# Developing LSB-certified applications

## Five steps to binary-compatible Linux applications

George Kraft, IV

October 01, 2002

The Linux Standard Base is a big step toward ensuring binary compatibility among Linux® applications, and it should greatly reduce the amount of testing and validation required for operation on multiple platforms. In five straightforward steps, George Kraft, chairman of the Linux Standard Base, shows you how to build an LSB-certified application.

News flash. Linux has taken the next evolutionary step beyond the source code compatibility of UNIX. Instead of porting or rebuilding source code from one Linux release or distribution to another, Linux has achieved binary compatibility between them all. The application developer only needs to build once for each Linux architecture, such as IA-32, PowerPC, or Itanium, then deploy.

Linux inherently has binary compatibility; the Linux Standard Base (LSB), however, has set some rules and guidelines that make this practical for applications. (See Related topics for a link to the LSB home page.) The path to shrink-wrapping an LSB applications for Linux requires you to code to the portability layer, use the correct ABIs, test with the LSB application checker, follow LSB packaging guidelines, and seek LSB certification. This article presents these five basic steps for using LSB resources to develop a binary-compatible application.

## Step 1. Code to the portability layer

Rather than let applications access operating system resources directly, open systems such as UNIX and Linux have a portability layer to which applications are coded. This is the source code compatibility layer that all well-behaved applications should be be written to. In general, this coding layer is the set of POSIX application programming interfaces (APIs). (See Related topics for a link to more information on the development of a single UNIX specification.) However, since we know that "GNU's Not UNIX," the LSB has redefined POSIX for Linux, hence the LSB specification standard.

There are UNIX-branded systems to ensure source code compatibility for applications among UNIX systems, and there are LSB-branded systems to ensure binary compatibility for Linux applications; however, applications must be coded to use the APIs or application binary interfaces (ABIs) within the scope of the compatibility definition. Applications must not use private interfaces of the operating system or subvert the portability layer by accessing operating system resources directly.

### Platform-tuned apps

Applications or middleware that access system resources directly are said to be "platform tuned." This is sometimes done for optimization purposes, but it greatly diminishes the possibility that the software will be binary compatible between systems or releases. This kind of software is sometimes platform branded to work for a particular release of an operating system, which is fine if that is the intent. This style of programming, however, is outside the scope of the LSB.

Applications that can conform to the LSB specification are likely candidates to be binary compatible among the set of LSB-branded systems. In general, applications have the potential to become LSB compliant if they can limit themselves to using only these 14 system libraries: libc, libdl, libm, libutil, libcrypt, libz, libpthread, libncurses, libX11, libXext, LibXt, libICE, libSM, and libGL.

If an application cannot limit itself to the interfaces of the libraries listed above, then -- to minimize runtime errors -- the application must either bundle the non-specified library as part of the application, or it must statically link the library to the application. However, the libraries themselves must be LSB compliant by using only the interfaces of the libraries listed above.

## Step 2. Use the correct ABIs

Restricting your development to the source API specification is not enough for binary compatibility, because different releases and different systems have different versions of the libraries. To become binary compatible, you must develop to the *ABI*. Similarly to Solaris, Linux is able to version individual ABIs. So, if an imaginary function, `kraft(x,y)`, currently returns a *double,* but a more recent version of the ABI returns an *integer,* then you must use the version that was specified to return the original aggregate data type. To do this, the LSB has created stub libraries. You can link to the LSB stub libraries and to the LSB runtime linker; then the LSB-specified ABI alone can be used. Either the application build will get an unresolved symbol error because it was using something not specified by the LSB, or the LSB stub libraries will ensure the correct ABIs per the binary specification.

