XPC C/C++ Unit-Test Libraries User Manual

1.1.2

Generated by Doxygen 1.8.9.1

Sat Oct 10 2015 12:02:24

Contents

Chapter 1

The XPC Suite with Mingw, Windows, and pthreads-w32

Author

Chris Ahlstrom

Date

2012-01-09

Last Edit:

2015-10-05

License:

$XPC_SUITE_GPL_LICENSE$ (See xpc_mingw_license_subproject)

Chapter 2

Licenses, XPC Library Suite

Library:

xpc_suite and its libraries, applications, and documents

Author

Chris Ahlstrom

Date

2008-02-24

Last Edit:

2015-10-05

License:

$XPC_SUITE_GPL_LICENSE$

This module provides the license text for the XPC library suite.

Chapter 3

GNU Automake

This section describes the GNU automake setup for the XPCCUT libraries and test applications.

3.1 GNU Automake

GNU *automake* is a tool for describing project via a simplified set of Makefile templates arranged in a recursive (hierarchical) fashion. *Automake* provides these benefits:

- Shorter, simpler `makefiles` (really!).
- Support for a large number of targets, with almost no effort.
- The ability to configure the files for different development platforms.

Although the term "simplified" is used, *automake* is still complex. However, it is much less complex than writing a bare `makefile` to do the same things an *automake* setup will do. The *automake* setup will do *a lot* more.

Other systems have come along because of the complexity of *automake*. A good example is *cmake* (`http://www.cmake.org/HTML/index.html`). Another good example is *Boost.Build* (`http://www.boost.org/boost-build2/`). However, these systems seem to end up being just as complex as *automake*, since they, too, need to handle the multifarious problems of multi-platform development. We really wanted to transition to *Boost.Build*, but couldn't get it to handle *doxygen* in the way we wanted, even though that product is supposed to be supported.

The present document assumes that you are familiar with *automake* and have used it in projects. The discussion is geared towards what features *automake* supports in the **XPC** projects.

3.2 The 'bootstrap' Script

First, note that one can run the following command to jump right into using the `bootstrap` script:

```
$ ./bootstrap --help
```

Each **XPC** sub-project used to have its own `bootstrap` script, but maintaining them was too difficult. So, for **XPC 1.1**, there is only one `bootstrap` script. In addition, we've off-loaded some of its functionality to a more flexible `build` script.

The **XPC** project provides a `bootstrap` script to handle some common tasks. This script makes it easy to start a project almost from scratch, configure it (if desired), clean it thoroughly when done, pack it up into a tarball, and prepare it for making.

The `bootstrap` script does a lot of what one has to do to set up a project. Thus, it also serves as a description of the operations required in setting up almost any **GNU** *automake* project.

The `bootstrap` script also does a lot more cleanup than the various "make clean" commands provided by *automake*.

One reason the `bootstrap` command does so much is that, even at this late date, we are *still* finding that there are features of *automake* that we don't know how to use.

When run by itself, `bootstrap` first creates all of the directories needed for the project:

```
$ mkdir -p aux-files
$ mkdir -p config
$ mkdir -p m4
$ mkdir -p po
$ mkdir -p include
```

Some of these directories may already exist, of course.

The user will normally create `src` for the library code, and `tests` for unit-tests and regression tests. The user may create other directories if the project is more complex. A `doc` directory is commonly needed to hold documentation. Our projects also include a `doc/dox` directory for **Doxygen** documentation.

`bootstrap` recreates all of the **GNU** management files it needs to create. For **XPC 1.1**, we no longer bother keeping up with the following files:

```
ABOUT-NLS
AUTHORS
NEWS
INSTALL
TODO
```

We avoid them by using the `foreign` keyword in the `Makefile.am` files and in the *automake* call, to avoid *automake* complaining about "missing files".

Some of the files are automatically generated by the **GNU** *autoconf* tools.

Then `bootstrap` makes the following calls:

```
$ PKGCONFIG=yes              (or no if not available)
$ autopoint -f
$ aclocal -I m4 --install
$ autoconf
$ autoheader
$ libtoolize --automake --force --copy
$ automake --foreign --add-missing --copy
$ cp contrib/mkinstalldirs-1.9 aux-files/mkinstalldirs
```

The actions above are the default actions of the `bootstrap` script. Command-line options are available to tailor the actions taken, to implement some common operations:

```
--configure
```

The option above specifies that, after bootstrapping, run the `configure` script with no special options. This is no longer the default action of the `bootstrap` script. It is now preferred to use the `build` script.

```
--no-configure
 -nc
```

The option above specifies to not proceed to run the `configure` script. Simply bootstrap the project. Use this option, and later run `./configure` or `build` for building for debugging, stack-checking, and many other options.

```
--enable-debug
--debug
 -ed
```

The options above cause the `bootstrap` script to run `configure` with the `-enable-debug` and `-disable-shared` flags.

Again, it is now preferred to use the `build` script for this purpose.

```
--cpp-configure
```

The option above specifies to configure the project with the **XPC**-specific `-disable-thisptr` option. This option is good for usage in **C++** code, where the developer is passing unit-test structure objects by reference to **C++** functions. Since the item is a reference to an actual (non-pointer) class member, it is guaranteed to exist, and therefore null `this` pointer checks are unnecessary. This option can be combined with the `-debug` option.

Warning

> This option will make a version of the unit-test (unit_test_test in the tests directory) that will segfault due to attempts to use the null pointers supplied in some of the tests.

In most cases, the C++ developer will want to treat the resulting library as a *convenience library*:

`http://sources.redhat.com/autobook/autobook/autobook_92.html`

As such, the `-disable-thisptr` version of **XPC** libraries should *not* be installed by the developer. Otherwise, **C** unit-test applications will lose the benefits of null-pointer checks. The **XPC CUT++** build process links in this specially-built version of the **CUT** library into the **CUT++** library, and the install process does make sure that the**C** header files are also installed.

The following `bootstrap` options for **XPC 1.0** no longer exist in **XPC 1.1**.

```
--make, -m
--generate-docs
--no-generate-docs
--no-automake
```

The following options still apply, but they operate on *all* **XPC** projects as whole:

```
--clean
  clean
```

The options above specify to delete *all* derived and junk files from the project.

```
--debian-clean
```

The option above specifies to just remove the files created by running the various debian/rules script options.

The default of the `bootstrap` script is to bootstrap the project. To set up other configurations, manually call the desired `configure` command. For example, to generate (for example) debugging code using static libraries:

```
./bootstrap
./configure --enable-debug --enable-coverage --enable-thisptr --disable-shared
```

Also see the `build` script, which is the new preferred (and easier) method of configuring (and building, and testing) the projects.

3.3 The 'build' Script

First, note that one can run the following command to jump right into using the `build` script:

```
$ ./build --help
```

The `build` script is new with **XPC 1.1**.

It can be run after the `bootstrap` script is run.

It supports the same *configure* process as `bootstrap`, but it also add the following features:

- Build *out-of-source* configurations. This allows the developer to create directories suchs as `release` and `debug` in which to build the code.
- Perform a whole-project *make*, which builds all of the **XPC** source code, `pkgconfig` files, `*.po` files, and documentation.
- Selectively build a single project or the documentation.

The following `build` options cause the projects to be configured with certain options and built in a special output subdirectory:

```
Name           Directory      Build Options

--release      release        None
--debug        debug          --enable-debug --disable-shared
--cpp          cpp            --disable-thisptr
--build name   name           None
```

These options work by creating the directory, changing to it, and running a `../configure` command. This action creates only `Makefiles` that mirror the original directory structure, and cause the modules from the original directory structure to be compiled and built into the new directory structure. These options offer a way to maintain multiple builds of the same code.

The following option supports selecting what to build:

```
--project name
```

The potential values of name are (or will be):

```
all            Build all projects and the documentation. Default.
doc            Just build the documentation.
xpccut         Build the xpccut C project.
xpccut++       Build the xpccut++ C++ project.
xpc            Build the xpc C project.
xpc++          Build the xpc++ C++ project.
xpcproperty    Build the xpcproperty project.
xpchello       Build the xpchello project.
```

The default action, after configuration, is to perform the *make*:

```
--make
--no-make
```

The rest of the options are also found in `bootstrap`. See the `-help` option.

Now, if the `all` project is selected, the following sequence of actions will occur:

(NOT YET FULLY IMPLEMENTED)

1. xpccut
   (a) Build `xpccut` normally in the selected out-of-source directory (preferably `release`, but `debug` or a directory name you pick yourself will also work).
   (b) Build `xpccut` with `-disable-thisptr` in the `cpp` directory.
   (c) Build `xpccut++` normally in the selected out-of-source directory, linking it to the`-disable-thisptr` version of `xpccut`.

