Linux Standard Base Development Kit for application building/porting

Rajesh Banginwar
Intel Corporation
rajesh.banginwar@intel.com

Nilesh Jain
Intel Corporation
nilesh.jain@intel.com

Abstract

The Linux Standard Base (LSB) specifies the binary interface between an application and a runtime environment. This paper discusses the LSB Development Kit (LDK) consisting of a build environment and associated tools to assist software developers in building/porting their applications to the LSB interface. Developers will be able to use the build environment on their development machines, catching the LSB porting issues early in the development cycle and reducing overall LSB conformance testing time and cost. Associated tools include application and package checkers to test for LSB conformance of application binaries and RPM packages.

This paper starts with the discussion of advantages the build environment provides by showing how it simplifies application development/porting for LSB conformance. With the availability of this additional build environment from LSB working group, the application developers will find the task of porting applications to LSB much easier. We use the standard Linux/Unix `chroot` utility to create a controlled environment to keep check of the API usage by the application during the build to ensure LSB conformance. After discussing the build environment implementation details, the paper briefly talks about the associated tools for validating binaries and RPM packages for LSB conformance. We conclude with a couple of case studies that demonstrate usage of the build environment as well as the associated tools described in the paper.

1 Linux Standard Base Overview

The Linux* Standard Base (LSB)[1] specifies the binary interface between an application and a runtime environment. The LSB Specification consists of a generic portion, gLSB, and an architecture-specific portion, archLSB. As the names suggest, gLSB contains everything that is common across all architectures, and archLSBs contain the things that are specific to each processor architecture, such as the machine instruction set and C library symbol versions.

As much as possible, the LSB builds on existing standards, including the Single UNIX Specification (SUS), which has evolved from POSIX, the System V Interface Definition (SVID), Itanium C++ ABI, and the System V Application Binary Interface (ABI). LSB adds the formal listing of what interfaces are available in which library as well as the data structures and constants associated with them.

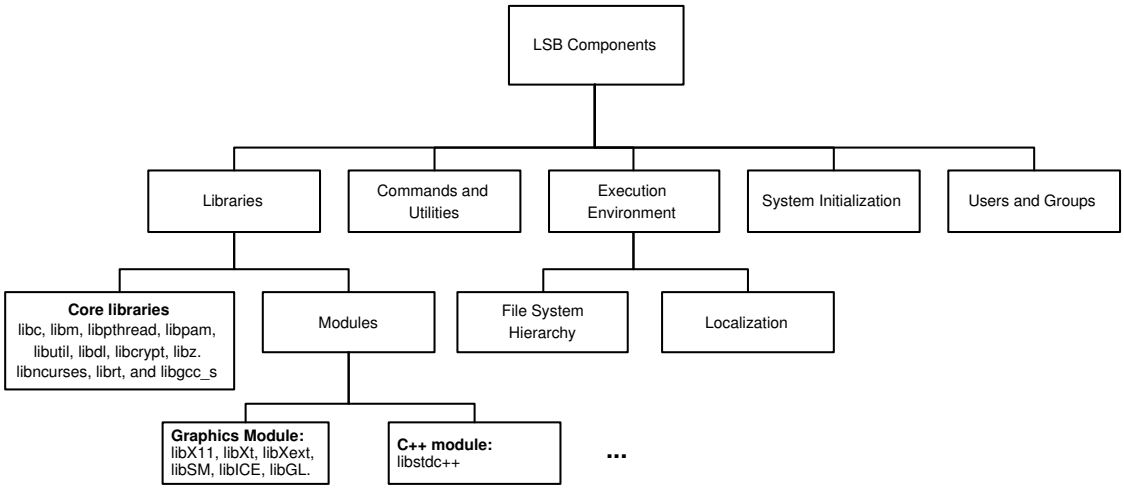

Figure 1: LSB Components

1.1 Components of LSB 3.0

Figure 1 shows the components of LSB 3.0 including the set of libraries covered in the specification. For applications to be LSB compliant, they are allowed to import only the specified symbols from these libraries. If application needs additional libraries, they either need to be statically linked or bundled as part of the application.

