CHAPTER TWELVE

Using the Sample Implementation

The LSB Sample Implementation (LSB-si) is a minimal LSB-conformant runtime environment used for testing purposes. Think of it as a tiny Linux distribution that you have complete control over. It can be booted as a standalone system, hosted inside a virtual machine, or run in a change-root mode. It is particularly convenient to use hosted mode as a way to enable an LSB test environment to exist on a system without disrupting the base system, thus reducing the need for multiple dedicated LSB test machines.

LSB-compliant applications should be tested inside the LSB-si to insure they haven't picked up any distribution-specific quirks. The LSB Certification program, as outlined in Chapter 7, requires that an application be tested under the LSB-si.

The LSB-si can also be used in a variety of validation scenarios where there's a desire to limit the installed environment to the minimal set required by the LSB.

12.1 Understanding the Sample Implementation

The LSB-si is an example runtime environment implementation. It is the accumulation of elements that are both explicitly required and implied by the LSB Written Specification, plus any necessary dependencies to make the runtime environment work.

The LSB-si has three main purposes. First, it is an example implementation to demonstrate the features and correctness of the LSB Written Specification. Second, it serves as an example of building an LSB-conformant system from current open source packages without a lot of special patching. Thirdly, the LSB-si is a testing sandbox to help evaluate the LSB compliance of an application. If an application fails because it requires something not found in the LSB-si, then this is an opportunity to trim or rework the dependencies of an application to help better conform to the LSB.

The LSB-si is a *sample implementation*, not a *reference implementation*. The difference is that a sample implementation demonstrates one way to conform to the specification, while a reference implementation serves as a baseline for conformance validation. In a reference implementation scenario, other implementors are required to be compatible with all of the reference implementation, including all the features and bugs unintentionally present in it. In addition, a reference implementation constrains the evolution of other implementations because even legitimate enhancements or fixes would deviate from the reference, thus breaking adherence.

The LSB is an ABI specification designed to capture the functional requirements of a conforming system without locking in any specific implementation, versions of software packages, and so on, as being the standard. By specifying the behavior of required elements and avoiding a reference implementation, the LSB permits conforming runtime environments and applications to evolve within the ABI confines of the Written Specification.

12.2 Setting Up the Sample Implementation

The LSB-si can be run in one of three different modes: the `chroot` LSB-si, the UML LSB-si, and the bootable Knoppix LSB-si. Software for each mode can be downloaded from the LSB Download Web site.[1]

1. `http://www.linuxbase.org/download/`

12.2.1 Sample Implementation Using `chroot(1)`

The `chroot` LSB-si is distributed as a compressed tarball that should simply be unpacked by the superuser (Example 12.1). Later, other pieces can be layered on top of it, and packages for additional functionality can be installed inside it. For illustrative purposes, the IA32 (x86) architecture will be used in the following examples. For your situation, you may need to substitute the appropriate LSB architecture and version.

Example 12.1: Installing LSB-si on IA32 (x86) Architecture

```
# mkdir /opt/lsbsi-ia32
# cd /opt
# tar jxvf /tmp/lsbsi-ia32-2.0.1.tar.bz2
```

The LSB-si can be started by running the `/sbin/chroot` command (Example 12.2; some distributions place this command in `/usr/sbin`).

Example 12.2: Starting LSB-si by the `/sbin/chroot` Command

```
# /sbin/chroot /opt/lsbsi-ia32
```

To simulate a login session, use this command as shown in Example 12.3.

Example 12.3: Login Session

```
# /sbin/chroot /opt/lsbsi-ia32 /bin/bash -l
```

When working in the `chroot`, only files inside the `chroot` tree will be accessible. Since this limits the usefulness of the environment, some steps should be taken ahead of time to make the necessary files available. One approach is to simply copy files from the `chroot` into a suitable location. If another window or terminal session is available on the host system, it is also possible to copy files into the `chroot` tree the same way, whenever they are needed (Example 12.4).

Example 12.4: Copying Files into `chroot`

```
# cp mypackage.rpm /opt/lsbsi-ia32/tmp
```

Another approach is to establish bind mounts before starting the `chroot`. These bind mounts enable portions of the filesystem of the host to appear to be inside of the `chroot`. Bind-mounting a user's home directory to the same location in the `chroot` will make that user's files appear in the same path inside the `chroot` as on the host system. Example 12.5 shows setting up access to the files in the home directory of `username`.

