libs11n Porting Guide

For libs11n version 1.2.x

s11n-devel@lists.sourceforge.net - http://s11n.net

ACHTUNG #1: *this is a "live" document covering an in-development software library. Ergo... it may very well contain some misleading or blatantly incorrect information! Please help us improve the documentation by submitting your suggestions to our mailing list!*

Abstract:

This document provides a general overview of the libs11n source tree, mainly for purposes of assisting in porting libs11n to platforms other than its “native” GNU/Linux host.

This document assumes that the reader is well versed in C++ and knows what libs11n is:

http://s11n.net

Maintainer: stephan@s11n.net (list: s11n-devel@lists.sourceforge.net)

Table of Contents

1 Preliminaries

1.1 License

> *"You cannot guaranty freedom of speech and enforce copyright law."*
>
> *Ian Clarke*
>
> *"This [document] is encrypted with ROT26 encoding. Decoding it is in violation of the Digital Millennium Copyright Act."*
>
> *Anonymous Software Developer*

This documentation is released into the Public Domain.

1.2 Disclaimers

The obligatory disclaimers include:

1. This manual will make *no sense whatsoever* to most people. It is target at experienced C++ programmers ("intermediate level" and higher), and makes many assumptions about prior C++ knowledge.
2. This software and documentation are PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
3. Reading disclaimers makes you go blind. ;)
4. Writing them is even worse. :/
5. This list of disclaimers might not contain all the necessary disclaimers.

And, finally:

libs11n's author (Stephan Beal) has limited access to platforms other than GNU/Linux. Thus any porting notes for other platforms may be incorrect or untestable/untested. You have been warned.

1.3 Feedback

Please send any and all feedback to our development list:

s11n-devel@lists.sourceforge.net

1.4 An invitation

This project is always interested in feedback from users, be it in the form of criticism, code patches, test reports, documentation, or fan mail. This document, in particular, cannot be made 100% complete without the help of users of various computing platforms. If you are using (or would like to use) libs11n on a platform other than GNU/Linux, your input on this manual would be greatly appreciated.

1.5 Document conventions

We will use the following conventions throughout this document:

All relative filesystem paths are assumed to be relative from the top-most directory of your libs11n source tree unless specifically stated otherwise.

C++ source code is formatted like this:

```
#include <s11n.net/s11n/s11nlite.hpp>
...
```

Shell code (e.g., to be entered at a Unix shell prompt) is formatted like this:

```
prompt > some_command argument1 argument2 ... argumentN
```

If you are viewing this document in black and white, you might be able to tell the difference between C++ and shell code blocks by their differing drop-shadows. If not, context should make clear what type of code you're looking at.

1.6 Trademarks

The following trademarks (or registered trademarks) are known to appear in this document:

- Windows is a registered trademark of Microsoft Corporation
- UNIX is a registered trademark of The Open Group. Here we use *Unix* to generically refer to Unix-like operating systems (Linux, BSD, Solaris, etc.).
- Solaris is a registered trademark of Sun Microsystems
- Linux is a registered trademark of Linus Torvalds
- Macintosh is a registered trademark of Apple Computer, Inc.

Other trademarks may be used here and not properly acknowledged. Where this is the case, it is a case of ignorant omission and not a wilful act of negligence.

1.7 Credits

The following people have contributed, perhaps indirectly, to this document:

- Ashran did the first port of libs11n to Win32 platforms.
- Pete Harlow provided the template/skeleton for this document.
- Aaron Dalton did the initial FreeBSD port, which inspired the creation of this document.

2 An introduction to the source tree

The libs11n source tree structure is based on several years of experience in maintaining various library source trees. While it does not have a “perfect” structure, especially for cross-platform purposes, its structure is fairly logical, straightforward, and maintainable. This chapter will introduce you to the source tree, giving the reader a grasp on where the various parts of the library live.

Note that when we speak about “the” source tree here, we mean the “native” source tree. That is, the one which is used as the master copy of the sources. This is the source tree used for libs11n's native platform, GNU/Linux.

2.1 Getting the sources

The source code can be downloaded from either of the following sources:

- http://s11n.net/download/
- http://sourceforge.net/projects/s11n

The code is managed via CVS [Concurrent Versioning System], and CVS copies can be checked out by following the instructions listed on the SourceForge project page.

