

The Linux Standard Base

7. Juni 2004

Note légale

Dieser Beitrag ist lizenziert unter der GNU Free Documentation License.

Zusammenfassung

The Linux Standard Base (LSB) is an open source project to increase binary compatibility among Linux distributions and enable software applications to run portably. It thus helps developers focus on adding value to their software, rather than spending time dealing with verification and porting issues for multiple Linux distributions. This has value not just for closed-source programs, but also as a mechanism for wider availability of open source projects. The LSB utilizes a three-pronged strategy of behavioral descriptions, tests, and example (proof-of-concept) implementations which are each used to help validate the others. By avoiding specifying things at the level of "version X.Y of package ZZZ" the LSB does not hamstring continuing rapid evolution of open source packages and Linux distributions: conforming systems are required to continue to provide LSB-described behavior, but can add additional interfaces and libraries, and even different versions of existing interfaces. This paper examines the origins, evolution and future directions of the LSB project, the case for standards in the open source space and how they provide value to software developers, and some case studies of how the LSB can be used to produce portable software.

"Through the definition and testing of operating system interfaces, the LSB creates a stable platform that benefits both developers and users.- Linus Torvalds

1 The Linux Standard Base

1.1 Abstract

The Linux Standard Base (LSB) is an open source project to increase binary compatibility among Linux distributions and enable software applications to run portably. In this paper we review the case for standards for open source software, the technology and concepts that form the foundation of the LSB, the current status, and examine challenges and future directions for the LSB.

The LSB is an application binary interface (ABI) specification. As such, it describes a set of programming interfaces and services that will be available to applications 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 example implementation to arrive at a stable platform. We also look at the process by which the LSB is developed, including community participation and voting rights earned through participation.

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. Of particular importance is the imminent LSB 2.0 release, which adds a C++ ABI and modularizes the specification. We also include an example showing the LSB porting process.

1.2 Introduction

1.2.1 The Need for a Linux Binary Standard

The GNU/Linux operating system (hereafter *Linux*) is an aggregation of the Linux kernel, packages from the GNU project, and many other open source software packages. The aggregations are usually called *distributions*. According to the Linux Weekly News distribution list (<http://lwn.net/Distributions>) there are 375 different

Linux distributions. Each serves particular purposes and targets, and the diversity in and of itself does not indicate a weakness. But it is also clear that there is ample potential for incompatibility between distributions.

Free Software and other open source software carry a certain promise of compatibility: with the source code available users have the freedom to change the code or compile an older version if a new version has introduced incompatibilities. But there is a limit to how far many users are willing to go in such pursuits; for them the value of free software is that it gives them control of their own computing environments, but they almost certainly don't want to spend their time trying to build/keep compatibility between different distributions. And not all software can end up being open source.

The extent to which compatibility has actually been a problem varies depending on the viewpoint of the observer. On the one hand, the core of Linux is much the same - the distribution companies and projects have tended to choose largely the same software as the foundation for their distributions, and this brings a level of compatibility. But the open source projects that are aggregated into Linux evolve over time, and often introduce intended or unintended incompatibilities as part of that evolution, the aggregators make different decisions on when to take up these new versions, and additionally make changes to suit their purposes. Distributions have generally been pretty 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 of course limited to that distribution.

Other compatibility concerns have come from differing packaging schemes and different file system layout policies.

Despite the above claim of reasonable compatibility, it is almost certain that every veteran of using Linux has a few war stories where a program fails after an OS upgrade, or a program pulled off the net won't work, etc. 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 on any other combination. 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 if Linux is to become more widely used among those who see their computer more as a tool than as a development platform.

A binary standard helps mitigate these issues.

1.2.2 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 to a standard allows the package to target the full list of distributions that conform to that standard
- Reduced expenses: portable code leads to more code reuse and less time spent building and testing on different combinations
- Increased confidence: as the standard interface set remains stable and vendors get used to keeping it that way, there's less chance of versions and patches causing application breakage.
- Faster development through the increased number of standard interfaces: there is less tendency to spend time tune the application for a specific distribution, when usually the small gains bring little benefit

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 build to a binary standard will help identify potential code portability problems and unexpected dependencies, and the code becomes more widely usable with less support burden on the project team, which can then spend more time innovating.