2. xpc

   (a) Build `xpc` normally in the selected out-of-source directory (preferably `release`, but `debug` or a directory name you pick yourself will also work).

   (b) Build `xpc` with `-disable-thisptr` in the `cpp` directory.

   (c) Build `xpc++` normally in the selected out-of-source directory, linking it to the`-disable-thisptr` version of `xpc`.

3. Build `xpcompmgr++` normally in the selected out-of-source directory.

4. Build `xpcproperty` normally in the selected out-of-source directory.

5. Build `xpchello` normally in the selected out-of-source directory.

3.4 The configure.ac Script

The **XPC** projects support a number of build options. Some are standard, such as compiling for debugging and internationalization support. Some options are peculiar to **XPC**, such as conditional support for error-logging and pointer-checking.

As usual in **GNU** *automake*, the options are supported by switches passed to the `configure.ac` script, supported by corresponding *m4* macros in the `m4` directory. Here are the options:

```
  Switch              Default  Status/Description

--enable-debug           no    Compiles for gdb
--enable-lp64            no    Very problematic at present, useless
--enable-stackcheck      no    Makes segfaults much more likely, a good test
--enable-thisptr         yes   Activates thisptr() null-pointer check
--enable-errorlog        yes   Provides run-time logging at various verbosities
--without-readline       no    readline() support enabled by default
--enable-coverage        no    Coverage testing using gcov
--enable-profiling       no    Turns on gprof support
--enable-motorola        no    Compiles for MC68020 to check the code
```

The Motorola option requires a version of *gcc* called something like *68000-gcc*, which is a version of *gcc* built for cross-compiling. We still have a lot more to learn about this issue. It isn't ready, and may be wrong-headed anyway.

Some of the options can take additional parameters to tweak how they work. The options that can do that are:

- `-enable-debug`. In addition to the standard `yes` and `no` values, also accepted are `gdb` (the default) and `db` (the BSD/UNIX variant). See Debugging using GDB and Libtool for more information.

- `-enable-coverage`. See Building For 'gcov' Usage for more information.

- `-enable-profiling`. See Building For 'gprof' Usage for more information.

These options are implemented by the `AC_XPC_DEBUGGING` macro call in the **XPCCUT**'s `configure.ac` file.

3.5 Making the XPCCUT Library

This section assumes that you have bootstrapped and configured the library in some manner. It also assumes that you want to install the library for usage by another project.

3.5.1 Making the Library for Normal Usage

There's not much to explain here. Once the **XPC CUT** library is bootstrapped and configured, the rest of the process is very simple and standard at this point:

```
$ make
$ make check    [or "make test"]
# make install
```

This sequence of steps installs the library (`libxpccut` in static and shared-object forms), the `unit_*.h` header files, the HTML version of the documentation, a rudimentary *man* page, a primitive (and usually incomplete) Spanish translation for the error and information messages, and a *pkg-config* file that can be referred to in the `configure.ac` file of projects that want to link properly to the **XPC CUT** library.

There is one special project that is deeply dependent on the **XPC CUT** library, and that is the **XPC CUT++** library and test application. This special case is discussed in the next subsection.

3.5.2 Making the Library for XPC CUT++

The **XPC CUT++** library is a C++ wrapper for the **XPC CUT** library. As such, it basically requires the **XPC CUT** library to be built at the same time. It does not require the **XPC CUT** library to be installed fully, however. Instead, the library is built and linked into the **C++** version of the library, and only the header files are installed.

Rather than have the developer deal with all of this, the `bootstrap` script for the **XPC CUT++** library looks for the **XPC CUT** library source-code in the same base directory (e.g. `xpc_suite`. If it finds it, it bootstraps it configures it with the same user options as passed to the `bootstrap` script, plus a "secret" option to disable the this-pointer checking, for a little extra speed.

Then the **XPC CUT** library is built.

When the **XPC CUT++** library is built, the **XPC CUT** library is then added to the **C++** version of the library, so that **C++** applications need include only one library.

We're also working on a **CUT++** sample application that will help us make sure that the **XPC CUT++** combines properly the **C** and **C++** components, and installs properly the **C** and **C++** header files. See the xpc_cutsample_↩ introduction page for more information.

3.6 Linking to the XPC CUT Library

Using the **XPC CUT** library in an external application requires that the **XPC CUT** library be installed, as noted by the simple instructions in the previous section.

The application project must make some settings in the following project files:

1. `configure.ac`
2. `app/Makefile.am` (as applicable)
3. `tests/Makefile.am` (as applicable)

In the application's `configure.ac` file, the following line is necessary:

```
PKG_CHECK_MODULES([XPCCUT], [xpccut-1.0 >= 1.0])
```

This directive causes the `XPCCUT_CFLAGS` and `XPCCUT_LIBS` macros to be defined as if the following commands were run, and the output assigned to the respective macros:

```
$ pkg-config --cflags xpccut-1.1
$ pkg-config --libs xpccut-1.1
```

The respective outputs of these commands are

```
-I/usr/local/include/xpc-1.1
-L/usr/local/lib/xpc-1.1 -lxpccut
```

These values need to be added to the command lines. First, in `configure.ac`, the `CFLAGS` value needs to be augmented:

```
CFLAGS="$CFLAGS $CFLAGSTD $COMMONFLAGS $XPCCUT_CFLAGS"
```

In the `Makefile.am` in the application directory, the following are needed. First, expose the `-L` and `-l` directives noted above.

```
XPCCUT_LIBS = @XPCCUT_LIBS@
```

Note that the `XPCCUT_CFLAGS` macro is automatically carried through to the application's subdirectory as part of the *automake* process.

For a good example, see the xpc_hello_mainpage page and the **XPC** *Hello* application it describes.

3.7 Libtool

This section describes *libtool* and contains notes from other documents in the suite.

3.7.1 Libtool Versioning

We support two types of version in the **XPC** suite:

- Package versioning
- Libtool versioning

Package version indicates when major changes occur in a package, and also indicates what sub-directory the libraries are installed in. For example, the current package version of the **XPC** suite is `1.0.4`. The libraries and header files are included in directories with names of the form `xpc-1.0`. The `4` increments whenever we have made enough changes to make a new release. If the changes modify or delete an existing interface, then the `0` will get incremented to `1`, and the new storage directory will become `xpc-1.1`. If we rewrite any major component of the suite, then the `1` in `1.0` will increment, and the new storage directory will become `xpc-2.0`. We don't anticipate this kind of major change, though.

Although package version is visible to the user, and partly determines the compatibility of the **XPC** libraries, the actual interface versioning is determined by *libtool* versioning. The following discussion is a succinct description of this link:

`http://www.nondot.org/sabre/Mirrored/libtool-2.1a/libtool_6.html`

Imagine a simple version number, starting at 1. Consider each significant change to the library as incrementing this number. This increment reflects a new interface. *libtool* supports specifying that interface versions from `i` to `j` are supported.

libtool library versions are described by three integers:

- `current`. The most recent interface number that this library implements.
- `revision`. The implementation number of the current interface.
- `age`. The difference between the newest and oldest interfaces that this library implements.

The library implements all the interface numbers in the range from `current - age` to `current`.

If two libraries have identical `current` and `age` numbers, then the dynamic linker chooses the library with the greater `revision` number.

The **XPC** suite actually abused the distinction between the package and *libtool* version information. So, for **XPC** package version `1.0.4`, we are resetting the *libtool* version to `0:0:0`

When we make public the next version with any changes at all, the new versions will be:

- **Package**: `1.0.5` (patch++)
- **libtool**: `0:1:0` (revision++)

If, instead of just any simple change, we added an interface:

- **Package**: `1.0.5` (change in patch number)
- **libtool**: `1:0:1` (current++, revision=0, age++)

Our reasoning here is that the additional interface cannot break existing code, so that the same installation directory, `xpc-1.0`, can be used.

If we changed or removed an existing interface, or refactored the whole set of projects, old code will break, so we'll need a new installation directory.

