Because Linux is an open operating system, you can configure and assemble it to suit specialized purposes. However, while variety and choice are beneficial for users, heterogeneity can vex software developers who must build and support packages on a multitude of similar but subtly different platforms. Fortunately, if an application conforms to the Linux Standard Base (LSB) and a Linux distribution is LSB-compliant, then the application is guaranteed to run. Discover the LSB, and learn how to port your code to the standard.

The Linux® Standard Base (LSB) is a specification and collection of tools and test suites that help Linux software developers increase compatibility among Linux software distributions. Applications and distributions can be certified compliant with the specification, providing assurance to users that certified software is compatible. This tutorial describes the LSB and explains how to vet your application code to conform to it.

Prerequisites

To benefit from this tutorial, you should have experience with the C or C++ programming language as well as the typical Linux software development environment and its cadre of tools, including the compiler, linker, system libraries, configuration and build utilities, and optionally packaging tools.

You should also have experience installing software from the command line and have at least a modicum of experience with administering and maintaining a Linux system, such as configuring a file system, starting and stopping network services, and adding system services. If you're running Ubuntu or Debian GNU/Linux, you should also have some experience with the apt package manager.

Before you start the tutorial, you must install the qemu package for your Linux distribution. For additional speed, install kvm. Alternatively, you can easily install VMWare Workstation on any computer running Microsoft® Windows® XP and run a virtualized Linux instance on the Windows XP platform. You can download a trial version of VMWare Workstation at no cost and use it to create one or more virtual machines (VMs). After you've created a VM, you can save it and switch to the no-cost VMWare Player to replay it.

You may also find it useful to download and read the LSB 4.0.0 specification appropriate for your target hardware platform. Hardware-specific LSB specifications are available for seven popular architectures: the IA32, IA64, PPC32, PPC64, S390, S390X, and AMD64 processors.

If you wish to certify your application, please visit the Linux Foundation's LSB Certification Program.

To compile and experiment with the examples in this tutorial, you must have a computer running either the Linux or Windows XP operating system. The qemu application, or VMWare Workstation on either of those platforms, provides a Linux environment for your work.

If the application you wish to port depends on software libraries other than those ratified as part of the LSB specification for your processor, you must install those libraries on the virtual system, as well. To avoid confusion, it's ideal to install application-specific libraries separately from the standard libraries.

The Need for a Standard

According to the DistroWatch website, at least *25* significant Linux distributions currently exist. If you include other variations -- some optimized for size (such as Damn Small Linux), some for convenience (such as Knoppix), some tailored to a specific audience (such as Fedora Core), and still others customized for a specific country (such as Red Flag Linux)--that rather staggering number of choices easily doubles or triples. Linux is the only modern operating system with so many variations.

On one hand, the do-it-yourself culture of Linux encourages diversity and specialization. On the other hand, such proliferation can be overwhelming, especially if you're an independent software vendor (ISV) faced with developing, selling, and supporting a Linux application. If the "UNIX® Wars" of the 1980s splintered ISVs' chances of success, the large number of versions of Linux seems to promise nothing less than wholesale obliteration of the market for rich Linux applications. Imagine the expense of building and supporting a complex application on more than a handful of Linux variants. That's one reason the Linux Standard Base is so useful.

Promoting Interoperability

In an attempt both to encourage diversification *and* to provide consistency, the Free Standards Group (now The Linux Foundation) created the *Linux Standard Base*, an "open source project to develop and promote a set of standards [to] increase compatibility among Linux distributions and enable software applications to run on any compliant system" (according to the LSB Web site).

Specifically, the LSB defines a consistent Linux platform. A Linux *distribution* is LSB-compliant if it implements (at least) the LSB platform. (Nothing precludes a version of Linux from providing additional features.) A Linux *application* is LSB-compliant if it consumes only the services that the LSB platform provides. If the application requires additional libraries, it provides those itself, either during installation or through static linking. (Of course, the libraries must be both self-sufficient and LSB compliant as well.) See the sidebar **LSB terminology** for more details.

