Building applications for the Linux Standard Base

C. Yeoh

The goal of the Linux™ Standard Base (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 Linux system. There are currently LSB specifications available for the Intel Architecture IA-32™ processors and for the 32- and 64-bit PowerPC™, Itanium™, 31- and 64-bit zSeries™, and AMD64™ architectures. This paper describes the process of building LSB-compliant applications, and covers the use of the LSB development environments, testing of binaries, and packaging.

Development of the Linux** kernel was started by Linus Torvalds in 1991. By 1992, some early Linux distributions such as MCC (a Linux distribution from the Manchester Computing Centre), TAMU (a distribution from Texas A&M University) and SLS (Softlanding Linux System) were easily available over the Internet. Distrowatch.com, a comprehensive Web site following Linux distributions, has over 350 distributions in its database.[1]

Standards for operating systems can reduce the incompatibilities between various implementations. The POSIX** standard is an example of an API (application programming interface) standard which has helped keep a certain level of commonality among UNIX** implementations. Where there are large numbers of implementations, as is the case with Linux, widespread core compatibility makes it possible to have applications that work correctly on many implementations.

The Linux Standard Base (LSB) is a set of operating-system standards with the goal of increasing compatibility among Linux distributions and enabling software applications to run on any Linux-compliant system.[2] The LSB is developed and promoted by the LSB workgroup of the Free Standards Group (FSG), an independent, non-profit organization dedicated to accelerating the use of free and open-source software by developing and promoting standards.[3] Other examples of standardization workgroups of the FSG are OpenI18n (which addresses issues of internationalization), OpenPrinting, and the Open Cluster Framework (which defines standard clustering APIs).

The LSB workgroup is divided into several subprojects, each of which is responsible for a major component of the project; such as, its specification, futures, testing, sample implementation, example

applications, and the build environment. There are tools available to test distributions as well as the third-party applications that will run on those distributions.

■ Depending on its intended use, people have differing views on what constitutes a Linux system ■

The project was first announced in May of 1998, and the first official version of the LSB specification was released in June 2001. At first only the IA (Intel Architecture)-32** architecture was supported, but the 2.0 version released in 2004 supports seven architectures: IA-32, Itanium**, 32- and 64-bit PowerPC*, AMD64**, and the 31-bit and 64-bit z/Architecture*. The breadth of interfaces has also increased, with support of C++, C99, SUSv3 (Single UNIX Specification version 3) and IPv6 (Internet Protocol version 6) also recently being added to the specification.

This paper, which describes the process of building LSB-compliant applications, is organized as follows. In the following section, we describe the structure of the project and its development. In "Specification documents," a brief overview of the structure and contents of the specification documents is given. "LSB build tools" describes the tools that should be used to build an application. The following two sections explain the issues relevant to packaging and the File Hierarchy Standard. "Using external libraries" presents an application which uses non-LSB-standard libraries. "Making the xpaint application LSB-compliant" presents an example of the process required to build an application, while "Application battery" describes a pool of applications built to be compliant, which are available to use as a reference for commonly encountered problems. "Testing" covers the tools available to test an application for compliance.

DEVELOPMENT OF THE LINUX STANDARD BASE

The Linux Standard Base project is an open-source project and is run much like any other open-source project. It is not controlled or operated by any company, and contributors come from a broad range of areas, from programmers to documentation writers and people with backgrounds in developing international standards. Like many projects, there has been an ebb and flow of contributors, with some having participated continuously from the start of the project, some having helped out in a few specific areas, and others who have just recently joined. There are participants from distribution developers, application developers, and those simply interested in standardization of Linux. Most of the direct participants are paid to work on the LSB, (unlike most open-source projects), but this phenomenon is waning.

There is significant industry support for the project, with developers from large companies such as IBM, Intel, and Hewlett-Packard actively involved. Representatives from the major distributions such as SUSE, Mandrake, the Debian** project, and Red Hat, as well as the Open Group which has been involved in standards development for a long time, regularly participate in the development of the LSB.

The LSB bases its standard completely on the work of other open-source projects, most of which do not run according to fixed schedules, nor are they necessarily synchronized with each other. This can present a challenge to standardization, as often it will be advocated that a much better version of some component of the standard is almost ready or has recently become available and should be adopted. Some distribution developers want to use the latest cutting-edge technology, while others would prefer to be more conservative, and use only features that have been tested for a much longer period of time. This may cause conflict as to what is the most appropriate course to take.