- **Package**: `1.1.0` (minor++, patch=0)
- **libtool**: `1:0:0` (current++, revision=0)

If the changes were really major (a rewrite of the libraries):

- **Package**: `2.0.0` (major++, minor=0, patch=0)
- **libtool**: `1:1:1` (current++, revision++, age++)

3.8 pkgconfig

GNU *pkgconfig* is a way to help applications find out where packages are stored on a system. The common places to store them are (in descending order of likelihood on a Linux system):

```
/usr            Apps installed by the default package manager.
/usr/local      Apps installed from source code.
/opt            Solaris and Java flavored apps such as OpenOffice
$HOME           Apps installed by the user in his/her home directory.
```

As an example, we were trying to install totem (a media player for the Gnome desktop environment) on a Gentoo system. The Gentoo emerge system provided an old version, 1.0.4, but we wanted the latest, which happened to be 1.5. Thus, we went to the totem site and downloaded the source. Running configure revealed a dependency on "iso-codes". However, that package was masked on Gentoo.

Since it was a Debian package, we found it at

`http://packages.debian.org/testing/misc/iso-codes`

and downloaded the source tarball. We did the `./configure; make; make install` mantra, and then retried the configuration of the totem build.

Still missing. We looked in `/usr/lib/pkgconfig`, but no `iso-codes.pc` file. Nor was it in `/usr/local/lib/pkgconfig`, which is where we expected to find it.

So we look in the iso-codes build area, and see the iso-codes package-configuration file. We copied it to `/usr/local/lib/pkgconfig`.

The totem configuration was then built with no problem.

3.9 Endorsement

A rousing endorsement of Libtool and Pkgconfig from Ulrich Drepper:

http://udrepper.livejournal.com/19395.html

The second problem to mention here is that not all unused dependencies are gone because somebody thought s/he is clever and uses -pthread in one of the pkgconfig files instead of linking with -lpthread. That's just stupid when combined with the insanity called libtool. The result is that the -Wl,–as-needed is not applied to the thread library.

Just avoid libtool and pkgconfig. At the very least fix up the pkgconfig files to use -Wl,–as-needed.

Hmmm, we need to remember that "at the very least" advice....

3.10 References

1. http://wiki.showmedo.com/index.php/LinuxBernsteinMakeDebugSeries Discusses an improved version of automake, called "remake".
2. http://www.gnu.org/software/libtool/manual/automake/VPATH-Builds.html Discusses building the object files in a separate subdirectory.
3. http://www.nondot.org/sabre/Mirrored/libtool-2.1a/libtool_6.html Discusses versioning in libtool.
4. http://www.adp-gmbh.ch/misc/tools/configure/configure_in.html Walks the user through a configure.ac file; very helpful.
5. http://www.geocities.com/foetsch/mfgraph/automake.htm Provides a nice summary of autoconf/automake projects.

Chapter 4

XPC Basic and Subversion

Author

Chris Ahlstrom

Date

2008-02-24

Last Edit:

2015-10-05

License:

$XPC_Basic_GPL_LICENSE$ (See xpc_Basic_license_subproject)

Chapter 5

Using Git

5.1 Introduction to Using Git

This document is a concise description of various processes using the *Git* distributed version control system (DV↩
CS).

5.2 Setup of Git

Like other version-control systems, Git has configuration items and work-flows that need to be tailored to your needs.

5.2.1 Installation of Git

Obviously, the easiest way to set up Git on Linux is to use your distro's package-management system.

Failing that, or if you want the very latest and greatest, you can get the source code and build it yourself:

`http://git-scm.com/downloads`

On Windows, that same URL will get you an installer executable. After that, you're on your own for monitoring and retrieving updates. Another option is to use the *msysGit* project: `http://msysgit.github.io/` Actually, that gets you the very same executable!

5.2.2 Git Configuration Files

There are three levels of configuration for Git. Each succeeding level overrides the previous level.

- `/etc/gitconfig`. The system-wide configuration. Using `git -system config` affects this file. On MSYS for Windows, the `msys/etc/gitconfig` file is used.
- `~/.gitconfig` (Linux) or `$HOME/.gitconfig` (Windows, where HOME is `C:/Users/$USER`. The user's pan project configuration. Using `git -global config` affects this file.
- `myproject/.git/config`. The project's own configuration.

5.2.3 Setup of Git Client

For our purposes, the command-line Git client is quite sufficient. Start with the `gitk` package if you are interested in a GUI.

The `git config -global ...` command will set up your preferences for ignoring certain files, for coloring, email, editors, diff programs, and more. Specific instances of this command are shown below.

1. *Setting up Git features*. This process includes setting up preferred handling for:
   (a) Whitespace.
   (b) Line-endings.
   (c) Differencing code.
   (d) Merging code.
   (e) Programmer's editor.
   (f) Aliases for Git commands.
   (g) Colors.
   (h) Template of commit message.
   (i) And much more.
2. *Setting up 'git-ignore'*. This process provides a list of files, file extensions, and directories, that you do not want Git to track, as well as subsets of those that you do want Git to track anyway.

5.2.4 Setting Up Git Features

There are additional configuration options. Rather than discuss them, we will just list the commands and show the `~/.gitconfig` file that results.

```
$ git config --global user.name "Chris Ahlstrom"
$ git config --global user.email ahlstrom@bogus.com
$ git config --global core.excludesfile ~/.gitignore
$ git config --global core.editor vim
$ git config --global commit.template $HOME/.gitmessage.txt
$ git config --global color.ui true
$ git config --global core.autocrlf input
$ git config --global diff.tool gvimdiff
$ git config --global merge.tool gvimdiff
$ git config --global alias.d difftool
```

`commit.template`:

```
subject line

what happened

[ticket: X]
```

Another option is `core.autocrlf` (what about `core.eol`?). If set to 'true', then it will convert LF into CR-LF line-endings. Useful for those unfortunate Windows-only teams. For Linux teams, or for mixed-OS teams, see the "input" setting above. That setting will leave you with CR-LF endings in Windows checkouts but LF endings on Mac and Linux systems and in the repository.

There are many more helpful configuration items, such as those that deal with OS-specific line-endings or the disposition of white space. Other "config" options to check out: `core.pager`; `core.whitespace`; `user.↩ signingkey`; `color.*`; and, on the server side, `receive.fsckObject`, `receive.denyNonFast↩ Forwards`, and `receive.denyDeletes`.

After this, your `.gitconfig` should look like this:

```
[user]
   name = Chris Ahlstrom
   email = ahlstrom@bogus.com
[core]
   excludesfile = /home/ahlstrom/.gitignore
```

And this is the file that results:

```
[user]
   name = Chris Ahlstrom
   email = ahlstrom@bogus.com
[core]
   excludesfile = /home/ahlstrom/.gitignore
   editor = vim
[diff]
   tool = gvimdiff
[difftool]
   prompt = false
[alias]
   d = difftool
[color]
   ui = true
```

You can check your current settings with this command:

```
   $ git config --list
```

5.2.5 Setting Up the Git-Ignore File

Building code generates a lot of by-products that we don't want to end up in the repository. This can happen if the `git add` command is used on a directory (as opposed to a file).

First, look at the sample git-ignore file `contrib/git/dot-gitignore`. Verify that the file extensions all all to be ignored (not checked into source-code control), and that the list includes all such files you can conjur up in your imagination. Next, copy this file to the `.gitignore` file in your home directory.

Finally, run

```
   $  git config --global core.excludesfile ~/.gitignore
```

5.2.6 Setting Up Git Bash Features

It is very useful to have your command-line prompt change colors and show the Git branch that is active, whenever the current directory is being managed by Git.

Note these two files in the `contrib/git` directory in this project:

- `dot-bashrc-git`, to be carefully inserted into your `~/.bashrc` file.
- `dot-git-completion`, to be copied to `~/.git-completion`, which is sourced in that .bashrc fragment.

What these script fragments do is set up your bash prompt so that it will show the Git branch that is represented by the current directory, in your command-line prompt, if the directory is part of a Git project.

5.3 Setup of Git

There are three "locations" where a Git repository can be created:

- Your personal computer
- Your local network
- A remote network (on the Internet, hosted by a hosting site)

5.4 Setting Up a Git Repository on Your Personal Computer

A restrictred, but still useful use-case is to set up a Git repository on the same computer you use for writing and building code.

Assuming you have a nascent project directory in existence, and want to start tracking it in Git, you perform the following steps (under your normal login):

1. `cd` to the project's root directory. Make sure there are no files that you don't want to track, or that they are covered by your system or user git-ignore file.
2. Run the command `git init`. The result will be a hidden `.git` directory.
3. Now run `git add *` to add all of the files, including those in subdirectories, to the current Git project.
4. Run `git status` to verify that you have added only the files that you want to track.
5. Run a command like `git commit -m "Initial project revision."`

5.5 Setting Up a Git Repository on Your Personal Computer

If your personal machine is on a home/local network, you can clone your new Git repository on another home machine. The following command assumes you use SSH even at home to get access to other home computers.

```
$ git clone ssh://homeserver/pub/git/mls/xpc_suite-1.1.git
```

This repository will be referred to as a remote called "origin". This remote assumes that "homeserver" has the IP address it had when the clone was made. However, outside of your home network, you may have to access "homeserver" from a different IP address. For example, at home, "homeserver might be accessed as '192.168.↩ 111.18', the IP address of 'homeserver' in `/etc/hosts`, while outside your home network, it might be something like '77.77.77.77', accessed from the same client machine remotely as 'remoteserver'.

