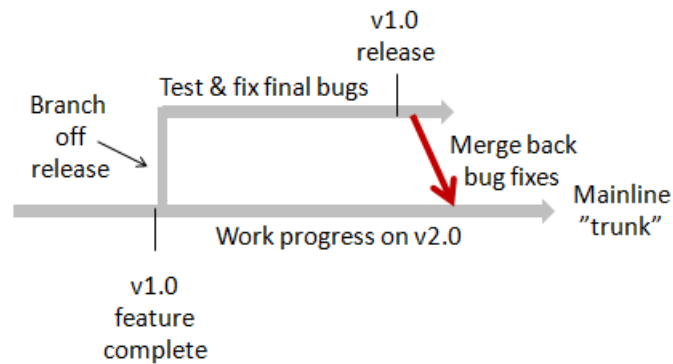


Figure 13 - Release branches in Subversion

This enables the majority of the team to get a head start on working on version 2.0, the next major release, while the smaller testing and release team can make minor bug fixes found during final testing in a separate branch before release. Once the release of version 1.0 is made, any final modifications made during final test and release, are merged back to the mainline development to make it up-to-date too.

FEATURE BRANCHES

Feature branches are used in a different scenario compared to release branches. A feature branch is created when complex work needs to be done in parallel to other development work. This for example can happen when adding a major new feature will destabilize the trunk for a long time, thus creating problems for other, unrelated, development work.

By making the new feature in a branch of its own, complex development can be performed in an isolated sandbox, thus ensuring that its development does not interfere with other development, or vice versa.

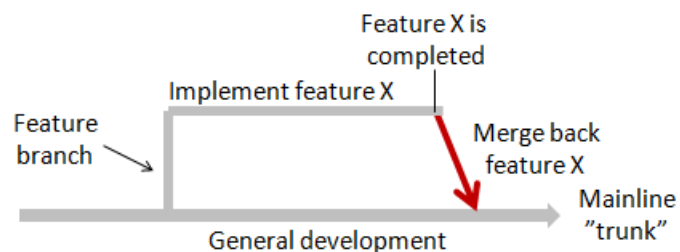


Figure 14 - Feature branches in Subversion

Most companies want to ensure that the code in the mainline trunk compiles and is stable at least daily. If changes to be committed are large enough to make the mainline trunk unstable and impossible to compile for days, then such development should probably be performed in a parallel feature branch instead to ensure the mainline trunk is stable and can be compiled at any given time.

However a problem can arise if many changes are being committed to the mainline trunk. Then the code base in the feature branch and the mainline trunk can diverge in such a way that it becomes difficult to merge the feature branch back into the mainline trunk again.

To solve this problem, the mainline trunk should be merged into the feature branch sufficiently often, to ensure that the feature branch is kept up-to-date with other code changes from the mainline.

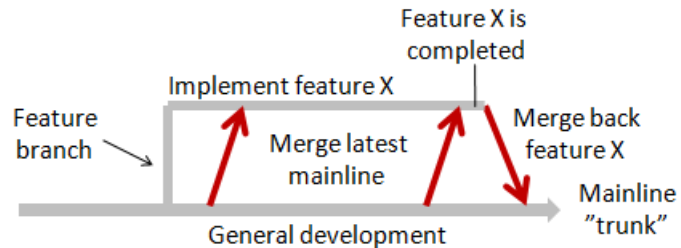


Figure 15 - Feature branches in Subversion

Finally, when the development is completed in the feature branch, it is merged back into the mainline trunk and the branch is deleted. The mainline will then contain all the changes made in both the feature branch and the mainline itself. In other words, there is a completely new version of the trunk.

In large software projects, it is not uncommon to have multiple feature branches in parallel development at the same time, where separate teams work on specific implementations of separate features. Feature branches are a useful mechanism for minimizing the risk that the work done by one team interferes with the work of another.