To help resolve this problem, a set of guidelines has been developed to decide what qualifies for inclusion in the standard. Generally, the requirements are that it be considered best practice; that there is an implementation with a "no strings attached" license (e.g., BSD/LGPL [Berkeley Software Distribution/ Lesser GNU Public License] licenses); that there is a stable ABI (application binary interface); and that the upstream developers are supportive of standardization. For final inclusion of a feature, it is also necessary to have test suites to test implementations utilizing this feature and a consensus among distribution developers that it is a feature they want to support.

Depending on its intended use, people have differing views on what constitutes a Linux system. A result of this is that there has been contention on how broad a range of features the LSB specification should cover. The requirements for an embedded system are significantly smaller than those of a desktop system. The LSB 1.x versions of the specification targeted something between these implementations (that is, the requirements for most server-type systems), but there has been consistent pressure for both more and fewer requirements to be included. In order to address this, LSB 2.0 split features into building-block-like modules. Although only one configuration of modules is currently supported by the certification program, in the future this division will allow for embedded and desktop system configurations to be certified. It also has the benefit of making it easier for other standardization groups to reuse the work of the LSB and vice versa.

Although there are other Linux standardization efforts, they are complementary to the LSB rather than competing. For example, the China Electronics Standardization Institute has created a Chinese Linux Standard that builds upon the LSB documents.[4]

Although ISO (International Organization for Standardization) has not been directly involved in the development of the LSB specification, a subset of the LSB 2.0.1 specification has been submitted to ISO/IEC (International Electrotechnical Committee) JTC (Joint Technical Committee) 1 as a publicly available specification (PAS) submission. At the time of writing, the submission is undergoing analysis by the SC22 (Subcommittee 22) group. As explained in "Specification documents," there are strong links between the LSB specification and the ISO/IEC 9945 standard, and some developers are working on both standards.

Despite the name of the project, the Linux Standard Base specification is not intended to be Linux-kernel specific. Theoretically, it should be possible for a BSD-based system to become LSB-compliant, and it is expected that Sun will announce that Solaris** is compliant with the LSB specification.[5]

SPECIFICATION DOCUMENTS

It is important to understand that the LSB specification is a binary standard, specifying an application binary interface (ABI). This is different from some other well-known standards, such as SUS or POSIX, which are source or application-programming-interface (API) standards. Specifying an ABI instead

■ The Linux Standard Base specification is not intended to be Linux-kernel specific ■

of an API standard means that compliant programs are portable across compliant implementations, without the need for recompilation. This has advantages for software developers, as they do not need to build, test, and maintain a separate version of their application for each distribution they wish to support. There are now many users of Linux systems who either do not have the skill required to compile applications themselves or do not wish to spend time doing so, and it is to their advantage to have prebuilt binaries available. Widespread binary compatibility also helps distribution developers, as it makes available a larger pool of applications that will work with their product. Binary compatibility is a basic requirement for the large-scale consumer market.

An ABI defines a low-level interface between a program and the libraries and operating-system services that it uses. This contains details such as the format of the files, calling conventions, C++ "name mangling" conventions, and symbols available and their corresponding versions (if applicable). For example, the System V ABI[6] document is one of the core specifications upon which the LSB is built.

The LSB specification is a set of specification documents. It is divided in two ways, by the functionality supported and by the architecture. The functionality documents are split into several modules, such as core, embedded systems and packaging, graphics, and desktop. This concept was introduced in the 2.0 version of the specification, in order to allow for different LSB module configurations to be certified. For example, a distribution targeted at the embedded-systems market may only support the core and embedded-systems modules; whereas, a desktop system may implement all of them. The exact configuration of modules that are available and able to be certified is still in development.

Because the specification is a binary standard, a significant portion of the information is architecture-specific. Therefore the module documents are further divided, such that there is one document that specifies the architecture-independent information, and one additional document per architecture that contains the architecture-specific details.

■ The Linux Standard Base specification gives two options for packaging an application ■

Where possible, established standards are referenced, rather than duplicating the information. For example, Linux system developers in most cases follow the POSIX 1003.1 standard. The LSB leverages the work done by the Austin group who developed this standard by referencing a subset of that document. In some cases, the behavior as implemented on Linux systems is slightly different, and this is noted in the LSB specifications.

The LSB specification does not cover everything that would be found in an ordinary Linux installation. Currently, it covers the core areas of each system, and coverage is expanding as resources become available. Although, as described earlier, the information is split into various documents, the specification contains a list of interfaces and the libraries in which they reside, a set of commands and utilities, definitions of the object format of executables, runtime linking requirements, and package format definitions.