Example 12.5: Bind Mounts

```
# mkdir /opt/lsbsi-ia32/home/username
# mount -o bind /home/username /opt/lsbsi-ia32/home/username
```

This does not cause the user account itself to exist in the `chroot`. The **useradd** command should be used to add the account. The account should be given the same `userid` as it has on the host system.

The LSB-si limits itself to commands and features required by the LSB Written Specification, which is a set chosen to be able to minimally install, configure, and administer installed software. This has some testing benefits, but this also means there likely are a few missed components. These are not hard to work around, but the LSB-si philosophy leaves them out; by having to install them specifically, you become aware of these areas that are actually not standard across systems.

The **rpm** tool needs access to `/proc` to calculate disk space (the mounted filesystem table is found through `/proc`). However, `/proc` is not part of the LSB Written Specification. The simplest workaround is to add a line describing it to the `/etc/fstab` of the LSB-si so that you can simply say `mount /proc`. Example 12.6 shows such a line, through it is best to simply copy the line from `/etc/fstab` on the host system.

Example 12.6: Adding `/proc` to `fstab`

```
none  /proc  proc  defaults  0 0
```

Depending on your requirements, you may also need to perform similar steps for other pseudo filesystems such as `/dev/shm` and `/dev/pts`.

Some of the commands you may be accustomed to may be missing, including editors and development tools. Many developers have found it useful to work in a windowed environment, with one window running the LSB-si `chroot` and others set up to be able to edit and otherwise modify files as necessary.

Example 12.7 illustrates copying an installable package into the `chroot` tree, starting the `chroot`, installing the package, and running it. As this example will utilize the X Server on the host system, an additional setup step to enable access to this server is included before the `chroot` is started.

Example 12.7: Installing a Package on `chroot`

```
# cp lsb-xpaint-2.6.2-3.i486.rpm /opt/lsbsi-ia32/tmp
# xhost +localhost

# /sbin/chroot /opt/lsbsi-ia32
# cd /tmp
# rpm -i lsb-xpaint-2.6.2-3.i486.rpm
# rpm -qa
# export DISPLAY=localhost:0
# /opt/lsb-xpaint/bin/xpaint
```

For networked applications, some additional support is probably needed. For example, the `/etc/services` file is not described by the LSB, and thus is not present in the `chroot`. If needed, simply copy it over from the host system.

Also, if a server (daemon) is to be tested, it will first need to be set up to listen on the appropriate port; secondly, any conflicting service on the host

system must be disabled so the connection actually reaches the daemon in the LSB-si `chroot`.

The base LSB-si does not have the **inetd** or **xinetd** servers, so by default these cannot be used to listen for service requests. The `lsbsi-`*`arch`*`test` package can be installed to provide **inetd**. Example 12.8 shows a normal run of `lsb-runtime-test` to validate the LSB-si. The `lsb-test` package is previously installed. One part of testing requires the **syslog** daemon, so the `syslogd` from the host system is shut down first. The test setup requires a true login, so **inetd** is started to watch over a **telnet** daemon.

Example 12.8: Using the `lsb-test` Package to Run **inetd**

```
# sh /etc/init.d/syslog stop
# /usr/sbin/chroot /opt/lsbsi-ia32
# mount /proc
# /sbin/syslogd -m 0
# /usr/sbin/inetd
```

Now connect to the LSB-si `chroot` using telnet, and log in as a user defined inside the LSB-si (Example 12.9).

Example 12.9: Connecting to `chroot`

```
$ telnet localhost
```

12.2.2 Sample Implementation Using UML

The UML LSB-si runs off a *root filesystem* that is encapsulated in a file. The LSB-si will not be able to "see" outside this filesystem except if running `hostfs`, in which case the filesystem tree of the host can be mounted and accessed. This is the easiest way to get files, such as packages, to be installed into the UML LSB-si.

It is also possible to copy files into the `root_fs` (root filesystem file) of the UML LSB-si. This is a little tricky, so proceed carefully (Example 12.10).

