



Tutorial

Using the LSB to Increase Application Portability

***Mats Wichmann
Intel Corporation
mats.d.wichmann@intel.com***



Objectives

- Understand the LSB Model for applications
- Introduce the LSB Build tools
- Learn how to configure, build and check LSB programs and libraries
- Answer Lots of Questions

Contents

- Introduction and LSB Overview
- The Basics: Installing and using the LSB Tools
- LSB Tools in Practical Use
 - Analysis of tool output
 - Configuring and building with tools
 - Static linking and using extra shared libraries
 - C++
- Summary, Thoughts, Resources



LSB Tutorial

Introduction and LSB Overview



Binary Compatibility

- The LSB describes a Runtime Environment for applications
- A *contract* between the runtime and the application:
 - The runtime will provide a particular environment in the form of specified libraries, interfaces and commands
 - The application will use only the features of that environment
 - It will avoid libraries, functions and commands for which there is no specification

Quick Starter

- LSB Tools Install

- Yum repo (example `/etc/yum.repos.d/lsb.repo`):

- `[lsb-devel]`
`name=LSB Development`
`baseurl=http://ftp.freestandards.org/pub/lsb/snapshots/repositories`
`/yum/repo-ia32`
`enabled=1`
`Gpgcheck=1`

- `# yum install lsb-task-sdk lsb-task-app-testkit`

- Debian repo (`/etc/apt/sources.list`):

- `deb http://ftp.freestandards.org/pub/lsb/snapshots/repositories/ \`
`debian lsb main`

- `# aptitude install lsb-task-sdk lsb-task-app-testkit`

Common Complaint

- *"LSB Development Tools Are Too Hard to Figure Out"*

- Let's see...

- ```
$ cat hello.c
#include <stdio.h>
#include <unistd.h>
main()
{
 printf("hello world: %d\n", getpid());
}
$ gcc hello.c -o hello-1
$./hello-1
hello world: 14250

$ lsbcc hello.c -o hello-2
$./hello-2
hello world: 14260
```

- That was difficult...

# The LSB Specification

- The LSB specification presents a stable runtime environment for applications
  - Required shared libraries, including
    - A description of the behavior of public interfaces
    - Symbol versioning as needed (e.g. glibc interfaces)
  - An LSB dynamic linker
  - Guidelines on packaging and install locations
  - Commands needed for installation and startup

# LSB Components

- In addition to the written specification, the LSB Project also produces:
  - Test tools to measure compliance
  - Development tools to help build applications
  - Example runtime
  - Example applications

# Runtime LSB Compliance

- For a *runtime*, compliance means providing the functionality to run a compliant application
  - Does not mean commands are LSB conforming
  - Does not restrict the runtime from providing additional libraries, commands, etc.
  - Even possible to implement on a non-Linux system
    - For example, in a VM
- Indicate compliance by running the tests plus the example applications
- *The Linux Foundation* info on LSB certification info at <http://www.linux-foundation.org/en/Certification>

***All major Linux OS flavors are LSB compliant***

# Application LSB Compliance

- For an *application*, compliance means able to run on a compliant system
  - Use only LSB specified interfaces and libraries if possible
    - Statically link with non-LSB specified libs
    - Bundle non-LSB shared libs with the application
  - Link against LSB runtime linker
  - Follow guidelines for installation
  - Prove compliance by passing LSB Application Checker and testing on two LSB systems and LSB SI

***Very few applications are certified as LSB compliant***

# LSB Application Certification

- LSB Certification is also available for applications
  - Run compliance tests
  - Complete FHS checklist
  - Package in LSB format
- Two other (informal) models for applications:
  - Self-certified - follow the model, but don't submit to certification authority
  - "Requires LSB"
    - Require an LSB runtime, but also uses extra stuff not covered by LSB
    - Please document extensions - good data for future LSB directions!
    - Useful to limit the scope of "differences", LSB forms common base



# LSB Tutorial

*Installing and Using the LSB Tools*



# Tools

- How to make sure the LSB build rules are followed?
  - The spec is lengthy – don't try to memorize it!
    - By reference, includes many shelf-feet of specs
  - Existing projects would have to retrofit build rules
    - And those changes would not apply to non-LSB builds
- The answer: tools
  - Toolkits to control compile and link (two available)
  - LSB-clean stub libraries and headers
  - Application checker
  - **New:** LSB Application Testkit Manager (atk-manager)
    - web-based wrapper around lsbappchk
    - links results to LSB Developer Network (LDN) for help