For users, a binary standard 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 presence of such a standard also offers the prospect of a supply of ready-to-install applications that can run across a range of systems from multiple suppliers.

1.3 The LSB Project

1.3.1 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 1998, with the support of many key figures including Linus Torvalds, Linux International, leaders of the distribution vendors, and even the FreeBSD project. The mission statement of the LSB Project 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 helps develop a specification more quickly as it adjudicates disputes over unclear aspects: if in doubt, the reference platform wins. But in the Linux space, this model was not practical. In an environment of many peer implementations, 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 which were not available, and would have added no real value to the market situation. In reality, a reference platform would have diminished the value of distributors' core software and overly constrain new development, and without the key element of a supportive community, 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. In this model the sample implementation has no legal standing, the final decision on interpretations is always done against the written specification.

1.3.2 LSB Project Organization

The LSB project was founded as an open source project, with open participation, merit-based selection, etc. While retaining this structure, the LSB 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 as several related subprojects; the subproject leaders make up an LSB steering committee and operations are coordinated by an elected chairperson.

Community Driven As stated, the LSB operates as an open source project. Specification materials are licensed under the GNU Free Documentation License and software components are released under various Open Source compatible licenses. 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 bug tracker (bugs.linuxbase.org).

As is typical of an open source project, the LSB has no mandate to *impose* requirements 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 are granted no-cost FSG individual membership, which conveys a 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. Eventually, we expect to operate a project along the lines of Linux Kernel Janitors, where a pool of small tasks is collected and taken care of.

1.3.3 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 these definitions, behavioral tests are developed to measure whether an implementation follows the specification. An important detail is that the LSB only talks about services which must be available to LSB programs; 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. This is handled through versioning: if the LSB version exists and works as specified, there is no problem with providing an interface of the same name with different behavior, as long as the symbol version is different. Thus, existing implementations are able to continue to provide their value add.

Continued innovation and deployment of new software is possible; this activity simply happens outside the LSB space without disturbing it. When new features have become compelling and widely accepted, they can be brought in to the LSB as part of the specification.

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 layered manner. The Sample can be run as a chroot session, or inside a virtual machine (UML, VMware, etc), 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.

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.