2.2 Code portability

The libs11n core source code uses only ISO-standard C++ features. That said, we use features of C++ which are relatively new and not supported on “older” compilers. As a general rule, any compiler which is less-compliant than GNU gcc 3.2.x will not be able to build libs11n. GNU gcc 3.2+ are believed to be suitable, though we do not explicitly test compatibility with arbitrarily older compilers (nor do we support older compilers nor compilers which do not support ISO C++ standards).

The optional plugins support necessarily uses platform-specific APIs for loading shared libraries (DLLs).

All of that said, the *source code* is very portable while the *build tree* is not. The library is primarily developed on

GNU/Linux machines and the build tree is exceedingly Linux-centric. Thus porting libs11n primarily consists of copying the source files into a build system which works on a particular target platform.

Despite being GNU/Linux-centric, the build tools are known to fail on Debian Linux systems because of “unusual” linking requirements on that platform. (We mean “unusual” in the sense of “different than most Linux distributions, but not technically wrong”, and it is not intended to offend any Debian users.) Likewise, the build tools don't work as-is on Cygwin environments because of odd linking rules on that platform.

2.3 libs11n as a static library

libs11n cannot be built as a static library. More correctly, it can be built but will not work as expected. This is a side-effect of the way the classloader works and the fact that unused symbols in a static library are not imported into client code which links to the library. There are reportedly some linker-specific kludges to work around this, but they have not been explored (nor are there plans to explore them).

Thus, building libs11n requires one of the following:

- Build it as a shared library.
- Compile it directly in to client code. The source code is designed to make it simple to include libs11n's code directly into client projects.
- Build it separately and link the resulting object files directly in to client code.

2.4 Overview of code layers

libs11n is broken down into several layers:

- The core library (`src/s11n/`) provides the API necessary for de/serializing data to/from memory but does not provide any file/stream i/o support. The core library includes the often-mentioned classloader API.
- The i/o layer (`src/io/`) provides the “standard” i/o layer. As the library manual describes, clients are free to implement their own i/o layer if they like. Doing so is not trivial, however, and requires implementing one's own file format handlers (amongst other things).
- s11nlite (`src/lite/`) is a wrapper around the core library and the i/o layer. It is the “recommended” client-side interface, due to its simplicity, though client code is also free to directly use the core and default i/o layers if they need to.
- Plugin support (`src/plugin/`) is used by the core library if it is enabled in s11n_config.hpp. This allows the core to transparently load client-side serializable types dynamically from DLLs. The i/o layer optionally uses this to dynamically load new i/o handlers.
- Serialization proxies (`src/proxy/`) are header files for plugging in support for serializing STL and common POD types. These are not used by the above-mentioned layers, and are intended for use by client-side code.
- s11nconvert (`src/client/s11nconvert/`) is a simple tool for converting s11n-compatible data sources between any support formats. It also acts as a demonstration of how to use some of the library's features.

These layers are described in detail in the library manual.

2.5 Source tree structure

The source tree's top-level directory contains the following “significant” directories:

- `doc/` = the documentation, including the library manual and Doxygen support files.
- `include/` = initially empty, “public” libs11n headers are symlinked here during the build process. This symlinking serves several purposes, which we will discuss later on. All libs11n-related headers formally live somewhere under `include/s11n.net/s11n/`, and this is how implementation files reference them (e.g., `#include <s11n.net/s11n/foo.hpp>`).
- `src/` = various subdirectories containing sub-parts of the library. Sub-modules include:
    - `s11n/` = the core library
    - `io/` = the “standard” i/o layer. That is “one possible implementation”, since the core library is ignorant of all i/o details and interfaces.

- `lite/` = the s11nlite abstraction layer.
- `proxy/` = headers implementing/registering various s11n proxies, including support algorithms for proxies. These are not part of the core itself, but does provide the implementations for serializing STL containers and common POD types.
- `client/` = applications which use the library, namely sample programs and s11nconvert.

- `toc/` = the build tools. They only work on platforms hosting GNU versions of common system tools, like GNU sed, GNU tar, GNU make, etc. For porting purposes, this directory can normally be ignored/omitted.

For porting purposes, the `src` subdirectory is primarily of interest, and the `include` directory may also be interesting (see below for more info on that directory).

2.5.1 The philosophy of the include/ directory

Often times in large projects, disparate parts of the project become inter-dependent on each other. This normally shows up as references to header files of other parts of the project. To avoid such circular dependencies, and to help enforce a clean build structure, this library uses a tried-and-true approach to handling header files. We will briefly describe it here...

