The Linux Standard Base: The Road to Linux Compatibility

Mats Wichmann, Intel Corporation

Stuart Anderson, Free Standards Group

Abstract

As Linux gained in popularity, worries that it could suffer from the fragmentation that hurt the UNIX industry drove prominent members of the open source community to convene the Linux Standard Base (LSB) project. In this paper we review the technology and concepts that make the LSB project work, the current status, and examine challenges and future directions for the LSB.

The LSB is an application binary interface (ABI) specification and as such describes a set of programming interfaces and other services that will be available at run time. But a specification alone is not sufficient; we will examine how the LSB uses a three pronged strategy of specification, test and implementation to arrive at a stable platform. We also look at the process by which the specification is developed, including community participation and voting rights earned through "sweat equity".

We next consider the progress the LSB has made at providing a binary compatibility contract between operating system and application, and discuss some current shortcomings and planned evolution. The developments underway for the forthcoming version 2.0 of the LSB include modularization of the specification to enable the development of usage profiles such as server, workstation, or embedded; and a full C++ ABI.

In the final sections, we describe the process for individuals and groups to bring new candidates for standardization to the LSB project. Issues to consider include how to evaluate whether an existing project is ready for standardization, which requires answering questions about license openness, stability of interfaces, dependencies on other projects, general acceptance (i.e., is this project "best practice" in its field?), degree of support across distributions, demand for the feature to become a standard and availability of tests and a sample conforming implementation to validate the specification that will be developed.

Introduction

The Need for a Linux Binary Standard

The GNU/Linux operating system (hereafter Linux™) is an aggregation of the Linux

kernel and many open source software package. The aggregations are usually called distributions. As Linux gained in popularity, a large number of different distributions became available. The interesting thing is that there is no precise definition of what makes up a Linux distribution.

In this environment, potential incompatibility between Linux versions became a concern. The extent to which compatibility has actually been a problem varies depending on the viewpoint of the observer. On the one hand, the distribution companies and projects have tended to choose largely the same software as the foundation for their distributions - the core of Linux is much the same. On the other, the versions chosen have not been the same, and have evolved over time, being taken up at different rates to meet the needs and release schedules of the various distributions. As open source packages evolve through major versions there are often intended incompatibilities; distributions have generally been good about carrying old and new versions together when that is necessary for compatibility, although the promises are only in the forward direction, and are limited to that distribution. Other compatibility concerns have come from differing packaging schemes and different file system layout policies.

In practice, binary compatibility has been relatively good between versions of the same distribution, and at the source code level it is also good, with some exceptions, many of which have to do with dependencies on the existence and correct version of other packages. At the binary level, however, it is common to see precompiled software come with a precise description of the distribution and version for which it was built, with few expectations that it will work anywhere else. Commercial software may even come with statements that the product is unsupported except on the very limited list printed on the box. Clearly, this is not a desirable situation for long term health.

Benefits of a Binary Standard

For application developers, having a standard application binary interface for Linux means:

- A larger market for applications; rather than targeting a single specific Linux distribution, building for the LSB allows the package to target the full list of certified distributions
- Improved portability; a larger number of features on certified distributions are ``plug-compatible''
- Faster development through the increased number of standard interfaces; there is less tendency to tune the application for a specific distribution
- More innovation is possible; reduced time spent porting and testing applications

These points apply to open source developers as well as commercial, although the benefits may be realized a bit differently. For example, the process of preparing a package to compile in LSB mode will help identify potential code portability problems and unexpected dependencies.

For users, the LSB enables availability of a standardized binary application platform for

Linux from multiple suppliers. Users have freedom of choice rather than being locked in to a single supplier. The LSB also offers the prospect of a supply of shrink-wrapped applications that can run across a range of systems from multiple suppliers.

The LSB Project

History of the Linux Standard Base

To attain the benefits of a binary standard and forestall the possibility of any real divergence between Linux versions, the LSB project was founded in 1997, with the support of many key figures including Linus Torvalds, leaders of the distribution vendors, and even the FreeBSD project. The mission statement of the LSB Project is:

> The goal of the LSB is to develop and promote a set of standards that will increase compatibility among Linux distributions and enable software applications to run on any compliant system. In addition, the LSB will help coordinate efforts to recruit software vendors to port and write products for Linux.