Of particular interest to developers porting from UNIX to Linux systems is a document[7] produced by Andrew Josey, which summarizes the incompatibilities between POSIX and the LSB. Sometimes the differences are accidental and transitory as updated releases are made. Others are the result of intentional decisions to differ in design or implementation. There is an ongoing effort to understand the reasons for these differences, and an attempt is being made to harmonize the two standards.

LSB BUILD TOOLS

An LSB-compliant implementation is not required to construct an LSB-compliant binary. Moreover, an LSB-compliant distribution is not necessarily a suitable environment to build a compliant binary without the use of the tools described later in this section. Although the default build environment provides the services and interfaces required by the specification (such as header files and the default linkage of symbols in the libraries), it may not match the standard, nor is it required to do so. The LSB project has not released any tools that properly check an implementation for its suitability in natively building an application, and it is recommended that tools which are suitable for this always be used.

In practice, apart from the issue of only using functionality supported by the LSB, there are three other main issues in creating a binary that is compliant: LSB header files, stub libraries, and the LSB linker. These issues are described in the next subsections.

LSB header files

For interfaces supported by the standard, the number and size of parameters for functions, the size of any values returned by a function, and the size of variables is specified. Also, the meaning of specific values of parameters for certain functions has also been standardized. For example, the lseek library interface, for which POSIX 1003.1 specifies an API as:

```
off_t lseek(int fildes, off_t offset, int whence),
```

it also specified that the third parameter, whence, can be passed the macro values SEEK_SET, SEEK_CUR, or SEEK_END. As the LSB is an ABI standard, it further defines the size of the types (for each architecture), and the values and meanings of the macros for the third parameter, as follows:

```
#define SEEK_SET   0   /* Seek from beginning of file. */
#define SEEK_CUR   1   /* Seek from current position. */
#define SEEK_END   2   /* Seek from end of file. */
```

The size and number of the parameters of functions and the definitions of the macros can change over time. If the version of glibc (the GNU C library) on the system which builds the application does not have header files which conform to the ABI, then it would be possible to create a binary that would work correctly on the system where it was built, but would not behave as expected on an LSB-compliant system.

Using the previous example, if the definitions of the SEEK modules changed to the following:

```
#define SEEK_CUR  0   /* Seek from current position. */
#define SEEK_SET  1   /* Seek from beginning of file. */
#define SEEK_END  2   /* Seek from end of file. */
```

then when an application that ran on an LSB-compliant system tried to do an absolute seek, it would do a relative one and vice-versa.

To prevent this problem from occurring, an application should be compiled using header files that are known to conform to the specification. The LSB project has produced a set of header files that are known to be compliant for each supported architecture. All of the interfaces supported by the standard are stored in an SQL (structured query language) database. The information in the database is used to generate both the specification and the header files used by the build environment.

Stub libraries

Some of the shared libraries supported by the LSB, such as *libc*, use symbol versioning, wherein the specification lists a specific symbol version for each interface. Each interface in one of these libraries has a version string associated with it. This allows for multiple ABI-incompatible versions of the same function to be contained within the same shared library. When a binary is linked and the shared library to which it is linking contains more than one version of a symbol, the default version defined by the shared library is used.

Although an LSB-compliant distribution must supply all of the required interfaces with the specified versions, these do not necessarily have to be the default versions. It would not be an uncommon situation to be building an LSB-compliant application on a system where newer versions of the required interfaces are the default. Ordinary compilation would result in a non-compliant binary.

To address this problem, a set of stub libraries has been produced for the build environment containing all of the interfaces required by the specification. When a shared library implements symbol versioning, the appropriate version is used for the interfaces. Much like the LSB header files, these stub libraries are generated directly from the information in the LSB database, ensuring that it is synchronized with the specification document.

Using the LSB stub libraries instead of the shared libraries on the development system when linking a binary ensures that the correct interfaces are used. Because the shared libraries contain only the interfaces required by the specification, if an application uses an interface outside of the specification, a link error occurs. It is much better to detect these sorts of problems as early as possible in the development process, as opposed to later, when using a checking tool. The stub libraries are intended to be used for the production of the binary to be packaged, and not just as a method to test if the build system would build a compliant binary.