Header files themselves are typically stored in the same directory as the implementation files (the .cpp files). In practice, this is normally easier for developers to work with than physically separating the include files and implementation files. Based on this, one might expect the implementation files to have #includes similar to the following:

```
// myfile.cpp
#include "myfile.hpp"
```

In this library, however, the implementation files use fully-qualified includes, like this:

```
// myfile.cpp
#include <s11n.net/s11n/myfile.hpp>
```

At build-time we symlink `myfile.hpp` to `include/s11n.net/s11n/myfile.hpp` and add `TOP_SRC_DIR/include` to the compiler's INCLUDES path. This setup helps ensure the following:

- It is impossible to build the library if two sub-modules have a circular dependency on each other because the headers are symlinked directly before compilation of a sub-module. Note that this "impossibility" is only guaranteed on a clean (re)build, and not on subsequent builds (where the symlinked headers are already in place).
- The library internally uses the headers in the exact same way that client code is expected to use them.
- That public header files can be cleanly separated from internal header files (those only used by implementation files) without having to physically separate the source files. We do this by simply not symlinking/installing internal header files and internally #including them using `"myfile.hpp"`, as opposed to `<s11n.net/s11n/myfile.hpp>`.

Several years of practice, across many different projects, has shown this approach to be generally superior to the two most common alternatives of:

- Physically separate the include files and their implementations. This is logically similar to the symlinking process, except symlinking "abstracts" this copy away, following the often-observed "another layer of redirection" principal. In practice, this physical separation is normally a nuisance for maintainers. It also allows circular dependencies to crop up.
- Keep the headers and implementation files together and use `"myfile.hpp"` includes instead of `<myfile.hpp>`.

We could of course argue the merits of the various approaches, but the approach described above is the one we use and that's not going to change, so there isn't much point in arguing anything here.

2.6 The build tools ("toc")

The libs11n tree uses a set of custom build tools called "toc" ("the other ./configure"). Toc is available as a standalone project (http://toc.sourceforge.net), but the standalone copy is not needed for building libs11n – it has its own copy of toc. Unlike most build systems, toc is designed to live entirely in the project source tree, and is dependent only on common system tools (sed, perl, make, tar, gcc, etc.).

Several people have asked why GNU Autotools are not used. Don't get me started – suffice it to say that i have a deep-seated hatred for the GNU Autotools, based on years of maintaining source trees which use(d) them. While the toc tools are not as cross-platform as GNU Autotools, they should work on the majority of systems which host GNU system tools (including almost all Linux platforms).

This document will not go into much detail about how to maintain the toc-related files, as they are not normally necessary when porting the build tree to non-Linux platforms. They can, for most purposes, be ignored by porters.

2.7 Optional add-ons

The standard build tree has optional support for two add-on libraries:

- `libzfstream` adds client-transparent support for saving/loading files using zlib and bz2lib compression techniques. It is available from http://s11n.net/download/ and requires zlib and/or bz2lib to function. If enabled, the `s11n::io` code (under `src/io/`) automatically makes use of it when reading files, and automatic compression of files can be enabled client-side by including `<s11n.net/zfstream/zfstream.hpp>` and calling `zfstream::compression_policy()`, as described in the `zfstream` API documentation.
- `libexpat` is the *de facto* standard XML parser for Unix platforms and is available from http://expat.sourceforge.net. In libs11n, enabling this support only adds one additional serializer (i/o handler) to the library, and does not otherwise affect the library.

If the configuration process finds these libraries, they are enabled automatically and flags are set in `s11n_config.hpp` to alert the sources that these features are available.

3 Generated source code

Some of the source code for libs11n is generated by the configure process or external applications. When porting to other platforms, such generated files must normally be pre-generated on a Linux host or generated on the target host via another mechanism.

The main generated files are described below.

3.1 “at-files”

Several of the generated files are built from so-called “at-files”. At-files are simply template text files similar to input files conventionally used by the GNU Autotools, with tokens being surrounded by at-signs, e.g. `@AT_TOKEN@`. By convention, such input files have a “.at” extension.

The configure/build process uses a tool (`toc/bin/atsign_parse`) to parse out the @AT_TOKENS@ and replace them with values determined by the configuration process. The `atsign_parse` script is a standalone Perl program and can be used independently of the other build tools.

