XPC C/C++ Unit-Test Libraries

1.1.3

Generated by Doxygen 1.8.17

1 XPC C Unit Test Library

Note that you can build the detailed-design documentation by running *doxygen* against the `doxygen.cfg` in directory `xpccut-1.0/doc/dox`.

1.1 Introduction to the C Unit-Test Library

The **XPCCUT** library is a *cross-programming C unit test* library that provides a flexible and easy-to-use unit-test framework.

We don't pretend this library does everything needed for good unit-testing. Here's what it does:

- Provides flexible support for unit-test options including:
  - Test descriptions.
  - Quiet running (little or no output) or verbose operation.
  - Timing the tests.
  - Running portions of the test in an interactive manner.
  - Returning a single pass/fail value ("0" means "passed", "1" means "failed").
- Provides for some organization of the unit-tests:
  - By group (e.g. all tests for a given class).
  - By case (e.g. a single member function of a class).
  - By sub-test (e.g. an important corner case for a function). This selectability makes it easy to focus on problematic cases during debugging.
- Allows for easy discovery of the exact failed test or sub-test.
- Provides support for semi-automated regression testing.
- Generates a clean-looking and concise test report.

Here's what it *doesn't* do:

- Provide a way to generate a unit-test framework.
- Provide links to requirements for a project.
- Provide for complete automated unit-test coverage. (However, instructons for coverage-testing are provided in this documentation).
- Provide for easy profiling. (However, instructons for profiling are provided in this documentation).

Information about the options supported can be found in the documentation for the `unit_test_options.c` module.

1.2 Additional Features

The **XPCCUT** library also has support, some of it merely rudimentary, for the follwoing features.

- Internationalization via the GNU gettext facility.
- Additional `configure` options for debugging, enabling certain pointer checks, test-coverage, profiling, and stack-checking.

1.3 Dependencies

The **XPCCUT** project has dependencies at two levels: user and developer. The user will normally use a tar file with a `configure` script in it, while the developer will use the `bootstrap` script to construct the former script. Hence, the developer must install more dependencies.

1.4 User Dependencies

- *GNU C/C++ compilers*.
- *doxygen*. Necessary for building the HTML documentation.
- *graphviz*. Necessary for building the diagrams that are included in the HTML documentation.
- *pkg-config*.

1.5 Developer Dependencies

- *autoconf*.
- *autoconf-archive*.
- *automake*.
- *libtool*.
- *texlive*.

1.6 Generating the PDF Document

Given that you are reading this documentation in a browser, most likely, you have already made the documentation via these steps:

```
$ cd xpccut-1.0/doc/dox
$ make
$ cd html
$ $BROWSER index.html
```

Therefore, creating a PDF version of this documentation in a single, large document is a trivial additional step.

```
$ cd xpccut-1.0/doc/dox
$ cd latex
$ make pdf
$ mv refman.pdf XPCCUT-1.0-Reference_Manual.pdf
$ cp XPCCUT-1.0-Reference_Manual.pdf to_wherever_you_want
```

The PDF document is about 360 pages, nicely laid out, with diagrams, a table of contents, an index, a navigable side-bar, and hyperlinks, all easily kept up to date as long as you modify your code-comments diligently with your favorite programmer's editor.

Try doing that with a **WYSIWYG** word-processor like *Microsoft Word* or *OpenOffice Writer*!

1.7 Using the XPC CUT Library

Since this library is supported by *automake*, complete instructions are provided in the *XPC Automake* document, in it's *Usage* subsection. This document is part of the **XPC** suite, and is found in the `xpc_suite` directory, which is the parent directory of `xpccut`.

Instructions for *Windows* builds are a bit more difficult, since there is no real convention for handling library project locations for either installation or header files.

At some point, the reader may be able to find examples in the other XPC suite projects. We'll note them here as we finish them.

1.8 References

1.8.1 Internal References

Also see the sidebar for access to references that may not be presented here (e.g. the source code).

1. Licenses, XPC Library Suite
2. Debugging using GDB and Libtool
3. Unit Test Coverage and Profiling
4. Making a Debian Package
5. XPC Application License
6. XPC Library License
7. XPC Documentation License
8. XPC Affero License

1.8.2 External References

There are a lot of other unit-test facilities out there. Be sure to try as many as possible, to see which one hits the sweet spot for your projects.