As the LSB expands its scope, future specification versions will include more libraries.

In addition to the Application Binary Interface (ABI) portion, the LSB specification also specifies a set of commands that may be used in scripts associated with the application. It also requires that applications follow the filesystem hierarchy standard (FHS)[7].

Another component of the LSB is the packaging format specification. The LSB specifies the package file format to be a subset of the RPM file format. While LSB does not specify that the operating system (OS) distribution has to be based on RPM, it needs to have a way to process a file in RPM format correctly.

All LSB compliant applications use a special program interpreter: `/lib/ld-lsb.so.3` for LSB version 3.0 instead of the traditional `/lib/ld-linux.so.2` for IA32 platforms. This program interpreter is executed first when an application is started, and is responsible for loading the rest of the program and shared libraries into the process address space. This provides the OS with a hook early in the process execution in case something special needs to be done for LSB to provide the correct runtime environment to the application. Generally, `/lib/ld-`*`arch`*`-lsb.so.3` or `/lib64/ld-`*`arch`*`-lsb.so.3` is used for other 32– or 64–bit architectures.

The next section discusses issues involved in porting/developing applications to LSB conformance along with the basic requirements for the same. The section ends with the overview of LSB development kit to help with the task. The subsequent sections discuss alternate standalone build environments and case studies showing real applications ported to LSB.

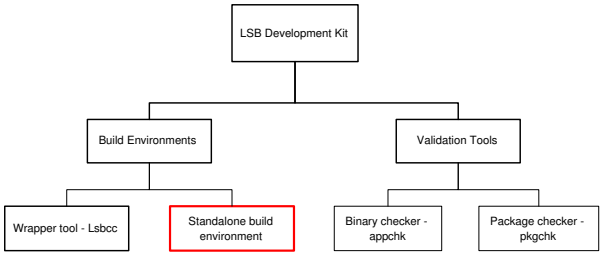

Figure 2: LSB Development Kit

2 Porting/Developing applications to LSB

This section starts the discussion with requirements for porting or developing applications to LSB. The application binaries will include executables and Dynamic Shared Object (DSO) files.

- Limit usage of DSOs to only LSB-specified libraries. Applications are also limited to import only LSB-specified symbols from those libraries.
- Use LSB-specific program interpreter `/lib/ld-lsb.so.3` for IA32 and `/lib/ld-`*`arch`*`-lsb.so.3` for other LSB-supported architectures.
- Use ELF as specified by LSB for created binaries.
- Use LSB-specified subset of RPM for application package.

For many application developers it may be a non-trivial task to port or develop applications to LSB. The LSB WG provides a development kit shown in Figure 2 to assist application developers in this task.

The LDK mainly consists of build environments to assist application developers with porting/development of applications to LSB and validation tools to verify for LSB conformance of application binaries and packages. LSB WG today has `lsbcc/lsbc++`—a `gcc/g++` wrapper tool which serves as a build environment as discussed in a subsection below. The second build environment which we are calling a standalone build environment is the topic of discussion for this paper. Before we discuss that build environment in detail, let's talk about the validation tools and existing build tools briefly.

2.1 Validation Tools in LDK

There are two validation tools delivered as part of LDK. These tools are to be used as part of LSB compliance testing for application binaries and packages.

1. `appchk`: This tool is used to validate ELF binaries (executables and DSOs) for their LSB conformance. This tool will work hand-in-hand with the build environment as discussed in the later sections of this paper. The LDK Case Studies section details the usage of this tool.
2. `pkgchk`: This tool new for LSB 3.0 is used for validating application packages. The tool makes sure that the package uses the LSB specified RPM file format. It also validates the installation aspect of the package for FHS conformance.