Originally, the LSB intended to create a common reference implementation for the base of a Linux system, backed by a specification. A reference platform model may help develop a specification more quickly as it adjudicates disputes over unclear aspects. But in the Linux space, there were problems with this model. In an environment of many peers, a reference implementation must either be chosen from among them, or developed independently of them; in either case the non-chosen must then commit to being bug-compatible with the reference. Choosing one would have alienated the other distributions, while having the LSB Project develop their own required resources not available, and added no real value to the market situation. In addition, a reference platform would diminish the value of distributors core software and overly constrain new development, and without this key element of the community in support, it could not succeed.

Consensus was eventually achieved around a different approach: a written behavioral specification of the services *available to LSB programs* instead of an implementation defined by package/version pairs. This new focus was realized as a three pronged approach that included a Written Specification which defines the behavior of the system, a formal Test Suite which measures an implementation against the specification, and a Sample Implementation which provides an example instantiation of the specification.

LSB Project Organization

The Linux Standard Base is now a Workgroup of the Free Standards Group (FSG). The FSG is a not-for-profit organization dedicated to accelerating the use and acceptance of open source technologies through the development, application and promotion of standards. The FSG supports the LSB project's efforts, and operates a certification program that allows conforming systems and applications to use a special logo immediately identifying them as LSB Compliant. The FSG, as a company, can receive

funds through memberships and donations, and this is beneficial to the LSB project as a means for some financial support.

To ensure progress on the various tasks the LSB undertakes, the project operates several related subprojects; the subproject leaders make up an LSB steering committee and operations are coordinated by an elected chairperson.

Community Driven

The LSB operates as an open source project. Specification materials are licensed under the GNU Free Documentation License and software components are released under an Open Source license. Project source code (including specifications) and buglists are openly maintained; these have recently migrated from SourceForge to the LSB's own cvs repository (`gforge.linuxbase.org`) and Bugzilla (`bugs.linuxbase.org`).

As is typical of an open source project, the LSB has no mandate to *impose* its technology on anyone; instead the work needs to stand on its own merits and by how open and inclusive the process is. As such, participation in the LSB is open to all interested parties, and many have contributed. Voting eligibility for those few major issues that need to go to a vote is determined by participation: those who contribute actively to the project earn a vote. LSB participation is not tied to FSG membership, although the inverse relationship exists: active participants in the LSB become considered FSG individual members who can vote in FSG elections.

How to Participate

The easiest way to participate is to join some LSB mailing lists and start to become familiar with the activities; the LSB website should give an overview of active subprojects, schedules, plans, etc. Another way to begin participating is to look at the list of open bugs and see if any look interesting to work on.

The LSB Specification

The LSB is a behavioral specification, which means it describes how things behave, not how they are implemented. For a given interface, the correct calling parameters and possible returns and error outcomes are defined. From this definition, behavioral tests are developed to measure whether an implementation follows the specification. An important detail is that the LSB is a specification of services available to LSB programs only; an implementation is allowed to provide other services, and it is allowed to provide the same services in a different way to non-LSB programs. Thus, existing implementations are able to continue to provide their "value add" as long as they can also provide the necessary services to LSB programs. Continued rapid innovation and deployment of new software is possible; this activity simply happens outside the LSB space without disturbing it.

This model also makes it possible to provide the LSB as a compatibility layer on top of an operating system, even a non-Linux operating system. The LSB Sample

Implementation, one of the three prongs in the LSB strategy described above, is delivered in this manner. The Sample can be run as a chroot session, or inside a virtual machine, providing an LSB runtime environment on a system that may not itself be a conforming system. Thus, as a matter of terminology, a conforming system is referred to as an LSB Runtime instead of as an LSB Linux Distribution or other such expressions.

It is perhaps easiest to consider the LSB concept as a contract between systems and applications: a conforming runtime promises to provide a set of services, that work in the specified manner, to conforming applications; and conforming applications promise to use only those services that are required of the runtime, and only in the way specified.