# Build Toolkits

- Compiler wrappers for C and C++
  - Restricted build in standard environment
  - Better for builds that require non-LSB features at build time
    - In particular, for GNU-style configure scripts
- Chroot environment
  - Isolated environment in a chroot
  - Explicitly import any non-LSB commands, libraries
    - More control over what's picked up by builds
  - Better for self-contained builds

# Installing the Tools

- You need the following packages
  - `lsb-appchk`, `lsb-build-base`, `lsb-build-chroot`, `lsb-build-cc`, `lsb-build-c++`
  - **May want:** `lsb-build-desktop`, `lsb-build-qt3`, `lsb-build-qt4`
- Available several different ways
  - individual pkgs
  - tarball-with-installer-script ("sdk")
  - Possibly from distro-provided pkgs
  - repo "task" package `lsb-task-sdk`
- System should already have lsb support installed
  - try, for example (on an rpm-based system):  

```
$ rpm -q --whatprovides lsb
```

# A Trivial Program

- Let's see the application checker in action:
  - Hello world (straight up, with a twist):

```
$ cat hello.c
#include <stdio.h>
#include <unistd.h>
main()
{
 printf("hello world: %d\n", getpid());
}
$ gcc hello.o -o hello-1
$./hello-1
hello world: 14250
```

# Checking the Program

```
- $ lsbappchk hello-1
lsbappchk for LSB Specification 3.0.4
Checking binary hello-1
Incorrect program interpreter: /lib/ld-linux.so.2
Header[1] PT_INTERP Failed
Found wrong interpreter in .interp section: /lib/ld-
linux.so.2 instead of: /lib/ld-lsb.so.3
```

- That's not an LSB binary...



# The Compiler Wrapper

- lsbcc / lsbcc++ wrappers are invoked as if they were the compiler:
  - Fiddle with arguments, then invoke real compiler
  - Make sure stub libraries are used for linking
  - Make sure all other libraries are linked statically
  - Use LSB headers in preference to system headers
  - Try:

```
lsbcc --lsb-verbose hello.c
```

# lsbcc Configuration

- The wrappers are “configured” through environment variables
  - LSBCC - name of C compiler to call, if it isn't `cc`
  - LSBCCXX - name of C++ compiler to call, if it isn't `c++`
    - Provide a way to use a different version or alternate compiler, e.g.  
`LSBCC=gcc-3.4` or `LSBCC=icc`
  - LSBCC\_SHAREDLIBS - additional dynamic libs
    - Otherwise statically links all libraries not in the LSB
    - Use with care!
  - LSBCC\_DEBUG - value selects tracing options
  - LSBCC\_WARN - value selects warning options
- As of LSB 3.1, also command-line options
  - Take the form `--lsb-foo`

# Example using lsbcc

- Build, check, run:

```
$ lsbcc hello.c -o hello-2
$ lsbappchk hello-2
lsbappchk for LSB Specification 3.0.4
Checking binary hello-2
$./hello-2
hello world: 14986
```

# The lsb-buildenv chroot

- A chroot providing a controlled, isolated build environment
  - Build toolchain: gcc, binutils, etc
    - gcc config adjusted to give sensible results without calling separate command (i.e. lsbcc)
  - Required libraries
  - LSB stub libraries and header files
  - A copy of RPM for packaging
  - Start and stop scripts
  - An ssh setup to enter the chroot
- Just install the lsb-buildenv package
  - and Run the startup script

# Non-LSB Code

- What if we “break” the source from an LSB viewpoint?
  - `$ readelf -s /lib/libc.so.6 | grep getpid`  
000bb160 g DF .text 00000032 GLIBC\_2.0 \_\_getpid  
000bb160 w DF .text 00000032 GLIBC\_2.0 getpid
  - Change the code to use the internal `__getpid` in `hello2.c`:
  - `$ cat hello2.c`  
#include <stdio.h>  
#include <unistd.h>  
main()  
{  
 printf("hello world: %d\n", \_\_getpid());  
}
  - `$ cc hello2.c`  
`$ ./a.out`  
hello world: 11185

# Non-LSB Code (cont'd)

- Still a working program natively, but has an additional LSB conformance problem - a non-spec interface:
  - `$ lsbappchk a.out`  
lsbappchk for LSB Specification 3.0.4  
Checking binary a.out  
Incorrect program interpreter:  
/lib/ld-linux.so.2  
Header[ 1] PT\_INTERP Failed  
Found wrong interpreter in .interp section:  
/lib/ld-linux.so.2 instead of:  
/lib/ld-lsb.so.3  
Symbol `__getpid` used, but not part of LSB