In this case, you need to add a remote name to supplement 'origin'. For example:

```
$ git remote add remoteaccess ssh://remoteserver/pub/git/mls/xpc_suite-1.1.git
```

So, while working on your home network, you push changes to the repository using:

```
$ git push
```

or

```
$ git push origin currentbranch
```

while away from home, you push changes to the repository using:

```
$ git push remoteaccess currentbranch
```

5.6 Setup of Git Remote Server

5.6.1 Git Remote Server at Home

Setting up a Git "server" really means just creating a remote repository that you can potentially share with other developers.

The repository can be accessed via the SSH protocol for those users that have accounts on the server. It can also be accessed by the Git protocol, but you'll want to run that protocol over SSH for security. A common way to access a remote Git repository is through the HTTP/HTTPS protocols.

5.6.2 Git Remote Server at GitHub

Let's assume that one has a body of code existing on one's laptop, but nowhere else, and that no Git archive for it already exists. Also, let's assume we want to first set up the archive at GitHub, and then clone it back to the laptop to serve as a remote workspace for the new GitHub project. Finally, let's assume you've already signed up for GitHub and know how to log onto that service.

The process here is simple:

1. Convert your local project into a local Git repository.
2. Create your empty remote repository on GitHub.
3. Add your GitHub repository as a remote for your local Git repository.
4. Push your master branch to the GitHub repository.

Convert local project to local Git repository.

1. Change to your projects source-code directory, at the top of the project tree.
2. Run the command `git init` there to create an empty git repository..
3. Run the command `git add .` to add the current directory, and all sub-directories, to the new repository.
4. Run the command `git status` if you want to verify that all desired files have been added to the repository.
5. Run the command `git commit -m "short message"` to add the current directory,

Create new repository at GitHub.

1. Sign up for a GitHub account, if needed, then log into it. Also be sure to add your computer's public key to GitHub, if you haven't already. (How to do that? We will document that later.)
2. On your home page, click on the "plus sign" at the upper right and select "New Repository".
3. Verify that you are the owner, then fill in the repository name and description.
4. Make sure it is check-marked as public (unless you want to hide your code and pay for that privilege). There's no need to initialize the repository with a README, since you are importing existing code. No need to deal with a `.gitignore` file or a license, if they already exist in your project.
5. Click the "Create Repository" button.
6. Once the repository is created, take note of the URLs for it:
   - **HTTPS**. `https://github.com/username/projectname.git`
   - **SSH**. `git@:github.com:username/projectname.git`
7. Add this new remote repository to your local copy and make it the official copy, called "origin"; `git remote add origin git@github.com:username/projectname.git`
8. Push your current code to this repository: `git push -u origin master`

The `-u` (or `-set-upstream`) option sets an upstream tracking reference, so that an argument-less `git pull` will pull from this reference, Once you push the code, you will see a message like:

```
Branch master set up to track remote branch master from origin.
```

Note that the URL for the home-page for the project is

```
https://github.com/username/projectname
```

This URL can be used when adding GitHub as a remote.

To clone the new GitHub repository to another computer:

```
git clone git@github.com:username/projectname.git
```

Note that some of this information is also available in Scott Chacon's book, "Pro Git", from Apress.

TODO: talk about pushing and pulling your own changes, and pulling changes made by people who have forked your project.

5.7 Git Basic Commands

We've already covered the `config` and `init` commands, and touched on a few other commands. In this section, we briefly describe the most common commands.

You can get information using `git help`. Adding the `-a` option lists the many command of git. Adding the `-g` option lists some concepts that can be presented. Try the "workflow" concept:

```
$ git help workflows
```

The result is a man page summarizing a workflow with Git.

There is also a nice man page, gittutorial(1).

5.7.1 git status

First, note that git commands can be entered in any directory or subdirectory of a project. Yet the command will still operate on all files and directories in the project, starting from the root directory of a project. If files are to be displayed, their paths are shown relative to the current working directory.

The `git status` command examines the files and determines if they are new, modified, or unchanged. If new or modified, they are shown in red. The command also determines if they have been added to the commit cache. If so, then they are shown in green.

5.7.2 git add

When files are shown as red in the `git status` command, that means they have been modified. If you think they are ready, you can use the `git add` command to add them to the commit cache; they will then show up as green in the `git status` listing.

You can add any or all modified files to the commit cache. If you add a file, and then modify it again, you will have to add it again.

Now, if some files are still red, but you want to commit the green files anyway, it is perfectly fine to do so.

5.7.3 git commit

This command takes the files in the git commit cache (they show up as green in `git status`) and commits them. The most common form of this command is:

```
$ git commit -m "This is a message about the commit".
```

Keep the message very short, around 40 characters. One can leave off the message, in which case one can add this message, plus a much longer description, in the text editor that one set up git to use.

5.7.4 git stash

This command stores your current local modifications and brings the files back to a clean working directory. Once finished, you can recall the stash and continue onward. Great for double-checking the older revision of the code.

5.7.5 git branch

Branching is a complex topic, and not every workflow can be documented. Here, we describe a straightforward workflow where there's a master branch, and occasionally one side-development branch that ultimately gets merged back to master. We will assume for now that no conflicts occur, and that branches occur serially (one developer, who finishes the branch and merges it back before going on to the next branch.)

5.7.5.1 Create a branch

Our repository is at a certain commit in `master` at the current latest commit in a repository. We will call this commit `cf384659`, after the hypothetical checksum of that commit.

The `HEAD` pointer points to `master`, which points to `cf384659`.

Let's create a branch called "feature":

```
$ git branch feature
```

Now, `feature` points to `cf384659`, just like `master` does. `HEAD` still points to `master`. This means that we are still working in `master`.

To switch to the "feature" branch so that we can work on it:

```
$ git checkout feature
```

Now `HEAD` points to `feature`.

There's a git trick that let's you create a new branch and check it out in one command:

```
$ git checkout -b 'feature'
```

Verify that you're in the new branch; the asterisk will point to the new branch:

```
$ git branch
* feature
  master
```

In this branch, let's edit a file or two and commit them.

```
$ vi file1.c file2.c
$ git commit -m "Added part 1 of feature."
```

Now there's a new commit just beyond `cf384659`, we'll call it `f1958ccc`. `feature` now points to `f1958ccc`, and `HEAD` still points to `feature`.

We're not done with `feature`, but we suddenly realize we want to add something to `master`. Let's set `HEAD` back to `master`

```
$ git checkout master
```

That was fast! Now we can make some edits, commit them, and then go back to working on `feature`:

```
$ git checkout feature
```

Now, git won't let you change branches if files are uncommitted, to protect from losing changes. Before you change branches, you must either *commit* the changes or *stash* them.

After a number of edits and commits on the "feature" branch, we're ready to tag it for ease of recovery later, and then merge "feature" back into "master".

5.7.6 git ls-files

This command is convenient for finding files, especially ones that did not get added to the archive because they were unintentionally present in a `.gitignore` file. The following command shows those "other" files:

```
$ git ls-files -o
```

5.7.7 git merge

TODO

5.7.8 git push

TODO

5.7.9 git fetch

TODO

5.7.10 git pull

TODO

5.7.11 git amend

TODO

5.7.12 git svn

TODO

5.7.13 git tag

TODO

5.7.14 git rebase

TODO

5.7.15 git reset

TODO

5.7.16 git format-patch

TODO

5.7.17 git log

TODO

5.7.18 git diff

TODO

5.7.19 git show

TODO

5.7.20 git grep

TODO

5.8 Git Workflow

TODO

5.9 Git Tips

```
git status
git status -sb
git checkout -b
git log --pretty=format:'%C(yellow)%h %C(blue)%ad%C(red)%d
   %C(reset)%s%C(green) [%cn]' --decorate --date=short
git commit --amend -C HEAD
git rebase -i
```

Also see the section on Using Git with Subversion for using the Git-to-Subversion bridge.