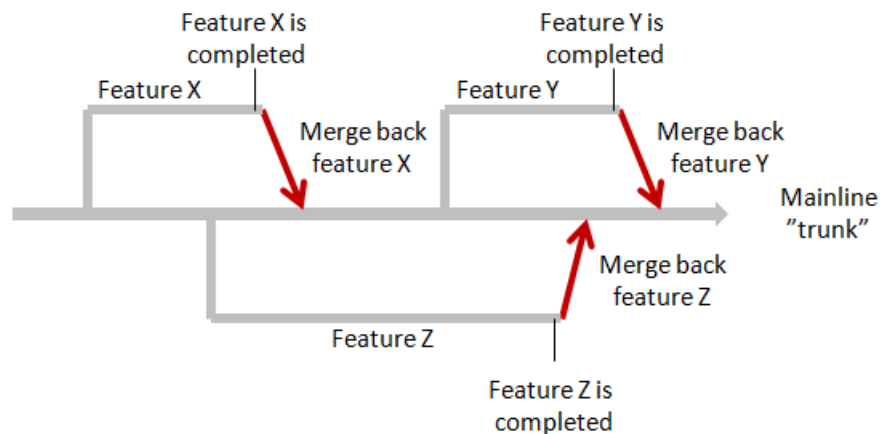


Figure 16 - Feature branches in Subversion

As mentioned, it is recommended that the latest code changes from the trunk is merged into the feature branches often, to make sure the code changes implemented in the branch actually works with the latest developments being performed in the trunk.

By separating major feature development into their own branches by using parallel feature branches, complex changes to a version of the application can be performed in a systematic and orderly fashion without worrying about unwanted interaction or impact on or from other teams.

TAGS

An important concept in version control systems is that of tags. Those who understand the concept of DNS in networking will find a ready analogy to tags. The idea behind DNS is to provide a way for human beings to use familiar terms when asking for network locations. It is much easier to remember www.google.com than to ask for it by its IP address of 74.125.91.103.

By analogy, a tag is a symbolic name for the source code state in the repository at a specific revision number. A tag is a named snapshot of the project at a specific time. For example, a tag can be created on every release, every customer delivery, for every code review meeting, etc. The main purpose of tags is to identify the repository state at a specific time by giving it a symbolic name for later reference by that name. Tags simplify retrieving the state of the repository at a specific revision number, as you can refer to the tag name instead of the revision number when you want to check-out exactly the same software configuration at a later time.

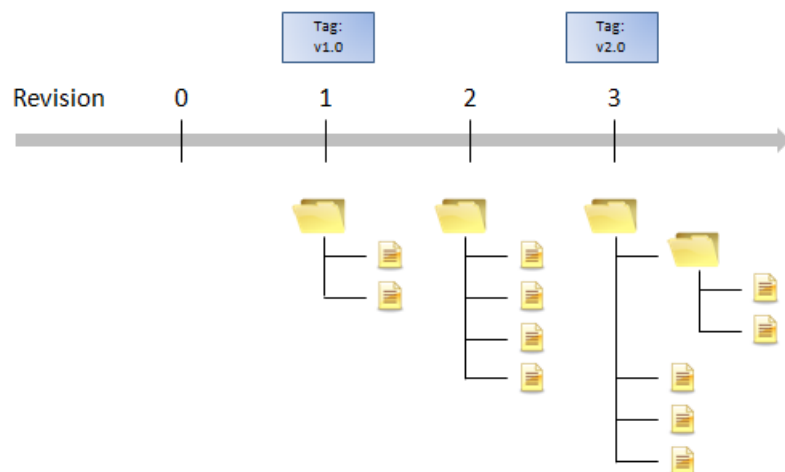


Figure 17 - Tags in Subversion

For example, the revision number when making a BETA release of some software product might be 2577. So a tag can be created with the name “v2.1 BETA”, which is easier to remember than revision number 2577. Both are the same thing. Tags and revision numbers are merely two different ways of referring to a specific state of the source code tree in the repository; a numeric and a symbolic.

It is good practice to create a tag with a descriptive name on each important repository state, such as on each Alpha and Beta version, each release candidate and the final release of each product version.

Note: Tags are called “labels” in some version control systems.

KEYWORD SUBSTITUTION

Certain keywords can be entered as placeholders into project files and Subversion can automatically substitute dynamically generated “live data” for the keywords.

Assume for example that a file header comment in a C file looks like this:

```
/*  
Revision:  
$Rev$:  
  
Author:  
$Author$:  
  
Date:  
$Date$:  
*/
```

When you commit this file to the repository, these keywords will be replaced with dynamically generated “live data”:

```
/*  
Revision:  
$Rev: 67 $:  
  
Author:  
$Author: mark $:  
  
Date:  
$Date: 2011-01-15 08:42:00 -0400 (Sat, 15 Jan 2011) $:  
*/
```

When you later open the files in your source code editor, the file headers thus contain dynamically updated information on what revision this file is at, who made the last change and when, etc. This is a boon to anyone who has tried to update comments by hand in modules that contain this kind of information.

SYSTEM CONFIGURATIONS

Most version control systems, including Subversion, are built around the client/server model. The Subversion server software is typically installed on a team server, and Subversion clients can then be used on developer PC's to interact with the version control system on the server

However, using version control system software like Subversion is equally important if you are the only developer in the project. In such a scenario, the Subversion server software can be installed in the client PC (typically the developer's laptop), and thus run in the same computer as the Subversion client software.