Example 12.10: Copy Files into `root_fs`

```
# mkdir /mnt/lsbsi
# losetup /dev/loop0 /opt/lsb-umlsi/lib/root_fs
# mount /dev/loop0 /mnt/lsbsi
# cp lsb-lynx-2.8.4-1.i486.rpm /mnt/lsbsi/tmp
# umount /mnt/lsbsi
# losetup -d /dev/loop0
```

The UML LSB-si is started by running the User-Mode Linux kernel; a script is provided by the installable (RPM or Debian) packages to do this with appropriate arguments. If it has not been run previously, the `Configure` script will be run now to set up important parts of the hosted UML Sample Implementation (Example 12.11).

Example 12.11: Running the `Configure` Script

```
# /opt/lsb-umlsi/bin/umlsi
```

An advantage of using the UML is that it allows the whole system, kernel and all, to be run in user space. This enables the testing and debugging of an LSB-supporting kernel without actually installing a new kernel on the host system. In addition, UML more stringently isolates the LSB-si from the hosting system.

12.2.3 Using a Bootable Sample Implementation

The base tarball plus the `lsbsi-`*`arch`*`-boot` tarball are nearly enough for a complete standalone bootable system. The two missing pieces are a Linux kernel and a bootloader setup, plus some final configuration needs to be done manually. Examples 12.12 to 12.15 show the steps to build a bootable LSB-si partition on an already-bootable system. You will need to augment further to put these on a "fresh" system with no existing Linux installation.

Locate an unused partition, such as `/dev/sda9`, and make an appropriate filesystem on it (the `-j` option to **mke2fs** is used to create a journaling (ext3) filesystem). This partition will now contain most of a runnable system. There

are a few more things that need to be done to it, such as building the correct filesystem mount table and setting up a kernel that works appropriately for this system.

Example 12.12: Creating a Journaling Filesystem

```
# mke2fs -j /dev/sda9
# mkdir /mnt/lsbsi
# mount empty-partition /mnt/lsbsi
# cd /mnt
# tar -xvjf /tmp/lsbsi-ia32-2.0.0.tar.bz2
# cd lsbsi
# tar -xvjf /tmp/lsbsi-test-ia32-2.0.0.tar.bz2
```

1. Handcraft an appropriate `/etc/fstab` that describes the root filesystem, swap (if any), and any additional mounts, including special filesystems such as `/proc`, `/dev/pts`, and so on.
2. Copy over the `modules` directory matching a bootable kernel that already works on this system (Example 12.13).

Example 12.13: Copying the `modules` Directory

```
# cp -r /lib/modules/2.4.18-24.8.0 /mnt/lsbsi/lib/modules
# cp /boot/vmlinuz-2.4.18-24.8.0 /mnt/lsbsi/boot
```

3. Copy over a working `/etc/modules.conf` or similar, as appropriate for the system you will be running (Example 12.14).

Example 12.14: Copying `/etc/modules.conf`

```
# cp /etc/modules.conf /mnt/lsbsi/etc
```

4. Add instructions to boot the kernel of your choice using the appropriate partition to boot to. For example, to boot using **lilo**, you might add the contents of Example 12.15 to `/etc/lilo.conf`.

Example 12.15: `/etc/lilo.conf` Instructions to Boot to **lilo**

```
image=/boot/vmlinuz-2.4.18-24.8.0
        label=lsbsi-boot
        root=/dev/sda9
        initrd=/boot/initrd-2.4.18-24.8.0.img
        read-only
```

If you're using a bootloader such as **lilo** that requires running a command to install the new configuration, don't forget to do so!

12.2.4 Sample Implementation Using Knoppix

Knoppix[2] is a bootable CD that runs GNU/Linux in memory without disturbing the operating system installed on the hard disk. The LSB has a version of Knoppix with the LSB-si configured on it. This disk is useful for demonstrating the LSB-si or for a tutorial lab use. When used for a demo or in a tutorial lab, there should be no concerns about what kernel, compiler, or **glibc** libraries are installed. Almost everyone can simply insert the disk into their computer and reboot to run Knoppix LSB-si.

12.3 Application Testing Using the Sample Implementation

The LSB-si is equipped with a copy of **rpm**, to meet the requirement that a conformant platform be able to install RPM packages of the specified format. Note that **rpm** itself is not required by the LSB Written Specification. The RPM database in the LSB-si is populated with a minimal set of `provides`. As required by the specification, it provides a versioned **lsb** package dependency that matches the LSB Written Specification version, that is, `lsb=1.3` for LSB version 1.3, and `lsb=2.0` for LSB version 2.0. The LSB Written Specification permits, but does not require, an implementation of a later version to also support an earlier version, but the LSB-si does not attempt multiple-version compatibility.