2.2 `lsbcc/lsbc++` – Existing build tool

In the last few years, the LSB WG has been providing a compiler wrapper, called `lsbcc` and `lsbc++`, as a build tool for application porting. `lsbcc` or `lsbc++` is used wherever build scripts use `gcc` or `g++` respectively. The wrapper tool parses all of the command line options

passed to it and rearranges them, inserting a few extra options to cause the LSB-supplied headers and libraries to be used ahead of the normal system libraries[6]. This tool also recognizes non-LSB libraries and forces them to be linked statically. Because the LSB-supplied headers and libraries are inserted into the head of the search paths, it is generally safe to use things not in the LSB.

With these simple steps many of the applications can be ported to LSB by simply replacing `gcc` with `lsbcc` and `g++` with `lsbc++`. In this method, the host environment is used for the build process; hence sometimes it may be difficult to reproduce the results on multiple platforms due to environment differences. This issue is not specific to the `lsbcc` wrapper build environment, but a common problem for many build systems. The build environment discussed in this paper addresses this issue by creating a standalone environment. Another shortcoming of the `lsbcc` approach is that the wrapper tools rely on the usage of `gcc` as compiler and configure-make process for application building. If the application relies on tools like `libtool` which modify the compiler command lines, `lsbcc` may not work correctly without additional configuration changes to produce LSB-compliant results. Similarly, usage of other compilers may not be possible as the wrapper tool relies on the command line option format used by `gcc`. For similar reasons, the tool may require additional configuration in certain customized build processes which may not rely on traditional config-make like build scripts.

3 LDK Standalone build environment

The standalone build environment is created using the standard Linux utility `chroot`. The isolated directory hierarchy is built from source packages and is completely independent of its host environment. With the development of this tool application developers will have a choice between the wrapper tool discussed above and the standalone build environment discussed here. From now on we refer to this standalone build environment as simply the build environment unless otherwise explicitly noted.

The concept of this build environment is derived from the Automated Linux from Scratch (ALFS)[2] project to create an isolated environment. The build environment comes with basic build tools and packages required for common application building. These tools are preconfigured so that the applications built produce LSB-conformant results. The application developer may add more tools/packages to this build environment as discussed later.

Since the application build happens in an isolated environment, except for some minor changes to Makefiles, the application developers do not need to change the build process. Since the whole mechanism is independent of the compiler as well as build scripts used, this build environment will work for most application development situations.

The build environment provides a set of clean headers and stub libraries for all the symbols included in the LSB specification. Applications are restricted to use only these symbols to achieve LSB conformance.

The build environment when used as documented will help produce the LSB-conformant application binaries. We recommend using the build environment from the beginning of application development cycle which will help catch any LSB porting issues early, reducing overall cost of LSB conformance testing.

The remainder of this section discusses the

build environment implementation in details. In addition to providing information on how it is used and accessed, the section also describes how the build tools are configured and/or updated.

3.1 Build environment Structure

Like a typical Linux distribution, the build environment has a directory hierarchy with `/bin`, `/lib`, `/usr`, and other related directories. Some of the differences between this build environment and a Linux distribution are the lack of Linux Kernel, most daemons, and an X server, etc. To start this build environment the developer will need root privileges on the host machine. The `lsb-buildenv` command used for starting the build environment behaves as follows:

```
Usage: lsb-buildenv -m [lsb|
nonlsb] -p [port] start|stop|
status
```

By default when used with no options, the environment will be configured for LSB-compliant building. The option `non-lsb` will force it to remain in normal build mode. This option typically is used for updating the build environment itself with additional packages/tools. The default sshd-port is set at 8989.

The `lsb-buildenv` command starts the `sshd` daemon at the specified port number. To access and use the build environment the user will need to `ssh` into the started build environment. By default, only the `root` account is created; the password is set to `lsbbuild123`. Once the user is logged into the build environment as `root`, he/she can add/update the user accounts needed for regular build processes.

```
$ ssh -p 8989 root@localhost
```

The build environment comes with the LSB WG-provided headers and stub libraries for all the LSB 3.0-specified libraries. These headers and stub libraries are located in the `/opt/lsb/include` and `/opt/lsb/lib` directories respectively. It is strongly recommended against modifying these directories.