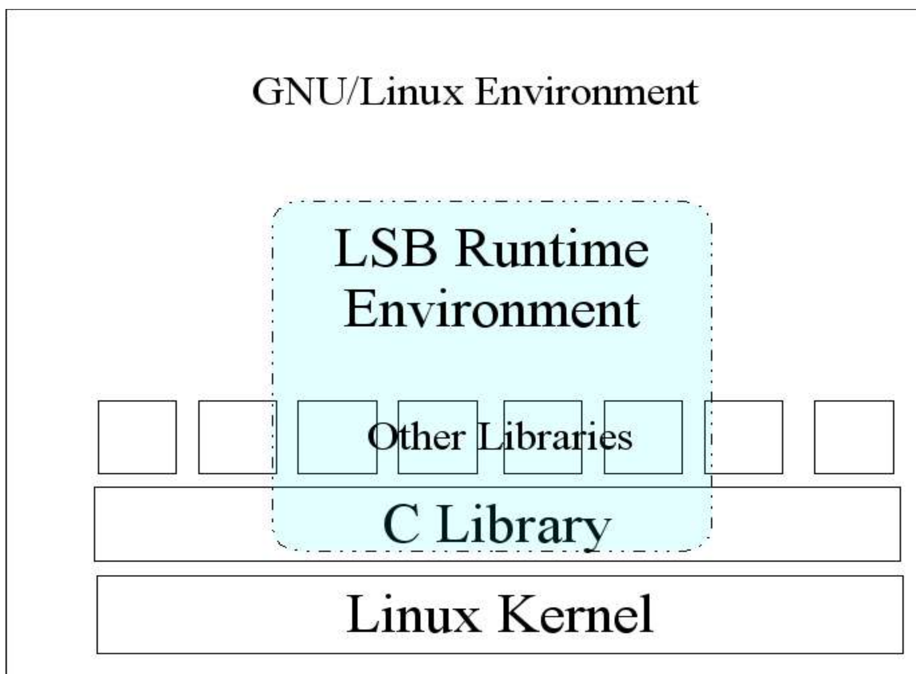

Figure 1. The LSB Subset

The LSB Specification is actually made up of a set of specifications for a binary portability environment. As it describes a binary environment, some details, such as the sizes of fundamental data types or structure layouts, and symbol versions in libraries, differ between architectures. For example the size of a C language "long" is different between 32-bit and 64-bit architectures. To clearly describe this, the specification is split into a generic piece that is common to all architectures (gLSB) and a set of architecture-specific pieces (archLSB). Most of the information is contained in the gLSB document; the set of architecture-specific variances has turned out to be quite small.

To the extent possible, the LSB builds on existing standards rather than duplicating effort and writing its own specification for each interface. Two such standards are the Single Unix Specification (SUS) which has evolved from POSIX and the SVID, and the System V Application Binary Interface (ABI). The LSB uses the ELF object format definitions from the ABI, and the interface behaviors from the SUS, and adds the formal listing of which interfaces are available in each library and the data structures and constants associated with them. As described earlier, the LSB is a codification of existing practice. Thus, in some cases, the specification references a base standard for a

feature and describes variances from that base standard. These are kept to a minimum to leave it possible to implement an LSB runtime without using the precise packages that a typical Linux system would use.

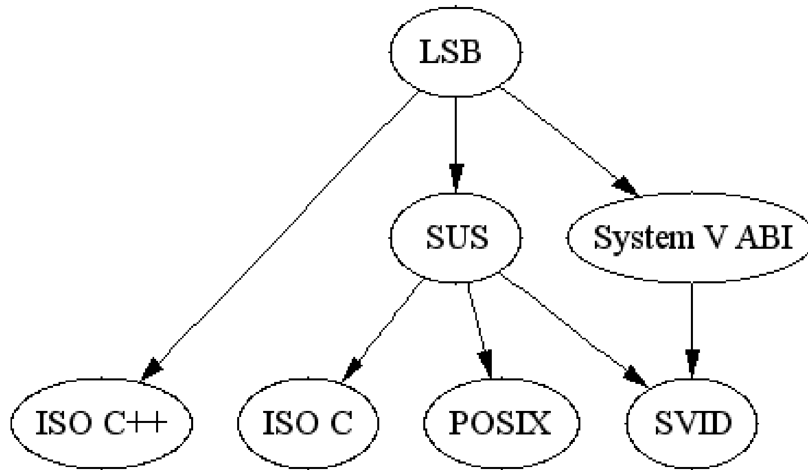

Figure 2. Specification Hierarchy

LSB Specification Components

Conceptually, the LSB is made up of a few distinct pieces and a fair bit of "glue".

- Binary file format details
- Libraries and library interface details
- Runtime program loader
- Packaging format
- Filesystem layout

The first two parts make up the ABI, and are described in more detail in later sections.

As of LSB 1.3, the following shared libraries are specified by the LSB. Applications using these libraries must use only the interfaces specified for these libraries. Any other libraries used by the application must either be provided with the application and built LSB conforming, be provided LSB conforming by another LSB package, or statically linked into the application. This list of libraries is an area for future growth.