LSB linker

The LSB specification requires the use of a non-standard runtime linker. For example, on IA-32 Linux systems, ordinarily the runtime linker used by applications is /lib/ld-linux_so.2. During the development of the specification, it was realized that there would be cases where a distribution would want to ship two versions of a given library: one which is LSB-compliant, and another which exists either for backward compatibility reasons or is a newer version that has some features which the distribution wants to make available. Occasionally developers of libraries accidentally make ABI-incompatible changes without either incrementing the symbol-version numbers or shared-library versions. This normally would make it impossible to ship two ABI-incompatible versions. When all LSB-compliant applications use the LSB linker, the linker can arrange for the LSB-compliant version of the shared library to be used; whereas, the ordinary Linux linker ensures that all other applications use the distribution-specific shared library.

The manner in which the runtime linker is used by a binary varies from compiler to compiler. For example with gcc (GNU compiler collection), either the flag -Wl,–dynamic-linker=/lib/ld-lsb.so.2 can be passed to gcc, or alternatively the specifications ("specs") file can be modified to change /lib/ld-lsb.so.2 to be the default.

The name of the LSB linker is architecture-specific. For example, for the 32-bit PowerPC architecture, it is /lib/ld-lsb-ppc32.so.2, and for 64-bit PowerPC, it is /lib64/ld-lsb-ppc64.so.2. This allows for distributions

to support more than one LSB architecture simultaneously. Most commonly, this occurs when 64-bit

■ Statically linking a library into an application is the most straightforward method to preserve compliance ■

distributions also want to be 32-bit-compliant, but it leaves open the possibility for emulations of other architectures to be LSB-compliant as well.

Tools

Two programs have been developed to build compliant binaries, lsb-build-cc and lsb-build-chroot, each using a different strategy to achieve the same goal. They both require the same LSB header files and stub libraries, and these are supplied separately in a package called lsb-build-base. In order to build a compliant C++ binary, it is also necessary to use the lsb-build-c++ package.

These programs are not intended to be an integrated development environment (IDE), but instead are tools that can be used in conjunction with an IDE. All of the LSB build environment packages can be downloaded from http://www.linuxbase.org/download/#lsbdev.

The lsb-build-cc program

The lsb-build-cc strategy is to supply a program which is invoked instead of the gcc C compiler when compiling a program. This program, called lsbcc, supplies extra arguments and modifies existing arguments passed to it, and then invokes gcc itself, passing on these arguments. The modifications and additions are made so that gcc does not use the standard search path for header files and libraries, but instead uses a path which points to the LSB-compliant ones first, before searching the ones supplied by the distribution.

For build systems that honor the CC environment variable, just setting CC to lsbcc results in the binaries being built in an LSB-compliant way. It is important to have this variable set when running configure scripts because using lsbcc can change what functionality is visible to be used for building the binary.

A program called lsbc++ is also supplied with the lsb-build-cc package, which performs a job similar to that of lsbcc, except that it wraps the g++ program for building C++ programs.

The lsb-build-chroot program

The lsb-build-chroot package creates a chroot (i.e., a restricted environment) that programs can be compiled in. Inside the chroot, /usr/include contains LSB-compliant header files, and the stub libraries are contained inside /usr/lib. Also within this environment, the specs file for gcc is modified so that the LSB runtime linker is used in any binaries generated.

The environment can be configured to selectively incorporate parts of the host system into the restricted environment. Home directories of specified user accounts are automatically made available. An ssh (secure shell) daemon is configured to run within the chroot so it is possible to login to the environment. This removes the need for root access that would normally be required to enter the chroot.

The lsb-build-c++ package

The lsb-build-c++ package is required to build a C++ program. This package contains header files relevant to C++. Unlike the header files for other parts of the LSB specification, the C++ header files are not generated from the database, but instead are a snapshot from a specific version of gcc. The LSB database is currently unable to store information of the complexity required by C++.

LSB 1.3 versions of this package also contain a static libstdc++ library which has been compiled to be LSB-compliant. This was necessary as LSB 1.3 did not natively support C++, and it was necessary to statically link the runtime library. As C++ is part of the specifications for LSB 2.0 and above, a stub library of libstdc++ is included with lsb-build-base for the 2.0 versions of the build environment.

Comparing the lsb-build-cc and lsb-build-chroot programs

For most applications, when the build process is fairly straightforward and adheres to standard practice such as the use of the CC environment variable, lsb-build-cc is the simplest way to build a compliant application. It has been better tested and is more mature than lsb-build-chroot. However, there may be some circumstances where lsb-build-chroot is

preferable, as lsb-build-cc does not cope with situations where there are hard-coded references to compilers, headers, or library include paths. Configure scripts can be fairly smart in picking up header files from the system which may contain information that conflicts with LSB specification, and lsb-build-cc is not always able to detect this. The lsb-build-chroot program presents a sanitized environment, with non-compliant headers and libraries not visible unless explicitly imported.