Subversion also includes a special local single-user mode that works completely without the server software. In that mode, the Subversion client operates directly on the local repository without any need for network server hardware and database server software.

You can thus connect to a Subversion repository in three different ways:

- Multi-user mode with remote server software. The Subversion client connects to the Subversion server software over a network. Subversion acts as both a team collaboration and a code management tool.
- Single-user mode with local server software. The Subversion client connects to the Subversion server software running on the same computer. Subversion acts as a code management tool.
- Local single-user mode without server software. The Subversion client connects directly to the local Subversion repository without any need to install the server software. Subversion acts as a code management tool.

The Subversion server runs as a GUI-less background daemon/service (either on a team server or on a single developer's computer). The server-side is controlled and configured using command line tools and configuration files.

The default Subversion client is command line based, using the **svn** command with various options. However, there are many alternative Subversion clients available, including GUI based ones.

This white paper will provide an overview of the benefits of using the **Atollic TrueSTUDIO®** C/C++ embedded development IDE as a deeply integrated Subversion client, for use in embedded development.

GETTING SUBVERSION

The Subversion software can be downloaded here:

<http://subversion.apache.org>.

The Subversion manual is available here:

<http://svnbook.red-bean.com>.

Many books on Subversion are available from various publishers too.

BENEFITS OF IDE INTEGRATION

Even though all Subversion features can be accessed using the default **svn** command line client, or some other external GUI client, this is not optimal for daily use. The reason is that anything that causes the developer to shell out of the development environment to accomplish a task that is properly considered as integral to application development is distracting, causes loss of focus, and ultimately, loss of productivity. For efficient use, your embedded systems C/C++ IDE should include a deeply integrated Subversion GUI client. In this setup, common version control system operations are a part of the critical workflow, not a diversion.

This section outlines how Subversion can easily be accessed from within the **Atollic TrueSTUDIO®** C/C++ embedded development IDE, using an easy-to-use yet powerful graphical user interface.

CONNECTING TO SUBVERSION

Before the **Atollic TrueSTUDIO®** IDE can integrate with a Subversion repository, the repository must first be created on the server side. This is done using the Subversion administrative server tools. Additionally, the IDE must connect to the Subversion repository using proper network settings and security credentials. Normally, IT staff within a company provides this kind of service.

Connecting to Subversion repositories is done using the **SVN Repositories** view. This view is opened by using the menu command **Windows->Show view->Other**. Then expand the SVN folder to display various options.

Double-click on the **SVN Repositories** view to open it:

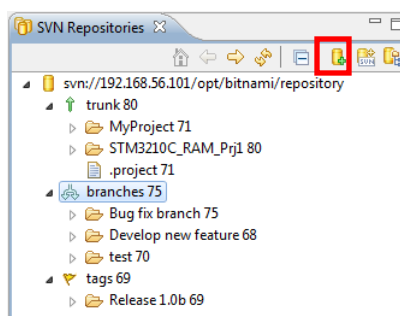


Figure 18 – The Atollic TrueSTUDIO® Subversion repositories view

If the IDE is already connected to one or more repositories, they are listed in the view and you can drill down and see the repository structure (projects and their files, branches, tags, etc).

If you want to connect to one more Subversion repository, click on the toolbar button with the green “+” sign. A connection dialog box is now opened and you can enter the network connection information and security credentials for your Subversion repository:

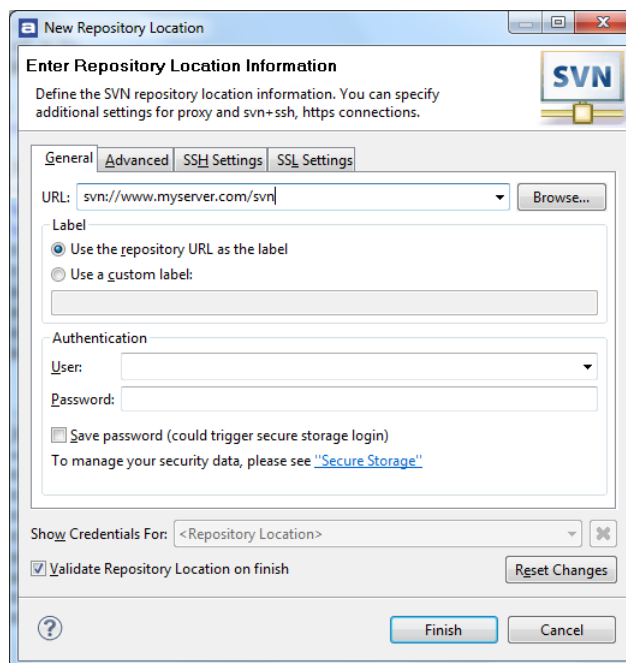


Figure 19 - The Atollic TrueSTUDIO® Subversion repository configuration

Additional tabs provide facilities for detailed configuration, in terms of Subversion usage and SSH or SSL security settings.