Libc related:	libc	libm	libpthread	libutil	libcrypt	libdl
LibX11 related:	libX11	libXt	libXext	libSM	libICE	libGL
Miscellaneous:	libz	libpam	libncurses			

Table 1: Required LSB Libraries

In addition to the ABI portion of the LSB, the specification also describes a set of commands that may be used in scripts associated with the application. It is important to note that this is not intended to be a full user environment; commands are only required

by the LSB when they have a specific administrative purpose such as supporting portable installation and startup sequences.

Another key component of the LSB is a program interpreter. The program interpreter (sometimes called runtime linker) is the first thing that is executed when an application is started, and is responsible for loading the rest of the program and shared libraries into the process address space. The LSB requires a unique name for the program interpreter which is linked to LSB conforming binaries. This provides the operating system a hook early in the process execution in case something special needs to be done to provide the correct runtime environment to the application. In practice, most LSB runtime environments have provided LSB conforming behavior from their regular system libraries, and just made the LSB program interpreter a symbolic link to the regular system program interpreter, but this requirement allows something different to be done if necessary.

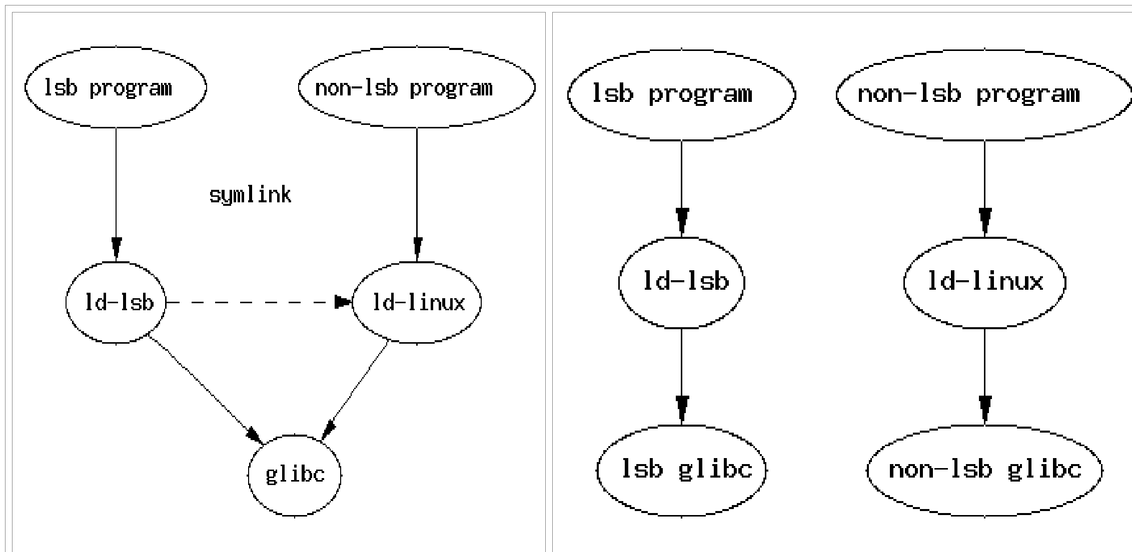

Figure 3. Purpose of LSB Linker

To facilitate building not just portable binaries, but also portable packages to install those binaries, the LSB specifies a packaging format. This is a subset of the RPM file format. The LSB does NOT specify that the distribution has to be based on RPM, only that it has some way of correctly processing a file in the RPM format.

The final major component of the LSB is the inclusion of most of the Filesystem Hierarchy Standard (FHS). The FHS describes the location of system components, and describes the part of the filesystem space that is reserved to add-on applications. The rules for using this space are designed to avoid naming conflicts amongst different applications and different vendors. A portion of this is a registry of provider and package names which is administered through the Linux Assigned Names and Numbers Authority (LANANA).

What is an ABI?

An Application Binary Interface is a set of rules and definitions that describe how

source code is to be transformed into a binary format which can be processed by an operating system, and executed on a particular processor architecture. Normally, this is all handled by the compiler and toolchain (assembler, linker, etc.), so developers don't have to worry about the details, but if object files produced by two different compilers using two different sets of rules are used together, unpredictable results can occur. The purpose of an ABI it to allow for the correct interoperability of object files, no matter how they were produced.

An ABI has two major parts. The first is the set of rules and definitions that describe the object file format. These are the things typically handled by the compiler and tool chain. For Linux, the ELF file format is used. This is the same format that is used by other operating systems like System V and FreeBSD, but there are specifics particular to Linux, and to each processor architecture supported by the LSB.