When it is necessary to use a compiler other than gcc, some reengineering of lsbcc and lsbc++ is necessary. The amount of work required depends on how "command line compatible" the compiler is with gcc and binutils. In contrast, this should be a fairly straightforward process with lsb-build-chroot, and LSB-compliant headers and libraries are contained in /usr/include and /usr/lib.

For reasons of efficiency, lsb-build-chroot utilizes a Linux feature called "bind mounts," which allows one part of a file system to be mapped into another. It is not a copy, but rather the same content is made available in another part of the directory hierarchy, analogous to the hard linking of directories. Caution must be exercised because without properly understanding how bind mounts work, it is easy for someone with root privileges to accidentally remove important parts of the system such as /lib and /bin when cleaning out the build environment.

For many applications, initially, it can take a developer considerable time to configure an lsb-build-chroot system to import all of the non-LSB requirements into the build environment. Further development of the tool should alleviate this problem.

PACKAGING

The LSB specification gives two options for packaging an application. The first is a packaging format which is based on RPM** (Red Hat Package Manager). It is a subset of RPM Version 3 that removes some options, such as trigger scripts which are not implementable in the context of some package management systems.

All LSB-compliant implementations must be able to handle the installation of files for this LSB package format. It is important to note that the LSB specification does not require an implementation to supply the RPM program or use the RPM database. It merely has to be able to process the package, install the individual files, and run any installation scripts. For example, a Debian-based system could use the program alien to convert the file into a "deb" (a package in the Debian packaging format), which would then be installed using dpkg.

The LSB does not require an implementation to have a database to handle the management of files installed, nor any commands to perform an installation. These aspects are implementation specific. Strict requirements for the naming of a package were added to the specification to ensure that no clashes occurred. All packages must start with the lsb- prefix. If the name contains only one hyphen, then the name must be registered with the Linux Assigned Names and Numbering Authority (LANANA). LANANA is part of the Free Standards Organization, and contact information can be found at http://www.lanana.org. Alternatively, if more than one hyphen is used in the name, then the portion between the first and second hyphen must be either a LANANA registered name or a fully qualified domain name in lowercase, which is owned by the person or organization distributing the package.

The second option for someone distributing an application is to provide a package in a format of his or her choice, along with an installation program which is itself LSB-compliant. This program would handle unpacking the package, installing any files, and executing any installation scripts.

When an installation is done in this manner, it avoids the implementation's native package manager. This can remove or degrade the package manager's ability to implement some very useful features such as file collision detection and package integrity checking. Therefore, a decision not to use the LSB package format should be made only after serious consideration.

Package dependencies

An LSB-format package must have certain dependencies. An LSB-1.3-compliant package depends only on lsb. An LSB-2.0-compliant package must demonstrate through its package dependencies which LSB modules are required by the application. The specification lists the following possible dependencies: lsb-core-<arch>, lsb-graphics-<arch>,

lsb-core-noarch, and lsb-graphics-noarch, where <arch> indicates the LSB architecture name.

Applications that require the graphics libraries should depend on one of the lsb graphics dependencies; others simply depend on one of the lsb core dependencies. The noarch dependencies are provided for applications that depend only on the architecture-independent portions of the specification. All of these dependencies are provided by an LSB-compliant distribution.

For example, a 64-bit PowerPC graphics-based application would depend on lsb-graphics-ppc64, and a 64-bit zSeries non-graphics application would depend on lsb-core-s390x. A package that contains only shell scripts should depend on lsb-core-noarch.

An LSB package should never depend on distribution-specific dependencies. However, it is allowable for the package to depend on dependencies provided by other LSB-compliant packages.

FILE HIERARCHY STANDARD

The LSB specification references the File Hierarchy Standard (FHS), and all LSB-compliant applications must be FHS compliant. The following discussion is not an exhaustive coverage of the requirements to be FHS compliant, but highlights some of the important requirements and those that are specific when read in conjunction with the LSB specification.

All LSB-compliant applications are considered to be add-on application software packages, and as such should be installed into the /opt hierarchy (see FHS 2.3 Section 3.13). The provider or package name must be registered with LANANA. To ensure that there is no conflict of init or cron job script names, these must also be registered with LANANA.

Not all of the commands listed in the FHS are present on an LSB-compliant system. Because only those commands and utilities explicitly defined in the LSB specification are ensured to be in an compliant distribution, an application must not attempt to use any others.

With the introduction of the /media directory, FHS 2.3 clarified where removable media such as CD-ROM and floppy discs should be mounted in the system.