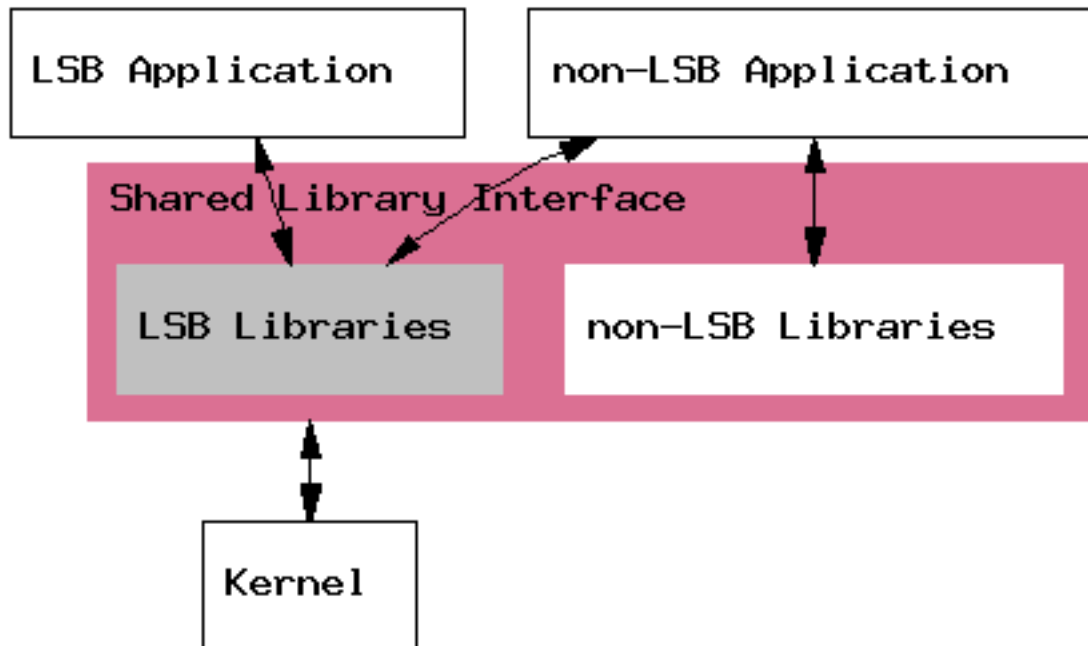


Figure 1. The LSB Concept

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 pointer is different between 32-bit and 64-bit architectures. To avoid numerous lists of *if arch A then X else if arch B then Y*, the specification is split into a generic piece that is common to all architectures (*generic LSB*) and a set of architecture-specific pieces. Most of the information appears in the generic LSB 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 that has evolved from the System V ABI and the interface behaviors from the SUS, and adds the formal listing of interfaces available in each library and the data structures and constants associated with them.

As a codification of existing practice, the LSB specification references a base standard for a feature and describes variances, if any from that base standard. These are kept to a minimum to leave it possible to implement an LSB runtime without constraining the implementation to any specific package. For example, the *cp* command may most often come from the *coreutils* package, but by avoiding requiring all of the extra command options found in that version, some other version of *cp*, such as perhaps the one from *Busybox*, might also meet the requirement.

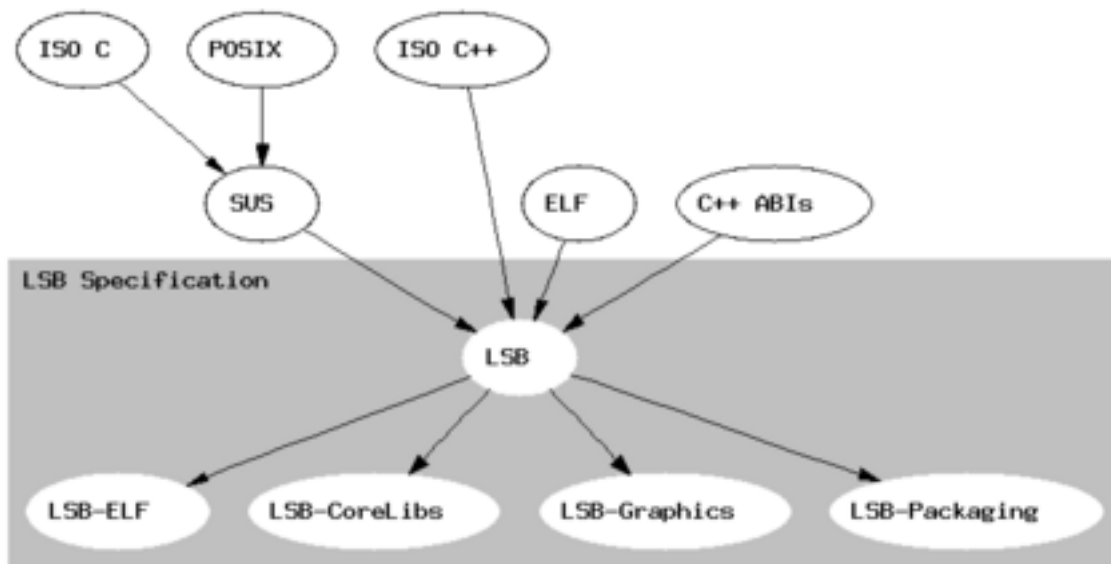


Figure 2. Specification Hierarchy

1.3.4 LSB Specification Components

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
- Minimal command set
- Runtime program loader
- Packaging format
- Filesystem layout

Prior to LSB 2.0 all of these were presented as a single document with matching supplements for each architecture. With LSB 2.0, the specification is modularized into component pieces as shown in Figure 2 above: ELF,

LSB, Packaging (including Filesystem layout) and Graphics (split off from the core LSB). Each component has a set of architecture-specific supplements.