X11 and OpenGL headers are exceptions to this and are located in `/usr/X11R6/include` although they are soft-linked in `/opt/lsb/include/X11`. These headers are taken from the Release 6 packages from X.org. The stub libraries related to all X libraries specified in LSB are located in `/opt/lsb/lib`.

3.1.1 Tools and Configuration updates

As discussed earlier the build environment is equipped with all the standard C/C++ build tools like `gcc` compiler suite, `binutils` package, etc. The goal for this build environment is to minimize the changes the application developer needs to make in the build scripts for the build to produce LSB-compliant results. The build tools are modified/configured to help produce LSB-conformant results as discussed below:

- **Compile time changes:** As discussed above, LSB provides a clean set of header files in the `/opt/lsb/include` directory. The `gcc` specs file is updated so that the compiler looks for this directory before continuing looking for other system locations. The string `-I /opt/lsb/include` is appended to the `*cpp_options` and `*cc1_options` sections in the `gcc` specs file.
- **Link time changes:**
  - By default the link editor (`ld` on most systems) is configured to look

in `/lib`, `/usr/lib`, and some other directories for DSO files. For the build to produce LSB-compliant results, we need to make sure the linking happens only with the LSB-provided stub libraries. For this, the default search path link editor uses to search for DSOs is changed to `/opt/lsb/lib` by configuring the `ld` build process at the time of creating/building this build environment. The `ld` is built with the following command: `./configure -with-lib-path=/opt/lsb/lib`

- Add `-L /opt/lsb/lib` to `*link` section of the `gcc` specs file to restrict the first directory accessed for libraries
- Remove `%D` from `*link_libgcc` section of `gcc` specs file. This will disallow `gcc` to add `-L` option for startup files.
- Set dynamic linker to `ld-lsb.so.3` by updating the `gcc` specs file by appending `*link` section with `%{!dynamic-linker: -dynamic-linker /lib/ld-lsb.so.3}`.

3.2 Packaging structure

The build environment comes with the most commonly needed packages pre-installed. Commonly used development (devel) packages are also pre-installed. As it is not possible to guess exactly what each application developer will need (since each build process is unique in requirements), the build environment comes with a populated RPM database to help the user add new packages as needed. This RPM database is built from scratch during the building of all the packages installed in the build environment. As no binary RPM is used for creating the build environment, Linux distribution-specific dependencies are avoided.

We use the CheckInstall [3] tool for populating RPM database in the build environment. This tool works by monitoring the steps taken by `make install` process and creates an RPM package which can then be installed. Please refer to the relevant reference listed in the Reference section for further documentation regarding this tool.

This RPM database may be used by the application developer if he/she needs to add/update a package required for a given build process. If for some reason (like dependency issues) a binary RPM cannot be installed, we suggest building and installing the package from source code by starting the build environment in non-lsb mode. Although not recommended, the user can always copy the relevant files manually into the build environment from the host machine.

4 Typical LSB porting process

This section discusses the process involved in porting the application to LSB. The subsection below discusses how LDK can be used during active development of application. Figure 3 shows the porting process in the form of a flow diagram.

- The first step is to run the existing application binaries through `appchk`. This will identify all the DSOs and symbols used by the application binaries that are not specified by LSB.
- The next step is to remove any unnecessary library dependencies where possible. Review all the makefiles (or similar scripts) to make sure the application is not

linking with any libraries that it does not need.

- If `appchk` is reporting that the application binary is dependent on a DSO not specified in LSB, there are two options to fix that:
  - The first option is to use static version of the library. This way the application will not depend on the concerned DSO.
  - If for some reason (licensing issues, etc.) that is not possible, the required functions will need to be implemented by the application developer avoiding the usage of that library or creating an application-specific DSO with those functions. When an application-specific DSO is created, it needs to be certified along with the application binary.
- For changing the usage of DSO to static library the Makefiles need to be updated manually. Remove `-l` options used during the linking phase for the concerned library. Include the corresponding static library in the linker command line.
- The next step is to perform `configure` and `make` (or similar scripts) as required by the application. Since the build environment is configured to use LSB-provided headers by default, the user may see some compilation errors. Typically these errors result due to usage of internal (although exported) or deprecated symbols. The developer will need to fix these by using the appropriate symbols for the given situation. The case study below shows one such situation. Another type of error occurs when a used symbol is not part of LSB although the concerned library is partially specified in LSB. The application developer needs to find alternatives to