2. `http://www.knopper.net/`

The LSB project has developed a package verification tool **lsbpkgchk** that should be run and whose output should be examined carefully. As of this writing, there were still some false negatives—warnings or errors that would not prevent a package from installing correctly (it appears that there is no version of **rpm** that 100% follows the rules in the LSB Written Specification). See the **lsbpkgchk** release notes for more information.

Packages need to be installed with normal dependency checking. Do not resort to installing with `rpm --nodeps`, as one of the purposes of testing in the LSB-si is to show that a given package can properly be installed by a strictly conforming implementation, and bending the rules by bypassing the dependency checking of **rpm** weakens this test.

Please report any issues back to the LSB team, preferably by filing a bug in the LSB Bugzilla[3] bugtracker.

Example 12.16 shows the installation of an **rpm** package. The package should now be ready to test in the normal way.

Example 12.16: Dependency Checking

```
# rpm -i /tmp/lsb-lynx-2.8.4-1.i486.rpm
```

12.4 Building the Sample Implementation

It is not necessary to compile the LSB-si for testing; the official packages released by the LSB project should be used for this purpose. However, it may be instructive to see how the LSB-si is constructed to get to a clean implementation of the LSB Written Specification. The remainder of this section describes the process.

The LSB-si is built from clean upstream package sources. The build instructions are captured in a set of XML files that serve as input to a tool called **nALFS**; the concept is derived from the Linux From Scratch[4] project.

3. `http://bugs.linuxbase.org`

4. `http://www.linuxfromscratch.org`

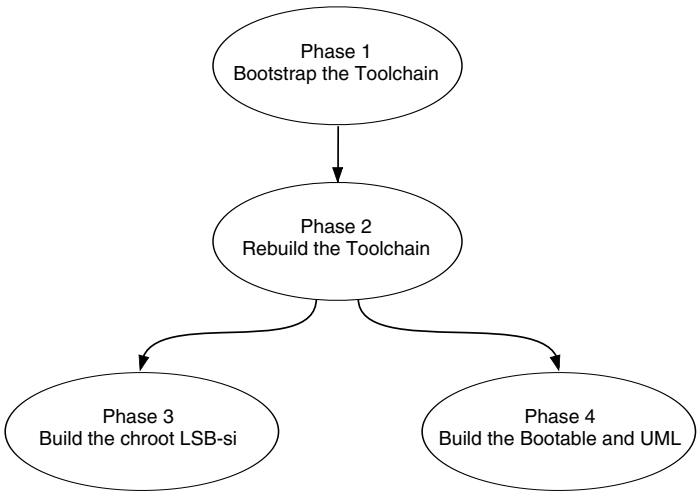

Figure 12.1: LSB-si Build

The build is a multistage process (Figure 12.1), so that the final result has been built by the LSB-si itself, and the majority of dependencies on the build environment of the host machine are eliminated. Ideally, all the dependencies would be eliminated, but in practice a few minor things may leak through. In particular, the initial stage of the LSB-si build now does not do the GCC `fixincludes` step as this pulled in some details of the host system in the "fixed" header files that were then used throughout the build process.

The **first phase**, or bootstrap, of the LSB-si build is to produce a minimal toolchain of static binaries as shown in Figure 12.1. Packages such as `gcc`, `binutils`, `kernel-headers`, and `coreutils` are built.

The **second phase** of the build is to use the bootstrap as a `chroot` environment to build a more complete toolchain as shown in Figure 12.1. As binaries are rebuilt, the new ones are installed on top of the old static copies built in the bootstrap phase so that by the end of the second phase, we have a complete development environment, using all dynamic libraries. This environment has the characteristic that it is entirely isolated from the details of the build host environment, since none of the tools from the build host have been used to compile the final binaries and libraries.

To reduce the rebuild time, the bootstrap phase is copied to another location before starting, and the copy is used as phase 2. During LSB-si development, there tend to be few changes to the bootstrap, but many to the later phases. For a released LSB-si source tree, this really doesn't matter except that it increases the space requirements of the build area a bit. Thus the bootstrap copy of second phase is not essential for the build strategy, but rather a convenience for LSB-si developers.