1.8.2.1 Test Frameworks

1. `http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks`
2. `http://code.google.com/p/googletest/wiki/GoogleTestDevGuide`

1.8.2.2 Coding Style

1. `http://code.google.com/p/google-styleguide/`
2. `http://www.gnu.org/prep/standards/standards.html`
3. `http://lxr.linux.no/linux/Documentation/CodingStyle`
4. `http://www.research.att.com/~bs/bs_faq2.html`

2 XPC C++ Unit-Test Library

This file introduces the **XPC CUT++** unit-test library. This document is fairly complete, but still a work in progress.

Note that you can build the detailed-design documentation by running *doxygen* against the `doxygen.cfg` in directory `xpccut-1.1/doc/dox`.

2.1 Introduction to the C++ Unit-Test Library

The **XPC CUT++** library is a "cross-programming C++" library for a flexible, but fairly simple, unit-test framework.

It is actually primarily (but not completely) a **C++** wrapper for the **XPC CUT** library, so please peruse XPC C Unit Test Library for more information. You may need to generate that library's documentation to read about it.

We don't pretend this library does everything needed for good unit-testing. Here's what it does:

- Provides flexible support for unit-test options including:
  - Test descriptions.
  - Quiet running (little or no output).
  - Timing the tests.
  - Running portions of the test in an interactive manner.
- Provides for some organization of the unit-tests:
  - By group (e.g. all tests for a given class).
  - By case (e.g. a single member function of a class).
  - By sub-test (e.g. an important corner case for a function).
- Provide support for automated regression testing.
- Generates a clean-looking and concise test report.
- Allows for fairly easy discovery of the exact failed test or sub-test.

Here's what it *doesn't* do:

- Provide a way to generate a unit-test framework.
- Provide for complete unit-test coverage.
- Links to requirements for a project.
- Allow for easy profiling.

In particular, it shouldn't be considered as a replacement for the popular CppUnit project at `http://cppunit.sourceforge.net/cppunit-wiki`.

2.2 Automake Notes Specific to XPC CUT++

Using *automake* to build a multiple library project is interesting.

We want to incorporate the specially-compiled version of `libxpccut` (the **C** library for which `libxpccut++` is a class wrapper) in this **C++** library, because the special version of `libxpccut` does not check null pointers, and will never be installed.

One way to do it is to modify the `Makefile.am` to include the **C** source-code files from the `xpccut-1.1` directory. This looks clumsy, but ties the **C** and **C++** source files together logically.

Another way is to treat the special version of libxpccut as a *convenience library* and add its contents to libxpccut++. This method has the disadvantage that we have to provide code in the bootstrap script to explicitly build the **C** version of the library with XPC_NO_THISPTR defined.

Let's try the second way first, even though it is clumsier. We want to learn more about the LIBADD macro.

Both methods illustrate how clumsy it can be to treat handle a library that is completely dependent upon an independent library.

In order to incorporate the object modules in libxpccut.a into libxpccut++.la, we make changes to the src *and* tests Makefile.ams. These two changes have the advantage of leverage libtool better.

First, src/Makefile.am. All we have to do is add this definition:

```
libxpccut___la_LIBADD = ../../xpccut-1.1/src/libxpccut.la
```

Next, tests/Makefile.am.

```
LIBDIRS = -L../src/.libs
libraries = -lxpccut++
```

Here, all we've done is removed the explicit inclusion of the libxpccut library from the make process.

We leave the DEPENDENCIES alone, just in case the developer modified libxpccut while fixing bugs.

The final result is that the src build adds the libxpccut library to the libxpccut++ library. This is the actual call that links them:

```
/bin/sh ../libtool --tag=CXX   --mode=link
   g++  -g -O2  -Wall -Wextra -pedantic -D_REENTRANT -DAPI_VERSION=""
   -ggdb -O0 -DDEBUG -D_DEBUG -DNWIN32 -fno-inline -version-info 1:1:0
   -o libxpccut++.la -rpath /usr/local/lib
   cut.lo cut_options.lo cut_status.lo
   ../../xpccut-1.1/src/libxpccut.la
```

Now, once we got the cut_unit_test application built using the libxpccut library with the XPC_NO_THI↩ SPTR option in force, we found that our unit test application would segfault. We had uncovered some severe issues with our easy classes, due to their default constructors. This resulted in some changes in architecture to make the classes more self-contained and safer to use.

2.3 References

1. http://www.gnu.org/software/autoconf/manual/automake/Libtool-Convenience-↩ Libraries.html Discusses how to build a library from 'convenience' libraries.

3 Debugging using GDB and Libtool

This file describes the very basics of debugging using *gdb*.

3.1 Introduction to Debugging

This document describes how to do basic debugging.