The figure below shows the many areas for which the LSB defines (or will eventually define) standards. The areas are *libraries*, the basis of all Linux applications; the *execution environment*, including mandates for where certain critical system files reside and how the system provides for localization services; *system initialization*; *common commands and utilities* to be found across all LSB-compliant systems; and *user and group management.*

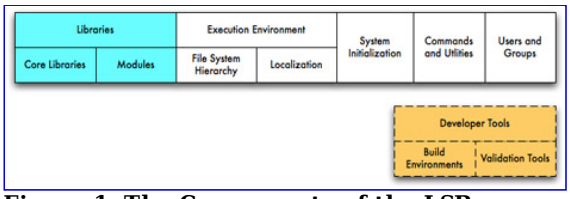

Figure 1. The Components of the LSB

LSB Terminology

When discussing the LSB, you'll frequently hear (and read) the terms *conformance*, *compliance*, and certification.

1. The LSB defines the requirements for a runtime and an application to be in *conformance* with the specification.
2. An application or a distribution is in *compliance* with the LSB if its developer has realized all the suitable LSB requirements and claims that the software *conforms*.
3. An application or distribution receives *certification* if it successfully demonstrates *compliance* through official review.

In this tutorial, you will learn how to create compliant applications. You cannot self-certify your application, although you can run many tests that greatly increase the likelihood of certification.

Application developers are probably most concerned with the blue and orange boxes.

In blue, the LSB stipulates a set of core libraries and additional modules that an LSB-compliant application may use. The core libraries include libc, libm, libpthread, libpam, libcrypt, libz, libncurses, librt, and libgcc_s. Modules include X Window System Version 11 graphics (libX11, libXt, libXext, libSM, libICE, and libGL) and the C++ standard library, libstdc++.

In orange--and not part of the LSB specification--is a set of valuable companion tools to validate applications, packages, and soon, static libraries, as LSB-compliant.

Binary Compatibility

The LSB specification is a binary compatibility standard. An application binary (compiled, executable code) that is LSB-compliant can run unchanged on any LSB-compliant Linux distribution that is built for the same processor architecture as the application.

Binary compatibility is remarkably different from source compatibility. An application that is source-compatible requires the software to be recompiled in place, using the local system's compilers and libraries--two significant founts of variability and unpredictability. Source compatibility doesn't guarantee that the executable will run at all, let alone run as expected.

The Benefits of Consistency

Think of the LSB as a kind of binding agreement between a distribution and an application. The distribution "agrees" to provide *at least* the specification's libraries and interfaces, and the application "agrees" to use *only* those libraries and interfaces, providing any other software that's needed. To ensure that both parties adhere to the agreement, the LSB provides tools, guidelines, and validation services to certify Linux distributions and applications as LSB-compliant.

Adherence to the LSB benefits distribution vendors, ISVs, and users alike.

For the distribution vendor

- A distribution vendor can promote its compliance with the LSB, assuring customers that LSB-compliant applications will run consistently and reliably on its product.
- After its distribution is certified LSB-compliant, a vendor can re-focus its energies on differentiating the distribution with novel features.

For the ISV

- An application developer can create and certify a single binary and deploy it to any LSB-compliant distribution.
- An ISV need only test and support the single certified binary. Additionally, if the binary is sufficiently internationalized, a single version of an application might suffice for all customers, regardless of nationality.

For the customer or user

- The user can pick and choose among competing LSB-compliant distributions, either to deploy the most appropriate flavor of Linux or to avoid vendor lock-in.
- The user has many Linux applications to choose from.

Given all the dynamics listed above, distribution vendors, ISVs, and users reap rewards from each unique investment in Linux. In turn, Linux benefits as well.

The LSB Sample Implementation

As described above, the LSB is a "contract" between an application and a distribution to adhere to the LSB specification. Ignoring the complications of making a Linux distribution compliant, how can a Linux application developer--you--ensure compliance?