# Compiling non-LSB Code With Tools

- The stub libraries will prevent linking this program:
  - `$ lsbcc hello2.c`  
`/tmp/ccqrdLCg.o: In function `main':`  
`/tmp/ccqrdLCg.o(.text+0x17): undefined reference to`  
``__getpid``  
`collect2: ld returned 1 exit status`
- And so does the chroot:
  - `chroot> cc hello2.c`  
`/tmp/ccyWFmlG.o: In function `main':`  
`/tmp/ccyWFmlG.o(.text+0x17): undefined reference to`  
``__getpid``  
`collect2: ld returned 1 exit status`

# C++ Builds

- C++ code building is not much different -
  - Builds with c++ won't follow the LSB rules, use lsbc++ instead

```
• $ cat hellow.cc
#include <iostream>
int main() {
 std::cout << "hello world" << std::endl;
 return 0;
}
$ g++ hellow.cc -o hellow-1
$./hellow-1
hello world
```

# C++ with lsbcpp

- **\$ lsbappchk hellow-1**

```
lsbappchk for LSB Specification 3.0.4
```

```
Checking binary hellow-1
```

```
Incorrect program interpreter: /lib/ld-linux.so.2
```

```
Header[1] PT_INTERP Failed
```

```
GNU_EH_FRAME not checked!
```

```
Found wrong interpreter in .interp section: /lib/ld-linux.so.2 instead of: /lib/ld-lsb.so.3
```

```
$ lsbcpp hellow.cc -o hellow-2
```

```
$./hellow-2
```

```
hello world
```

```
$ lsbappchk hellow-2
```

```
lsbappchk for LSB Specification 3.0.4
```

```
Checking binary hellow-2
```

```
GNU_EH_FRAME not checked!
```

# The Isbsi

- The Sample Implementation (Isbsi) is a minimal runtime
  - A Proof of Concept of easily building an LSB system
  - Tries to include only what's required by LSB spec
  - Built using the Linux-from-scratch (LFS) model
    - see <http://www.linuxfromscratch.org>)
  - Useful for catching non-LSB usage
    - If it's not in the LSB, the Isbsi (probably) doesn't have it!
  - The Isbsi (plus a kernel and a few extra bits) can be booted standalone or in a virtual machine, or can be run hosted as a chroot or uml.

# Installing the lsbsi

- The easiest way to use the lsbsi is the user mode Linux (uml) form – lsb-umlsi (ia32 only for now)
- Alternatively, the lsbsi can be used as a chroot by unpacking three tarballs as root (substitute desired architecture):
  - `# tar -xjf lsbsi-core-arch-2.0.3.tar.bz2`
  - `# cd lsbsi-core-arch`
  - `# tar -xjf lsbsi-graphics-arch-2.0.3.tar.bz2`
  - `# tar -xjf lsbsi-test-arch-2.0.3.tar.bz2`

# Starting the uml lsbsi

- Configuration questions asked on first startup
- Example startup (after config questions):
  - `# /opt/lsb-uml-si/bin/uml-si`  
various boot messages  
log in as *root*, password *lsbsi123*
  - Adding users/groups is a one-time activity (substitute the desired user/group):  
`# groupadd -g 500 mats`  
`# useradd -u 500 -g 500 mats`  
`# passwd mats`
  - "hostfs" is used to mount host's root on `/mnt/disk`:  
`lsbsi:/root# mount | grep hostfs`  
`none on /mnt/disk type hostfs (rw)`

# Umlsi Configuration

- Further configuration of umlsi is possible
  - Some changes affect the way the umlsi is started: edit startup script `/opt/lsb-umlsi/bin/umlsi`
    - For example, uml defaults to using xterm. If you don't want this, follow examples to change
    - Also can do networking setup here
  - Other changes are made in the umlsi's filesystem, and affect the way it boots
    - For example, the number of windows to start is set in `/etc/inittab`