It is currently somewhat incomplete. For code-coverage and profiling, see Unit Test Coverage and Profiling

3.2 Basic Debugging

We won't pretend to teach debugging here; this section is just to get you started.

See some of the items in References for much more information.

3.2.1 Building Without Libtool Shared Libraries

The first thing to note is that it is easier to debug if the project is built with only the static libraries in *Libtool*. With *libtool* and shared libraries, you have to use *libtool* to run the debugger, as described in section Debugging a Libtoolized Application.

However, if the application is built using only the static libraries, then the library code is part of the application, and the application can be debugged normally. To build the application without shared libraries, they must be disabled. Run (or re-run) the 'configure' script with the following options:

```
$ ./configure --enable-debug --disable-shared
```

If you know you want to debug the project before you bootstrap it, then bootstrap it this way:

```
$ ./bootstrap --enable-debug
```

This command sets up the make system, then executes the `./configure` command shown above, including the –disable-shared.

3.2.2 GDB

3.2.3 CGDB

3.2.4 DDD

3.3 Debugging a Libtoolized Application

An application that has been created using Libtool, but that has not yet been official installed, requires some special handling for debugging.

As with a normal application or library, the source-code a build system for a project reside in a small directory hierarchy with directories such as `src`, `include` (or, in our projects, `include/xpc`), and `tests`.

In a normal library, the library file (archive) is created in the `src` directory.

This library is linked to the test application, which is created in the `tests` directory, and is a binary file of ELF format.

But there are some differences once Libtool is used:

- src.
  - libxpccut.la. Instead of a single static archive (library) that is created, this file is created. It contains some information about the shared library, static library, installation location, and other items of information that describe the newly-created library.
  - .libs. The newly created static library is deposited here, instead. Here is what is in this hidden directory:
    - The original, static library, libxpccut.a.
    - The shared library, libxpccut.so.0.1.1.
    - Links to the static and shared library, similar to what one would see when the library is installed.
    - All of the object modules for each source-code module, *.o.
    - A copy of libxpccut.la, called libxcpcut.lai, that differs only in having a tag *installed* set to *yes* instead of *no*.
- tests.
  - unit_test_test. Instead of a binary executable file, this item is actually a script. When executed, it looks as if the original application is being executed directly. But it is actually a script that sets up so the shared libraries can be found and accessed while the unininstalled (and hidden) application runs.
  - .libs. Just as with the src/.libs directory, this hidden directory contains the actual binary for the application. There are two binaries there – we don't yet know what the difference is – a topic for the future.

Try running the following command:

```
$ gdb unit_test_test
```

gdb will complain:

```
".../xpccut-1.1/tests/unit_test_test": not
in executable format: File format not recognized
```

This makes sense, since unit_test_test is a script. In order to debug the application, we have to let Libtool execute the debugger:

```
$ libtool --mode=execute gdb unit_test_test
$ libtool --mode=execute cgdb unit_test_test
$ libtool --mode=execute ddd unit_test_test
```

Commands like these can also be used to execute valgrind and strace, too.

3.4 References

- http://heather.cs.ucdavis.edu/~matloff/debug.html Norm Matloff's Debugging Tutorial
- http://nostarch.com/debugging.htm N. Matloff and P.J. Salzman (2008) "The Art of Debugging with GDB, DDD, and Eclipse", a very good tutorial book.

4 Unit Test Coverage and Profiling

This file provides a tutorial on using the **GNU** test-coverage and profiling tools.

4.1 Introduction to Coverage Testing and Profiling

This document describes how to use *gcov* (coverage testing) and *gprof* (code profiling), using the **XPCCUT** unit-test project as an example. As befits the purpose of that project, that's where this document can be found.

Coverage testing analyzes the code to determine which functions an application exercises during its run.

Code Profiling analyzes executing code to determine where most of the CPU time is being spent.

Together, both methods ensure that your code is as well-tested and as fast as you can make it.

4.2 Installing 'gcov' and 'gprof'

The means of installing *gcov* and *gprof* vary with your Linux (or BSD) distribution. (We won't even get into *Solaris*.)

gcov is provided as part of the *gcc* package, and *gprof* is provided by *binutils* package (in Debian, anyway).

Other potentially helpful (Debian) packages are available:

- `lcov`. A set of PERL scripts that provide colored HTML output with overview pages and various views of the coverage.
- `ggcov`. A GUI for browsing C test-coverage results.

