Reprinted from the

Proceedings of the Linux Symposium

Volume Two

July 21th–24th, 2004
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Jes Sorensen, *Wild Open Source, Inc.*
Matt Domsch, *Dell*
Gerrit Huizenga, *IBM*
Matthew Wilcox, *Hewlett-Packard*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Increasing the Appeal of Open Source Projects

Experiences from the LSB Project

Mats Wichmann
Intel Corporation / LSB Project
mats.d.wichmann@intel.com

Abstract

It is often said that open source projects will "win" or "lose" based purely on technical merit. Experiences from the LSB Project's interface standardization efforts indicate there are some concrete steps an open-source project producing interface libraries for general use can take to make the project more usable for a wider audience, leading to greater chance of widespread acceptance. Such projects have a reasonable chance of becoming standards, whether de-facto or by inclusion in formal specifications such as the LSB.

The evidence is that projects ready for large-scale use typically meet most of a set of criteria that include: demand; stable, well-documented interfaces; comprehensive interface and regression tests; an easily-deployed (portable) working implementation; and an appropriate choice of license. With the exception of demand, most of these criteria can be consciously worked towards. The paper will present some case studies of libraries that have successfully been incorporated into the LSB specification. It will also discuss some tools the LSB has developed that may help in describing public interfaces and developing tests, and discuss some ways in which portability of the code base can be improved.

1 Introduction

The Free Software and Open Source Software models present some unique concepts which seem to work best when the software is widely used and there's an active feedback loop to debug and improve the software. In order for this to be possible, it's important that some core requirements that apply to all software are attended to in this space as well: consistency and compatibility, documentation, and ease of use. If the software is to hard to deploy or make use of, the user base will remain small and the synergy which is so important to these projects will be harder to achieve.

While ease of use is a concept that is hard to measure for the developer as it means different things to different users, for an individual user it's pretty easy to tell when an application or library is not easy enough to use—it's painful to install, get running, or program to, making it hard to use it to solve the problem at hand. Where money did not change hands to obtain the software, the likely response will be to give up and look for a different solution, while what we as developers would rather have is feedback about the problems and suggestions for improvement. Often lacking a "marketing department" to drive requirements (whatever one may think of such a situation), this feedback is crucial to the open source process.

The Linux Standard Base (LSB) project

(`http://www.linuxbase.org`) aims to drive the creation of a consistent runtime environment for applications. Drawing from the experiences of the LSB project, we will examine a pair of issues, one on either side of the "runtime environment" boundary: building better libraries, and making applications (and libraries) easier to deploy.

2 Building Better Libraries

Libraries are an effective mechanism to provide for code reuse.

In Linux, libraries are normally provided as shared objects, although they may also be provided as static archives. Some libraries are foundational in that they are expected to be used by a broad variety of applications, such as the GNU C library, which is used by all programs; or the GNOME glib, which is used (directly or indirectly) by all graphical applications written to GNOME. Other libraries may export a programming interface specific to one application family such as libMagick for ImageMagick.

If a project produces libraries which are to be usable by others there are some particular issues that apply.

2.1 Stable Interfaces

A library provides certain programming interfaces which are available to programs to use (external), and probably also contains interfaces which are not intended to be used outside the library (internal). The set of external interfaces provides the Application Programming Interface (API). As programmers become familiar with the library, they will want the API to provide some stability so that they don't always have to recode their programs when the library is revised.

When a program is linked with a shared library, it will contain references to library interfaces which are resolved at runtime by the dynamic linker. The runtime instantiation of the library interface set provides the Application Binary Interface (ABI), and programmers will want the ABI to remain stable as well, or their programs may work incorrectly run against a different version of the shared library then it was originally linked against.

The dilemma for the library developer is that it's hard to get it (completely) right the first time. Bugs will be found, often the design will be found to be limiting or even incorrect, or the library may simply need to evolve to meet new needs. It would be terribly limiting to never be able to evolve the library just because users and developers demand stability. Fortunately, there are some techniques that can be used to make life a little easier.

A useful step is to identify the intended API and make sure that is all the library exports to programmers. If the API is designed as an abstraction layer distinct from the internal implementation, considerable freedom will be available to modify the library "under the covers" while still keeping the ABI stable. It's worth taking the time to design the API in this manner. It is also very useful if programmers cannot reach the internal routines which may need to change—experience has shown that if an interface can be found, someone will find a way to use it. A linker script can be used to export the desired symbols, hiding the others:

```
{
    global:
        lsbfoo;
    local:
        foo*;
};
```

A linker script is used when build-

ing a shared library by including the `-version-script=`*scriptname* directive in the gcc link line.