ELF and the libraries make up the ABI, which will be described in more detail shortly.

As of LSB 2.0, the following shared libraries are required 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. The list of libraries is an area for future growth.

Tabelle 1: Table 1: Required LSB Libraries

Core Libraries:	libc, libm, libpthread, libstdc++, libcrypt, libdl, libpam, libncurses
Graphics Libraries:	libX11, libXt, libXext, libSM, libICE, libGL

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 identifying required shared libraries, performing runtime relocation and other setup tasks. The LSB requires that each architecture define a unique program loader name that can be distinguished from the regular program loader for that architecture. 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.

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*).

1.3.5 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 application developers don't have to worry about the details. The purpose of an ABI is to allow for the correct interoperability of object files, no matter how they were produced, including by different compilers.

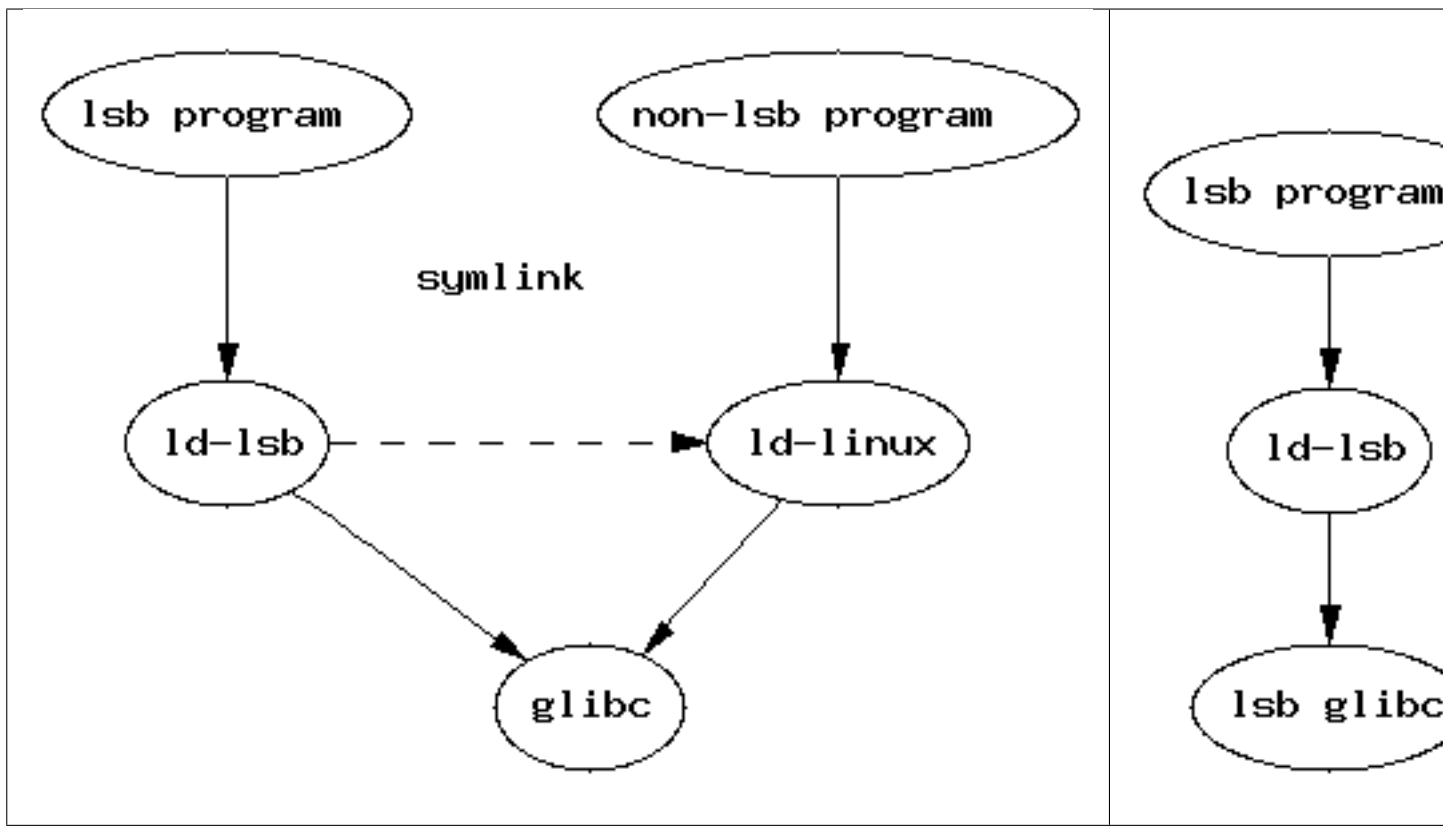
An ABI has two major parts. The first is the set of rules and definitions that describe the object file format. These are the things typically handled by the compiler and toolchain. For Linux, the ELF file format is used. This is the same format that is used by other operating systems such as Solaris 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 implementors to write 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:

Tabelle 2: Figure 3. Purpose of LSB Linker



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

When compiled for ia32, 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 any function are passed (on the stack), and the size and ordering of the arguments to *printf*. 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 library part of the 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 libraries with which it can be linked (including library version number if required) is defined by the ABI. The LSB ABI defines the program interpreter for an ia32 system 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 there's a reference to *_libc_start_main* although we didn't include that in our source program. That was added by the toolchain; such hidden symbols must also be accounted for by the ABI, since a conforming library needs to supply those routines in order to execute the conforming binary.

The ABI defines how aggregate types (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.

1.3.6 C++ ABI

With LSB 2.0, there is also a C++ ABI. The language mapping for C++ is much more complicated than the simple mapping which is used for C. Fortunately, an industry group led by Mark Mitchell at Code Sourcery did a lot of this work for the Intel Itanium architecture. This work has been extended to other architectures, and is 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 requirement 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 reversible. 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*, proved to be quite a challenging task. For the C library, there is a lot of consistency across 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 necessary for the C libraries.

1.4 Building Conforming Applications

The previous description has been primarily for runtime and tool providers. The rules for an LSB conforming application are relatively simple:

- Use only LSB-specified shared libraries
- Use only LSB-specified interfaces in those libraries
- Link with the LSB linker
- Package according to the LSB packaging guidelines

The reason for the prohibition in the first rule is that a conforming implementation is only *required* to ship the libraries in the LSB, any other libraries may or may not be present. Thus, an application can't count on them.

It is permissible under the LSB rules to link statically with needed libraries, or to build them dynamically and include them with the application, both of which solve the availability problem.

With some care, it is not unreasonable to take care of the first and third rule in project makefiles, but the second can be tricky. The LSB project does produce an application checker that reports on non-LSB interfaces used by a compiled program. But overall, getting all three right would require a fair bit of work, especially for complex builds. So the LSB project has developed a trick which helps build conforming programs more easily.

All the information about libraries and interfaces is captured in a database, which is used to generate parts of the specification and certain tests. This helps keep details in sync. The same database also generates a set of stub libraries, one for each library required by the specification, containing entry points for all the interfaces and definitions of all the global data elements in the specification - and only those. When a candidate program is linked against these stub libraries, it has the effect of failing the link if there are references to other routines in those libraries. The stub libraries are *only* ever used for link-time symbol resolution, at run-time the system's normal libraries are used.

The trick is put into effect by using a compiler wrapper program named *lsbcc* (there's a corresponding *lsbc++*). When invoked as the compiler for a candidate LSB application, it checks the command line and intercepts link references to non-LSB libraries and turns them into static links. It also makes sure the stub libraries will be searched first by the linker, and a set of spec-clean header files will be searched first by the compiler. It then passes the modified command line on to the regular compiler which does the work. The result of this fiddling is that a program will either compile and link, in which case it is almost certain to be LSB conforming, or it will fail to link in a way that indicates the non-LSB usage fairly clearly.