If you're a developer, you likely know your code very well and can work diligently to meet the obligations of LSB compliance. However, it may not always be obvious that your code depends on a non-standard (non-LSB) component, especially as a typical Linux distribution has thousands of components strewn about the system.

For example, perhaps your code depends on a non-compliant string library found only on your favorite distribution, or perhaps the string library used was added locally and recently by your system administrator. As an another example, perhaps your code depends on a specific *version* of a library. Version W of the library is LSB compliant, but version Y--the version you depend on--is not part of the LSB specification.

Tools to Help You Migrate

To help find and remove such dependencies--which is the crux of making your code LSB compliant--the LSB provides some build environment utilities (about which more later) and the Sample Implementation.

The Sample Implementation helps you validate that your application executes successfully in a compliant runtime environment. You can think of the Sample Implementation as a bootable mini-Linux--an environment that proxies for a full LSB-compliant distribution. You can install it on any Linux system, even one that is not compliant. It also contains header files, stub libraries, and a compiler wrapper that simulate a software build environment on a compliant Linux system. These can help you identify source code that depends on non-LSB libraries and application binary interfaces.

Of course, you could use an LSB-compliant distribution as your test environment, but the Sample Implementation has a few advantages. It only includes what's defined in the LSB, and several versions of it exist--one for each version of the LSB specification. Hence, if you wish to verify that your application is LSB 3.0 and LSB 3.2 compliant, you can install the respective Sample Implementations on a single test system rather than set up two Linux systems. The Sample Implementation is the preferred method to validate your application against a newly ratified LSB specification before real-world, compliant systems are available.

The Target Platform

For this tutorial, you'll use LSB 4.0.0 as the target platform. This version was the most recent LSB specification at the time this tutorial was written.

Versions 3.2.0 and 3.1.0 are the previous versions of the LSB. You'll find that many popular Linux distributions are certified with one of the earlier instances of the LSB.

Since LSB 4.0.0, the LSB build utilities, which are used outside the Sample Implementation, are not bound to a particular specification version and can be used to create applications compatible with any given LSB version greater or equal to 3.0.0. By default, the latest LSB version is used as a target. To change the target version, one should use either the `LSBCC_LSBVERSION` environment variable or the `--lsb-target-version` option of the compiler wrapper.

Setting up the Emulator

To run the LSB 4.0.0 Sample Implementation, you need an operating system capable of running the SI in a chroot environment. The community distribution Fedora 9 (also called FC9) is one flavor of Linux that can run the 4.0.0 Sample Implementation, and we will use it for this tutorial.

What if you don't have FC9 installed, or you wish to run your application on multiple instances of the LSB, or on a variety of Linux distributions? You could assemble a (somewhat large) collection of hardware or multi-boot among all your desired variations, but a far easier and less-expensive solution is to run each Linux distribution in its own emulator or virtual machine. In the following examples, we will be using the qemu emulator to boot from a Fedora Core 9 disk image.

Install qemu

Install qemu on your computer or a similar application, such as kvm or VMWare Player, and boot FC9 to run the LSB 4.0.0 Sample Implementation. For example, if you are running the Debian GNU/Linux or Ubuntu Intrepid Ibex, installing qemu merely requires installing its package with the following command in a terminal window. (In each of the following command lines, a preceding dollar sign (`$`) means to run the command as a normal user, and a preceding hash mark (`#`) means to run the command as root.)

```
$ sudo apt-get install qemu
```

You can also use the Synaptics package manager if you prefer a graphical interface.

Download a qemu Image

You can download an FC9 disk image from the Linux Foundation website. Because this image contains an entire operating system, it's quite large (1400 MB).

Running qemu

To start qemu, run the following command in a terminal window, in the same directory where you saved the disk image.

```
$ qemu fedora-9-i386.qcow2 -redir tcp:2222::22 &
```

The qemu application traps all mouse and keyboard input unless you press Ctrl-Alt. To set the focus back to the virtualized environment, click in the qemu window. The `-redir` option above will enable you to log into qemu with ssh from the console.