To simplify the compilation of an application with the correct ABIs, the LSB provides an `lsbcc` wrapper script. This `lsbcc` uses the LSB stub libraries with the correct ABIs, the LSB runtime linker (ld-lsb.so.1), and header files corresponding to the LSB specification. All you need to do is download the **lsb-base** and **lsb-cc** packages from the LSB (see Related topics for download links), and then integrate them into your normal build procedure:

```
$ rpm -i ftp://ftp.freestandards.org/pub/lsb/lsbdev/lsbdev-base-1.2.2-1.i386.rpm
$ rpm -i ftp://ftp.freestandards.org/pub/lsb/lsbdev/lsbdev-cc-1.2.2-1.i386.rpm
$ lsbcc -o myapplication myapplication.c
```

**or**

```
$ CC=lsbcc make myapplication
```

**or**

```
$ CC=lsbcc ./configure; make myapplication
```

Suppose `kraft` were an ABI provided by libc and it had different versions. If `myapplication` uses the `kraft` ABI, we could compare the application and the library images to see which ABI is actually used.

```
$ /usr/bin/objdump -T myapplication | grep kraft
08048330 DF *UND* 00000032 GLIBC_2.0 kraft
$ /usr/bin/objdump -T /lib/i686/libc.so.6 | egrep kraft
000bb160 g DF .text 00000032 GLIBC_2.0 kraft
000bb160 w DF .text 00000032 GLIBC_2.1 kraft
```

The correct ABI for `kraft` used by `myapplication` should be GLIBC_2.0 so that it will return a *double*. The library and the operating system are allowed to evolve, but the LSB refreshes less frequently to provide a stable runtime environment for applications.

### Terminology

LSB "conformance" requirements are stated in the written specification. LSB "compliance" is a developer's claim of conformance. LSB "certification" is proof of compliance through the certification program (see Related topics for a link to the program). "LSB" is a trademark of the non-profit Free Standards Group, Inc., where the Linux Standard Base workgroup operates.

## Step 3. Test with the LSB application checker

The LSB application checker, lsbappchk, is the primary test for LSB ABI compliance. Below is an example "Hello World" application that uses the `getpid()` API.

### Listing 1. "Hello World"

```
#include <stdio.h>

#include <unistd.h>



main()

{

    printf("hello world: %d\n", getpid());

}
```

Once compiled, you can check helloworld.c to determine if it is LSB compliant. The lsbappchk tool compares the ABI symbols used by an application to ABI symbols defined by the LSB written specification.

```
$ lsbcc -o hw_good helloworld.c
$ lsbappchk hw_good
lsbappchk for LSB Specification 1.2
```

```
Checking binary hw_good
```

We see from the lsbappchk output that it did not find any compliance anomalies; however, if we were to slightly change the above example to use the private function `_getpid()`, we see different results.

```
$ lsbcc -o hw_bad helloworld.c
/tmp/cc5CITzio.o: In function 'main':
/tmp/cc5CITzio.o(.text_0xd): undefined reference to '_getpid'
collect2: ld returned 1 exit status
```

From lsbcc's above standard error (stderr) output, we can see that lsbcc would not compile the application with a non-compliant ABI. Using the LSB lsbcc tool helps you avoid creating non-compliant applications.

But what if we used the native compiler?

```
$ cc -o hw_bad helloworld.c
```

The native compiler allows the application to compile with the system's private interface `_getpid()`, but we know from the LSB specification that this is wrong. In addition, we know that we should never use any interface that is prefixed with an underscore.

```
$ lsbappchk hw_bad
lsbappchk for LSB Specification 1.2
Checking binary hw_bad
Incorrect program interpreter: /lib/ld-linux.so.2
Symbol _getpid used, but not part of LSB
```

From lsbappchk's stderr output, above, we can see the application is associated to the wrong runtime loader and is using the non-compliant `_getpid()` ABI. So, even if you do not use lsbcc to catch non-conformance issues, you can still use lsbappchk later to validate the application. This methodology is not bulletproof, but it is a good indicator.

## Step 4. Follow LSB packaging guidelines

Once your application is built using the LSB headers and linked with the LSB stub libraries and runtime loader, you can start packaging your application the LSB way. The LSB specifies that you are to package your application in an RPM v3 formatted file. In addition, you may not use triggers, nor depend on the execution order of pre-install or pre-uninstall scripts. You are also limited to using only commands specified by the LSB in those scripts and in your application, because other commands are not guaranteed to be present or to behave in expected ways. The LSB does not specify the tool to install these RPM-packaged applications. You can use the rpm tool on RPM-based systems, and alien on Debian.