As an example of the kind of issues that must be specified, if a compiler emits a special section type which must be understood by the program loader in order to correctly run the program, then that section must be specified in the LSB, or it becomes impossible for others to implement a program loader that can correctly run that program.

The second major part of an ABI is the definition of which shared libraries will be present, and which interfaces are required in each library. This part of the ABI is always unique to the operating system.

Let's take a look at the classic example:

```
main()
{
    printf("%s %s\n","hello","world");
}
```

When compiled, the following assembly code is produced, which shows some of the interesting details of the object file.

```
        .file   "hellow.c"
.section        .rodata
.LC0:
        .string "world"
.LC1:
        .string "hello"
.LC2:
        .string "%s %s\n"
.text
        .align 4
.globl main
        .type    main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl $8,%esp
        addl $-4,%esp
        pushl $.LC0
        pushl $.LC1
        pushl $.LC2
        call printf
```

```
        addl $16,%esp
        leave
        ret
```

The ABI specifies how the arguments to `printf` are passed, and in what order they are passed. The assembly shows the stack being prepared, a call being placed to the routine, then the stack being cleaned up on return. This sequence matches that described by the ABI for this processor. Also note the mapping of function names in C is simple, the name is the same in the object as it is in the C source code.

To illustrate some of the details from the second part of an ABI, let us examine the resulting executable file from our example.

```
# ldd hellow
        libc.so.6 => /lib/libc.so.6 (0x40028000)
        /lib/ld-lsb.so.1 => /lib/ld-lsb.so.1 (0x40000000)
```

The program is dynamically linked, and the names of the libraries with which it can be linked is defined by the ABI. The LSB ABI defines the program interpreter to be ld-lsb.so.1, and the C library to be libc.so.6.

```
# nm hellow
         .
         .
08049538 D _DYNAMIC
08049610 D _GLOBAL_OFFSET_TABLE_
08049524 D __data_start
08049528 D __dso_handle
         U __libc_start_main@@GLIBC_2.0
080483d4 T main
         U printf@@GLIBC_2.0
         .
         .
         .
```

On Linux systems, symbols in the C library are versioned, so the ABI portion of the LSB must also define the required versions. Versioning allows an interface to migrate forward by changing the version number, and leaving another instance of the interface in the library with the old version number. In our example, the symbol `printf` is linked against version `GLIBC_2.0`, and will use this version at runtime, even if there are other versions of `printf` present in the library. Also note that the tool chain added a call to `__libc_start_main` for us, even though it was not in our source code. The name, and behavior of this interface must also be defined by the ABI so that a conforming runtime knows it needs to provide this interface. These sorts of "hidden" interfaces make developing a correct specification harder.

The ABI defines how aggregate types, such as structures and unions, are laid out in memory with respect to the order, offset and alignment of member fields, as well as tail padding. The ABI also defines constant values which are passed to and from functions found in shared libraries.

Accomplishments and Status

Release Summaries

LSB 1.0	June 2001	initial release of complete LSB
LSB 1.1	Jan 2002	2nd release: fixes and additions from LSB 1.0 experiences
LSB 1.2	June 2002	Certification program launched
LSB 1.3	July 2003	Itanium®, PowerPC™ architectures fully supported
LSB 1.9	July 2003	C++ technology preview.
LSB 2.0	Jan 2004	(forthcoming) C++ complete; modularization

Table 2. Release Schedule

Architecture Support

The LSB has been successfully grown to include Intel IA32, Intel Itanium, IBM PowerPC 32-bit and 64-bit, and IBM 390 (31-bit) and Z-series (64-bit). AMD64 is in review.

Runtime Compliance Success

A large number of runtimes are now LSB conforming. Those that have registered with the LSB certification program can be counted; there were almost two dozen certified for either LSB 1.2 or LSB 1.3 at the time of writing, including nearly all of what would be considered major distributions. Runtime providers have seen a benefit; their customers have asked for LSB compliance, and recently, several governmental agencies around the world have begun requiring LSB on Linux systems they purchase.

We have also found that having a test suite to measure behavior has reduced unintentional incompatibilities; the Linux kernel and certain libraries did not always do things the same way on all architectures, and the LSB test suite has helped detect several of those situations, now fixed.