It's quite possible, however, that some interface in the ABI will need to change in an incompatible way. To provide for this, the symbols making up the ABI can be assigned versions, leaving the possibility of changing the version. The following example shows the use of a linker script which exports two routines and assigns them version `LSBLIB_1.0`:

```
LSBLIB_1.0 {
    global:
        lsbfoo;
        lsbbar;
    local:
        foo*;
};
```

If the symbol version is changed, old binaries won't run against the new library as the symbol version in those binaries will not be found; while binaries compiled against the new library will pick up the new symbol version. It is also possible—and may be desirable—to provide both the old and the new version of the interface in the newer library, this way old binaries can continue to run, while new binaries will be linked against the newer version of the interface by default, but could also be explicitly linked against the old version. The following example shows creating a new symbol version set which is inclusive of the previous one, only the `lsbfoo` interface will get the version tag `LSBLIB_1.1`.

```
LSBLIB_1.0 {
    lsbfoo;
    lsbbar;
};
LSBLIB_1.1 {
    lsbfoo;
} LSBLIB_1.0;
```

To make this work, the GNU linker is needed, and some special directives (`__asm__(".symver realname, alias, version");` are needed in the code, so that the old routine can be bound to the old version and the new code to the new version. The GNU linker documentation has more details on this.

If a lot of interfaces need to change incompatibly, it is better to change the major version of the library. The library version will be bound into binaries compiled against it. With major changes, multiple versions of the library can be provided, giving compatibility for old and new code.

In the LSB project, symbol versioning is used for those libraries which are already normally built that way, essentially the GNU libc set. Adding symbol versioning is a nice way to avoid breaking compatibility if a small number of interfaces have to be changed in incompatible ways. The LSB specification calls out specific library versions which must be provided by a conforming runtime, and where the symbols are versioned, the specific symbol versions. As conforming runtimes may have evolved the interfaces in the manner described, a trick is used for linking LSB conforming applications: a set of stub libraries has been constructed which contains only the LSB interfaces, with the versions required by the spec, and these are used for link-time symbol resolution.

2.2 API Documentation

A factor in how useful a library is is the quality of api documentation The documentation must describe in detail the programming interfaces available, with function calling and return conventions, boundaries, and error con-

ditions. This is the kind of information traditionally captured in the "manpage." The best measure of the quality of API documentation seems to be whether assertion-based tests (see next section) can be developed completely from the documentation, or whether the source code must be referred to fill in the details.

It is especially useful to use a tool to automate a part of this process. There are a number of tools that understand how to produce documentation from commented source code, one example would be *doxygen* (`http://www.doxygen.org`) although documentation generators seem to be more commonly used with higher-level languages (e.g. Javadoc for Java, Pydoc for Python, etc.)

The advantages of a generator approach is that the interface descriptions in the documentation don't depend on human transcription to get them right in the first place, and then don't go out of skew if the interfaces in the code ever change. It's particularly galling to try to code to an interface that does not work as documented.

The LSB specification has to date included mostly libraries which are already standardized at the API level—for example, the GNU C library is designed to be compatible with POSIX specification, so the LSB specification for the C library is able to reference this existing specification for almost all of the functional descriptions. As the LSB seeks to expand the base to other important libraries found on Linux systems, the API documentation will have to be imported by copy or by reference into the specification, so the existence of such documentation has become an LSB selection criteria.

The LSB itself has a slightly different documentation problem, as it has to capture an ABI description to describe the binary interface programs will see. A single API prototype or structure definition has been captured the way it will be seen on each of the (currently seven) architectures the LSB supports, based on things like data model (sizes of integers and pointers, for example). The symbol versions matching the interfaces must also be captured. All of this information is represented in a MySQL database which is browsable on the web (`http://www.linuxbase.org/dbadmin`) but which is also used to generate LSB header files, the stub libraries mentioned in the previous section, and the portion of the LSB specification that contains library listings, interface listings, and data definitions.

The database is also used to generate test code. Of particular note, the LSB generates two test programs, one to test the presence of the libraries and interfaces on a runtime, and another to test that an application uses only the libraries and interfaces in the specification. The data for these two programs is generated directly out of the specification database.