USING EXTERNAL LIBRARIES

Since the current coverage of the LSB specification is relatively small compared to what is shipped in a standard Linux distribution, some applications may require functionality outside of the specification. In most cases it is possible to use non-LSB library functionality in an application and still remain compliant.

Static Linking

Statically linking a library into an application is the most straightforward method to preserve compliance. The library itself must be LSB-compliant, which may require the inclusion of further libraries to be statically linked. This may not always work, as a library may use an interface which is not in the LSB but resides in a library which is part of the LSB. An example of this would be a library which calls the strfry library interface, which on a Linux system based on glibc (the GNU C Library) resides in libc.so, but is not included in the LSB specification. In these sorts of cases, or situations where it is undesirable to statically link even more libraries, it is necessary to build a version of the library which does not use this functionality. At the cost of reduced functionality, some libraries can be configured during build time to not use other libraries.

By default lsb-build-cc links a library statically (rather than dynamically) if the library is not part of the LSB specification. When using lsb-build-chroot, one should only import the static version of the library into the environment.

Supplying shared libraries

An alternative to statically linking a non-LSB library is to dynamically link against the library and include a copy of the shared library in the package. Where this method is used, the shared library itself needs to be LSB-compliant. Like the binary itself, it is allowable for that shared library to be dynamically linked against other non-LSB-required libraries, but those libraries must also be included with the application under the same restrictions as the first shared library.

As lsb-build-cc by default only statically links non-LSB-specified libraries, it is necessary to instruct it to dynamically link the library through the use of the LSBCC_SHAREDLIBS environment variable. The value of LSBCC_SHAREDLIBS must be a colon-separated list of library names to be dynamically linked. When lsb-

build-chroot is used, simply importing the shared library into the environment causes it to be dynamically linked.

MAKING THE XPAINT APPLICATION LSB COMPLIANT

The xpaint application was one of the applications chosen to be part of the LSB application battery (described at length in the following section). The xpaint application is a fairly simple drawing program based on X11 (the X windowing system for bitmap displays), and is a good example of an application that exercises the graphics components of the LSB specification.

Several source code changes were required to build a compliant binary, though some of these were general bug fixes necessary to support the broad range of architectures supported by the LSB. A common example of this involved parts of the application that were not written correctly for 64-bit architectures. For example:

```
static void
scrollCB(Widget w, LocalInfo * l, XtPointer position)
{
  float *percent = NULL;
...
  if (!percent) return;
  if ((int) position>0)
}    *percent += 1.0/256.0;
```

On 64-bit architectures the XtPointer parameter is a 64-bit value, and casting it to an integer causes errors at runtime. Similarly, some files had prototypes for common library functions such as malloc and free that were incorrect (for all architectures). These were removed, and header files were included with the correct declarations. In a few cases sys_errlist was used, but because this is not part of the LSB, the code was changed to use the strerror library interface, which is part of the specification.

The last class of changes were those necessary to make the application FHS-compliant. The xpaint application uses the xmkmf tool to do most of its configuration, which made it easy to make the changes required:

```
xmkmf -DBinDir="/opt/lsb-xpaint/bin" -DLibDir=
  "/opt/lsb-xpaint/lib" \
-DEtcX11Directory="/opt/lsb-xpaint/etc" -DManPath=
  "/opt/lsb-xpaint/man"
```

And on installation:

```
make BINDIR = "/opt/lsb-xpaint/bin" LIBDIR =
  "/opt/lsb-xpaint/lib"
ETCX11DIR = "/opt/lsb-xpaint/etc" MANPATH =
  "/opt/lsb-xpaint/man"
install install.man
```

Although not encountered when building xpaint, applications that use configure scripts to build often run into problems. This is because configure scripts often do not test directly for a feature being present, but the developer assumes that it should be present. For example, some scripts assume that if a system uses a certain version of glibc (or above), then IPv6 support will be present. LSB 1.3 does not support IPv6, but the configure scripts detected a certain version of glibc and so configured the build to include IPv6 support. Despite this, the compilation may fail if the required header information is missing. Application developers should ensure that the configuration scripts look for the exact functionality desired.

APPLICATION BATTERY

The application battery is a set of open-source applications (such as Samba, Apache**, and TCL [Tool Command Language]) which have been built by the LSB team such that they are compliant with the LSB standard. Together they utilize a significant percentage of the functionality offered by the LSB specification. They are used as part of the procedure for certifying a distribution and also serve as a useful test for the LSB build environments.