To avoid name space collisions when installing LSB-conforming applications, the applications belonging to the base operating system or the distribution are to be installed in /sbin/, /bin/, or /

usr/. System administrators can build packages from source and install them into the /usr/local/ directory. However, third-party packages of add-on software must be installed in /opt/*<package>*/, where *<package>* is the name that describes a software suite. Associated files of these /opt/ applications may be in /var/opt/*<package>*/, /etc/opt/*<package>*/, and /opt/share/*<package>*/. Although the LSB specifies that the application is to be installed in /opt/, it is a good idea to make this a relocatable prefix in the RPM spec file. This will let the installer override that location and place the files elsewhere if there are mitigating circumstances.

Startup scripts of daemons, of course, must be placed in /etc/rc.d/. The name of an application's initialization script must be unique and use only the characters [a-z0-9]. To avoid name space problems, its name must be registered with the Linux Assigned Names and Numbers Authority (LANANA; see Related topics). If the script name is hyphenated, then either the leftmost string must be a name registered with LANANA, or it can be your fully-qualified domain name in lower case.

In addition to avoiding name space collisions in the filesystem hierarchy, LSB-conforming packages are prefixed with "lsb-". If the name of the package only contains one hyphen, its name must be registered with LANANA. If the package name contains more than one hyphen, then the area between the first set of hyphens must be either an LSB provider name registered with LANANA, or your fully-qualified domain name in lower case. For example, *lsb-java* may be a name registered with LANANA by Sun Microsystems, but there may be another java package name, *lsb-unregistered.org-java*, which is not registered.

## Step 5. Seek LSB certification

The last step in developing an LSB application is to get it certified. The idea is that any LSB application can run on any LSB distribution. Today, there are at least seven certified runtime environments (see Related topics for a list). The LSB is working to build the number of LSB applications.

You should note that there are only "LSB Certified" applications -- those that have gone through the certification process and have signed the LSB Trademark License Agreement. There is no such thing as an "LSB Compliant" application, because it is either certified or it is not. In addition, the owners of LSB Certified applications warrant that their applications pass the owner's own Functional Verification Test (FVT) on two LSB Runtime Environments (distributions) and on the LSB Sample Implementation. The certification and its warranty make an LSB application more valuable to the consumer.

Here's a checklist of what to do as an application developer during LSB certification:

- Register yourself on the LSB Certification Web site
- Self-test your application using the LSB-provided tests, and then upload the results
- Warrant that it passes your own FVT
- Complete the Conformance Statement Questionnaire
- Sign the LSB Trademark License Agreement

The above information is then confirmed by the Certification Authority. Finally:

- Pay the certification fees and sign the LSB Certification Agreement

## Conclusion

The goal of the LSB is for applications to run on any Linux distribution. The means by which the LSB achieves this is by distribution and application certification backed by a warranty of conformance. If your system or application is doing something non-conforming, then the customer can ask you to fix it per the specification. The well-defined ABI makes it possible to "shrink wrap" an application and put it on the shelf to sell at your local computer store. The "system requirements" list on the side of your product's box should say "LSB v1.2" instead of specific releases of Linux distributions.

Why limit your product's potential install base to just a few systems, when you can widen your marketplace to any LSB-compliant system? A side effect of the LSB is less software testing while letting it install on a larger set of Linux systems.

I challenge you to take a few moments to download the LSB development environment, rebuild your application, and then test it with the LSB application checker to see how compliant you are. You might find that you only need a few little tweaks to become compliant, or you might need to rewrite areas where you are using private system interfaces. In either case, you can make an informed decision about how compatible your application is and why.

## Related topics

- The Linux Standard Base home page is the main source for the written specification and more.
- The Single UNIX Specification page is the development area for the working group evolving the Single UNIX Specification.
- Name space assignments are provided by LANANA, the Linux Assigned Names And Numbers Authority.
- Read "Porting UNIX applications to Linux -- Hints and tips" for general considerations surrounding a Linux port as well as specific information on migrating applications to Linux on IBM zSeries machines.
- Build your skills in Linux systems administration with our certification exam study guides. Whether you choose to take the exams or not, our certification-prep tutorial series will immerse you in Linux fundamentals as well as advanced topics.
- Find more resources for Linux developers in the developerWorks Linux zone, including our newest how-to tutorials.
- Build your next development project on Linux with IBM trial software, available for download directly from developerWorks.