ACCESSING SUBVERSION

Once the IDE is connected to a repository, we can start working with it. To avoid information overload, *Atollic TrueSTUDIO®* has many menu options, toolbars and docking views that are hidden by default. This is also the case with the Subversion GUI client.

By default, Subversion can only be accessed by right-clicking on a file or folder in the **Project Explorer** view, and then select **Team** in the context menu. A Subversion sub-menu is then displayed (if the IDE is connected to a Subversion repository):

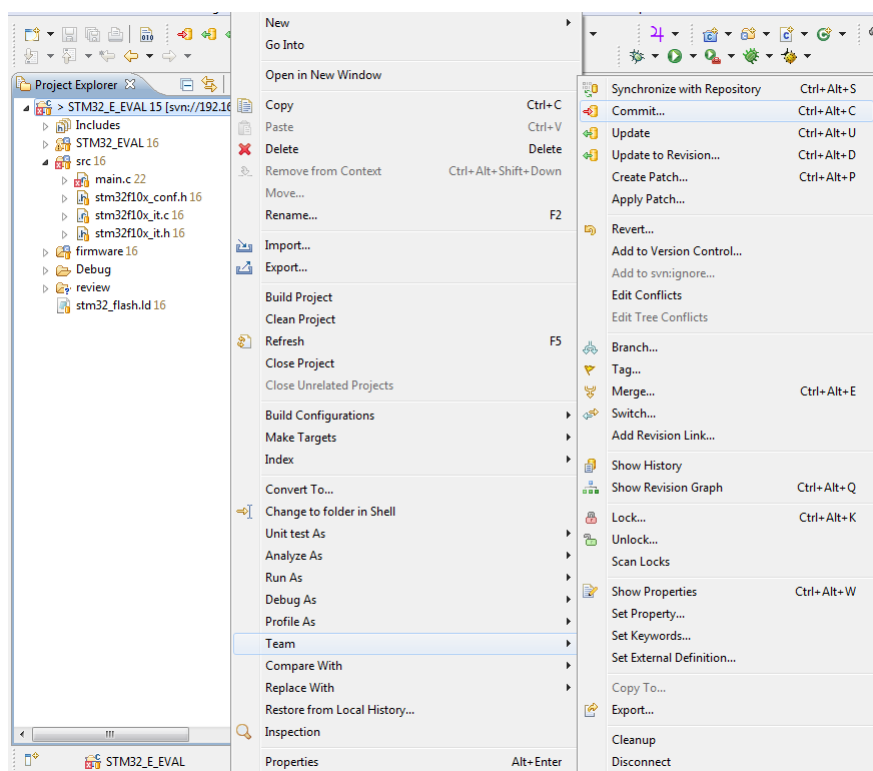


Figure 20 - Accessing Subversion from the Atollic TrueSTUDIO® project explorer view

Using this menu, you can access most of the client-side features of Subversion. But before using Subversion from within *Atollic TrueSTUDIO®*, you may want to enable two alternative ways of accessing Subversion:

- The Subversion toolbar
- The Subversion menu

Use the menu command **Window->Customize->Perspective** to open the perspective customization dialog box. Open the **Command Groups Availability** tab, and select the **SVN** checkbox. Click **OK** to save the changes.