# Starting the chroot lsbsi

- All chroot setup is manual
    - Example assumes you installed as `/opt/lsbsi`
    - If you want to telnet into the lsbsi, make sure your host isn't running a telnetd (including via inetd/xinetd)
    - Adding users/groups is a one-time activity (substitute the desired user/group)
    - ```
# mount -o bind /home /opt/lsbsi/home
# chroot /opt/lsbsi
# mount /proc
# groupadd -g 500 mats
# useradd -u 500 -g 500 mats
# passwd mats
# inetd
```
- and from another window:
- ```
$ telnet localhost
```
- (and log in)

# Testing in the lsbsi

- Example of testing in the lsbsi
  - assumes the uml lsbsi (adjust as needed):
    - \$ `cd /mnt/disk/home/mats/src`
    - \$ `./hello-2`  
hello world: 88
    - \$ `./hello-3`  
hello world: 89
    - \$ `./hello-1`  
-bash: ./hello-1: No such file or directory
  - Notice the lsbsi won't run the non-LSB binary!

# Summary – Part 1

- In this section, we have
  - Discussed the LSB binary compatibility “contract”
  - Used lsbappchk to examine binaries
  - Installed and configured both lsb build toolkits
  - Seen how the toolkits help build conforming binaries
  - Set up the sample implementation for testing



# LSB Tutorial

*LSB Tools in Practical Use*



# Part 2 Contents

- In this part of the Tutorial, we will see the tools in practical use:
  - Analysis, configuring and building with tools
  - Packaging issues
  - Static linking and using extra shared libraries
  - C++ issues

# Examining a binary

- Let's examine the rsync program as it appears on the system (no attempt at an LSB build):

- `$ lsbappchk /usr/bin/rsync`

```
lsbappchk for LSB Specification 2.0.4
```

```
Checking binary /usr/bin/rsync
```

```
Incorrect program interpreter: /lib/ld-linux.so.2
```

```
Header[1] PT_INTERP Failed
```

```
Found wrong interpreter in .interp section: /lib/ld-
linux.so.2 instead of: /lib/ld-lsb.so.2
```

```
DT_NEEDED: libpopt.so.0 is used, but not part of the LSB
```

```
DT_NEEDED: libresolv.so.2 is used, but not part of the LSB
```

```
Symbol poptGetContext used, but not part of LSB
```

```
...
```

```
Symbol getpass used, but not part of LSB
```

```
Symbol mallinfo used, but not part of LSB
```

```
Symbol glob64 has version GLIBC_2.2 expecting GLIBC_2.1
```

# Analysis

- How many of these are problems?
  - Wrong dynamic linker
    - Fix by linking correctly (e.g. Lsbcc)
  - Two non-LSB libraries are used
    - Need to be eliminated or worked around
  - Several non-LSB interfaces are used
    - Some come from the non-LSB libs, some seem to come from LSB libraries
  - One symbol version issue
- Note:
  - This is a hybrid example for illustration purposes (that is, I cheat)

# Building LSB

- Building an LSB application is easy:
  - Use only approved libraries and interfaces
  - Link it correctly
- Or not so easy:
  - Existing code probably uses other libraries and non-LSB interfaces
  - Existing build frameworks are not easy to retrofit with new build rules

# LSB Symbol Versions

- For libraries where symbols are versioned, such as libc, the LSB describes a specific version
  - The only reason for introducing a new version is an incompatible change, so versioning is a compatibility aid
  - The LSB compatibility contract promises the LSB version will continue be supplied even if new versions are added

# Symbol Versioning

- Let's examine some symbols:

```

• $ readelf -s /lib/libc.so.6 | grep glob64
 405: 000a0f30 3378 FUNC GLOBAL DEFAULT 11
glob64@GLIBC_2.1
 2084: 0009f9c0 3378 FUNC GLOBAL DEFAULT 11
glob64@@GLIBC_2.2
$ readelf -s /opt/lsbdev-base/lib/libc.so | \
grep glob64
 837: 00007798 5 FUNC GLOBAL DEFAULT 6
glob64@@GLIBC_2.1
 842: 00007798 5 FUNC GLOBAL DEFAULT 6
glob64

```

- Version GLIBC\_2.1 is the LSB version, and is found in glibc. When the LSB build tools are used, that version will be linked, and it exists at runtime – no problem.

# LSB Tools

- Portable code must:
  - Either be written to a minimal (least common denominator) function set
  - Or written to a richer set, but with conditional code
- The LSB build tools are designed to help
  - The chroot environment allows building with the standard toolchain, just enforces the rules
  - The lsbcc wrapper should drop in as a replacement for the compiler, if the build is designed properly (e.g. uses CC macro instead of built-in compiler name)

# Rsync configure

- Let's see if the configure script will do the right thing:
- `$ CC=lsbcc ./configure`  
configure: Configuring rsync 2.5.5  
... *(selected items only)*  
checking for gcc... lsbcc  
checking for inet\_ntop in -lresolv... yes  
checking for mallinfo... no  
checking for poptGetContext in -lpopt... yes
- This looks fairly okay
  - Configure tests for libresolv and libpopt worked, so it must be correctly static linking these libraries
  - A configure test detected that mallinfo doesn't exist (in the LSB environment) so it won't be used

# Rsync build

- At build time, however, some problems appear:

- \$ **make**