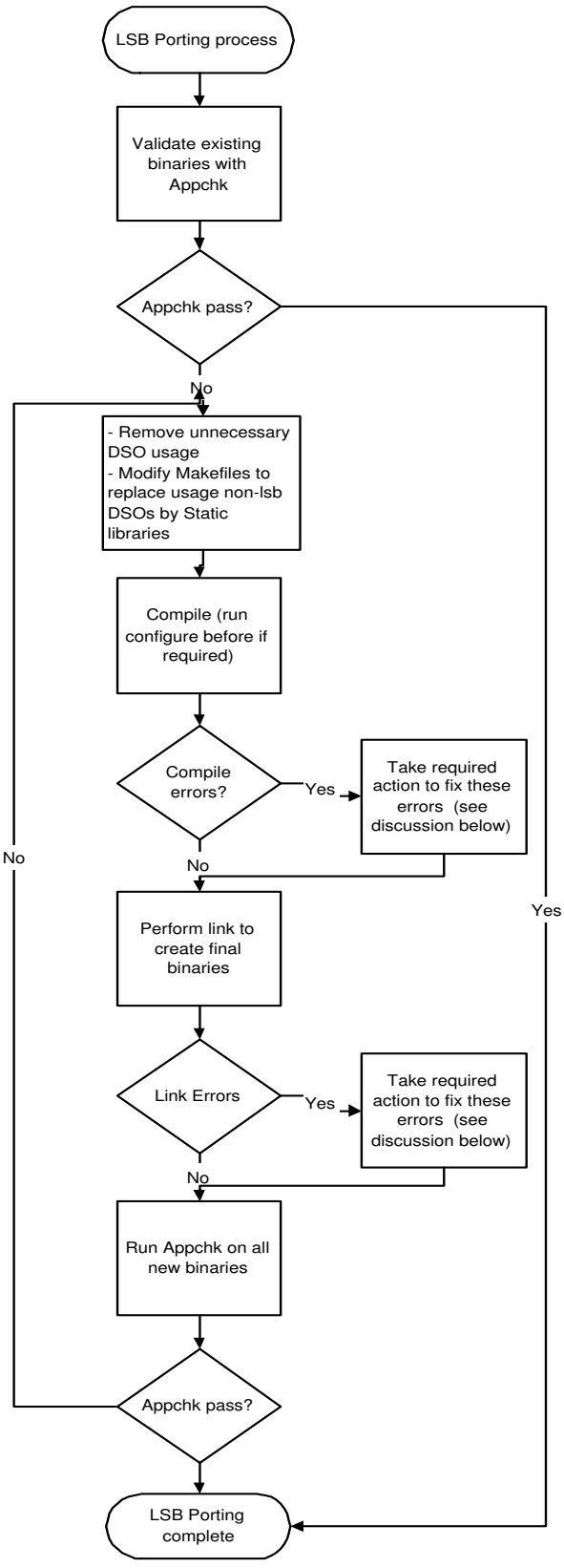

Figure 3: LSB porting process

such symbols that are covered by LSB or implement them as part of the application.

- The next step is linking to create final binaries for the application. If the Makefiles are correctly modified as discussed above, there should be minimal errors at this stage. The common error about "Symbol not defined" needs to be handled if certain deprecated or unspecified LSB symbols are used by the application and not caught in the compilation phase. Again the case studies below show couple of such examples.

4.1 LDK usage during application development

Other than porting the existing Linux applications to LSB, the build environment and the tools in LDK can be used by developers during the application development cycle. Regular or periodic usage of the build environment during the development cycle will help catch the LSB porting issues early in the development cycle, reducing overall LSB conformance testing time and cost. Such usage is highly recommended.