After you boot the disk image, you will need to answer a few questions, such as the name of the user account you wish to create. After you have created a user, reboot qemu and get root access by logging in via single user mode, as follows.

1. After qemu boots, at the prompt saying "Press any key to enter the menu", strike a key.
2. At the GRUB prompt, type `e` to edit the boot options.
3. On the next screen, arrow down to the line beginning with the word "kernel".
4. Type `e` to edit it.
5. At the end of the line, type a space and then the word `single` to specify single user mode.
6. Hit ENTER.
7. Type `b` to boot FC9 in single user mode.
8. At the root prompt (`#`), type the command `passwd` to change the password for root.
9. Reboot qemu.

Congratulations! You now have a copy of Fedora Core 9 running on its own disk image, entirely separate from your own Linux installation. If you started qemu with the command specified above, you can log into the emulator from your "real" computer's console with the following command.

```
$ ssh -p 2222 localhost
```

Logging into the emulator with ssh, as opposed to typing into the window displayed by qemu, will enable you to copy and paste example code, something highly desirable in this tutorial.

Porting Your Code to the LSB

You can now begin the process of porting your code to the LSB.

The General Approach

Porting an application to the LSB requires the following steps.

1. Copy your code to the new build system.

   The new build system might be an LSB-compliant Linux distribution running on separate hardware or, as in this case, an emulator or virtual machine.

2. Build your code and run the Linux Application Checker ("AppCheck") tool to scan your binary for symbols that are not expressly provided in the LSB specification.

   You can also use this tool to scan your static archives for suitability for use in an LSB-compliant application.

3. If AppCheck finds invalid symbols, change your code or the assembly of your code to bring it into compliance.

   For instance, if you're using a library that isn't part of the LSB specification, link to it statically so that the code is self-contained in your binary. (Again, this assumes that the code in the library itself is otherwise LSB-compliant.) Assuming that you've addressed all the issues, you can proceed to the next step.

4. Use the LSB Sample Implementation to build the code in a clean, compliant environment.

   If your code uses libraries that are not provided under the LSB, you must modify your build process to either install or link statically to those libraries. (Again, remember that all libraries must be LSB-compliant as well.)

5. If your code builds successfully, run the code in the LSB Sample Implementation.

6. If your target Linux system is LSB-compliant, run your code on that system.

7. Package your application.

   LSB-conforming systems promise to be able to install an LSB-compliant RPM package. However, you need not limit yourself to that format, with the caveat that the packaging technology you choose must itself work on an LSB-compliant system. For example, a shell script with a tarball is an acceptable installation format.

Now, let's build a simple application to demonstrate this process.

The LSB Build Environment Utilities

Before starting the LSB Sample Implementation, try the the build environment utilities, which you can install easily on virtually any Linux system. You can use the utilities lsbappchk and lsbpkgchk to quickly determine why your application and RPM package, respectively, aren't compliant with the LSB.

Download and Install the Build Environment Utilities

Log into qemu, then become root and type the following command.

```
# nano /etc/yum.repos.d/lsb.repo
```

Paste the following code into the nano screen.

```
[lsb-4.0]
name=LSB 4.0
baseurl=http://ftp.linuxfoundation.org/pub/lsb/repositories/yum/lsb-4.0/repo-ia32
enabled=1
gpgcheck=0
```

Type Ctrl-X and save the file. Now you can install packages from the LSB repositories into FC9. The following line will do just that, as well as install GCC (the C compiler, which is not present on the disk image), and a library we'll use later.

```
# yum install lsb-task-sdk lsb-appchk gcc pcre-devel
```

Installing the lsb-task-sdk package above installs the following packages in turn.