Application Compliance

The story has been less rosy for applications. At this writing, there are no certified LSB applications. The Application Battery, prepared by the LSB team itself, has eleven applications it considers Certification Ready - they meet all technical requirements but as the LSB Project is not the owner of those applications, it has not certified them. There are known to be several sucessfully built LSB conforming applications. To some extent, it was expected that applications had to trail conforming runtimes, but it has been more than a year since the certification program produced the first certified runtimes. We continue to study why this program has not been more popular; one of the conclusions is that it is still too hard to build a non-trivial LSB application (work is in progress to improve the build tools and documentation to help address this); another is that there are important features missing from the LSB that impede porting.

Reflections on LSB Component Selection

The features included in the LSB specification were strongly based on (then) current practice. In a later section a set of selection criteria for new features is described; a similar, though less formal process, was applied to selecting the initial feature set. Features were chosen by consensus; if a feature was not agreed to be current best practice, stable, and ready to standardize, it was left out. In some cases compromises had to be made; one form of compromise was the development of a higher-level wrapper that could support differing behaviors. An example of this was system startup scripts: some packages may need to install a script that starts the feature when the system is booted. Since consensus could not be reached on the location for a startup script, a new facility was defined which is passed the script and is responsible for installing it in the correct location for that runtime, the LSB then needed to say nothing about the location, only the name and behavior of the script-installer.

The conservative selection policy has proved both a blessing and a curse. The upside is that there has been surprisingly little trouble bringing runtimes into compliance because what was specified matched very closely what was already deployed, and those libraries have, in the main, been quite stable. The downside is that many commonly used Linux features were omitted because the participants could not all agree those features were ready to standardize, whether because of a lack of maturity, not widely accepted as best practice, lack of availability of a standards-level description or tests, or other reasons. Probably all of the omissions were technically the right decision, but been a barrier to LSB adoption by application developers when they see a favorite feature is missing. A large area for future development will be to rigorously examine all requests and grow the specification as needed to be more widely useful, yet still remain stable.

Figure 4 shows the results of an analysis tool the LSB project uses to help identify unmet dependencies in programs.

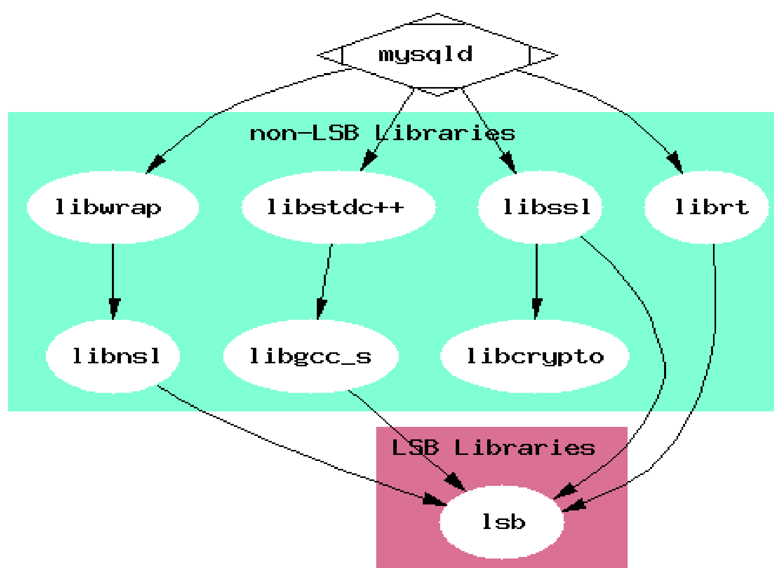

Figure 4. mysql Dependency Analysis

Current Work

LSB version 2.0 is currently in the final stages of development.

C++ ABI

One of the major new features for LSB 2.0 is the addition of an ABI for C++. The language mapping for C++ is much more complicated than the simple mapping which is used for C. Fortunately, an industry group lead by Mark Mitchell at Code Sourcery did a lot of this work for Intel® Itanium® architecture. This work has been extended to other architectures, and included in the current GNU Compiler Collection project.