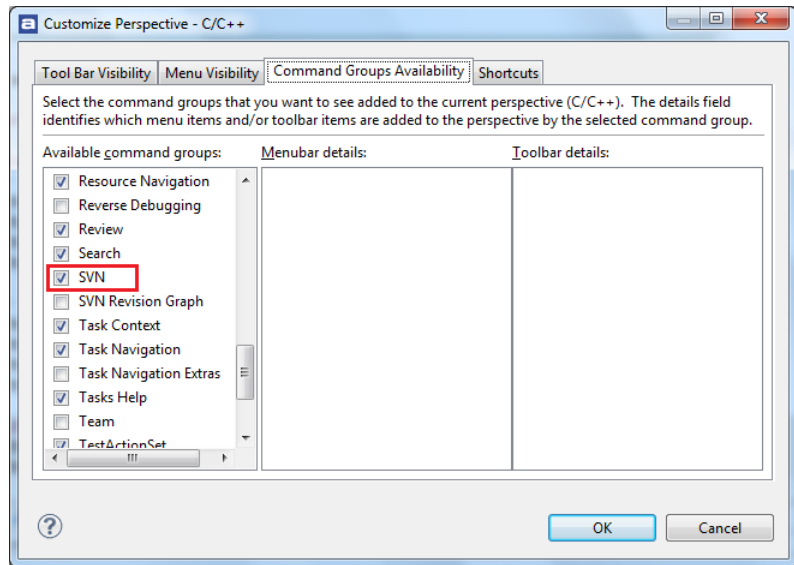


Figure 21 - Configuring Atollic TrueSTUDIO® for the Subversion menu and toolbar

An SVN toolbar and an SVN menu will now be displayed in the IDE. The toolbar provides quick access to the most commonly used features, and the menu contains menu commands for most of the Subversion client-side features:

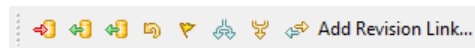


Figure 22 - The Atollic TrueSTUDIO® Subversion toolbar

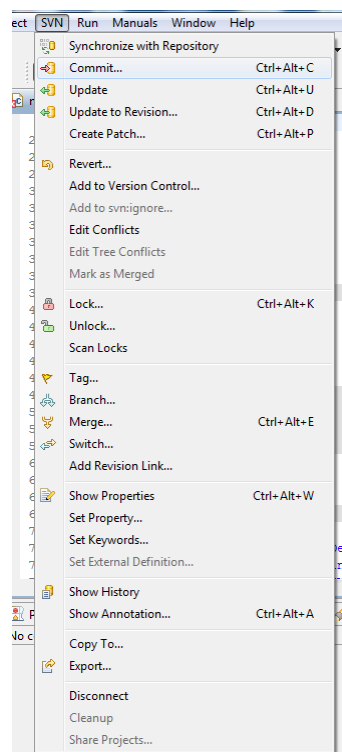


Figure 23 - The Atollic TrueSTUDIO® Subversion menu

Once you can access the Subversion GUI client from any of the 3 ways explained above, you can start to use the features.

BROWSING THE REPOSITORY

You can use the repository browser to inspect the state of centralized repository. To open the repository browser, use the menu command **Windows->Show view->Other...** and open the **SVN** category:

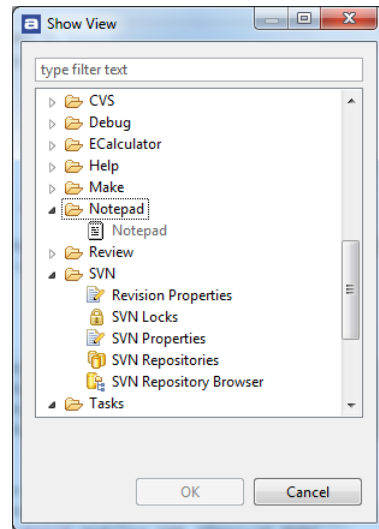


Figure 24 - The Atollic TrueSTUDIO® view selection dialog box

Double-click on the **SVN Repository Browser** view to open it:

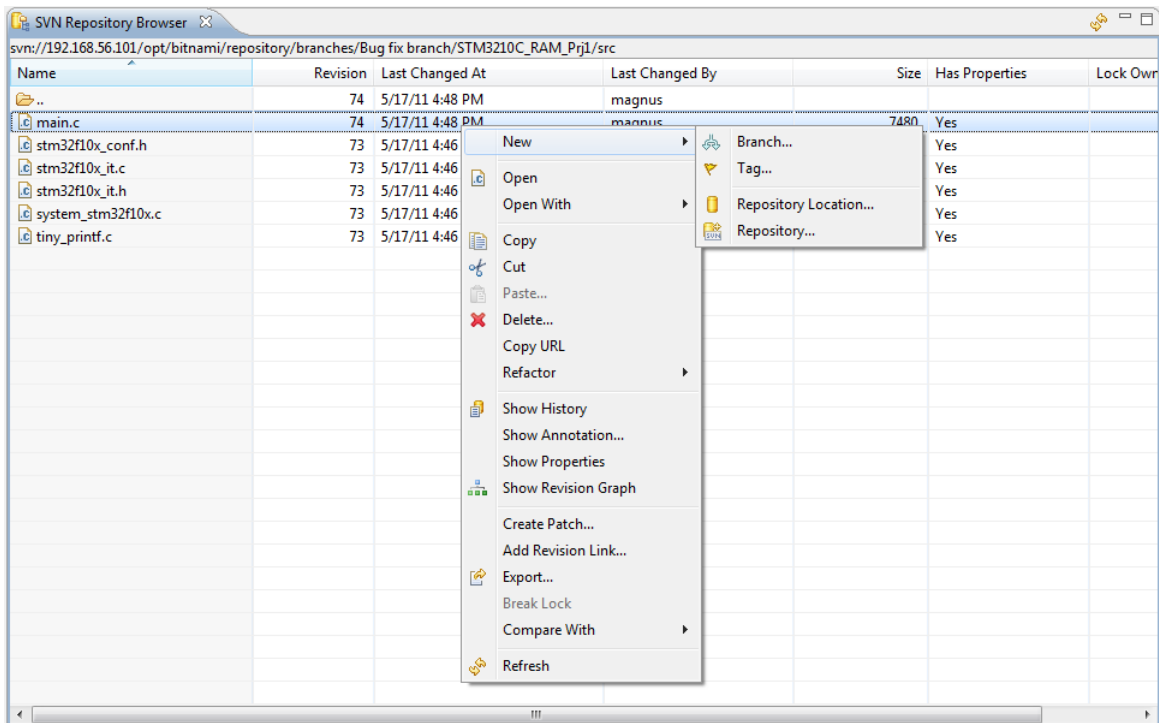


Figure 25 - The Atollic TrueSTUDIO® Subversion repository browser

You can navigate across the files and folders in the repository, and drill down to any level. Right-clicking on any resource brings up a context menu with relevant Subversion operations that can be applied to the selected resource in the repository.

COMMITTING CODE CHANGES

You can commit changes to one or more files using the Commit command. Before the files with changes are committed, the Subversion client display a dialog box where you can enter a commit comment, that is recorded for the future to explain the purpose of this code change.

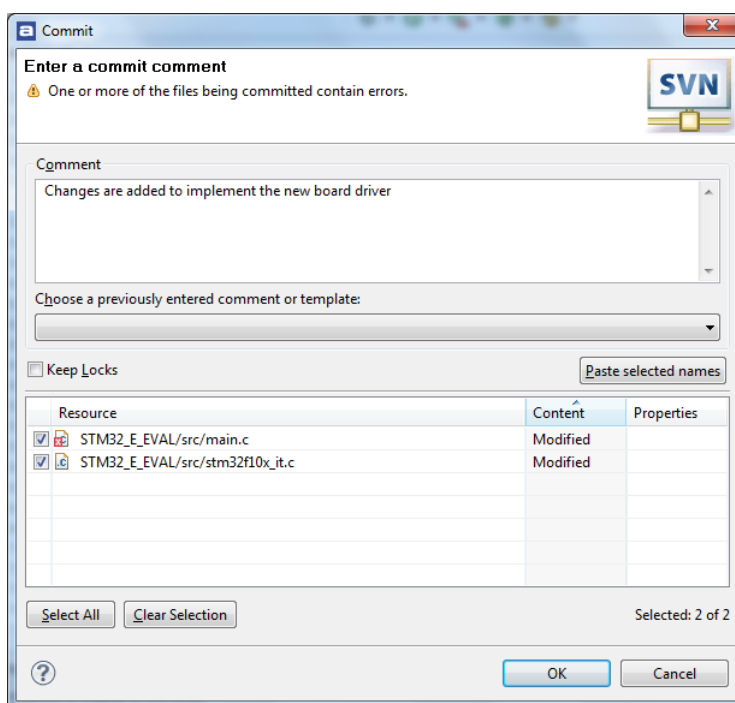


Figure 26 - The Atollic TrueSTUDIO® Subversion commit dialog box

Note that the top of the dialog box informs about the fact that one or more files to be committed have compiler warnings or errors! Additionally, the list of files to be committed is listed. By double clicking on any of the files, a graphical file difference utility is displayed to visualize the changes being committed to the repository.