Chapter 6

Using Git with Subversion

6.1 Introduction to the Git-Subversion Bridge

This document is a concise description of how to use the *Git* distributed version control system (DVCS) with a Subversion repository.

Git provides a "Subversion bridge", using a number of commands that start with `git svn`. These commands allow the developer to clone the Subversion repository, manage changes locally via the Git repository, and then push and pull changes against the Subversion repository.

The advantages of using Git to manage a Subversion repository:

1. The developer's machine has a complete copy of the repository. This copy makes a nice backup in case the *Windows* server goes down.

2. The copy of the repository is compressed, making it about one-half the size of the Subversion working copy. If the working copy is converted to a "bare" repository, the size goes down by another factor of 8 or so.

3. The developer can perform the full life-cycle of version control without a connection to the Subversion server:

   (a) Updates. One can do updates without a connection to Subversion, but branching and commits cannot be done.

   (b) Branching (local branches). Branching under Git is fast and flexible. Local branches can track remote branches.

   (c) Commits. One can happily commit often, and then *rebase* the commits down to a single commit for check-in to Subversion.

   (d) Merges. Git merges are said to be easier and more fool-proof than Subversion merges.

4. Commits can be rebased to a simpler, linear change, suitable for committing into the Subversion repository.

The disadvantages of using Git to manage a Subversion repository:

1. An additional learning curve, with a slightly more complex (but flexible) lifecycle.

2. Because of our chosen directory layout, Git cannot detect the existing branches in Subversion from the top level of a repository such as `dev`. There is a way around this issue by cloning only directories that have `branches`, `tags`, and `trunk` directly under them, such as the `project` directory.

All in all, although the Git/Subversion bridge is most useful to a developer who is often unable to connect to our internal Subversion repository, it can be also useful to any developer, and provides a decent backup (of sorts) of the Subversion repository.

6.2 Git/Subversion Usage Summary

This section provides a complete outline of how to set up and use Git to manage a Subversion repository. References to the details of each step are provided in this outline.

1. `git config -global ....` Set up your preferences for ignoring certain files, for coloring, email, editors, diff programs, and more. See Setting Up Git. Some samples:
   (a) `git config -global core.excludesfile ~/.gitignores`.
   (b) `git config -global user.name "Chris Ahlstrom"` and much more. See Setting Up Git Features.
2. `git svn clone https://svnserver:443/svn/....` Create a Git archive (working copy) of a Subversion repository, Fetching a Copy of a Repository, or a sub-project of the repository, Cloning a Standard Subversion Layout. We restrict ourselves to sub-projects that have the standard Subversion layout (`trunk/branches/tags`).

The process of using *Git* with *Subversion* is described in more detail in the following resources:

`http://trac.parrot.org/parrot/wiki/git-svn-tutorial`

`http://progit.org/book/`

The latter URL also points to a way to get a paperback copy of the book. We recommend it.

6.3 Setting Up Git

Setting up Git requires the following processes:

1. *Installing Git*. Installing Git depends on your Linux distribution or what Windows packages you want to use. Installation is beyond the scope of the present document.
2. *Setting up 'git-ignores'*. This process provides a list of files, file extensions, and directory that you do not want Git to track.
3. *Setting up Git features*. This process includes setting up preferred handling for:
   (a) Whitespace.
   (b) Line-endings.
   (c) Differencing code.
   (d) Merging code.
   (e) Programmer's editor.
   (f) Aliases for Git commands.
   (g) Colors.
   (h) And much more.

These settings can be made system-wide, globally (for all Git archives under your control), on a single archive, and, I believe, even on a single directory.

Here's a summary of one possible process for working with a Subversion repository:

1. `git svn clone https://svnserver:443/svn/dev`. Fetch a copy of the repository.
2. `git clone -bare dev dev.git`. If desired, convert it to a much smaller bare repository.
3. `git branch -a -v`. View all branches (local and remote) of the new copy of the repository. The `-v` option, if present, shows addition information.

4. `git svn fetch` or `git svn rebase`. Get any changes from the Subversion repository; the latter also updates your copy to HEAD.
5. `git add my_modified_files` and `git commit`. Commit your modified (included by *git add*) files.
6. `git commit -a`. The same as running the two commands in the previous step. Note that each commit will be checked in as a separate Subversion revision.
7. `git svn dcommit`. Commit your local Git-committed changes to the Subversion repository.
8. `git branch` and `git checkout`. Local branching.
9. `git svn rebase -i`. Use an interactive rebase to combine a number of commits before submitting them to Subversion.

6.3.1 Setting Up Git-Ignores

Building the code generates a lot of by-products that we don't want to end up in the repository. This can happen if the `git add` command is used on a directory (as opposed to a file).

First, look at `gitignores.txt` a verify that the file extensions all all to be ignored (not checked into source-code control), and that the list includes all such files you can conjur up.

Next, copy `gitignores.txt` to the `.gitignores` file in your home directory.

Finally, run

```
$  git config --global core.excludesfile ~/.gitignores
```

After this, your `.gitconfig` should look like this:

```
[user]
   name = Chris Ahlstrom
   email = ahlstrom@bogus.com
[core]
   excludesfile = /home/ahlstrom/.gitignore
```

6.3.2 Setting Up Git Features

There are additional configuration options. Rather than discuss them, we will just list the commands and show the `~/.gitconfig` file that results.

```
$ git config --global user.name "Chris Ahlstrom"
$ git config --global user.email ahlstrom@bogus.com
$ git config --global core.excludesfile ~/.gitignores
$ git config --global core.editor vim
$ git config --global color.ui true
$ git config --global diff.tool gvimdiff
$ git config --global alias.d difftool
```

And this is the file that results:

```
[user]
   name = Chris Ahlstrom
   email = ahlstrom@bogus.com
[core]
   excludesfile = /home/ahlstrom/.gitignore
   editor = vim
[diff]
   tool = gvimdiff
[difftool]
   prompt = false
[alias]
   d = difftool
[color]
   ui = true
```

There are many more helpful configuration items, such as those that deal with OS-specific line-endings or the disposition of white space.

6.4 Git/Subversion Processes

This section discusses various building blocks for operating on a repository.

6.4.1 Fetching a Copy of a Repository

One can get a copy of a repository easily using the `git svn clone` command. However, the disadvantage of this command is that one actually gets a working copy of the repository, with all of the code, and that can be a lot of data. See the next sections, Fetching a Branch From a Repository or Making a "Bare" Copy of a Repository, for alternative setups.

Getting a copy of a repository is a simple command, though it will take quite awhile to finish its work:

```
   $ git svn clone [-s] [-r M:N] https://svnserver:443/svn/dev
```

The `-s` switch indicates that the standard `trunk`, `branches`, and `tags` layout is used by that URL.

Although our repository has those directories in each sub-project, the top-level directory, `dev` does not, and so we cannot use that option for cloning the repository. Thus, we cannot recover branch information; we can only get the documents themselves. Still, significant work can be done with this use case.

The `-r` option restricts the checkout to a range of revisions. Getting all of them from `dev` takes about an hour and about 7.5 Gb of space, because Git has to check out and copy each revision.

The command will initialize a new Git repository, and then fetch a working copy of the repository. Then it will compress the Git blobs to save quite a bit of space.

As each Subversion revision is extracted and fetched, information on it is shown. For example:

```
r395 = 1c121cf3f6fcc4dd771134d682431e34cff7e2b7 (refs/remotes/xpc_original)
```

The command will initialize a new Git repository, and then fetch a working copy of the repository. Then it will compress the Git blobs to save quite a bit of space.

As each Subversion revision is extracted and fetched, information on it is shown. For example:

```
r395 = 1c121cf3f6fcc4dd771134d682431e34cff7e2b7 (refs/remotes/xpc_original)
```

When this command completes, one can list all the branches that now exist for this Git repository.

```
   $ git branch -a
   * master
     remotes/git-svn
```

Note that none of the Subversion branches can be seen, because `dev` does not have the standard `trunk`, `branches`, and `tags` layout.

Because the branching history of this clone cannot be recovered, we do not recommend cloning the whole repository. Instead, see Cloning a Standard Subversion Layout for cloning any single project, in the repository, that does have the standard Subversion `branches`, `tags`, and `trunk` layout.

6.4.2 Fetching a Branch From a Repository

This section covers two cases:

1. You want to fetch only a single branch from a repository.
2. You want to refresh your Git repository with a branch that was just created using `svn copy`

6.4.2.1 Fetching an Existing Branch From a Repository

This section very simply notes that one does not need to clone the whole repository... one can simply clone a subdirectory of it. For example:

```
$ git svn clone https://svnserver:443/svn/dev/project/branches/mysubproject/
```

Fetching a whole branch is useful only when you aren't going to use Git or Subversion merging facilities to merge the branch back into the main trunk, and don't care about archiving information about the merge. For example, you might just want to clone the directory so that you can use Git to easily generate two different revisions of the directory for peer-review purposes.