With C++, what looks like a simple class instantiation can actually result in several additional pieces of data being generated by the toolchain. An example is the easiest way to describe this:

```
#include <iostream>

using std::cout;

class foo {
protected:
        int _bar;
public:
        virtual void baz()=0;
};

class bar : foo {
public:
        inline bar() {
                _bar=0;
        };
        inline void baz() {
                cout << "bar " << _bar << " times\n";
                _bar++;
        }
};

class bar baz;

main()
{
        baz.baz();
        baz.baz();
}
```

The instantiation of the class `bar` has resulted in the following symbols being generated by the compiler.

```
# nm hw | c++filt
         .
         .
         .
08048754 T main
0804880e W bar::bar()
```

```
08048834 W bar::baz()
0804887a W foo::foo()
080489c0 V vtable for bar
080489e0 V typeinfo for bar
08048a00 V typeinfo name for bar
08048a08 V vtable for foo
08048a14 V typeinfo for foo
08048a1c V typeinfo name for foo
08049d2c B baz
         .
         .
```

For the class `bar`, three additional data structures are added, a Virtual Table, a Run-Time Type Info table, and a string containing the name of the class. Because class `bar` is derived from class `foo`, three data structures are added for it also. Under the right conditions, other data structures may also be produced.

In the C++ language mapping, symbols names are *mangled* from the name used in the source code. This mangling creates a name that uniquely describes the entire function signature, including return type and parameter types. This results in ordinary looking source names like `bar::baz()` becoming `_ZN3bar3bazEv` in the object file.

A subtle point caused by the C++ Standards requirment for identifying function signatures is that types are reduced to underlying types before mangling. On architectures where the type `size_t` is implemented as an `int`, the function signature `func1(char *, size_t)` is reduced to `func1(char *, int)` before being mangled. While the C++ standard has good reasons for this requirement, it has two undesirable side-effects.

First, name mangling is not fully reversable. This means that once a function name is mangled, the unmangling of that name can go as far as the reduced signature. At first glance, unmangled names may not look quite like the original, if any reduction took place.

Second, name mangling results in different symbol in the object files on different architectures. On a 64 bit architecture, our previous example is reduced to `func1 (char *, long)` instead because `size_t` is implemented as a `long` instead of an `int`.

Because of these complications, the second part of the ABI, the definition of `libstdc++.so`, has turned out to be a very interesting task. For the C library, there is a lot of consistancy accross the architectures. With few exceptions, the symbol names are the same for every architecture. For `libstdc++.so` a substantial number of the symbols are mangled differently on different architectures. This causes the database schema and code for the testing tools to be more complicated than was neccessary for the C libraries.

Modularization

The LSB 1.x specification series has been several years in the making, and was suffering

some growing pains. The key issue was a lack of flexibility: the LSB was a single "big gulp" specification, a runtime had to support all of it. On one end of the spectrum, there was simply too much material to support for certain profiles, such as embedded systems. On the other end, prospective new features were hampered by the need to show usefulness to all possible users of the LSB.

The approach taken was to restructure the specification into components known as modules, giving the flexibility to combine different sets of modules to fit particular uses, known as profiles.

From the external view, this modularization is largely virtual: LSB 2.0 looks very much like LSB 1.3 with a C++ ABI added, and the effect of the modularization should be nil. The work is simply an enabler to allow future growth in the directions mentioned.

Widening Participation to Topic Expert Groups

With LSB 2.0, two new concepts are introduced with the purpose of making it easier for groups other than the LSB to propose features for standardization. The intent is to enable a process where interested expert groups can build a specification that "plugs into" or is "built on top of" the LSB.

LSB Module

One new approach is for an interested group to develop their own module to "plug into" the LSB and then propose it as an LSB module to the Free Standards Group. If accepted as an LSB module, the FSG will take on the module as part of the LSB certification program. The submitting group may choose to apply to be an FSG workgroup (a peer of the LSB), or apply to become an LSB subproject - normally the scope of the module will guide this choice.

Examples of LSB Modules include "Base Graphics", "GUI Elements" and "Desktop" (the latter two still under consideration).

Built on LSB

The other new approach is for an interested group to develop a specification that is independent of the LSB, but requires an LSB runtime as a basis. In this case the project may apply to the FSG for "Built on LSB" status. Application is required in order to use the LSB trademark in this manner and interested groups may also apply to have the FSG build a certification program. However, should the interested group wish to simply build on top the the LSB without working with the FSG, all they need to do is download and follow the forthcoming document "Building Linux Based Solutions and Standards on top of the LSB" (check the FSG website).

How to Add Features to the LSB

Prior to LSB 2.0, any new feature added to the LSB had to be considered on the basis of being appropriate to deploy everywhere, because there was only one way of considering

the LSB: as a whole.

There are now three avenues for promoting a feature into the LSB "specification family". One is to have the feature added to an existing module. The second is to build the feature as a new LSB module. The third is to build the feature as module which requires an LSB runtime, but which is not managed by the Free Standards Group.

While the details may vary, all three should go through a relatively similar process.