3.2 <s11n.net/s11n/s11n_config.hpp>

This header is the primary configuration file for the source code, and defines, amongst other things, macros which determine:

- The library version number, license, URL, maintainer's email address, etc.
- Whether or not to use some “add-on” features, like libzfstream, libexpat, and plugins.
- Defines some shared paths for the library's components or add-ons.

This header is generated at configure-time from the file src/s11n/s11n_config.hpp.at. When editing the file for non-GNU platforms, it is best to “assume the worst” and disable any optional features (e.g. libzfstream, libexpat, plugins) and get it compiling before trying to enable any optional features.

3.3 <s11n.net/s11n/plugin/plugin_config.hpp>

This header is intended to be used only if plugins support is enabled (via `s11n_config.hpp`). It defines which DLL loader to use, the shared plugins path, and file extensions used by DLLs (e.g. “.so” on most Unix platforms and “.dll” on Windows platforms). It is generated at configure-time from `src/plugin/plugin_config.hpp.at`.

3.4 Serializer implementations

The majority of the serializers (i/o handlers, which all live under `src/io/`) use GNU flex to implement their input-parsing code. This has benefits and drawbacks. We won't go into the benefits here, but will cover some of the drawbacks because some specifically affect the portability of the source tree.

As of this writing (August 2006), the generation of flex-based output is only known to work with GNU flex 2.5.4. This version has been the "standard" version of flex for many years. Unfortunately, newer versions of flex are starting to be installed on Linux distributions but generate non-compiling code (at least in C++ mode). Versions newer than 2.5.4 cannot currently be used by libs11n. For this reason, the generated flex files are stored in CVS along with their source files. Generally speaking, generated files are not stored in CVS because they are easily creatable from their source files, but these files are an exception. The flex input files are all named `src/io/XXX/XXX_serializer.flex`, where XXX is one of the following: compact, funtxt, funxml, parens, simplexml, wesnoth. [Note that the expat serializer does not use flex, but uses libexpat's input-parsing capabilities instead.]

The standard downloadable source tree contains the generated files (`XXX_serializer.flex.cpp`) and only uses flex to regenerate them if it finds version 2.5.4 on the system.

Long-term, it would be really nice to reimplement the serializers to not rely on flex. This is, however, a significant amount of work, and is not planned any time in the near-term future (i am *so sick* of writing input parsers!). The compact_serializer can be reimplemented relatively easily without writing a complex parser, but the others have more complex grammars than the compact_serializer class, and therefore benefit from being generated by a parser generator tool like flex.

3.5 `libs11n-config` and `libs11n.pc`

The configure process generates the following extra files which are often useful by client-side build processes:

- `libs11n.pc` is intended for use by the `pkg-config` tool. This file is generated by `toc/tests/create_pkg-config_data.sh`, which is run as part of the configure process and is not usable standalone.
- `libs11n-config` is a shell script which can output useful information about the library, including precompiler and linker options required by client code. Run it without arguments to get a list of options. It is created from the input file `toc/tests/PACKAGE_NAME-config.at`.

These files are not strictly necessary when porting to non-Linux platforms but might be helpful, particularly when porting to other Unix-like platforms.

4 Platform-independent source tree ("nobuildfiles")

As part of the standard release process, we release an archive named `libs11n-VERSION-nobuildfiles.zip`. That file contains a "flattened" copy of the build tree. All generated files are copied directly into the source tree and the configuration header (`s11n_config.hpp`) is set up to disable platform-specific features. Additionally, the (normally-symlinked) headers are copied directly into their "real" destination (`include/s11n.net/...`) because symlinks are not supported on all platforms.

This release is intended to be used as a basis for including libs11n directly into client-side project trees or for creating platform-specific libs11n build trees.

The "nobuildfiles" release should build as-is on Unix platforms hosting GNU Make and GNU gcc by simply doing the following:

```
~/top_src_dir/ > cd src
make
```

The sample makefile can be used as a starting point for porting the tree to other Unix platforms. It is specifically designed to demonstrate the inter-dependencies of the various source code files without providing any "advanced" build support, like dependencies tracking.

4.1 Creating a nobuildfiles tree

The nobuildfiles source tree can be generated from the main source tree by doing the following on a fully-built copy of the master source tree, like so:

```
~/top_s11n_src_dir/ > make nobuild
```

That will generate the nobuildfiles source tree, which is independent of code generators and such and is

suitable for direct porting into arbitrary project trees or for use as a basis for porting libs11n to arbitrary non-Linux platforms.