There is much more information on building LSB programs described in some of the references; a deeper discussion here is beyond the scope of this paper.

1.5 LSB Accomplishments and Status

1.5.1 Release Summaries

Table 3: Table 2. Release Schedule

LSB 1.0	June 2001	initial release of complete LSB
LSB 1.1	January 2002	2nd release containing fixes and additions from experience gained with 1.0
LSB 1.2	June 2002	Certification program launched
LSB 1.3	January 2003	Itanium, PowerPC 32-bit architectures
LSB 1.9	July 2003	C++ technology preview; s390 and s390x architectures
LSB 2.0	July 2004 (est)	C++ complete; modularization; PowerPC (64-bit) and AMD64 architectures

1.5.2 Architecture Support

The LSB has been successfully grown from a single architecture, Intel IA32, to include Intel Itanium, IBM PowerPC 32-bit and 64-bit, IBM 390 (31-bit) and 390x (64-bit), and AMD64.

1.5.3 Runtime Compliance Success

There are now many LSB conforming runtimes. Most of those have become LSB certified: there were almost two dozen for either LSB 1.2 or LSB 1.3 at the time of writing, including nearly all of what would be considered the 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.

The process of putting in place a test suite to measure behavior has helped both smooth out unintentional incompatibilities between distributions and differences between architectures: the Linux kernel and certain libraries did not always do things quite the same way on all architectures. Some participants feel that this has been LSB's largest contribution to date.

1.5.4 Application Compliance

While conforming applications had to wait for there to be a base of conforming systems in place first, the lack of applications has been the biggest disappointment. The LSB project has, as part of its proof-of-concept implementations, built a number of open source programs LSB conforming. Some of the projects producing those programs, such as Samba and Apache, now look likely to be providing an LSB package as one of their standard builds. At this writing, a small number of commercial applications were completing the certification process for LSB 1.3.

The LSB project continues to study the relative dearth of conforming applications to see what can and should be done. It is telling that commercial developers who understand the LSB uniformly praise it as valuable, but may nonetheless not use it. One preliminary conclusion is that those commercial applications who have been committed to a Linux product for a while have already addressed possible incompatibilities in their own way: usually by making partnerships with a few distributions that are key to their business. They may be reluctant to impact those agreements and so LSB becomes, rather than a way to consolidate ports, simply an additional Linux port to be developed and maintained. These same vendors probably also have product commitments to Linux versions that are too old to be LSB conforming and so again, an LSB port looks less attractive. The solution to this appears to be a combination of education and time. As it becomes more likely that every target system can be LSB conforming the value of an LSB port will rise. Applications not already on Linux may not see these issues as problems for them.

Another possible issue is that the LSB specification is too restrictive - key features are not available, making it technically impossible to complete an LSB port. The largest such feature has clearly been the lack of a C++ ABI, which is addressed with the forthcoming LSB 2.0 specification.

1.6 Future Directions

The LSB 1.x specification series was several years in the making, and was suffering growing pains. The key issue was a lack of flexibility: the LSB was a monolithic specification, a runtime had to support all of it to claim LSB conformance. On one end of the spectrum, there was simply too much material for certain uses, such as embedded systems. On the other end, prospective new features, especially for large systems, were hampered by the need to show usefulness to all possible consumers of the LSB; even if vital to one audience they might not be to others.

The new modular specification lowers the barrier to inclusion a bit. A new feature will be targeted to a particular module, or perhaps be a new module itself; it needs to be shown to be necessary to the consumers of that module, but not necessarily to all consumers of the LSB.

1.6.1 How to Add Features to the LSB

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.

1.6.2 Feature/module criteria

The LSB Futures subproject, which has been 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 (that is, dependencies are either in the LSB or progressing in parallel to be added to the same version of the LSB).