This intermediate phase 2 of the build can be used as an LSB Development Environment; in effect, this is what it does when building the final phase. The final phase does not have a compilation environment, as that is not part of the LSB Written Specification. The intermediate phase 2 is designed to be used as a `chroot` environment; using the compiler directly (not in a `chroot`) won't work as things will point relatively to the wrong places. Although the intermediate phase 2 is for the same architecture as the host machine, it is more like a cross-compilation environment. Note that producing a more usable build environment is a future direction; the current intermediate phase is not officially supported as such and the bundle is not part of the released materials.

The **third phase** is the construction of the actual LSB-si as it will be delivered as shown in Figure 12.1. In this phase, the completed second phase is used in a `chroot` as the development environment, and each package is then compiled and installed to a target location in the LSB-si tree. During the third phase, care is taken not to install unnecessary binaries or libraries, because an upstream source package will often build and install more than is required by the LSB, and these need to be pruned from the final tree.

Since the LSB team has already anticipated several uses for the LSB-si that require more than the core set, there exists a **fourth phase** that builds add-on bundles that can be installed on top of the base LSB-si bundle to provide additional functionality as shown in Figure 12.1. There are currently three subphases of the fourth phase: the first one to build additional tools required for running the `lsb-runtime-test` suite on the LSB-si, the second to build additional binaries to make a bootable system, and the third to build additional binaries to make a User-Mode Linux system. The fourth phase is built by the second phase build environment just like third phase is, and is completely independent of the third phase. That is, if one had a completed second phase, one could start off a fourth phase build without ever building the third phase

and it would work fine. It is likely that in the future, there will be additional fourth phase subphases to include in a build environment.

12.4.1 Sample Implementation Build Process

The source code for the LSB-si Development Environment can be obtained from the LSB CVS tree. The code can be checked out in several ways: as a snapshot either by release tag or by date, or as a working cvs directory (even if you're not an LSB developer, having a working directory can let you check developments more quickly by doing a "cvs update"). For an example using a release-tag snapshot, see the build instructions in Section 12.4.2.

You can browse the CVS tree Web interface to determine the available release tags.

You will also need to check out (or export) the `tools/nALFS` directory to get the build tool. Again, see Section 12.4.2 for an example.

Source code for the patches to the base tarballs is in the CVS tree in `si/build/patches`. These patches should be copied to the package source directory. The base tarballs must be obtained separately. Once the build area has been configured, a provided tool can be used to populate the package source directory.

The same tool (`extras/entitycheck.py`) can be used to check if all the necessary files are present before starting a build. With a `-c` option, it will do a more rigorous test, checking md5sums, not just existence. Every effort has been made to describe reliable locations for the files, but sometimes a project chooses to move an old version aside after releasing a new one (if they have a history of doing so, the location where old versions are placed is probably already captured). The packages are also mirrored on the Free Standards Group Web site.[5] Still, retrieval sometimes fails; `entitycheck.py` will inform of missing files and the expected locations are listed in `extras/package_locations` so it's possible to try to fetch the missing packages manually.

5. `http://ftp.freestandards.org/pub/lsb/impl/`

12.4.2 Sample Implementation Build Steps

1. Obtain LSB-si sources from CVS:

```
$ export CVSROOT
$ CVSROOT=":pserver:anonymous@cvs.gforge.freestandards.org:\
/cvsroot/lsb"
$ cvs -d $CVSROOT export -r lsbsi-2.0_1 si/build
```

   Use `-D now` instead of the release tag to grab the current development version.

2. Configure the build environment. There's a simple configuration script that localizes the makefile, some of the entities, and other bits. The main question is where you're going to build the LSB-si. The default location is `/usr/src/si`. Make sure the build directory exists and is in a place that has enough space (see the note at the end of this section).

```
$ cd src/si
$ ./Configure
```

   Answer the questions.

3. From here on, you'll need to operate as superuser, as the build process does mounts and the `chroot` command, operations restricted to root in most environments.

4. Copy patches to their final destination (substitute your build directory if not using the default):

```
# cp patches/* /usr/src/si/packages
```

5. Check that the package and patch area is up to date:

```
# python extras/entitycheck.py -f
```

6. You're now ready to build the LSB-si:

```
# make
```

7. If there's a problem, **make** should restart the build where it failed. If the interruption happened during the intermediate LSB-si phase, it is likely that the whole phase will be restarted; this is normal.