6.4.2.2 Fetching a New Branch From a Repository

Let's say you've already got the complete repository in Git, and can see all the branches.

Then someone adds a new branch, using `svn copy URL1 URL2`. They tell you about it, and you do a `git branch -a`, but you do not see the new branch. How do you get the branch visible

```
$ git svn fetch
```

Git will tell you a little story about what it is doing, ending with this output:

```
Successfully followed parent
r587 = 9c607e137c9ae27cb4027ef20c6c953bffed2484 (refs/remotes/bugfix_branch)
```

Now `git branch -a` will show you that the new branch is present. You can then set up to track it:

```
$ git checkout --track -b local/bugfix_branch remotes/bugfix_branch
```

6.4.3 Cloning a Standard Subversion Layout

Right now, a lot of our work is done in the `branches` directory of the `project` project. That project *does* have a standard subversion layout, and we can clone that project and have access to the existing branches.

This task can probably be done with `git svn clone`, but here we will break that command into *git init* and *git fetch* commands.

First, make a directory to hold the new Git working directory, and change to it:

```
$ mkdir project
$ cd project
```

Second, initialize the Git repository, telling Git to use only the `project` project directory. Otherwise, it will get the whole `dev` repository, which is not what we want.) Also tell Git that the directory has the standard Subversion layout.

```
$ git svn init --no-minimize-url -s https://svnserver:443/svn/dev/project/
```

This command results in a new hidden `.git` directory, inside the directory called `project`.

Note the `-s` (standard Subversion layout) and `-no-minimize-url` (don't move up to the repository root) options.

Next, get all of the repository data, including the branches and all of the revisions, with the following command.

```
$ git svn fetch
```

This will take awhile. Once done, the working copy will have the latest copy of the Subversion `trunk`. Since we have been working off the `branches`, this copy of the trunk is very old – revision 200, dated 08 Aug 2011. At least it is small, only 268 Mb. We won't be using it.

The following command and output shows that we're on the *master*, which is a copy of `trunk`.

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

The following directory listing for the `project` directory verifies that it is a duplicate of `dev/project/trunk`:

```
$ ls
Configuration Data  Production  Projects  README  Scripts  Tools
```

The following command will show all the branches of `project`:

```
$ git branch -a
* master
  remotes/xpc_bugfix
  remotes/bugfix
  remotes/mybranch
  remotes/tags/Gold
  remotes/tags/new_code
  remotes/trunk
```

At this point, we could create a new branch. For this section, though, let's just look at an existing branch. Run the following command from your new `project` directory:

```
$ git checkout bugfix
```

This command checks out around 6400 files (very quickly). You can then verify that the contents match the current contents of `bugfix` in the Subversion archive.

Now let's switch to a different branch.

```
$ git checkout xpc_alternate
```

This command is even faster, it just moves "HEAD" to the correct commit.

You can then verify that the contents match the current contents of `xpc_alternate` in the Subversion archive.

Then you can switch back to `bugfix` and verify that it again matches the Subversion archive.

Note that the `git checkout` command is similar to the `svn switch` command. Like `git checkout`, the Subversion `switch` command pulls in the differences needed to convert your current branch to the branch to which it is switching. However, `svn switch` gets the changes from the repository, not locally, and shows the files being updated, so it is not as fast as `git checkout`.

At this point, the advantages of Git branching over Subversion branching should be obvious:

1. *Fast branch switching*. There's a lot less copying of data, and that copying is all done locally.
2. *Clean directories*. The directory structure is not littered with branch directories.

However, there is a disadvantage to overcome. After checking out a branch, we see what is called a "detached HEAD":

```
$ git checkout bugfix
$ git branch -a
* (no branch)
  master
  remotes/xpc_bugfix
  . . .
  remotes/trunk
```

The `(no branch)` item is tagged as the current branch. What this means is that, though we have the code for bugfix, we don't have a true branch name for it, because the branch we checked out is remote. We need a local branch name to work with.

We need to create a new local branch that *tracks* the remote Subversion branch.

There are two ways to do it.

```
$ git branch --track local/bugfix remotes/bugfix
$ git checkout local/bugfix
```

Or, in one command:

```
$ git checkout -b local/bugfix remotes/bugfix
```

Then we can see that we have a new local branch, `bugfix_2`, that tracks the remote branch, `bugfix`.

Note the two variations of the command. The `local/` portion is a convention that some people use, and is optional.

Another option (*BE CAREFUL, and study the meaning of this command online, first*) is:

```
$ git reset --hard remotes/bugfix
```

```
$ git checkout local/bugfix_2
$ git branch -a
* local/bugfix_2
  master
  remotes/mysubproject1
  remotes/mysubproject2
  remotes/mysubproject3
  remotes/bugfix
  . . .
  remotes/trunk
```

Normally, if we want to remind ourselves what branch we're in:

```
$ git branch
* local/bugfix_2
  master
```

Now we can edit our code, and commit to the local Git repository (not to the remote Subversion repository) as often as we feel necessary:

```
$ vi new_file.c
$ git add new_file.c
$ vi old_file_to_modify.c
$ git commit -a -m "Message about the commit."
```

If you want to combine a number of commits into one, in order to present a simpler history to Subversion, run this command:

```
$ git rebase -i
```

Then follow the directions in the edit session that is entered.

Also, it may recommend you use a command like the following, even though you've already set up tracking.

```
$ git branch --set-upstream local/bugfix remotes/bugfix
```

Once you've committed and rebased, you can finally commit to Subversion, doing a dry-run to double-check what will be dcommited, first:

```
$ git svn dcommit --dry-run
$ git svn dcommit
```

One more thing. If you've edited other files in the meantime, the dcommit will fail because the index holding the modified files has not been updated. If you really do not want to commit these recently-changed file yet, then stash the modifications:

```
$ git stash
$ git stash list
$ git svn commit
$ git stash apply
```

The first command saves the modifications, and the second shows what is stashed. The third command will no longer be aborted.

The fourth command then brings back the modified files.

Warning

Be sure to study Creating Remote Subversion Branches in order to fully understand handling Subversion branches.

6.4.4 Making a "Bare" Copy of a Repository

Once a repository has been cloned, it can be converted into a "bare" repository. A bare repository does not have a working tree. Rather, it has only the Git information and the archived documents. Thus, it is a lot smaller. Plus, it can be copied to a thumb drive so that someone else can clone it.

Bare repositories are especially important for the common case of a remote repository used only as a push and pull source (i.e. not used directly for editing).

The problem with a regular repository, created with `git svn clone`, that remote push and pull do affect with the working tree, and one gets a warning when pushing to a regular repository.

We can avoid this by using a bare repository for the clone. However, in migrating from svn, the `git svn clone` always creates a working tree.

The solution is to convert it to a bare repository after the cloning.

First clone the repository as discussed in Fetching a Copy of a Repository or, preferably Cloning a Standard Subversion Layout. Let's say we've done the latter, and have a project repository. Then convert it to a bare repository by cloning its `.git` hidden directory:

```
$ git clone --bare project/.git project.git
```

Note that, by convention, bare repositories always have the extension `.git`.

Then you can clone this bare repository in another working area in order to get an editable version.

```
$ git clone /path/to/project.git
```

This results in a new project directory that has the source code in editable format.

Some things to note:

1. The bare repository is *much* smaller than the cloned working copy. For example, the cloned `dev` repository is about 8 Gb in size, while the bare repository, containing basically the same information, is only about 400 Mb in size.
2. You don't have to clone a whole repository, even for doing work on it. You can clone subdirectories as well.

3. You can copy the whole `project.git` bare repository to another machine, either via the network or via a USB stick. You can use that remote copy as a private repository that can be accessed only by people that have accounts on that machine. `http://gitref.org/remotes/` explains how to keep your various copies of the repository in synch using `git push`, `git pull`, and `git remote`.
4. However, at this time, using a remote repository to synch with a Subversion repository is beyond the scope of this document. This document assumes you will interact only with the local repository you created to interact directly with the Subversion repository.

Some things to note:

1. The bare repository is *much* smaller than the cloned working copy. For example, the cloned dev repository is about 8 Gb in size, while the bare repository, containing basically the same information, is only about 400 Mb in size.
2. You don't have to clone a whole repository, even for doing work on it. You can clone subdirectories as well.