```
isbcc -I. -I. -g -O2 -DHAVE_CONFIG_H -Wall -W -c
clientname.c -o clientname.o
clientname.c: In function `client_sockaddr':
clientname.c:120: warning: implicit declaration of function
`IN6_IS_ADDR_V4MAPPED'
clientname.c:120: dereferencing pointer to incomplete type
clientname.c:127: storage size of `sin6' isn't known
clientname.c:127: warning: unused variable `sin6'
clientname.c: In function `compare_addrinfo_sockaddr':
clientname.c:213: dereferencing pointer to incomplete type
```

- These look like problems. . .

# Rsync problems

- Let's check the source:
  - See `clientname.c` lines 120, 127, 213
  - All are bracketed in `#ifdef INET6`
  - LSB 1.3 didn't support IPV6 (added in 2.0)
- Now check the configure script:
  - See configure lines 3070 and then 836
  - It's always on for "Linux" but we can configure it off with `--disable-ipv6`
  - This is an example of a questionable configure choice in the source, better to feature test for the capability than to make decisions based purely on OS type

# Rebuilding

- Redo the configure and rebuild:

- `$ make`

```
. . .
lsbcc -g -O2 -DHAVE_CONFIG_H -Wall -W -o rsync rsync.o . .
. -lpopt -lresolv
authenticate.o: In function `auth_client':
/home/mats/lwe2003/rsync-2.5.5/authenticate.c:278: undefined
reference to `getpass'
collect2: ld returned 1 exit status
make: *** [rsync] Error 1
```

- `getpass` is not an LSB interface (remember *appchk* *did* warn us), and was not checked for or worked around by `configure`

# Patching rsync

- The LSB project wrote a patch to add a `getpassword` routine and call it – it seemed the most expedient hack change

- Note there's more in the patch – we'll see another bit soon

- ```
$ make distclean
```
 - ```
$ patch -p1 < rsync-2.5.5.patch
```
  - ```
$ ./configure --disable-ipv6
```
 - ```
$./make
```

# Checking the result

- If that builds correctly, we need to check and test
  - \$ **lsbappchk rsync**  
lsbappchk for LSB Specification 2.0.4  
Checking binary rsync
  - \$ **mkdir /tmp/foo**
  - \$ **./rsync -avuzt . /tmp/foo**  
building file list ... done  
./  
.cvsignore  
...
- So far so good. Still need to do a thorough test
  - Also remember to test in the lsbsi

# Packaging

- LSB packaging rules:
  - Use rpm package format (more later)
  - Follow the FHS for file locations
    - LANANA registered package names or providers
  - Use only LSB commands in install, startup scripts
    - For example, can use POSIX shell, but not Perl, unless you supply LSB-conforming Perl with your package

# RPM

- The LSB specifies rpm package format
  - Not the rpm command!
  - All systems required to install, conversion is acceptable
  - Do not depend on an rpm package database
    - Only count on the lsb dependency
- For the Application Battery, the LSB project uses rpm to package only
  - Can build with rpm, but it's work to modify the rpm build rules the right way
  - Optimization flags and other machine and distribution specifics may get in the way
  - The expectation that you can rebuild from srpm may not hold unless the right macro files are in place

# Rebuild for Packaging

- For rsync we can handle the FHS issues mostly via the configure option *prefix*
- One patch is needed to a header (rsync.h line 29)(already in the patch we applied)

```
-#define RSYNCD_CONF "/etc/rsyncd.conf"
+#define RSYNCD_CONF "/etc/opt/lsb-rsync/rsyncd.conf"
```

- Configuration now needs these flags:

```
$ CC=lsbcc ./configure --disable-ipv6 \
--prefix=/opt/lsb-rsync
```

# “Installing”

- We then build, and install to a temporary location:
  - `$ make`
  - `$ make install prefix=/var/tmp/build`
- Now we can use a simple rpm spec file to package
  - Needs: name, version, etc.
  - Only %files really matters beyond the header
  - %pre, %install, etc. can be empty
  - May be better to leave empty rules out entirely
- An example on the next page. . .

# rsync spec file (cont'd)

- Summary: A program for synchronizing files

```

Name: lsb-rsync
Version: 2.5.5
Release: 3
Vendor: The Linux Foundation
License: GPL
Group: Appbat/file transfer
Buildroot: /home/mats/LSB/appbat/pkgroot/lsb-rsync
AutoReqProv: no
Requires: lsb-core >= 2.0
%description
LSB conforming version of rsync