For an application developer, the battery is a useful resource of examples of problems encountered when porting an application and the methods used to fix them. The application battery subgroup does not release source packages, but the procedure used to build the applications is available in CVS (Concurrent Versions System). The information can be accessed anonymously with the following commands:

```
$ cvs -d:pserver:anonymous@cvs.gforge.
  freestandards.org/cvsroot/lsb login
$ cvs -z3 -d:pserver:anonymous@cvs.gforge.
  freestandards.org:/cvsroot/lsb co appbat
```

The application battery information is downloaded into a directory named appbat. Some tools from the

Automated Linux From Scratch[10] project are used to build the applications. The configuration files are written in XML (Extensible Markup Language) and are easy to understand. Also stored in the repository are the patches which are applied to applications to achieve compliance.

TESTING

There are a number of tools currently under development to check that an application conforms to the LSB specification. At the time of writing, only lsbappchk is mature enough to use in the certification program, although it is expected that the lsbdynchk program will soon be added to the certification process with lsbpkgchk to follow later. These tools are described in the following subsections.

The lsbappchk program

The lsbappchk program tests individual executables for conformance to the LSB specification. It tests that the application dynamically links only the permitted shared libraries, and from them, links only the specified functions and global data. When extra shared libraries are shipped with an application, the program can be informed of these, and it will allow linkage to them. It will perform the same tests on any of those shared libraries in the same manner as it does the executable. The object format of the executable and any associated libraries are also checked.

Examples

The following is an example of running lsbappchk on the cat utility, which is shipped as part of the distribution. Note that while lsappchk is not LSB-compliant, it is not always expected or possible for all applications that are shipped with the distribution be compliant. An LSB-compliant distribution must supply all of the interfaces required by the specification (e.g., runtime libraries and utilities), but they do not need to be implemented such that they themselves as applications are compliant.

```
cyeoh@rockhopper:~$ lsbappchk /bin/cat
lsbappchk for LSB Specification 2.0.1.20040718
Checking binary /bin/cat
Incorrect program interpreter: /lib/ld-linux.so.2
Header[ 1] PT_INTERP Failed
Found wrong intepreter in .interp section: /lib/ld-linux.so.2
   instead of: /lib/ld-lsb.so.2
Symbol_overflow used, but not part of LSB-Core
Symbol fputs_unlocked used, but not part of LSB-Core
```

In the preceding messages, lsbappchk has detected that the standard Linux runtime linker is being used instead of the LSB-specified one. This commonly happens when the LSB build tools are not used to link an executable. It has also detected that the binary uses functions_overflow and fputs_unlocked, which are not in the LSB specification. By making some small code changes and compiling within an LSB build environment, this code could be made compliant easily.

The lsbappchk tool automatically generates a journal file, which must be submitted during the certification process. The file is human-readable and a summary can be produced by using the tjreport tool. The format of the journal file is fundamentally the same as that produced by the Test Environment Toolkit[11] (TET), although it does not use TET.

The lsbappchk program outputs warning messages which do not necessarily affect certification. To remove any doubt as to what is a certification problem and what is just a warning, the journal file explicitly classifies test results as failures if they would block certification.

The following example would test the program foo as it is dynamically linked against the bar library, which is shipped with the application:

```
lsbappchk -L/opt/foo/bin/foo/bar.so.1 foo
```

The full path name of the shared library must be used, and where multiple libraries are specified, they are searched in order. Libraries may be specified multiple times where interlibrary cycles exist.

By default, the tool only checks against the LSB-core specification. The -M flag must be used for applications which require other LSB configurations:

```
lsbappchk -M LSB-Graphics xpaint
```

Alternatively, the -A flag can be used to include interfaces from all LSB modules.

The lsbdynchk program

The lsbdynchk program is a tool that measures the conformance of an application as it is running. Although the lsbappchk tool can verify that only LSB-specified interfaces are used, it is unable to verify that they are used correctly. Taking the example of lseek in the section "LSB build tools," lsbappchk is unable to

determine when lseek is called that the first parameter is a valid file descriptor, or that the third parameter is one of SEEK_SET, SEEK_CUR, or SEEK_END.

The dynamic checker intercepts the library calls and, where possible, checks that the values of the parameters are within the range allowable by the ABI. The tool relies on the application under test being exercised in a manner that covers as much of the logic as possible. While it is unlikely that all code paths will be tested in all situations, lsbdynchk can be useful in detecting problems.