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

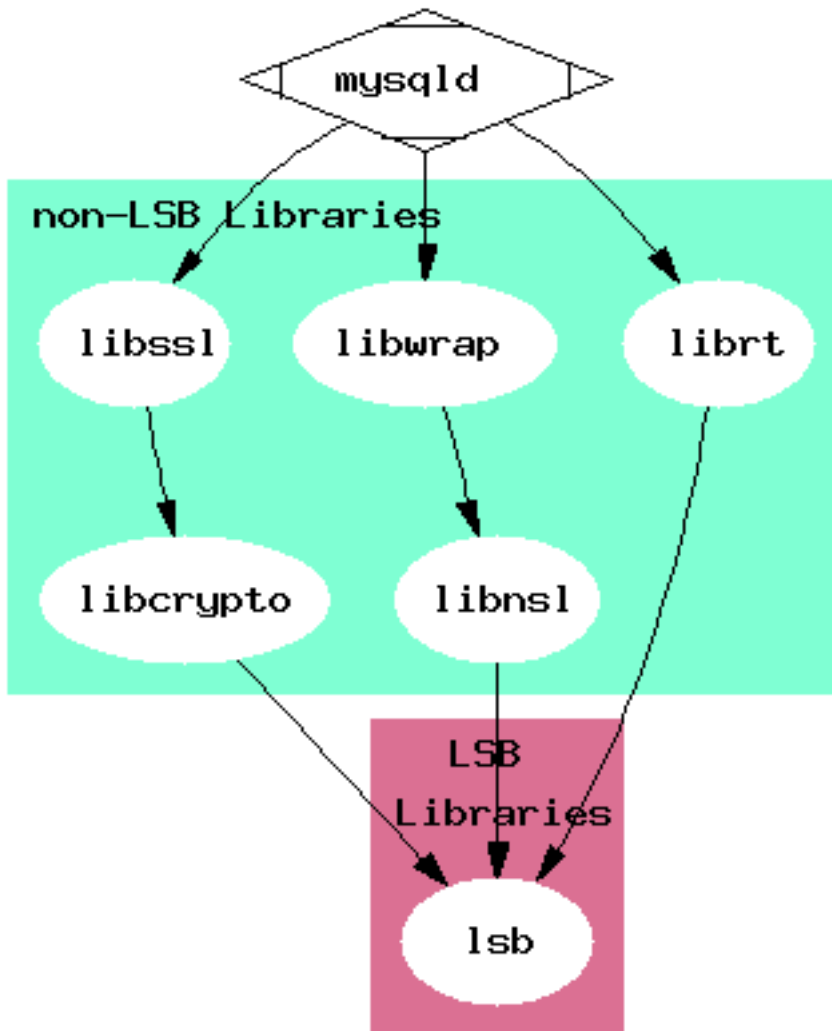


Figure 4. mysql Dependency

1.6.3 Completing a New Feature

The way to add a new feature 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 architecture supplement based on data gathered from the single existing implementation. Later it evolved that new runtimes were going to disagree with the supplement on glibc symbol versions, with the original having been built off unofficial patches applied to an earlier version of glibc than the one in which official support for that architecture was provided. This problem could not be compatibly resolved, and the supplement had to be withdrawn. The de-facto standard checklist item would have helped prevent this situation.

1.6.4 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 promote features for standardization. This enables 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 current LSB Modules include Graphics and Packaging. It is hoped, for example, that a group will follow the guidelines for a new module and propose a Desktop module for the LSB.

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).

1.7 Further Reading

- *Building Applications with the Linux Standard Base*, IBM Press/Prentice Hall, publication pending
- LSB website: <<http://www.linuxbase.org>>. Contains links to a variety of information about the LSB, papers and presentations, mailing list signup and archives, and links to the various 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>>

1.8 Legal Corner

The opinions expressed in this paper are those of the author in his role as a participant in the LSB project, and do not represent any official positions of Intel Corporation.

Linux is a trademark of Linus Torvalds.

UNIX a registered trademark of the Open Group in the United States and other countries.

LSB is a trademark of the Free Standards Group in the United States and other countries.

Intel and Itanium are registered trademarks of Intel Corporation.

PowerPC is a trademark of IBM Corporation.

AMD and AMD64 are trademarks of Advanced Micro Devices, Inc.