4.3 The 'gcov' Coverage Application

What is *gcov*? From the GCOV(1) man page.

```
gcov is a test coverage program.  Use it in concert with GCC to analyze
your programs to help create more efficient, faster running code and to
discover untested parts of your program.  You can use gcov as a
profiling tool to help discover where your optimization efforts will
best affect your code.  You can also use gcov along with the other
profiling tool, gprof, to assess which parts of your code use the
greatest amount of computing time.
```

The *GNU gcov* manual can be found at:

http://gcc.gnu.org/onlinedocs/gcc/Gcov.html

A good PDF version is available in Chapter 9 of the *GNU gcc* manual, PDF version:

http://gcc.gnu.org/onlinedocs/gcc.pdf

Also very helpful is *The Definitive Guide to GCC*, Chapter 6. See the References section for more details.

4.3.1 Building For 'gcov' Usage

An application to be analyzed with *gcov* needs to be compiled in a special manner:

- Use a *gcc* compiler.
- Use no optimization.
- Use the `-fprofile-arcs` option.
- Use the `-ftest-coverage` option.

In the **XPCCUT** project, this is all done easily with these commands:

```
$ ./bootstrap --clean                    (if needed)
$ ./bootstrap --no-configure             -or-
$ ./bootstrap -nc
$ ./configure --enable-coverage
```

Note the `-enable-coverage` option. It is implemented as part of the `m4/xpc_debug.m4` script, which is specified by the `AC_XPC_DEBUGGING` macro call in the **XPCCUT**'s `configure.ac` file.

In addition to the normal arguments, you can specify the following forms:

- –enable-coverage=gdb (the default)
- –enable-coverage=db

The first variant is the default, which specifies *GNU* **gdb** compatibility. The **gcc** flags for the **gdb** variant are:

```
-O0 -ggdb
```

The second variant supports the *BSD* **db** program. The **gcc** flags for the **db** variant are:

```
-O0 -g
```

Once the configuration has been set up, run *make* in the `src` directory.

```
$ cd src
$ make
```

Note that there are `*.gcno` files that are created by this compile, one for each C module that is built. Moreover, these are built both in the `src` and the `src/.libs` directories. (The latter is a hidden directory created by *libtool*.

Next, run *make* again in the `tests` directory.

```
$ cd ..
$ cd tests
$ make
```

Note, again, that there are `*.gcno` files that are created by this compile, one for each **C** module that is built in the `tests` directory.

4.3.2 Running for Coverage

Now we want to run the application once, to see how much module coverage we have.

General principles:

- It seems best to run the application from the project root directory, not from its `tests` directory.
- A file *`my_c_module`*`.gcda` file will be created in the `tests` directory by running the application.
- Additional `*.gcda` files will be created in the `src/.libs` directory.
- The run information in this file accumulates with each run. To start fresh, the `*.gcda` files have to be deleted.
- *Question*: What are the `*.bb` and `*.bbg` files referred to in some of the documentation? They don't appear in our project, so are perhaps obsolete.

Run the application (here, called `unit_test_test`).

```
$ ./tests/unit_test_test
```

Note that we are running it from the *root* of the project. This is necessary in order for files in the `src` directory to get their analysis done and their output files created.

The following items are generated:

- Generates `./tests/unit_test_test.gcda`
- In a *libtool* project such as this one, it also generates `./src/.libs/*.gcda` (one for each C module in `src`).
- However, no `*.gcda` files are generated in `src` itself.

4.3.3 Running 'gcov' for Analysis

Once the test data are generated, with *gcov* information, they need to be analyzed using *gcov*.

Here are some commands to try. These commands can be done inside the proper directory (either `src` or `tests`, as appropriate.)

```
$ gcov <i>my_c_module.c</i>
```

The command above must be run for which the analysis of coverage data is desired.

```
$ gcov -b my_c_module.c
```

The command above generates branch probabilities.

```
$ gcov -f my_c_module.c
```

The command above analyzes function calls.

All of the commands above are useful for analyzing a library module.

In the following command, both are done, and output goes to an additional log file that can be examined at leisure.

```
$ cd tests
$ gcov -b -f *.c > output.log
```

The run produces a `unit_test_test.c.gcov` annotated source file. If the `-f` switch was provided, then `output.log` will also contain the summary information.

To make it easier to do all of this, a `gcov` target is provided in the top-level `Makefile.am` make-file. As a synonym, a `coverage` target is also provided.

Although there is a lot to be looked at in the `*.gcov` file, let's limit ourselves to simply checking for lines that did not get executed. Examining the `unit_test_test.c.gcov` annotated source file, we can search for unexecuted lines – they have `#####` in them.