Note that the name “nobuildfiles” is marginally wrong and should be renamed. However, lots of docs use this name so we'll keep it around for at least the foreseeable future.

5 Plugins support

Because the dynamic loading of types is a platform-specific feature, the plugins layer for libs11n is an optional component. It is enabled or disabled in `s11n_config.hpp`. Optionally, it can be enabled and have a no-op (platform-independent) implementation, as described briefly below.

5.1 `plugin.*.cpp`

The files named `plugin.*.cpp` may require special treatment. Specifically, they should not be added to the compilation process. The file `plugin.cpp` includes *one* of the `plugin.*.cpp` files, depending on options set in `s11n_config.hpp` and/or `plugin_config.hpp`. These files are listed here:

- `plugin.noop.cpp` is a no-op implementation. This allows the s11n core to use the plugin API even though it can't load plugins. This is intended to simplify some of the libs11n internal implementation and potentially simplify some client code. This is the default plugin implementation.
- `plugin.dl.cpp` is used on platforms which have either `libdl` or `libltdl`. `libdl` is the “conventional” (but not quite standard) DLL loader for Unix platforms and `libltdl` is a more portable `libdl` workalike released by the GNU Project. Their usage is very similar, and thus they are packed into a single implementation file.
- `plugin.win32.cpp` is the Windows implementation. It uses Win32's `LoadModule()` to load DLLs.

Note that if plugin support is disabled in `s11n_config.hpp`, all of the `plugin*.*` files can be removed from the build process without adversely affecting the library. It simply won't be able to dynamically load serializers and client-side serializable types, both of which are admittedly seldom-used features. Alternately, enable the plugins support but disable the libdl/libltdl/LoadModule options to force the no-op implementation to be used.

6 Unix-like platforms (Linux, FreeBSD, Solaris, etc.)

Generally speaking, the libs11n code itself is portable as-is to a wide variety of platforms, Unix-like or not. The build process, however, is not always easily portable. When running on a system which does not have a full suite of GNU tools, it is recommended that one abandon the standard build tree and use build tools more suited to their platform.

6.1 FreeBSD port

There is a FreeBSD port of libs11n available here:

`http://www.freebsd.org/cgi/ports.cgi?query=s11n&stype=all`

As that is maintained by a third party, we cannot guaranty that it will always stay up to date, however.

As far as i am aware, the FreeBSD port is essentially the “nobuildfiles” tree with a modified Makefile.

6.2 Cygwin

The standard build tree “should” work under Cygwin but it does not because it does not properly accommodate the bastard Unix/Windows nature of Cygwin and unique linking requirements on the Cygwin platform. The “nobuildfiles” release can be used as a starting point for building libs11n under Cygwin.

6.3 GNU “auto, my ass!” Autotools

It is believed that the source code can easily be ported into a GNU Autotools framework. The code generation processes, while Unix-centric, would probably require a great deal of porting work for use in the Autotools, so porters are recommended to use the nobuildfiles tree as a starting point.

6.4 Potential linking considerations

The s11n classloader is modelled off of the work described in this paper:

http://s11n.net/papers/#classloading_cpp

In essence, it works by using anonymous objects to register loadable types with the s11n factory interface when an application (or DLL) is initialized. In the case of applications this happens pre-`main()` and for DLLs it happens (or should happen!) when they are opened.

One peculiarity of this approach is that on platforms using gcc it requires that client applications which use dynamically loadable types to be linked with the `-rdynamic` flag (also known as `-export-dynamic`). The term “dynamically loadable” actually refers to *all serializable types*, whether or not they are loaded via a DLL or compiled directly in to an application. *In theory*, if libs11n is directly compiled into an application, as opposed to being a linked library, this flag “might not” be needed. If libs11n is loaded from a DLL, the linker flag is certainly needed in order for the classloader to work properly.

Another peculiarity of this classloader model is that linking libs11n as a static library will not work because it internally uses the classloader to load types, such as the Serializers (i/o handlers). When linking against a static libs11n, the factory registrations never happen and the code will compile but will not be able to save/load except to/from memory (in the case of serialization to memory, we don't need a data format, and therefore don't need to load a Serializer). Even in the case of serialization to memory, it won't work if the serializables' registrations are not linked directly in to the application or in a DLL used by the application.