6.4.5 Rebasing (Updating) from the Repository

The following command pulls in any changes that were made to the Subversion repository since the repository was first cloned or last updates.

```
$ git svn fetch
```

The following command does the above, and it also sets the revision to the HEAD revision.

```
$ git svn rebase
```

These commands are analogous to `svn update`. The latter command is generally the one you want to use.

Note that this rebasing cannot be done with outstanding local changes in place. The changes have to be *add*ed to the Git index, and then have to be *commit*ted to Git. Then rebasing can be performed.

An alternative is to temporarily stash the modifications:

```
$ git stash
$ git svn commit
$ git stash apply              (or pop)
```

Git's stash feature actually uses a stack, so that more than one set of stashes can be made. See `git help stash`.

6.4.6 Committing Local Changes to the Repository

Unlike in Subversion, local changes to a file are not seen by Git until the developer makes them visible by doing a local *add*. Once visible to Git, the changes can be *commit*ted.

This feature is a great way to avoid having accidental edits committed, without having to remember to *revert* the accidental changes.

Also unlike Subversion, committed changes are not seen by the remote (original) repository, until the commits are *pushed* to the repository.

When a file is changed (whether modified, added, or deleted), git requires the following command to make the change visible to Git, and then, if the changes are still thought to be good, commit them:

```
$ git add <the_modified_files>
$ git commit -m "Message...."
```

You can combine adding the changes with committing them, if the files are merely modified:

```
$ git commit -a -m "Message...."
```

These changes are still local. They need to be *pushed* to the Subversion repository. This command will also bring your local copy up-to-date:

```
$ git svn dcommit
```

If conflicts occur, you have to first rebase the remote changes on top of your local changes:

```
$ git svn rebase
```

It's generally better to make sure your local repository is clean, though.

Also note that, for each time that you run `git commit`, a new Subversion revision will result. So, if you then the commit command five times, then `git svn dcommit` will record five new revisions into Subversion.

If you want to combine a number of commits into one, in order to present a simpler history to Subversion, run this command:

```
$ git rebase -i
```

Then follow the directions in the edit session that is entered. Basically, you will leave the first commit as a `pick`, and change the rest of them to a `squash`. Google will show you more use cases for *interactive rebasing*.

6.4.7 Handling Local Branching

Creating local branches in Git is a good way to work without interfering with the work of others.

```
$ git checkout master
$ git branch new_feature
$ git checkout new_feature
```

These commands make a local branch off the `master` branch, and call it `new_feature`. Changes committed to this branch that won't affect your master branch. One can switch back and forth between the local branch and the master branch.

This branch is connected to the trunk of the Subversion repository, so any changes dcommitted from this branch will be merged into the Subversion trunk and appear in the local trunk the next time `git svn rebase` is run, when the trunk is checked out.

We generally will not be using the `trunk` workflow.

6.4.8 Diff'ing Code

The cool thing about having a local Git repository is that your work can continue, even when the network is down. For example, if you need to peer-review some changes, you can do so even if there is no access to the Subversion server, as long as you rebased from the Subversion archive in time.

Doing differences is straightforward, but the output of the diff-tool built into Git is not really suitable for human consumption. So the first thing to do is set up Git to use your favorite diff-tool. It is easiest to set up to use certain well-known open-source tools, such as *vimdiff* and *kdiff3*, but you can set up for other tools using a wrapper script.

We've set up *gvimdiff*, as shown in this section of `~/.gitconfig`:

```
[diff]
   tool = gvimdiff
[difftool]
   prompt = false
[alias]
   d = difftool
```

The d alias merely allows one to save a little typing time – instead of type `git difftool ...`, one can type `git d ...`.

So now you want to review a branch of code that you've already obtained by a `git svn rebase`. The author of the changes has provided the name of the branch and the beginning and ending revision numbers. Let's assume the location of the branch on your system is `BRANCH`, which might have a value of `~/git/dev/project/branches/mysubproject`.

Run the following command to get a list of all the revisions that affected that path:

```
   $ git log $BRANCH > changes.log
```

Then review `changes.log` to find the revisions. For example, search for "@531". You might find something like this:

```
commit 5890c2d148f5ca7ecd2f82c34d3b583e60ed2a62
Author: ahlstrom <ahlstrom@aa7f1e7f-25a7-d04a-8b77-338273dbcc93>
Date:   Tue Apr 1 19:37:10 2012 +0000

    git-svn-id: https://svnserver:443/svn/dev@531
       aa7f1e7f-25a7-d04a-8b77-338273dbcc93
```

Grab the `commit` number. You only need the first few values, if they uniquely specify a commit number.

Get the commit numbers for the range of revisions. Use them in one of the following commands:

```
   $ git difftool 5890c2d148f5ca7ecd2f82c34d3b583e60ed2a62 85663ad65b198a80b469ea8b5ed19ff9353dfe40

   $ git difftool 5890c2d1 85663ad6
```

The result is that each file change is brought up, serially, in `gvimdiff`, and is easily examined, thanks to the very colorful output of that diff-tool and its feature of line-folding.

The comparable command in Subversion would be

```
   $ svn diff -r 604:610 --diff-cmd svndiff
```

where `svndiff` is a script that rearranges command-line arguments to a program like *gvimdiff* so that it properly compares the two versions of each file.

6.4.9 Creating Remote Subversion Branches

Cloning a Standard Subversion Layout talks about setting up a bridge to a Subversion project having the standard trunk/branches/tags layout, handling branches that are already in place. This section goes into more detail, and shows how to create Subversion branches using the git interface.

1. Start Subversion branch in Git.

   ```
   $ git svn branch -m "Branch for bug fixes" local/bugfix
   $ git checkout --track -b local/bugfix_2 remotes/bugfix
   $ git reset --hard remotes/bugfix   -or-
   $ git svn rebase
   $ (now you can happily hack on your code)
   $ (git commit, git commit --amend, git rebase -i HEAD~10, etc.)
   ```

   `_2` is appended to `bugfix` to avoid warning from Git about ambiguity. Could prepend `local/` to it instead. The *reset* command discards any local file modifications and also resets the Git index (staging area).
2. dcommit to Subversion. The `-dry-run` option checks that it will commit to the proper Subversion branch:

   ```
   $ git svn dcommit --dry-run
   Committing to https://svnserver:443/svn/.../project/branches/bugfix_branch ...
   $ git svn dcommit
   ```

3. Merging trunk into branch.

```
        $ git checkout master
        $ git svn rebase
        $ git checkout local/bugfix_2
        $ git merge --squash master
        $ git commit -m "Bring branch up-to-date with trunk"
```

4. Merging branch back to trunk.

```
        $ git checkout master
        $ git svn rebase
        $ git merge --squash local/bugfix_2
        $ git commit -m "Merge branch into trunk"
```

We won't generally be using the trunk in it, though, so see Merging Between Branches for more information.

Warnings:

1. Don't try to rebase already-dcommited changes. They do not belong to you anymore.
2. Always run "git svn dcommit --dry-run" before real commit to check in which branch your changes will be committed.
3. Always merge branches with "--squash". Otherwise git-svn will do it for you and can epic-fail sometime.
4. Although Subversion 1.5 and later can track merges, don't try to merge branches with pure svn and git-svn on same branch.

See `http://trac.parrot.org/parrot/wiki/git-svn-tutorial` for detailed information on how to handle branching in git-svn.

Also see `http://utsl.gen.nz/talks/git-svn/intro.html#howto`, which adds how to get incredible project compression with Git.

6.4.10 Merging Between Branches

In the Subversion archive, we don't really use `trunk` much (the last revision in it is currently `r200`).

Instead, we're using `mybranch` as a kind of provisional trunk. We create bug-fix branch by doing an `svn copy` of the `mybranch` URL. Then we do our editing and testing on the branch. Finally, we merge the branch into the `mybranch` directory, verify it, and then commit it.

This section explains how to do this process using the Git/Subversion bridge. It assumes that you have either tracked and checked out an existing branch as discussed in Cloning a Standard Subversion Layout or have created a Subversion branch using Git as discussed in Creating Remote Subversion Branches.

You have already edited your code, tested it, and have committed it back to the branch. For this example, the local branch is called `bugfix_2`, and it tracks the remote Subversion branch `bugfix`.

Now you want to somehow merge the changes from `bugfix` into `mybranch`.