The first one we find is

```
#####:  157:         unit_test_status_pass(&status, true);
```

and the second is

```
#####:  162:            fprintf(stdout, "  %s\n", _("No values to show ...
```

There are more, but let's just try to eliminate these two unexecuted lines by subsequent runs of `unit_test_test` with additional arguments.

```
$ ./tests/unit_test_test --group 2 --case 2
$ gcov -b -f *.c > output.log
$ vi unit_test_test.c.gcov
```

We tried to get the test-skipping mechanism to work, but looking at the `*.gcov` file, we see we did not succeed. Either another option is necessary, or we have a feature that is not completely implemented! Oh well, let's try the next unexecuted line.

```
$ ./tests/unit_test_test --show-values
$ gcov -b -f *.c > output.log
$ vi unit_test_test.c.gcov
```

This time, it worked, and we see that line 162 has now been executed.

We can keep at it, trying more options until we have tried them all, and verified that every line of code in the test application has been touched.

Next, we need to check the coverage of the library modules.

```
$ cd src
$ gcov -o .libs/ *.c                          -or-
$ gcov -b -f -o .libs *.c > output.log
```

This results in `*.gcov` files not in `.libs`, but in `src`. Let's look at just one, `portable_subset.c.gcov`. It has a lot of unexecuted code, because some of the code is meant purely for error conditions that a good coder will avoid easily. We don't really care about testing this module, since it is meant only for internal usage [another library provides more advanced versions of the functions in the `portable_subset.c` module]. However, let's add a test of one of the functions to the `unit_test_test.c` module.

The function to be tested is `xpc_nullptr()`, and it has its own test-case group in the `unit_test_test.c` module, test group *07*. Here is a summary of what we did to rebuild:

```
$ vi tests/unit_test_test.c include/xcp/portable_subset.c
$ cd src
$ make clean
$ make
$ cd ../tests
$ make clean
$ make
$ cd ..
$ ./tests/unit_test_test --show-values
$ cd tests
$ gcov -b -f *.c > output.log
$ cd ../src
$ gcov -b -f -o .libs *.c > output.log
$ vi portable_subset.c.gcov
```

And you can now verify that `xpc_nullptr()` no longer is left out of the test run.

By the way, if you want to start from scratch, change to the project's root directory, run the following command, and repeat the steps above.

```
$ ./bootstrap --clean
```

There is a lot more that can be tested for coverage. You are welcome to do that, and tell me what I have missed!

4.4 The 'gprof' Profiling Application

gprof is another GNU application for analyzing code. It works at run-time, tabulating whether the program spends most of its time. Hence, it is useful for find the places where optimization (or even fixes or design changes) would do the most good.

Other potentially helpful packages are available:

- `qprof`. A set of profiling utilities for Linux.
- `kprof`. A profiling front-end for gprof, Function Check, and Palm OS Emulator.

4.4.1 Building For 'gprof' Usage

An application to be analyzed with *gprof* needs to be compiled in a special manner:

- Use a *gcc* compiler.
- Use the `-pg` option to link in the profiling libraries.
- Add `-g` for line-by-line profiling.
- Use `-finstrument-functions` if you have your own profiling hooks.
- Avoid optimization.

In the **XPCCUT** project, this is all done easily with these commands:

```
$ ./bootstrap --clean                     (if needed)
$ ./bootstrap --no-configure              -or-
$ ./bootstrap -nc
$ ./configure --enable-profiling --disable-shared
```

Note the `-enable-profile` option. It is implemented as part of the `m4/xpc_debug.m4` script, which is specified by the `AC_XPC_DEBUGGING` macro call in the **XPCCUT**'s `configure.ac` file.

Also note the use of the `-disable-shared` option, which is a standard `configure` option. If this flag is not provided to `configure`, attempt to run the **gprof** command on the executable will result in the following error message:

```
gprof: unit_test_test: not in executable format
```

Apparently, using shared libraries will mess up the profiler.

In addition to the normal arguments, you can specify the following forms:

- –enable-profiling=gprof (the default)
- –enable-profiling=prof

The first variant is the default, which specifies *GNU* **gprof** compatibility. The **gcc** flags for the **gprof** variant are:

```
-pg -O0 -ggdb
```

The second variant supports the *BSD* **prof** program. The **gcc** flags for the **prof** variant are:

```
-p -O0 -g
```

If you want to do code-coverage at the same time, you should be able to add the `-ftest-coverage` option. I have not yet tried this, though. Let me know how it works.

Once the configuration has been set up, run *make* in the `src` directory.

```
$ cd src
$ make
```

Then run it in the `tests` directory.

```
$ cd ../tests
$ make
```

4.4.2 Running for Profiling

Change to the application's directory (*unlike for gcov?*) and run the following commands.

```
$ ./unit_test_test
```

The result of this run is a file called `gmon.out` in the `tests` directory.