In short, the linker must export all symbols in the application, and must not optimize away “unused objects”, in order to ensure that the classloader works properly.

7 Microsoft Windows® platforms

As of version 1.1.2, libs11n is known to compile on Win32 platforms. In August 2006 (version 1.2.4), libs11n's author accepted a job which requires that he have a PC running Windows. At that point, it became possible to officially support, or at least partially support, Windows platforms for libs11n.

This section is based on my very limited knowledge of the Win32 development tools – your mileage may vary.

7.1 General notes about libs11n under Win32

7.1.1 Exports and DLLs

We do not currently have libs11n running as a full-featured shared library under Windows. The reason for this is the “unusual” code requirements placed on DLLs by Windows. For example, it is necessary to explicitly flag “exportable” classes so that the Windows linker can make them available to code which wants to use the DLL. This requirement does not exist on Linux platforms, and therefore the proper exports are not in place in all of the code. The library may compile as a DLL, but will not work correctly because of missing exports.

The “proper” way to add exports to libs11n code is to include `<s11n.net/s11n/export.hpp>` and use the `S11N_EXPORT_API` macro instead of the Windows-specific `__declspec(dllexport)` marker. On non-Windows platforms the `S11N_EXPORT_API` macro expands to nothing (it is empty). For example:

```
#include <s11n.net/s11n/export.hpp>
...
class S11N_EXPORT_API my_class { ... };
```

There are some open questions about where exactly this export should go for class templates, and whether or not class templates have to be explicitly instantiated in order to properly export them. Initial research suggests that template types must be explicitly instantiated, but this requirement is not reasonable for libs11n, which doesn't know about all client-side types and therefore cannot explicitly instantiate them.

For more information on this problem, please refer to your favourite search engine or this Microsoft web page:

http://msdn2.microsoft.com/en-us/library/twa2aw10.aspx

Until these questions are satisfactorily answered (which at the moment seems impossible), it is unlikely that we will have a working libs11n DLL for Windows platforms.

i have attempted to work around the problem by building with the MinGW/Cygwin tools, but so far no luck.

7.1.2 Build flags

Here is an overview of known build flags which might have to be set in order to compile the libs11n sources under the various Visual C++ environments:

- The WIN32 macro must be defined (the exact value is unimportant). It is believed to be defined automatically by the compiler, so this should be a no-op.
- The INCLUDES path must point to the top-most include directory shipped with the source tree (normally named `include`). The implementation files all refer to the public headers using fully-qualified names (e.g., `<s11n.net/s11n/foo.hpp>`), so they can find the headers as long as your INCLUDES path is set properly. Some exceptional headers are "internal-only" and live in the same directory as the source files. Implementation files include such headers using local-style includes (e.g. `#include "foo.hpp"`).
- Turn precompiled headers OFF. They will not work here.
- Make sure that the "C++ Language Extensions" option is turned on (the default), otherwise the code will fail to compile in `<windows.h>`. Like it or not, the core Windows headers require some Microsoft-specific extensions to the C++ language.
- The option to "Keep Unreferenced Data" must be explicitly enabled or classloading will not work. If you get exceptions saying that serializers cannot be loaded (for example) then this is almost certainly the problem. This is a side-effect of the way the classloader works, which requires the creation of anonymous objects at application (or DLL) initialization time in order to register loadable types with their classloader. By default, VC++ wants to strip out those anonymous objects.

The `README.WIN32` file which comes with the source tree contains more detailed information about building libs11n on Win32 platforms.

7.2 Visual C++ Express

In August 2006 the first Win32 release of libs11n was made. It is packaged as a "solution" (project) file for Visual C++ Express, which is a freely-downloadable C++ compiler for Windows platforms, available from:

http://msdn.microsoft.com/vstudio/express/

Users may need to modify some parts (in particular, the INCLUDES path) to get it to compile on their system. It can, in any case, be used as a starting point for getting libs11n support into client-side applications.

8 Macintosh platforms

We currently have no information about building libs11n on any Macintosh platforms, including the BSD-based OS/X (also known as the "Darwin" platform). Likewise, we also have no port of the plugins layer for Macintosh platforms. In theory, the core library should be compilable, without plugins support, on arbitrary (but recent) Mac platforms.

9 The end(?)

This document can be considered to be a continual work-in-progress. If you believe that you can help us improve it, please get in touch:

s11n-devel@lists.sourceforge.net

Thanks for taking the time to Read The Fine Manual!