The LSB database schema and tools to extract data and build code (essentially a set of Perl scripts) are freely available for use by other projects, although they are probably mostly applicable to projects that support a large number of libraries and want to build similar test tools. They can be browsed from the LSB CVS tree (`cvs.gforge.freestandards.org`).

The summary is that while there's no magic to producing good documentation, it's important in producing a stable library that can be widely used. It's worth the time to see if some level of automation can help with the tasks, particularly if there are several areas that need to be kept in sync.

2.3 Interface Tests

Another area for consideration is detailed interface testing. Good tests allow checking

that interfaces perform as intended. The POSIX testing standard calls for such tests to be assertion-based, which means a written description of an intended behavior is produced, this is then used to develop the test case. The following example of an assertion is taken from the Open POSIX Test Suite (`http://sourceforge.net/projects/posixtest`:

mmap assertion 9 When MAP_FIXED is set in the flags argument, the implementation is informed that the value of pa shall be addr, exactly. If MAP_FIXED is set, mmap( ) may return MAP_FAILED and set errno to [EINVAL]. If a MAP_FIXED request is successful, the mapping established by mmap( ) replaces any previous mappings for the process' pages in the range [pa,pa+len].

Tests intended to operate at the source code level can be built and executed as part of the product build and are an effective way to catch regressions introduced during regular maintenance and development activity.

Binary level tests operate against an already built library, and are a way to test that a particular library is compatible with a particular API definition. Such tests increase the confidence of developers in the stability of the library.

In the LSB project, interface testing is the most important way of measuring a runtime against the LSB specification. However, the process of writing assertions and developing tests is not easy. It depends on a quality interface specification, good choice of testing methodologies, etc. There is little doubt that the most effective place for this work to take place is within the project itself. The source code file describing an interface can contain the interface, documentation, test assertions, and test code. All can be developed together without the kind of extra overhead incurred if each of the four items is developed separately by separate persons. The author is not aware of an existing toolkit which could automatically generate all of the necessary pieces from a single source file so endowed, but this would certainly make an interesting open source project of its own!

2.4 License Choice

The choice of license under which to release a library makes a considerable difference in who can use the libraries and how. This paper does not attempt a license recommendation as only the developer can know their own targets, needs and desires, which will guide the choice of license.

A Free Software license along the lines of the well-known GPL effectively restricts usage to programs under the same or compatible licenses. Such code cannot be used in closed source programs, even through dynamic linking, and also cannot be used by code under certain open source licenses that are not considered compatible, perhaps because they place some restriction on the user (one example might be a license that restricts usage to academic or personal use and disallows commercial use). The related LGPL license allows the use of the library by code of any sort through dynamic linking, but makes no similar provision for static linking. There are a variety of other licenses which grant greater or lesser freedoms in the ways the code may be used.

Some applications release code under dual licenses, for example a GPL-like license for those who can use it, and a separate license with commercial terms for those who cannot. It is also possible to release a package consisting of program code and library code with separate licenses for each.

As noted above, some licenses have compatibility clauses relating to how to code may be mingled with code under certain other licenses.

Various potential users of the code may have their own selection criteria that includes license choice. For example, the Debian project has a particular definition of "free" and consigns code which does not meet these criteria to the "nonfree" area.

Continuing with the use of LSB project experiences to illustrate, the LSB is concerned with functional interface descriptions, not with specific implementations. So the license of an *implementation* is not crucial—unless it's effectively the only implementation available, in which case it becomes a determining factor in practical use of the interface set.

An example may help clarify: the popular Qt toolkit was for a while the subject of some controversy in the open source community over its license terms, and a project was started to create an open source reimplementation of the Qt interface specification. When Qt licensing was changed to a dual license (one GPL-like, with a separate license for commercial developers) the open source reimplementation project was dropped as the problem people had with the previous license was resolved. However, the LSB project favors a "no strings attached" selection policy which suggests *against* the inclusion of a library where the only implementation doesn't allow a certain class of developers to just make use of the library in their code without arranging a commercial license.

The upshot is that choice of license needs to be considered very carefully.

3 Software Packaging and Deployment

The other major consideration this paper will examine is improving the accessibility of the software through producing a package that is easy to put into use. This discussion applies to both libraries and to complete applications.