4.4.3 Running for 'gprof' Analysis

Once the test data are generated, with *gprof* information, they need to be analyzed using *gprof*.

Here are some commands to try. These commands can be done inside the proper directory (either `src` or `tests`, as appropriate.)

```
$ gprof ./unit_test_test
```

One can add the following options:

- The `-b` option for brief output.
- The `-A` option for annotated source code.

Todo
- Make sure src and tests 'clean' targets get rid of the *.gc* files. CLEANFILES
- Note in xpc_64_bit.m4 that it is /cross/ compilation.
- Let –enable-gcov be the same as –enable-coverage
- Support 'make gcov' or 'make coverage'
- Figure out how to properly add '-lgcov' to LDFLAGS, (if needed)

4.5 References

- `http://gcc.gnu.org/onlinedocs/gcc.pdf` GNU gcc manual
- `http://gcc.gnu.org/onlinedocs/gcc/Gcov.html` gcov – Test Coverage Program
- `http://www.apress.com/book/view/9781590591093` (1st edition) and `http://book.pdfchm.com/the-definitive-guide-to-gcc-second-edition-9569/` von Hagen, William (2006) *The Definitive Guide to GCC*, 2nd edition. Apress, Berkeley, California.

5 GNU Automake

This module provides some notes on using GNU Automake. This section describes the GNU automake setup for the XPCCUT libraries and test applications.

5.1 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.

5.2 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.

5.3 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.

5.4 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.

5.5 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.

5.5.1 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.

5.5.2 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.

5.6 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.

5.7 Libtool

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

5.7.1 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++)

5.8 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.

5.9 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....

5.10 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.

6 Using Git

6.1 Introduction to Using Git

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

6.2 Setup of Git

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

6.2.1 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!

6.2.2 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.

6.2.3 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.

6.2.4 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
```

6.2.5 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
```

6.2.6 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.

6.3 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)

6.4 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."`

6.5 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
```

6.6 Setup of Git Remote Server

6.6.1 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.

6.6.2 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.

6.7 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).

6.7.1 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.

6.7.2 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.

6.7.3 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.

6.7.4 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.

6.7.5 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.)

6.7.5.1 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".

6.7.6 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
```

6.7.7 git merge

TODO

6.7.8 git push

TODO

6.7.9 git fetch

TODO

6.7.10 git pull

TODO

6.7.11 git amend

TODO

6.7.12 git svn

TODO

6.7.13 git tag

TODO

6.7.14 git rebase

TODO

6.7.15 git reset

TODO

6.7.16 git format-patch

TODO

6.7.17 git log

TODO

6.7.18 git diff

TODO

6.7.19 git show

TODO

6.7.20 git grep

TODO

6.8 Git Workflow

TODO

6.9 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 git_svn_page for using the Git-to-Subversion bridge.

7 Making a Debian Package

This file describes how to create a *Debian* package from an *automake*-supported project. We probably need to revisit this process, since the last time was about 2013.

7.1 Introduction

The **XPCCUT** library is a unit-test library for use in C programs. It also serves as the basis for a complementary C++ wrapper library. Its name stands for *Cross-Platform C, C Unit Test*.

The present page describes what we did to create a *Debian* package from this code.

This is our first try at making a Debian package. We welcome any ideas or criticism you have to offer.

7.2 Setup Steps

Here, we have to confess that the *Debian way* covers a lot of variations on a theme, and we are not sure that we've done everything in the proper manner.

The first thing to do is make sure your project is a **GNU autotools** project that builds and installs properly. This build setup can then be supplemented with **Debian** scripts located in a `debian` subdirectory of the project.

7.2.1 Initial Steps

1. Pick the conventions of your project carefully. All projects seem to have different layouts, Some layouts are not handled well by the debhelper tools, resulting in the need for some custom steps to be created.
2. Make sure the various automake targets work, as they are used by the debian scripts to perform these actions:
   - `make`
   - `make install`
3. Verify that the automake creates these items:
   - Directory `/usr/local/include/xpccut-1.1/xpc` (filled with header files)
   - Directory `/usr/local/share/doc/xpccut-1.1/html` (filled with documentation)
   - File `/usr/local/share/man/man1/xpccut.1`
   - File `/usr/local/share/locale/es/LC_MESSAGES/libxpccut.mo`
   - File `/usr/local/lib/libxpccut.a`
   - File `/usr/local/lib/libxpccut.la`
   - File `/usr/local/lib/libxpccut.so`
   - File `/usr/local/lib/pkgconfig/xpccut-1.1.pc`
4. `dh_make -c gpl -e ahlstromcj@gmail.com -n -l -p libxpccut`
5. Edit a bunch of files in the `debian` directory.