5 LDK Case Studies

This section discusses the real-life example of how LSB porting will work using this build environment. We consider two examples here to show different aspects of application porting. Since these examples are from the Open Source Software (OSS) projects they follow the optional `configure`, `make`, and `make install` model of building and installing software.

5.1 Example 1: ghostview 1.5

Ghostview[4] uses `xmkmf` to create the Makefile. When the application is built on a regular Linux machine, the `ldd` output for the ghostview binary is as follows:

```
$ ldd ghostview
libXaw.so.7 => /usr/X11R6/lib/libXaw.so.7
(0x00751000)
libXmu.so.6 => /usr/X11R6/lib/libXmu.so.6
(0x00b68000)
libXt.so.6 => /usr/X11R6/lib/libXt.so.6
(0x00af6000)
libSM.so.6 => /usr/X11R6/lib/libSM.so.6
(0x00ade000)
libICE.so.6 => /usr/X11R6/lib/libICE.so.6
(0x0024f000)
libXpm.so.4 => /usr/X11R6/lib/libXpm.so.4
(0x03c80000)
libXext.so.6 => /usr/X11R6/lib/libXext.so.6
(0x00522000)
libX11.so.6 => /usr/X11R6/lib/libX11.so.6
(0x00459000)
libm.so.6 => /lib/tls/libm.so.6 (0x0042e000)
libc.so.6 => /lib/tls/libc.so.6 (0x00303000)
libdl.so.2 => /lib/libdl.so.2 (0x00453000)
/lib/ld-linux.so.2 (0x002ea000)
```

Several of these libraries are not part of LSB yet and hence the application will not be LSB-compliant. To confirm that, run the `appchk` tool from LDK to find out exactly what is being used that is outside LSB's current specification:

```
$appchk -A ghostview
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.3
DT_NEEDED: libXaw.so.7 is used, but not part of the LSB
DT_NEEDED: libXmu.so.6 is used, but not part of the LSB
DT_NEEDED: libXpm.so.4 is used, but not part of the LSB
section .got.plt is not in the LSB
appchk for LSB Specification
Checking symbols in all modules
Checking binary ghostview
Symbol XawTextSetInsertionPoint used, but not part of LSB
Symbol XawTextReplace used, but not part of LSB
Symbol XmuInternAtom used, but not part of LSB
Symbol XawTextUnsetSelection used, but not part of LSB
Symbol XawScrollbarSetThumb used, but not part of LSB
Symbol XmuCopyISOLatin1Lowered used, but not part of LSB
Symbol XawTextDisableRedisplay used, but not part of LSB
Symbol XawFormDoLayout used, but not part of LSB
Symbol XawTextEnableRedisplay used, but not part of LSB
Symbol XmuMakeAtom used, but not part of LSB
Symbol XawTextGetSelectionPos used, but not part of LSB
Symbol XawTextInvalidate used, but not part of LSB
Symbol XawTextGetInsertionPoint used, but not part of LSB
```

The first message indicates the usage of `ld-linux.so` instead of `ld-lsb.so.3` as

dynamic linker. `DT_NEEDED` messages indicate the libraries which are not part of LSB specification but used by the application. The rest of the messages indicate symbols imported by the application but not specified in LSB.

Let's now look at how the build environment will help with porting this application to LSB and the steps users will need to go through in this process.

Step 1: Modify `Makefile` so that it does not use DSOs for the non-LSB libraries. Replace them with the static version of the libraries.

Step 2: Fix the compilation errors. In this case the errors included usage of symbols `sys_nerr` and `sys_errlist`. These are deprecated symbols and hence not part of LSB headers. The usage of these symbols is replaced by function `strerror`.

Step 3: Fix the link-time errors. In this case since the application uses three X libraries outside of LSB scope, we need to replace them with the corresponding static libraries.