Feature/module criteria

Over time, the LSB Futures subproject, which is charged with evaluating new-feature requests, has developed a checklist against which to measure new features. Following the checklist helps to ensure that features are not prematurely or incorrectly promoted as a standard. Key items on the checklist include:

- Sufficient demand or usage of the feature by developers
- Feature as implemented, is seen as a de-facto standard and is widely available
- There is a stable ABI
- There is a specification available, or resources willing to write it
- There are test suites available, or resources willing to implement one

Other considerations would be technology unencumbered by intellectual property constraints and dependencies being met (dependencies are either in the LSB or progressing in parallel to be added to the same version of the LSB).

Completing a New Feature

The way to add a new feature, then, is to insure an affirmative answer to all of the checklist questions. This applies whichever of the three avenues described above will be followed. The checklist will normally serve as a roadmap for the process by highlighting what remains to be resolved.

Each of these points has a key reason for being considered. If there is not sufficient demand across the spectrum of systems, it will be hard to get buyin to deploy it in all LSB runtimes. Note that this standard has to be applied in context: with the new module approach a feature that has insufficient demand for the baseline may still be appropriate for a module which appears only in a particular usage profile.

If the feature has several competing implementations that cannot be aligned with the standard, it is probably too early to standardize. The same is probably true of an interesting feature that is not in widespread use.

The binary interface itself for the feature should not be changing as the LSB will describe a particular instance of that ABI and if there are incompatible changes later, runtimes will have to continue to carry the old version and LSB programs will be pinned to the older version.

A precise description of the behavior of the feature is needed for inclusion into the specification; Documented by Source Code is not sufficient. The precise behavioral description also guides the development and/or evaluation of the tests for the feature. The tests, in turn, must be available so that compliance with the documented behavior can be proven.

An example of why the new-feature checklist needs to be rigorously applied: the LSB approved a 1.3 version of a new archLSB, having not realized that the two then-existing runtimes disagreed on glibc symbol versions, one having been built off an official upstream version, the other off "unofficial" patches applied to an earlier version. This problem could not be compatibly resolved, and that archLSB had to be withdrawn from 1.3. The “de-facto standard” checklist item could have helped prevent this situation.

Software deliverables of the LSB

In addition to the specification, the LSB workgroup also provides several software packages:

- Test Suites (Runtime Test subproject)

  One of the key components of the LSB from the beginning has been tests. Several different test packages are used to measure different aspects of the LSB:

  Runtime Test Suites
  : This is the extensive suite that is used to measure a runtime against the specification

  Application Test Suites - lsbappchk
  : Used to test an aplication for conformance to the LSB.

  Library Checker - lsblibchk
  : Used to verify that a distribution's libraries contains the correct set of interfaces, in the correct library.

  C++ ABI Checker - cxxabichk
  : A derivative of lsblibchk, this is a standalone tool which measures `libstdc++.so` against the LSB specification.

- Application Battery (Application Battery subproject)

  A set of common applications built as LSB conforming programs which are used to test the development environment, to test a LSB Run-time environment, and to serve as LSB porting examples.

- Development Environment (Development Environment subproject)

  A set of tools and files to facilitate building an application to conform to the LSB.

  lsbdev-base
  : Set of stub libraries and clean headers which should be used to build your application.

lsbdev-c++
: A package to provide C++ support for LSB 1.3 (which doesn't include C++ in the specifcation).

lsbdev-cc
: A compiler wrapper that can be used when building an application. This wrapper handles most of the adjustments which need to be made to correctly build an LSB conforming application.

lsbdev-chroot
: An alternate method for developing LSB applications that uses a chroot environment instead of a compiler wrapper.

- Sample Implementation (Sample Implementation subproject)

  A minimal runtime environment that serves both as an example of a LSB runtime environment, and as a tool for checking applications in a minmal LSB conforming environment. The SI can be used either in a chroot mode, or used with User Mode Linux to fully simulate a LSB runtime system. It should also run inside a virtual machine such as VMware.

These can all be found on the LSB website at `http://www.linuxbase.org/download/`

Further Reading

- Linux Standard Base website: `http://www.linuxbase.org`

  Contains links to a variety of information about the LSB, papers and presentations, mailing list information, and links to the subprojects.

- The Linux Assigned Names and Numbers Authority (LANANA): `http://www.lanana.org`
- Free Standards Group website: `http://www.freestandards.org`
- LSB Certification website: `https://www.opengroup.org/lsb/cert`
- Filesystem Hierarchy Standard `http://www.pathname.com/fhs`