7.2.2 Steps for Testing the 'rules' File

1. Bootstrap the project (`./bootstrap`) in order to create the Makefiles, if not already present.
2. Run `fakeroot debian/rules clean` to verify that cleaning results in no errors.
3. Run `debian/rules build` to verify that the source code and documentation build without errors or missing results.
4. Run `fakeroot debian/rules binary` to very that all of the installation steps work. (However, the *fakeroot* man-page says not to do this step! Do it anyway.) See the note below in case this fails.
5. Examine the directories created in `debian/tmp`. They will mirror an actual installation.
6. Examine the `libxpccut1` and `libxpccut-dev` directories created in the `debian` directory. They will also mirror the actual install, though only libxpccut-dev will include the documentation and header-file directories.
7. Examine the `*.deb` files (see below) that are created. You can use *Midnight Commander* (mc) or *Krusader* to see inside them and verify the layout of the installations.

7.2.3 Handling Build Failures

Sometimes building the binary will fail the first time you run it on a host, with a message like this:

```
You needed to add /usr/lib to /etc/ld.so.conf.
Automake will try to fix it for you
/bin/sh: line 6: /etc/ld.so.conf: Permission denied

That didn't work.  Please add /usr/lib to
/etc/ld.so.conf and run ldconfig as root.
/bin/sh: line 13: ldconfig: command not found
```

Just do what it says and try again.

7.2.4 Doing a Complete Rebuild

1. Go to the root directory of the project.
2. Run `dpkg-buildpackage -rfakeroot`

This will create, in the parent directory above the project, the following files:

1. `libxpccut-dev_1.1.0_i386.deb`
2. `libxpccut1_1.1.0_i386.deb`
3. `libxpccut_1.1.0.dsc`
4. `libxpccut_1.1.0.tar.gz`
5. `libxpccut_1.1.0_i386.changes`

Currently, however, we get the following error:

```
dpkg-buildpackage: warning: Failed to sign .dsc and .changes file
```

Fixed yet? If not, tell us how!

7.3 References

We used the following references to figure out what to do, though a lot of backtracking, interpolating, and trial-and-error was necessary:

1. http://debathena.mit.edu/packaging/
2. http://www.debian-administration.org/articles/337
3. https://help.ubuntu.com/ubuntu/packagingguide/C/basic-scratch.html
4. http://www.zoxx.net/notes/index.php/2006/08/09/22-create-a-debian-package-with-dh_make-and-dpkg-buildpackage
5. http://www.mail-archive.com/debian-mentors@lists.debian.org/msg18893.html
6. http://www.netfort.gr.jp/~dancer/column/libpkg-guide/libpkg-guide.html

Reference 5 was most helpful, while reference 4 goes through the whole process quickly, including how to make your own debian repository. Reference 6 is of great help in getting the conventions correct.

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

This file provides information about building XPC in the Mingw environment, along with how to use pthreads in Mingw and Windows.

8.1 Introduction

The **XPC** library suite can be built in **Windows** as well as **Linux**. But rather than use **Microsoft**'s *Visual Studio* to build it for **Windows**, we will use the *Mingw* project to support it.

And, at first, we will build the **Windows** support from with **Linux** (**Debian** and **Gentoo**), rather than in **Windows** itself.

(We will probably add a `vs2010` directory to hold a solution and a number of `*.vcxproj` files, but it is a low priority.)

8.2 Installing Mingw

Mingw and its related tools can be built from source code in various environments, then be installed. This build is a lot of work.

However, prebuilt packages exists for most environments, and that is what we'll use.

8.2.1 Installing Mingw in Debian

In **Debian GNU/Linux**, a number of packages exist to support *Mingw*:

- gcc-mingw32
- mingw-w64
- mingw32-binutils
- mingw32-runtime

To build **XPC** for *mingw*, try these commands for configuration:

```
$ ./configure --host=i586-mingw32msvc --with-mingw32=/usr/i586-mingw32msvc/
$ ./configure --host=amd64-mingw32msvc --with-mingw32=/usr/amd64-mingw32msvc/
```

8.2.2 Installing Mingw in Gentoo

Gentoo provides a script called *crossdev* that knows how to configure and build the various parts of *mingw*:

- sys-devel/crossdev. This package builds and installs
  - dev-util/mingw-runtime
  - dev-util/mingw64-runtime
  - dev-util/w32api