The most common way to install software on Linux must be to install a distribution-specific package that has already been prepared. This has many advantages, as it's configured, compiled, and tested for that distribution, and the package will be tagged with dependencies so the user can determine what else needs to be installed to make it work. It will normally have security update patches made available should such become necessary.

Of course, not every package can be chosen for distribution packaging, and it's quite possible that an interested user for your software may find that a package is not available at all, or just not available for her distribution of choice. This should pose no problem since by definition the source code is available, and the software can simply be built from source. Unfortunately, in many cases the *simply* is a misnomer since there may be dependencies on other software, toolchain versions, etc. that may prove to be impediments.

3.1 How Not to Install Software

Although probably everyone reading this paper has had some negative experiences of their own with software installation, by way of example here is a condensed version of a situation that befell the author, and indirectly provided the motivation for recording these thoughts here:

At one point, I became interested in doing some transpositions on a piece of music, and I thought there must be a piece of software that would help with this. There are certainly commercial PC-centric applications that do this very well but there must be something open source as well. Some searching turned up a promising application named *noteedit*. Surprisingly, **rpmfind** told me that the one distribution for which a current version was pack-

aged was Mandrake, luckily my distribution of choice. The package did indicate Cooker, which is Mandrake's early-access build tree, but since it was only a couple of weeks after that last release, I assumed the Cooker could not have migrated too far and it would probably work.

After obtaining and installing the package, plus an attendant library package as well as another library (libtse) also needed, I installed and tried to run the package. Alas, it had been linked against a different C++ library version and so had references to some symbols that were not in my C++ library and thus was not runnable.

My next effort was to download the noteedit and tse library tarballs and attempt to build them from source. This was not a great success either, as the configuration scripts kept reporting fatal problems due to missing build headers and libraries, of course I had to correlate these back to the packages they would be installed by and install those. After several cycles I abandoned this approach and went to the third try, going back to rpmfind and pulling down the source, rather than binary, rpms and trying to build from source that way. This ultimately yielded a runnable binary although not without some further pain which involved tweaking the rpm specfiles. And this success still came because some Mandrake user contributed a build to the Cooker, which although it was for the wrong version (from my point of view) could be adjusted at the source level to work. What if I were running something different?

3.2 Binary Software Distribution

A project can certainly make their software easier to check out if there's a binary package available. Even if packaged by some distributions (and for many projects even this does not happen, especially early on), there's still the question of reaching users of other distributions.

The difficulty with a project building binary packages is deciding what to build for: there are an endless number of combinations of distributions and versions, and only a small fraction could be targeted. Further, this potentially puts the project into a "distro support" mode, that is worrying about oddities on the particular distro/version they have chosen to build for. A better solution seems to be to build a portable (distro-neutral) version.

Producing a portable binary package as an example has many advantages for a project:

- One package works on multiple kinds of systems
- Users interested in the software can get it running quickly
- Bugreports don't have to worry about the user's build environment
- Bugreports will be against a known set of configure and build options

There's still plenty of use for users building from source as well, including trying out combinations the developers have not tried, but the opportunity to come up quickly should broaden the base of potential users since not everybody wants to go through building from source.

Of course a really good build procedure from source—which clearly identifies dependencies, is also very valuable. Configure scripts have the unfortunate habit of quitting on the first "fatal error," which means after you satisfy that build dependency you try again and occasionally run into another, and then another. In frustration, the author once coded a configure script which issues warnings (AC_MSG_WARN) instead of errors (AC_MSG_FAIL), setting a flag which

is used to signal a fatal error at the end of the script. The author is not sure this hack is a "really good build procedure" however!

3.3 Using the LSB to Build Binary Packages

If a portable binary package is a target, the LSB provides a good model. The LSB specification describes a runtime platform, and also describes some things about how the package is delivered.

To build a portable binary, a relatively short set of rules needs to be followed:

- Link with the LSB runtime linker
- Use only LSB-specified libraries with the correct version
- Use only LSB-specified interfaces and symbol versions from those libraries
- All other interfaces must be supplied with the application

The runtime linker has a distinct name for LSB programs. For example, on the IA32 architecture, `ld-lsb.so.1` is used instead of `ld-linux.so.2`. This allows an implementation to do something different for LSB programs, such as resolving against libraries in a different directory. This capability is rarely used: most runtimes simply make the LSB linker name a symbolic link to the regular linker.