The lsbpkgchk program

The aim of the lsbpkgchk tool is to verify that LSB applications packaged using the LSB package format are structured correctly. Fundamentally, the tool checks that the RPM format is correct and that features of RPM not supported by the LSB are not used. It also checks that the files within the package will be installed into areas consistent with the FHS.

CONCLUSIONS

The LSB specification is essentially a "trailing" standard. The LSB project avoids the invention of new technology, but instead documents and standardizes existing practice. It helps application developers communicate their requirements to the operating system developers. With the release of LSB 2.0, and the support of features such as C++, LSB compliance offers application writers a way to have their applications work well on a wide range of Linux platforms.

Further information about building LSB applications can be found at the LSB project site at http://www.linuxbase.org. The book *Building Applications with the Linux Standard Base* (published in October 2004) contains detailed information about the Linux Standard Base.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Institute of Electrical and Electronic Engineers, Inc., The Open Group, Linus Torvalds, Software In The Public Interest, Incorporated, Red Hat Incorporated, Sun Microsystems, Inc., or Apache Software Foundation.

CITED REFERENCES AND NOTES

1. "Linux Distributions - Facts and Figures," (November 2004), http://distrowatch.com/stats.php.
2. Linux Standard Base Project, http://www.linuxbase.org.
3. Free Standards Group, http://www.freestandards.org.
4. China Electronics Standardization Institute (CESI), http://www.cesi.ac.cn/.
5. "Solaris steals Linux's clothes," (November 17, 2004), http://www.computerweekly.com/articles/article.asp?liArticleID=135131&liArticleTypeID=1&liCategoryID=1&liChannelID=9&liFlavourID=1&sSearch=&nPage=1.
6. *System V Application Binary Interface, Edition 4.1*, The Santa Cruz Operation (1996), http://www.caldera.com/developers/devspecs/gabi41.pdf.
7. A. Josey, *Conflicts between ISO/IEC 9945 (POSIX) and the Linux Standard Base*, ISO/IEC Joint Technical Committee Technical Report (informative) (2003), http://www.jtc1.org/FTP/Public/JTC1/DOCREG/J1N71712.pdf.
8. The init scripts are the shell scripts bundled with applications which can start and stop the application. They are mostly used with applications which have server-based components and run during the booting or shutting-down phase of the operating system. They are also used to tell server-based applications to reread configuration files.
9. The cron job scripts are scripts run at defined times or intervals (e.g., once a day or at 1 AM every Thursday). They are often supplied as part of applications to do maintenance work.
10. Automated Linux From Scratch (ALFS)–News, http://www.linuxfromscratch.org/alfs/news.html.
11. TETworks: Home page for TETware, The Open Group, http://tetworks.opengroup.org.

GENERAL REFERENCES

Core Members of the Linux Standard Base Team, *Building Applications with the Linux Standard Base*, Prentice Hall, PTR, Upper Saddle River, NJ (October 2004).

Linux Standard Base Specification 2.0, Free Standards Group (2004), http://www.linuxbase.org/spec.

C. Yeoh, "Building LSB-compliant Applications," *Proceedings of the 9th International Linux System Technology Conference (Linux Kongress)*(September 2002), http://ozlabs.org/~cyeoh/presentations/blap-lk2002.pdf.

Standard for Information Technology, Portable Operating System Interface (POSIX), IEEE Std 1003.1–2003 (2003).

C. Yeoh, "Building LSB-compliant Applications," *Proceedings of the Linux.conf.au 2004*, http://ozlabs.org/~cyeoh/presentations/lca2004/index.html.

C. Yeoh, "Linux Standard Base," Projeto Software Livre Brasil 2004, http://ozlabs.org/~cyeoh/presentations/sl-2004/index.html.

Accepted for publication July 30, 2004.
Published online April 20, 2005.

Christopher Yeoh
IBM Server Group, 8 Brisbane Avenue, Canberra, ACT 2600, Australia (yeoh@au1.ibm.com) Christopher Yeoh has a B. Eng. degree in electrical and electronic engineering with first class honors and a B. Sc. degree in applied mathematics and computing science, both from the University of Adelaide. He has been using Linux since 1994 and became involved with the Linux Standard Base project in 2000, when he was working for Linuxcare. Mr. Yeoh joined IBM in 2001, at OzLabs in the IBM Linux Technology Center in Canberra,

Australia. He has been working on various aspects of the LSB, including test-suite development and the build environment, and is acting as the technical lead for the LSB build environment. He is also currently involved with K42, an open-source research operating system developed at the IBM Thomas J. Watson Research Center. Previously, he worked on the design and development of multidimensional graphical visualization and geographic-information-system products with real-time capabilities. ■