After compilation and linking, we use `appchk` to check for LSB conformance for the created binary `ghostview`:

```
$ appchk -A ghostview
appchk for LSB Specification
Checking symbols in all modules
Checking binary ghostview
```

If we run ldd on this binary we will see:

```
$ ldd ghostview
libXt.so.6 => /usr/X11R6/lib/libXt.so.6
(0x00af6000)
libSM.so.6 => /usr/X11R6/lib/libSM.so.6
(0x00ade000)
libICE.so.6 => /usr/X11R6/lib/libICE.so.6
(0x0024f000)
libXext.so.6 => /usr/X11R6/lib/libXext.so.6
(0x00522000)
libX11.so.6 => /usr/X11R6/lib/libX11.so.6
(0x00459000)
libm.so.6 => /lib/tls/libm.so.6 (0x0042e000)
libc.so.6 => /lib/tls/libc.so.6 (0x00303000)
libdl.so.2 => /lib/libdl.so.2 (0x00453000)
/lib/ld-lsb.so.3 (0x002ea000)
```

All these libraries are part of LSB and the `appchk` confirms that the symbols imported by the binary ghostview are specified in LSB. This shows the successful porting of the application to LSB.

5.2 Example 2: lesstif package

Lesstif[5] is an implementation of OSF/Motif producing following binaries:

```
bin/mwm
bin/uil
bin/xmbind
lib/libDt.so*
lib/libDtPrint.so*
lib/libMrm.so*
lib/libUil.so*
lib/libXm.so*
```

By default none of these binaries is LSB-compatible. On a regular Linux machine, we get the following output when we run `ldd` and `appchk` on `mwm`.

```
$ ldd clients/Motif-2.1/mwm/.libs/mwm
libXm.so.2 => not found
libXp.so.6 => /usr/X11R6/lib/libXp.so.6
(0x0042e000)
libXt.so.6 => /usr/X11R6/lib/libXt.so.6
(0x00af6000)
libSM.so.6 => /usr/X11R6/lib/libSM.so.6
(0x00ade000)
libICE.so.6 => /usr/X11R6/lib/libICE.so.6
(0x0024f000)
libXext.so.6 => /usr/X11R6/lib/libXext.so.6
(0x00522000)
libX11.so.6 => /usr/X11R6/lib/libX11.so.6
(0x00459000)
libXft.so.2 => /usr/X11R6/lib/libXft.so.2
(0x00705000)
libXrender.so.1 =>
/usr/X11R6/lib/libXrender.so.1 (0x00747000)
libc.so.6 => /lib/tls/libc.so.6 (0x00303000)
libdl.so.2 => /lib/libdl.so.2 (0x00453000)
libfontconfig.so.1 =>
/usr/lib/libfontconfig.so.1 (0x006ad000)
libexpat.so.0 => /usr/lib/libexpat.so.0
(0x006e4000)
libfreetype.so.6 => /usr/lib/libfreetype.so.6
(0x0598c000)
/lib/ld-linux.so.2 (0x002ea000)
libz.so.1 => /usr/lib/libz.so.1 (0x00532000)
```

```
$ appchk -A clients/Motif-2.1/mwm/.libs/mwm
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.3
DT_NEEDED: libXm.so.2 is used, but not part of the LSB
DT_NEEDED: libXp.so.6 is used, but not part of the LSB
DT_NEEDED: libXft.so.2 is used, but not part of the LSB
DT_NEEDED: libXrender.so.1 is used, but not part of the LSB
section .got.plt is not in the LSB
appchk for LSB Specification
Checking symbols in all modules
Checking binary clients/Motif-2.1/mwm/.libs/mwm
Symbol XmGetXmDisplay used, but not part of LSB
Symbol XmGetPixmapByDepth used, but not part of LSB
Symbol _XmMicroSleep used, but not part of LSB
Symbol XpmReadFileToImage used, but not part of LSB
Symbol _XmFontListCreateDefault used, but not part of LSB
Symbol XmeWarning used, but not part of LSB
Symbol XmRegisterConverters used, but not part of LSB
Symbol XmStringCreateSimple used, but not part of LSB
Symbol _XmAddBackgroundToColorCache used, but not part of LSB
Symbol _XmGetColors used, but not part of LSB
Symbol _XmSleep used, but not part of LSB
Symbol _XmBackgroundColorDefault used, but not part of LSB
Symbol _XmFontListGetDefaultFont used, but not part of LSB
Symbol XmStringFree used, but not part of LSB
Symbol XmCreateQuestionDialog used, but not part of LSB
Symbol XmMessageBoxGetChild used, but not part of LSB
Symbol _XmAccessColorData used, but not part of LSB
```

As explained in the previous case study, these messages indicate the usage of libraries and symbols not specified in LSB.