An application may only count on LSB libraries to be present on a conforming runtime, thus the restriction to link only with those libraries. If other libraries are needed, they can be statically linked, or provided in an application-supplied shared library. It is also possible to depend on *another* LSB-conforming package which supplies a shared library. Any such libraries must be constructed LSB conforming, which in practice means they need to watch their own dependencies on other libraries.

Some libraries may have more public interfaces than are described in the LSB specification. The most notable example is GNU libc. Even though these interfaces are likely to be present on every conforming system's version of those libraries, this is not required by the specification, and thus a conforming runtime may not count on them. For libraries which are symbol versioned, the binary must be linked against the symbol versions described in the specification.

While these rules are not terribly complex, it would be painful to modify build trees with many makefiles to apply them, so the LSB project supplies a compiler wrapper program `lsbcc` (as well as `lsbc++` for C++ programs) which applies the rules by fiddling with the compiler line before handing it off to the regular compiler, usually `gcc`.

If we get lucky, an LSB build can be as simple as:

```
CC=lsbcc ./configure
make
```

Of course it's not always this easy, and usually the problem is the use of libraries which are not in the LSB. The wrapper will actually turn references to non-LSB libraries into static links (the tool can be told to warn about this behavior as it's often useful to know what's happening behind your back). Sometimes static linking is a reasonable solution, sometimes packaging up the missing library in LSB mode is workable, and sometimes nothing will help but to lobby the LSB project to add the library—which will undoubtedly result in a polite request for help! The LSB still has quite a bit of evolving to do

and it's hoped that exposing it here will help identify the features which need to be added to future versions.

The other helpful aspect the LSB covers has to do with delivery of the software. Again, there are several areas:

- Portable format for the package
- Rules for where the package may place files
- Rules about names of packages to avoid clashes
- Special features such an an installer for startup scripts

The package format called out in the LSB specification is that used by the rpm package manager. This is a relatively portable format in that tools such as `alien` can convert these packages into other formats which can be handled by a system's package manager. There's no requirement that a runtime be rpm-based itself, and the only thing a package needs to (or is allowed to) depend on are provides for LSB modules (currently `lsb-core` and `lsb-graphics`) or other LSB packages.

It's also possible to deliver a package in other formats; in this case the rule is that the installer must be an LSB-conforming binary or an LSB-required command. A combination of a shell script and a tarball actually meet this requirement as both commands are required by the LSB specification. The use of other than the LSB package format is discouraged, however, as it makes it hard for system administrators to keep a view of what has been installed as would be the case if all software used the same package manager.

The File Hierarchy Standard (FHS) is imported into the LSB by reference and describes where an application may place files. To state these rules imprecisely, the package name serves as a tag, and it may install files into `/opt/`*tag*, `/etc/opt/`*tag*, and `/var/opt/`*tag*. This avoids clashes with distribution-provided packages and locally added software.

The naming of the package is also described by the LSB; essentially the rule is to register either a single package name, or a provider name, with the Linux Assigned Names and Numbers Authority or LANANA (`http://www.lanana.org`).

Finally, there are some provisions for things which don't fit into the above picture. For example, startup ("init") scripts and cron entries have to go in specific places. The LSB describes a special installer which may be invoked to create the links in the `/etc/rcX.d` directories.

With the specified behavior and tools, the LSB makes possible the creation of portable binary packages.

4 Summary

There are many considerations towards making software projects more popular. This paper has concentrated on only a small portion of those.

We have examined some issues towards making shared libraries useful. The assertion is that as a library becomes more Standard, whether that be a self-published standard or one promoted by a larger group or even a standards organization, it becomes easier for a wider audience to depend on it, software that uses it can be free of compatibility fears, and the larger community will lead to more and better feedback to continue to improve. Some steps that could help move a project towards such a state include developing solid interface specifications; stabilizing the interfaces as seen by

software through versioning, which leaves the freedom to continue to innovate while provide backward compatibility; and through comprehensive interface tests. We also looked at how choice of license plays into the usability of a library.

Another consideration towards usable software is lending the ability for potential users to get "on the air" with the software quickly, so they can evaluate it and see if it suits their needs without going through a lot of trouble. To that end, we looked at some benefits of projects delivering binary package in addition to source packages. The components of the LSB project which help in producing portable binary packages were also covered, to show how a project might be able to build a single binary package which helps the software become more accessible.

5 Disclaimer

The opinions expressed in this paper are those of the author and do not necessarily represent the position of Intel Corporation.