%files
%attr (- bin bin) /opt/lsb-rsync
%attr (- bin bin) /etc/opt/lsb-rsync
%attr (- bin bin) /var/opt/lsb-rsync
```

# LSB Application Battery Building

- The LSB project currently uses a separate build tool for the application battery called nALFS
  - xml-style files capture the build steps – see example
  - Packaging is done separately using simple rpm spec files such as the one just shown
  - This removes differences in rpm setup, and in rpm versions, from our build process
  - But it's a little cumbersome as examples...

# nALFS xml driver fragment

```
<package>
 <name>rsync</name>
 <version>&rsync-version;</version>
 <prebuild>
 <unpack> . . . </unpack>
 <patch> . . . </patch>
 <setenv> . . . </setenv>
 <configure>
 <base>&build_dir;/&rsync-directory;
 </base>
 <param>--disable-ipv6</param>
 <param>--prefix=/opt/lsb-rsync</param>
 </configure>
 </prebuild>
 <build> . . . </build>
 <postbuild> . . . </postbuild>
</package>
```

# RPM Requirement: Revisited

- While rpm format packages are preferred, there are other ways to deliver software
- The real rule is, an LSB conforming system must be able to install the package
  - It's required to if you use the rpm package format
  - Install can also be done using LSB-required commands, such as shell scripts + tar
  - Or can supply an installer with the package
    - Of course, it must be LSB-conforming itself

# Static Libraries

- The LSB model depends on dynamic linking
  - Link dynamically with libraries in the LSB spec
  - and with shared libraries built LSB conforming
- All other libraries must be linked statically
  - lsbcc handles this automatically
    - all `-lfoo` arguments are turned into `-static -lfoo`
    - Use `LSBCC_WARN=1` to report these changes
  - For chroot, must import the static lib into the chroot via the chroot's config file

# Dynamic Libraries

- Non-LSB shared libs are allowed
  - Must compile them LSB conforming
  - Must not count on them to exist on the target machine, which means:
    - Bundle them with the application or
    - Provide in another LSB conforming package, which can become a dependency
- lsbcc won't link these dynamic unless you tell it
  - lsbcc takes a library list e.g.: `LSBCC_SHAREDLIBS=tcl:tk`

# Building an LSB Shared Library

- The following steps illustrate building a working LSB shared library:

```
$ cat liblsb.c
#include <stdio.h>
void
lsbfunc() {
 printf ("Hello from lsblib\n");
}
$ lsbcc -c -o liblsb.o liblsb.c
```

- For extra fun, let's add symbol and library versioning:

```
$ cat liblsb.Version
LSBLIB 1.1 { lsbfunc ; } ;
$ lsbcc -shared -Wl,-soname=liblsb.so.2 \
-Wl,--version-script=liblsb.Version \
-o liblsb.so liblsb.o
```

# Linking with LSB Lib

- ```
$ cat hello3.c
#include <stdio.h>
void lsbfunc(void);
main() {
    printf ("Calling liblsb: ");
    lsbfunc(); }
$ lsbcc -c -o hello3.o hello3.c
$ lsbcc -o hello3-1 hello3.o -L. -llsb
/usr/bin/ld: cannot find -llsb
collect2: ld returned 1 exit status
```
- **lsbcc doesn't find the shared version, but it finds a static copy if one exists:**