- **lsb-appchk** verifies that a supplied binary only uses the dynamically linked symbols defined in the LSB.
- **lsb-pkgchk** verifies that an *application package*--a bundle used to install software on an LSB-compliant system--is valid. The lsb-pkgchk tool is intended for RPM packages only. However, the LSB does not mandate the use of RPMs to install software.
- **lsb-build-base** provides stub libraries and header files. While the stub libraries don't implement the functions defined in the LSB, they do mimic the actual dynamic libraries found on an LSB system. Hence, you can use lsb-build-base to build a compliant application.
- **lsb-build-c++** adds C++ support to the build environment.

- **lsb-build-cc** contains lsbcc, a wrapper around the GNU Compiler Collection (GCC) compiler that yields LSB-conforming applications. If your application uses a GNU-style configure script, you can easily modify your script to use lsbcc instead of the default system compiler, which is usually GCC. In some cases, you can directly replace GCC with lsbcc (for example, in a makefile).

An Example Program

Now as a regular user, create an empty C source code file called `echoargs.c`.

```
$ nano echoargs.c
```

Copy and paste the following source code into the nano editor screen. When compiled and executed, it will echo its command-line arguments (error-checking code has been omitted intentionally).

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

main(argc, argv)
int argc;
char *argv[];
{
  int i = 0;

  for (i = 1; i < argc; i++) {
   fputs(argv[i], stdout);
   putchar(' ');
  }

  putchar('\n');

  exit(0);
}
```

Compile the program with the local C compiler, which is now GCC.

```
$ cc -o echoargs echoargs.c

$ ./echoargs hello there, world!
hello there, world!
```

The code works as intended, but is it LSB-conforming? To find out, run the lsbappchk command.

```
$ /opt/lsb/bin/lsbappchk echoargs
```

Here is the relevant portion of the output of lsbappchk. Ellipses (...) indicate omitted text.

```
BIN: echoargs
LSB Application Checker Report
==============================

Binary echoargs:
  FAIL
  Incorrect program interpreter: /lib/ld-linux.so.2
...
section .gnu.hash is not in the LSB
  FAIL
  section .gnu.hash is not in the LSB
...
Dynamic Tag 0x6ffffef5 unknown
  FAIL
  Dynamic Tag 0x6ffffef5 unknown
```

The echoargs program is clearly not an LSB-conforming application.

Use the LSB Compiler to Make the Application Conform

Now, rebuild the same code using the LSB compiler--lsbcc--and run lsbappchk on the binary it produces.

```
$ /opt/lsb/bin/lsbcc -o lsb-echoargs echoargs.c

$ /opt/lsb/bin/lsbappchk lsb-echoargs

BIN: lsb-echoargs
LSB Application Checker Report
==============================

Binary lsb-echoargs:
```

```
...
  No failures found.

No LSB compliance issues were found.
```

Much better! The new binary is conforming. It was built with the LSB stub libraries using the LSB include files (header files) instead of the system header files. But does it run?

```
$ ./lsb-echoargs hey there
hey there
```

So it does. As another example, consider the listing below. This code snippet uses the open source Perl Compatible Regular Expressions (PCRE) library to add the power of regular expressions to traditional C. As useful as PCRE is, it's not part of the LSB specification.

```
#include <pcre.h>

int main()
{
  pcre *re;
  const char *error;
  int erroffset;

  re = pcre_compile("^[A-Z]", 0, &error, &erroffset, NULL);
}
```

Create an empty C source code file called `pcre.c` as you did above with `echoargs.c`, and then paste in this listing. Build it and check it with lsbappchk as follows.

```
$ cc -o pcre pcre.c -l pcre

$ /opt/lsb/bin/lsbappchk pcre
```

This time you should receive some additional error messages.

```
  FAIL
  DT_NEEDED: libpcre.so.0 is used, but not part of the LSB
...
  FAIL
  Symbol pcre_compile is used, but is not included in LSB 4.0 (Core & C++ & Desktop)
```

The functions declared by PCRE are flagged because they're not LSB-compliant. You can avoid these errors (assuming that the rest of the PCRE library is LSB-compliant) by adding the `-static` flag when you compile the application.

The same program compiled with lsbcc does not generate errors.