This package follows the typical OSS build process of `configure`, `make`, and `make install`. All the makefiles are generated at the end of configure step. What makes this package an interesting exercise is the usage of `libtool`. This tool is used for portability in the usage and creation of DSO and static libraries.

Let's now walk through the process of building this package for LSB conformance.

Step 1: Modify `Makefile` so that it does not use DSOs for the non-LSB libraries. Replace them with the static version of the libraries.

Step 2: There are no compilation errors observed for this package.

Step 3: The first linktime error we see is about the undefined reference to some of the `_Xt` functions. These functions exported from `libXt.so` are not part of the LSB specification even though most of the other functions coming from the same library are covered. In this case the reason for this exclusion happens to be the nature of these functions. Most of these are internal functions and not really meant to be used by applications. The workaround for this will be to use a static version of the library instead of DSO. All the makefiles using `libXt.so` are modified for this.

The next error we see is the usage of function `_XInitImageFuncPtrs`. This function is deprecated and private (although exported). The suggested function in this case is `XImageInit`. Make the required change in file `ImageCache.c`.

After the compilation and linking we use `appchk` to check for LSB conformance for the created binaries. The output is shown below:

```
$ appchk -A -L lib/Xm-2.1/.libs/libXm.so.2 -L
lib/Mrm-2.1 /.libs/\ libMrm.so.2 ?L lib/Uil-2
.1/.libs/libUil.so.2 clients/Motif-2.1/mwm/\
 .libs/mwm
appchk for LSB Specification
Checking symbols in all modules
Adding symbols for library lib/Xm-2.1/.libs/
libXm.so.2
Adding symbols for library lib/Mrm-2.1/.libs/
libMrm.so.2
Adding symbols for library lib/Uil-2.1/.libs/
libUil.so.2
Checking binary clients/Motif-2.1/mwm/.libs/mwm
```

This shows the successful porting of lesstif to LSB.

6 Future Directions for LDK

For the LSB Development Kit, we will continue to make the tools better and easier to use for application developers. As the LDK is maintained actively through the LSB Working Group, ongoing feedback will be included in the future development and active participation in the tools development is strongly encouraged.

One of the features we are actively considering is the integration of the LDK with Eclipse or similar IDE. Another area under consideration is a tool to help develop/create LSB conformance packages.

We would like to take this opportunity to encourage all application developers to use the tools discussed in this paper and provide feedback and feature requests to the LSB mailing lists. We strongly encourage ISV participation in this process and solicit their feedback on the available tools as well as LSB in general.

7 Acknowledgments

We sincerely thank Free Standards Group and its members for providing support to LSB project. We would also like to extend our thanks to a core group of LSB developers including Stuart Anderson, Marvin Heffler, Gordon McFadden, and especially Mats Wichmann for their patience and support during the development of the LDK project.

References

[1] Linux Standard Base at `http://www.linuxbase.org/`

[2] Automated Linux From Scratch project at `http://www.linuxfromscratch.org/alfs/`

[3] CheckInstall utility at `http://asic-linux.com.mx/~izto/checkinstall`

[4] ghostview at `http://www.gnu.org/software/ghostview/ghostview.html`

[5] lesstif at `http://www.lesstif.org/`

[6] `lsbcc` usage and details at `http://www.linuxjournal.com/article/7067`

[7] File hierarchy standard at `http://www.pathname.com/fhs/`

8 Legal

Proceedings of the Linux Symposium

Volume One

July 20nd–23th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Gerrit Huizenga, *IBM*
Matthew Wilcox, *HP*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Matt Domsch, *Dell*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*