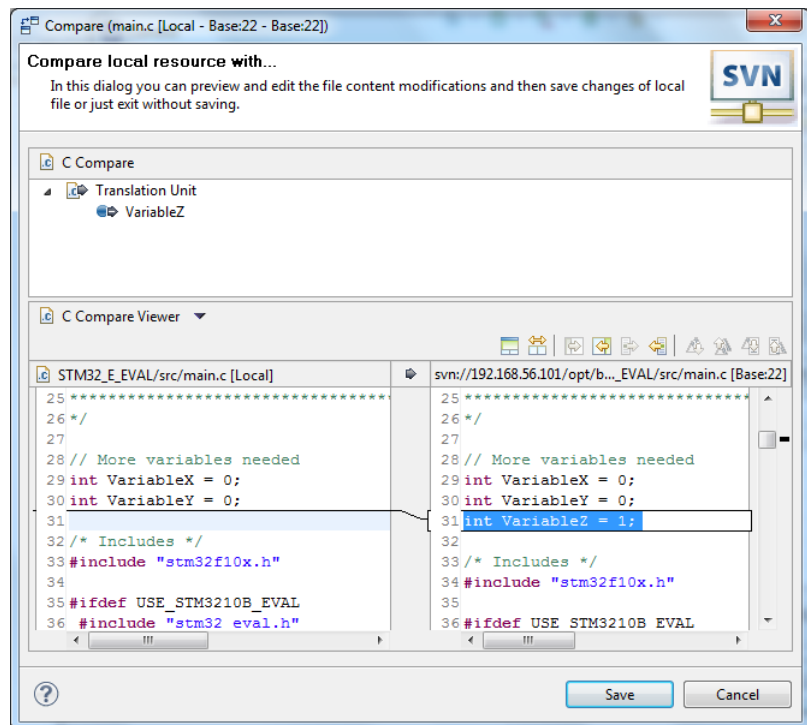


Figure 27 - The Atollic TrueSTUDIO® Subversion graphical file difference utility

Opening the file difference utility for any of the files to be committed not only shows differences, but provides a final possibility to review the code changes that are about to send into the central repository. This shows the utility of having the GUI for version control embedded within the C/C++ IDE, as necessary changes can be done immediately and without distraction.

BROWSING FILE HISTORY

The change history of any resource such as a file or a folder can be displayed using the **Show History** command. You can browse all the revisions (commit operations) that have modified this file, along with information on when it was made, by whom and for what purpose, by reviewing the commit comment. You can see what other files were committed at the same time for each revision as well.

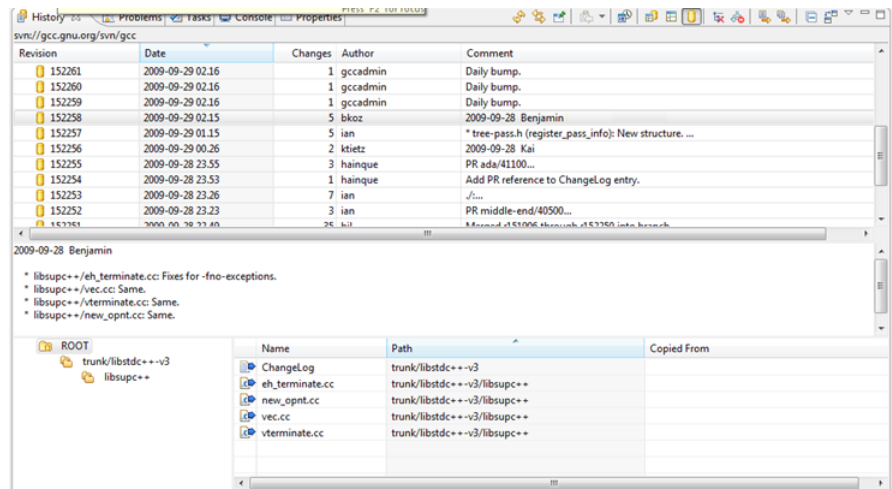


Figure 28- The Atollic TrueSTUDIO® Subversion history browser view

By selecting any two revisions of this file in the revision list, you can see a graphical file difference utility that visualizes how the selected two versions of the same file differ.

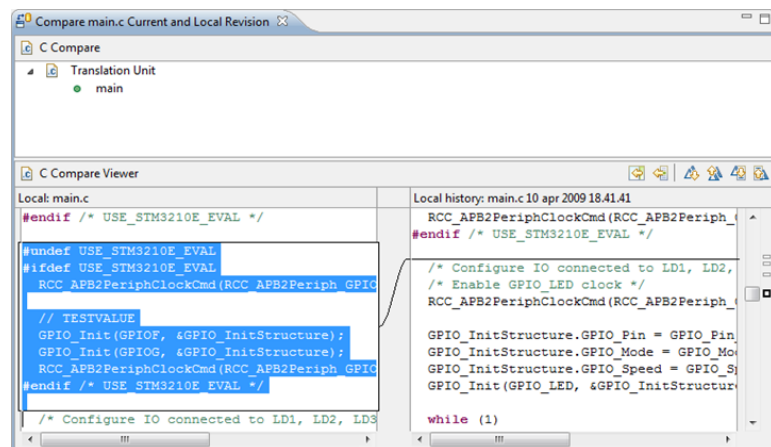


Figure 29 - The Atollic TrueSTUDIO® Subversion graphical file difference viewer

BROWSING LINE HISTORY

By selecting a file, you can get traceability of every line in that file by using the **Show Annotation** command. This opens the selected file in the editor, with a special color coded left-margin.