We already have the local branch `bugfix_2`. We need a local branch to trach the remote branch `mybranch`:

```
$ git checkout --track -b local/mybranch remotes/mybranch
```

Generally, it may be awhile before you actually do the merge, so you want to make sure that `mybranch` and `bugfix_2` are up to date:

```
$ git checkout local/bugfix_2
$ git svn rebase
$ git checkout local/mybranch
$ git svn rebase
```

We're not ready to merge just yet. We want to get an idea of the amount of merging and what will be merged. First, get a summary of the differences between our working branch, `local/bugfix_2`, and our merge destination, `local/mybranch`.

```
$ git checkout local/bugfix_2
$ git diff --name-status local/mybranch > diffs.txt
```

Examing `diffs.txt` will give you an idea of how many and which files were modified, added, or deleted.

If you want a more detailed look, then use your diff-tool:

```
$ git checkout local/bugfix_2
$ git difftool local/mybranch
```

The `local/mybranch` files appear in the left pane of your diff-tool, and the `local/bugfix_2` changes will appear in the right pane of the diff-tool.

We're ready to merge (locally) `local/bugfix_2` into `local/mybranch`. We set the destination branch as the current checkout, and then do the merge:

```
$ git checkout local/mybranch
$ git merge --squash local/bugfix_2
```

You will likely get a list of *CONFLICTS*. Edit each of the flagged files to remove the conflict and the conflict markers (just as in Subversion). Then tell Git they are resolved:

```
$ git add file1.c file2.c ...
```

If the conflict occurs because a file was removed, use:

```
$ git rm file.dll
```

Then commit all of the changes, whether due to the merge or due to the resolution of conflicts.

```
$ git commit -a -m "Made merge and resolved merge conflicts."
```

We've merged our bug-fixes into the local mybranch branch. Now we need to see that they took. You may need to set the upstream first, even though `mybranch` already tracks `remotes/mybranch`.

```
$ git branch --set-upstream mybranch remotes/mybranch
$ git rebase
```

Now do a dry run of dcommit to be sure it will be committed to the Subversion `mybranch` branch. Then actually do the commit.

```
$  git svn dcommit --dry-run
Committing to https://svnserver:443/svn/dev/project/branches/mybranch ...
$  git svn dcommit
```

6.5 Other Useful Commands and Workflows

This section presents brief and to-the-point commands for various workflows and actions.

6.5.1 Merging Subversion Branches Through Git

Here, we're assuming that one has cloned a Subversion repository as per Cloning a Standard Subversion Layout. Then one has created a Subversion branch, using either Subversion or the methods of Creating Remote Subversion Branches. So now one wants to make some edits and local commits on the branch, and then merge them into another branch, and make sure that the merge is properly committed to the Subversion repository.

```
 1. $ git checkout --track -b local/bugfix_branch remotes/bugfix_branch
 2. $ git svn rebase
 3. $ git commit ...
 4. $ git rebase -i
 5. $ git svn dcommit
 6. $ git checkout --track -b local/mybranch remotes/mybranch
 7. $ git svn rebase
 8. $ git diff local/mybranch local/bugfix_branch
 9. $ git branch
10. $ git merge --squash --no-commit local/bugfix_branch
11. $ git status
12. $ git commit -m "Merged bugfix_branch into mybranch."
13. $ git rebase -i             [optional]
14. $ git svn dcommit --dry-run
15. $ git svn dcommit
```

$ git branch –set-upstream local/bugfix_branch remotes/bugfix_branch

Here's what each step does:

1. Create a new branch called `local/bugfix_branch`, have it track the existing Subversion branch `remotes/bugfix_branch`, and check out that branch. Verify using `git branch`.
2. Fetch and merge any changes in the branch, to be up-to-date.
3. After each round of editing, commit your code to that branch. Note that it is committed locally.
4. If desired, rebase interactively to reorganize your commits for Subversion.
5. Commit all the changes to Subversion's version of the branch. This can be helpful in case you mess up later.
6. Create a new branch called `local/mybranch`, have it track the existing Subversion branch `remotes/mybranch`, and check out that branch. Verify using `git branch`.
7. Fetch and merge any changes in that branch, to be up-to-date.
8. Preview the changes made in `local/bugfix_branch` relative to the branch we want to merge into. If set up, you can replace `diff` with `difftool` (e.g. I use *gvimdiff* to view the differences between two files.) You can also use the `-name-only` option to get a simple list of files that changed.
9. Verify you are on the `local/mybranch` branch, the destination of the merge.
10. Merge the `local/bugfix_branch` branch into the current branch without committing the changes or moving the `HEAD`. You will want to resolve conflicts, perform spot checks, or perhaps even make a few additional edits at this point.
11. Verify that only the files desired, and all of them, are staged for commit. Use `git add` to stage any desired files that are not yet staged.
12. If all is well, commit the merge.
13. If desired, interactively rebase the merge to combine commits.
14. Verify that the merge changes will go to the expected Subversion repository. -15 Push the merge changes to Subversion.

After doing all that, it pays to go to the destination directory (`mybranch`) in a regular Subversion workarea and do an `svn update` to verify that all the files make it, and there are no conflicts. If there are conflicts, you probably want to resolve them with commands like the following:

```
$ svn resolve --accept=theirs-full Projects/libs/libatis/po/Makefile.am
```

If you do make more changes, be sure to be back to `local/mybranch` and run `git svn rebase` to keep your local branch up-to-date.

6.5.2 "Your branch is ahead..."

Let's say you issue the following command and see the following output:

```
$ git status
# On branch local/bugfix_branch
# Your branch is ahead of 'bugfix_branch' by 1 commit.
```

In normal circumstances, this simply means that you have not yet run the following command to get the Subversion archive up-to-date with the changes you committed in git. Therefore, you just need to commit to Subversion:

```
$ git svn dcommit
```

6.5.3 Git Garbage Collection

Run the following command in your Git repository, once in awhile, to keep it working fast. It makes sure that objects are packed together to increase the speed of accessing them.

```
$ git gc
```

This command re-compacts the data in your repository. Normally, git runs this command automatically every so often.

6.6 Feature Tables

Constructed from:

`http://git.or.cz/course/svn.html`

Function	Git	Subversion
Creating a new repository	git init git add . git commit	svnadmin create REPO svn import `file://REPO`
Track a project stored at URL	`git clone URL`	`svn checkout URL`
Track latest from the current project	git pull	svn update
Committing changes to edited documents	git commit -a	svn commit
Checking changes	git diff	svn diff | less
Checking changes more specifically	git diff REV PATH	svn diff -rREV PATH
Applying a patch created by the VCS	git apply	patch -p0

Obtaining project status	git status	svn status
Obtaining project status	git status	svn status

6.7 References

1. `http://stackoverflow.com/questions/871/why-is-git-better-than-subversion` A wide-ranging discussion comparing Git and Subversion
2. `http://whygitisbetterthanx.com/#svn` Lists the advantages of Git. Performances comparisons, sample workflows, and other diagrams.

Chapter 7

Debugging using GDB and Libtool

Author

Chris Ahlstrom

Date

2009-05-31

Last Edits:

2015-10-05

License:

$XPC_SUITE_GPL_LICENSE$ (see Licenses, XPC Library Suite)

No debugging is provided, since this documentation could be part of a more comprehensive documentation package covering many libraries and applications.

Chapter 8

Unit Test Coverage and Profiling

Author

Chris Ahlstrom

Date

2009-03-14

Last Edits:

2015-10-05

License:

$XPC_SUITE_GPL_LICENSE$ (see Licenses, XPC Library Suite)

No coverage_profiling is provided, since this documentation could be part of a more comprehensive documentation package covering many libraries and applications.

Chapter 9

Making a Debian Package

Author

Chris Ahlstrom

Date

2009-01-01

Last Edits:

2015-10-05

License:

$XPC_SUITE_GPL_LICENSE$ (see Licenses, XPC Library Suite)

Chapter 10

XPC C Unit Test Library

Author

Chris Ahlstrom ahlstromcj@gmail.com

Version

1.1

Date

2008-03-23

Last Edit:

2015-10-05

License:

$XPC_SUITE_GPL_LICENSE$ (see Licenses, XPC Library Suite)

Although this documentation could be part of a more comprehensive documentation package covering many libraries and applications, we include a directive anyway. If it causes problems as part of XPC suite of projects, substitute the following Doxygen directive:

Chapter 11

XPC C Unit Test Library

Chapter 12

XPC C++ Unit-Test Library

Library:

xpccut++

Author

Chris Ahlstrom

Version

1.1

Date

2008-03-28

Last Edits:

2015-10-06

License:

$XPC_SUITE_GPL_LICENSE$ (see Licenses, XPC Library Suite)

Index