If you use *layman* on **Gentoo**, be sure to check out potential issues with it:

`http://www.gentoo-wiki.info/MinGW`

You might want to add this line to `/etc/portage/package.use`:

```
cross-i686-pc-mingw32/gcc doc gcj multilib nptl openmp libffi -gtk
      -mudflap -objc -objc++ -objc-gc -selinux
```

Supported architectures:

- `i686-pc-linux-gnu`: The default x86 tuple for PCs.
- `x86_64-pc-linux-gnu`: The default tuple for 64-bit x86 machines (such as the AMD 64 and IA64 architectures).
- `powerpc-unknown-linux-gnu`: Support for PowerPC PCs, e.g. Apple Macintoshes.
- `arm-unknown-linux-gnu`: Support for embedded devices based on ARM chips.
- `arm-softfloat-elf`: For embedded devices with ARM chips without hardware floating point.
- `arm-elf`: For embedded devices with ARM chips with hardware floating point.
- `i686-pc-mingw32`: Supports cross-compiling for 32-bit Windows (toolchain based on mingw32).
- `i686-w64-mingw32`: Supports cross-compiling for 32-bit Windows (toolchain based on mingw64).
- `x86_64-w64-mingw32`: Supports cross-compiling for 64-bit Windows (toolchain based on mingw64).
- `avr`: Supports cross-compiling for Atmel AVR-MCU's.

As `root`, run

```
# emerge crossdev
```

Decide the target for which you want to build. Also decide where the cross-compiler is to be installed. Then run a command like the following, adjusting for your choices:

```
# crossdev --target i686-pc-mingw32 --ov-output /usr/local/portage
```

This action will cause about 6 packages to be built. We had to run it twice; the first time this warning came up:

```
!!! WARNING - Cannot auto-configure CHOST i686-pc-mingw32
!!! You should edit /usr/i686-pc-mingw32/etc/portage/make.conf
!!! by hand to complete your configuration
 * Log: /var/log/portage/cross-i686-pc-mingw32-binutils.log
 * Emerging cross-binutils ...
```

You can also run *crossdev* with the `-help` option to see what else you can do.

Here's what was installed on our **Gentoo** system in /usr/local/portage/cross-i686-pc-mingw32:

```
binutils -> /usr/portage/sys-devel/binutils
gcc -> /usr/portage/sys-devel/gcc
gdb -> /usr/portage/sys-devel/gdb
insight -> /usr/portage/dev-util/insight
mingw-runtime -> /usr/portage/dev-util/mingw-runtime
w32api -> /usr/portage/dev-util/w32api
```

To build **XPC** for *mingw*, try this command for configuration:

```
$ ./configure --host=i686-mingw32msvc --with-mingw32=/usr/i686-mingw32msvc/
```

8.2.3 Installing Mingw in Windows

We want install both *mingw* and *MSYS*.

TODO

8.3 Installing Pthreads

This section does not talk about installing the normal **Linux** *pthreads* package. It talks about installing *pthreads-win32* in **Windows** and in **Linux**.

8.3.1 Installing pthreads in Debian

On our **Debian** *squeeze* machine, *mingw* is installed in two locations:

```
/usr/amd64-mingw32msvc
/usr/i586-mingw32msvc
```

8.3.2 Installing pthreads in Gentoo

8.3.3 Installing pthreads in Windows

8.4 References

1. http://en.gentoo-wiki.com/wiki/Crossdev
2. http://www.gentoo.org/proj/en/base/embedded/handbook/
3. http://www.gentoo.org/proj/en/base/embedded/handbook/cross-compiler.xml
4. http://metastatic.org/text/libtool.html
5. http://dev.gentoo.org/~vapier/CROSS-COMPILE-HOWTO
6. http://en.gentoo-wiki.com/wiki/Mingw
7. http://www.gentoo-wiki.info/MinGW
8. http://wiki.njh.eu/Cross_Compiling_for_Win32

9 Licenses, XPC Library Suite

Library: xpc_suite and its libraries, applications, and documents

Author

Chris Ahlstrom

Date

2008-02-24 to 2022-06-01

License: XPC GPL 3

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

10 Todo List

Page Unit Test Coverage and Profiling

Make sure src and tests 'clean' targets get rid of the ∗.gc∗ files. CLEANFILES

- Note in xpc_64_bit.m4 that it is /cross/ compilation.
- Let –enable-gcov be the same as –enable-coverage
- Support 'make gcov' or 'make coverage'
- Figure out how to properly add '-lgcov' to LDFLAGS, (if needed)