```
$ ar crv liblsb.a liblsb.o
a - liblsb.o
$ lsbcc -o hello3-1 hello3.o -L. -llsb
$ ./hello3-1
Calling liblsb: Hello from lsblib
```

Linking with LSB Lib

- We need to tell lsbcc about the new shared library before it will link with it
- ```
$ LSBCC_SHAREDLIBS=lsb lsbcc -o hello3 \
hello3.o -L. -llsb
$./hello3
./hello3: error while loading shared libraries:
liblsb.so.2: cannot open shared object file: No such file
or directory
```
- Hmm, it was found at link time, but not at run time
  - Normally, the dynamic linker doesn't include the current directory
    - Security issue: it would matter where you ran a program from
  - FHS says we should put add-on software in `/opt`
    - We'll use `/opt/lsb` for this example

# Linking with LSB Lib

- `$ LSBCC_SHAREDLIBS=lsb lsbcc -Wl,-rpath,\`  
`/opt/lsb/lib -o hello3 hello3.o -L. -llsb`  
`$ readelf -d hello3`

Dynamic segment at offset 0x61c contains 23 entries:

Tag	Type	Name/Value
0x00000001	(NEEDED)	Shared library: [liblsb.so.2]
0x00000001	(NEEDED)	Shared library: [libm.so.6]
0x00000001	(NEEDED)	Shared library: [libc.so.6]
0x0000000f	(RPATH)	Library rpath: [/opt/lsb/lib]

- Now the binary contains information where to look, so we can copy over the library
  - Note we made liblsb.so but it wants liblsb.so.2 from the soname option earlier, so make a link:
- `# cp liblsb.so /opt/lsb/lib`  
`# ln -s /opt/lsb/lib/liblsb.so \`  
`/opt/lsb/lib/liblsb.so.2`  
`$ ./hello3`  
Calling liblsb: Hello from lsblib

# Checking with LSB Lib

- Let's make sure we made an LSB binary:
- `$ lsbappchk hello3`  
lsbappchk for LSB Specification 2.0.4  
Checking binary hello3  
`DT_NEEDED: liblsb.so.2 is used, but not part of the LSB`  
`Symbol lsbfunc used, but not part of LSB`
- lsbappchk also knows what the LSB libraries are
  - We have to tell it where to find additional shared libraries (note it wants the path to the file, not the directory)
- `$ lsbappchk -L/opt/lsb/lib/liblsb.so.2 hello3`  
lsbappchk for LSB Specification 1.3.3  
Adding symbols for library /opt/lsb/lib/liblsb.so.2  
Checking binary hello3

# LSB Library Summary

- Example intentionally used a lot of steps, but:
  - Build the library with `lsbccc`
  - Use `LSBCC_SHAREDLIBS` when linking to extend `lsbccc`'s idea of allowed shared libraries
  - Not allowed to install in “standard” places (`/lib`, `/usr/lib`), so need to tell binaries when linking (`-Wl,-rpath`)
  - To test, install the library by hand
    - If setting `soname` in the library, need to make the symbolic link to that name (normally done by `ldconfig`, but `ldconfig` is not part of the LSB)
  - `lsbappchk` needs to be passed the path to extra shared libs or it will report those as errors



# LSB Tutorial

*Summary : Thoughts : Resources*



# Summary

- In this Tutorial, we have examined:
  - The LSB application model
  - LSB tools for building applications
  - Techniques for “porting” an LSB application
  - Some information on LSB shared libraries
- Next steps will lead to more questions
  - Use the resources (in a few slides)

# LSB Project

- The LSB Project is about open standards
  - Built by the community, for the community
  - All the tools shown are free to use. modify, distribute
  - The spec is free to use and implement
- Please let us know if something is broken or crucial features are missing
  - Use lsb bugzilla or one of the mailing lists
  - You may be invited to help with the solution!

# What is the LSB Good For?

- When you can't or don't want to ship source
  - Closed source programs
- When source is hard or impractical:
  - User doesn't want to spent time on a source build
  - User has incomplete or no build environment, or wrong toolchain versions
  - Target has wrong versions of key packages
    - Remember, lsb=x.y covers it all
  - You only build on Blue Fish, user has White Cloud
  - It's too much trouble e.g. a push install on a grid
- And as a “porting base” to get you part of the way there...

# Legal

- Disclaimer:
  - The views expressed are those of the author and do not necessarily represent the position of Intel Corporation
  - Any brands and trademarks are the property of their respective owners



# Resources

- **Everything LSB:** [www.linux-foundation.org/en/LSB](http://www.linux-foundation.org/en/LSB)
- **Mailing lists linked off** [www.linux-foundation.org](http://www.linux-foundation.org)
  - **lsb-discuss** is the one to join
- **Bugs/requests to lsb project bugzilla:** [bugs.linuxbase.org](http://bugs.linuxbase.org)
- **Project repository:** [bazaar.linux-foundation.org/lsb](http://bazaar.linux-foundation.org/lsb)
- **Downloads:** [www.linux-foundation.org/en/Downloads](http://www.linux-foundation.org/en/Downloads)
- **FHS checklist:**  
[www.linux-foundation.org/test/fhschecklist.html](http://www.linux-foundation.org/test/fhschecklist.html)