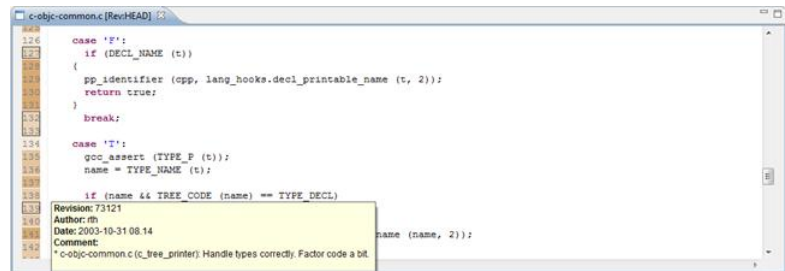


Figure 30 - The Atollic TrueSTUDIO® Subversion annotate viewer

The purpose of the **Annotation** view is to visualize when lines of code in that file were added or changed, by whom it was done and for what reason. Every line in the editor that has the same color in the left margin was committed to the repository at the same time (by the same commit operation), i.e. they were part in the same code change.

By studying the color coding of the left margin, it is thus possible to see what lines were added at the same time, or at different times. By moving the mouse cursor over the left margin, a tooltip display information about when that line, and all other lines with the same color, was added or modified, by whom, and for what purpose.

The annotation view thus shows 100% individual traceability of how every line of code in the selected file was entered into the repository.

THE REVISION GRAPH

While it is possible to browse the repository in textual form using other GUI views, the Subversion revision graph provides an exceptionally simple and informative way of getting an overview of the code development in the repository by means of a graphical chart.

Use the **Show Revision Graph** command to open the Subversion revision graph.

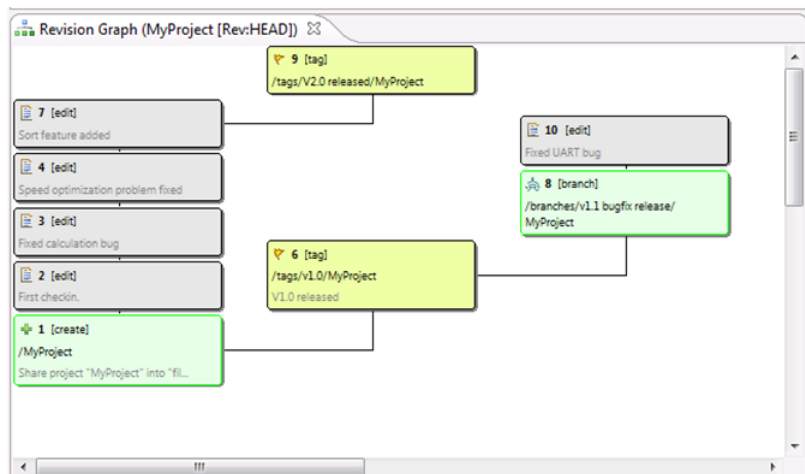


Figure 31 - The Atollic TrueSTUDIO® Subversion revision graph viewer

The revision graph provides a graphical overview of all commits, branches and merges, and tags. It is possible to filter the graph and showing only branches/merges and tags, thus not cluttering the graph with the entire history of commits. This allows the developer to focus on the structural changes to the repository, while retaining the option of examining further details of the change history as the situation demands.

By right-clicking on any of the graphical objects in the graph, a context sensitive menu is displayed where many Subversion operations can be performed right in the graph.

SUMMARY

As complexity and code size grow for each year, so does the problem of managing software and development efforts. Version control systems, like Subversion, address this problem, providing features for traceability, team collaboration and code management.

New embedded tools, like **Atollic TrueSTUDIO®**, integrates code management features right into the C/C++ development environment and provides a deeply integrated GUI client for Subversion, thus adding features for professional code management.

Atollic provides a family of well integrated tools for professional embedded systems development and debugging, team collaboration, static source code analysis, test automation and test quality measurement.

More information about Atollic, **Atollic TrueSTUDIO®**, **Atollic TrueINSPECTOR®**, **Atollic TrueANALYZER®** and **Atollic TrueVERIFIER™** products is available here:

www.atollic.com

www.atollic.com/truestudio

www.atollic.com/trueinspector

www.atollic.com/trueanalyzer

www.atollic.com/trueverifier