```
$ /opt/lsb/bin/lsbcc -o lsb-pcre pcre.c -l pcre

$ /opt/lsb/bin/lsbappchk lsb-pcre
```

The lsbcc wrapper modifies command-line arguments to the GCC compiler to use LSB header files and libraries and to avoid dynamic links to non-compliant LSB libraries. Here, lsbcc statically linked the PCRE code into the executable -- one solution to avoid library differences between one Linux platform and another.

If you use GCC and other tools to build your code, lsbcc is probably preferable to the chroot LSB Sample Implementation. In contrast, if you use a compiler other than GCC or have dependencies on a specific compiler, compiler options, or library or include file paths, the Sample Implementation is superior. The next sections demonstrate how to install and use the LSB Sample Implementation.

The Sample Implementation

We will install the Sample Implementation onto the FC9 disk image running on the emulator, and test our applications within that nested environment.

Installing the Sample Implementation

First, within qemu, download the lsbsi-chroot and lsbsi-tools binaries from the LSB SI Tools Release page. You will also need a special package to force installation of dependencies under Fedora.

Next, install the packages you downloaded.

```
# yum localinstall --nogpgcheck redhat-lsb-4.0-1.fc11.i386.rpm Lsbsi-chroot-4.0.0-1.i586.rpm Lsbsi-tools-4.0.0-1.i586.rp
```

Building and Testing Within the Sample Implementation

Within your FC9 emulation, you now have two complete environments: the FC9 environment and the Sample Implementation. For convenience and clarity, create an SI working directory and set two environment variables to simplify copying files between them. Substitute the directory where you saved the source code and executables you created above for the directory `/home/fred` below.

```
# FC9=/home/fred

# SI=/opt/lsb/si/chroot/tmp/work

# mkdir -p $SI
```

The `$FC9` directory can be your regular Fedora Core 9 working directory. The `$SI` directory is the drop for binaries that you created with the LSB build environment utilities and wish to run in the Sample Implementation. You can access this directory within the Sample Implementation as `/tmp/work`. So, to test `echoargs.c` in the chroot Sample Implementation, take the following steps.

```
# cd $FC9

# cp * $SI

# /opt/lsb/bin/si-chroot
```

You are now inside the Sample Implementation.

```
# cd /tmp/work

# ./lsb-echoargs hello there
hello there
```

The LSB-compliant executable you created with the build environment utilities under Fedora Core 9 should run with no problem within the SI. Now try compiling an executable within the SI and then running it there.

```
# cc -o si-echoargs echoargs.c

# ./si-echoargs hi from si
hi from si

# exit
```

Your application works, and you are now outside the SI. Change to the `$SI` directory, and then test and run the new executable under Fedora.

```
# cd $SI

# /opt/lsb/bin/lsbappchk si-echoargs
...
  No failures found.

No LSB compliance issues were found.

# ./si-echoargs hi from si outside
hi from si outside
```

The executable you built within the Sample Implementation should test and run without any problems, because Fedora Core 9 is LSB-compliant.

Summary

While the example code shown in this tutorial was simple, the same techniques used to build and test a few lines of code apply equally well to a few thousand lines of code. Use the lsbappchk tool to find questionable symbols. Try building your code with the LSB build environment utilities and then within the stand-alone LSB Sample Implementation chroot environment. Leverage a tool such as qemu, kvm, or VMWare to multiply your computing resources and run your targeted flavors of Linux.

With a little work, you can shape your application to be LSB-compliant, and then apply to have your application certified. Certified distributions and certified applications enable users to invest in Linux comfortably. The LSB provides uniformity, which--perhaps paradoxically--promotes choice.

And after all, choice is what Linux is all about.

Credits

This article by Martin Streicher originally appeared on IBM developerWorks. It has been revised for current technology practices by Ted Ts'o, Jeff Licquia, and Ron Hale-Evans.