- **Certification:** [www.linux-foundation.org/en/Certification](http://www.linux-foundation.org/en/Certification)
- **LANANA:** [www.lanana.org](http://www.lanana.org)
- **IRC:** [irc.freestandards.org](http://irc.freestandards.org) #lsb

# Contributing

- Requests in bugzilla, lsb-discuss list
- Comment on IRC channel
  - Virtual sprint (aka bug triage/discussion) on Fridays
- Use LSB Navigator
  - Soon we'll have a way to scan and upload app data to the DB
- Come to LSB BOF Friday
- Join the confcall (details in lsb-discuss ML archives)
- Come to an LSB meeting
- Or just join up and start contributing

# Backup

## Further Info



# What is LSB?

- An application portability standard for Linux
  - Application focused, doesn't talk about kernel
  - Describes core functionality, not a full system
    - Innovation beyond the baseline is encouraged
- An open source project
  - Operates as a workgroup of FSG
- An ISO standard

# Main Concepts

- Application Execution
  - ELF, libraries, interfaces, symbol versions
- Application Environment / Installation
  - commands, packaging

# Structure

- Reference existing standards where possible
- Describe "delta" where needed
- LSB contains the whole spec only where there is no other reference
- Document runtime requirements (ABI vs API)
- Generic part + arch-specific supplements

# Modular

- **Base**: core, graphics, c++
- **Desktop**: gtk, qt, graphics-ext, xml
  - lsb-desktop released in June 2006
- Module separation is practical (& political) more than functional
  - no effect on users
    - c++ separate module from core
    - qt separate module from gtk
    - graphics-ext represents “future directions” for graphics

# How it works: Libraries

- Runtime name (soname)
- Required symbols
- Symbol versions if used by library
- Behavior is defined by spec, not impl.
  - Tries to avoid requiring a specific upstream pkg. (there may be several choices) or version
  - Should be possible to write an implementation using only the spec

# How it Works: Applications

- Use what's in LSB, or solve availability problem
- Runtime linker ld.so must be LSB-specified name (e.g. /lib/ld-lsb.so.3)

# How it Works: Runtimes

- Provide a set of libraries that implement the requirements
  - Can be regular system libraries...
  - or one or more can be special to LSB
- Use runtime linker as a hook if need special libs or any other special processing
  - Most systems don't do anything special, LSB linker is just a symbolic link

# How it Works: Building

- Link against “stub” libraries
  - Contain right symbols and versions
  - Never used at runtime, just for symbol binding
- Two modes for building:
  - Wrapper around regular compiler, inserts path to stubs and makes sure LSB linker is used
    - Works with gcc, also tested with recent icc
  - Complete chroot build jail
    - Specific frozen versions of tools

# How it Works: Packaging

- Install to FHS-approved paths
- Use LANANA-approved names
- Use a slightly restricted rpm format
  - Basically omits things alien can't convert (triggers)
  - LSB is now the spec for the rpm file format
  - No requirement on runtimes to have rpm
- Depend on LSB-described meta-dependency
  - lsb=3.1
  - No need to worry about what packages make that up on any given distro

# More Packaging

- Other package formats allowed as long as “installer availability” problem is solved
  - e.g. A shell script and tar is cool as both commands are required by LSB anyway
  - Custom installer, if provided, is okay
- Discouraged, life is easier for admins if the system package manager is used
- Packaging is an area where there is still work to be done

# Tools for Apps:

- lsbcc, buildenv
  - lsbcc designed to allow LSB builds using the system toolchain: `do CC=lsbcc` (or `CXX=lsbc++`)
    - For configure: `CC=lsbcc ./configure` should work
  - buildenv chroot allows a self-contained build, doesn't use native system's toolchain
  - Both systems reduce need for “LSB-aware” modifications to existing build setups
- lsbappchk, lsbpkgchk
- dynchk (someday, if someone will finish it)

# LSB-Desktop

- Modules to support graphical/desktop apps
  - More work coming for LSB 3.2
- Four modules +1 opt, add these libraries:
  - **Gtk**: glib-2.0 gthread-2.0 gobject-2.0 gmodule-2.0 atk-1.0 pango-1.0 pangoxft-1.0 pangoft2-1.0 gdk-x11-2.0 gdk\_pixbuf\_xlib-2.0 gdk\_pixbuf-2.0 gtk-x11-2.0
  - **XML**: xml2
  - **Graphics-ext**: jpeg png12 fontconfig
  - **Qt3**: qt-mt
  - (optional Qt4) **Qt**: QtCore QtGui QtNetwork QtXml QtOpenGL QtSql QtSvg

# Tools for LSB-Desktop

- Natural extension to LSB 3.0 tools
  - Install extra build packages `lsb-build-desktop` and `lsb-build-qt3`
  - No extra options needed to build or check
- Only optional qt4 requires special options
  - `LSB_MODULES=qt4 lsbcc`
  - `lsbappchk -M LSB_Toolkit_Qt`