8. Building the add-on packages `lsbsi-test`, `lsbsi-boot`, and `lsbsi-uml` requires an additional step. This step is not dependent on the LSB-si (phase 3) step having completed, but it is dependent on the intermediate LSB-si (phase 2) step being complete:

```
# make addons
```

9. Now you can build the UML installable package (IA-32 build host or target only). This step is dependent on all of the other phases, including the add-ons, having completed:

```
# cd rpm
# make
```

The build takes a lot of space (around 1.4GB), and may take a lot of time. A full build on a fast dual-CPU Pentium 4 is about 2.5 hours; depending on architecture, memory, and processor speed it may take as much as 20 hours.

If the build stops, voluntarily or through some problem, there should be a fair bit of support for restartability, but this is not perfect. In particular, be cautious about cleaning out any of the build areas, as the package directory may still be bind-mounted. Each of the team members has accidentally removed the packages directory more than once, causing big delays while it's being refetched (it pays to make a copy of this directory somewhere else). Be careful! The makefile has a `clear_mounts` target that may be helpful.

12.5 Testing the Sample Implementation

The LSB-si (Figure 12.2) is subjected to the same testing regiment as described in Chapter 6:

1. `lsb-runtime-test`, with objective being zero failures
2. `lsblibchk`, to check that all the required libraries are present
3. Application Battery, running the FVTs for all the example applications

Install the LSB Runtime tests into the LSB-si `chroot` environment (Example 12.17).

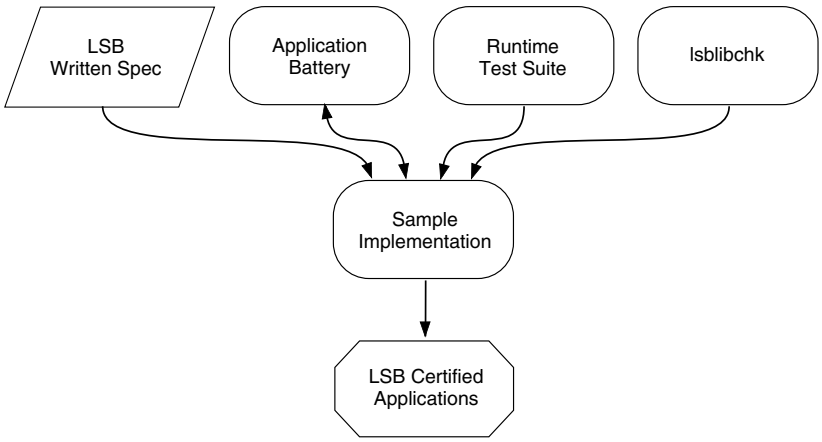

Figure 12.2: Sample Implementation

Example 12.17: Runtime Test Installation

```
# cp lsb-runtime-test-2.0-1.ia32.rpm /opt/lsbsi-ia32/tmp
# /sbin/chroot /opt/lsbsi-ia32
# cd /tmp
# rpm -i lsb-runtime-test-2.0-1.ia32.rpm
# password -d vsx0
```

As shown in Section 12.2.1, log in as user `vsx0`, then run the LSB Runtime test suite (Example 12.18).

Example 12.18: Running the Runtime Test Suite

```
vsx0$ telnet localhost
vsx0$ run_tests
```

12.6 PACKAGING THE SAMPLE IMPLEMENTATION

The LSB-si is released in several configurations for each of the following architectures:

1. **Base system** A tarball containing only what is required by the LSB Written Specification. Can be entered via `chroot`. For complete conformance, requires the underlying Linux kernel to be LSB-conformant.

2. **Testable system** Contains the base system plus additional files and programs needed to enable the LSB-si as a testing environment. This bundle can run the LSB Runtime test suite and LSB-conformant applications. Can be entered via `chroot`. Once running, it is possible to telnet into the environment. Complete conformance requires the underlying Linux kernel to be LSB-conformant.

3. **Bootable system** The files from the base system plus a kernel and other support to make it possible to boot a standalone. Some additional configuration will be necessary—for example, an appropriate boot configuration for the target hardware and configuration of necessary kernel modules.

4. **Hosted system** The files from the base system plus support to run User-Mode Linux, a kernel virtual machine that runs as a user-level application on the host system. User-Mode Linux may not be available for all LSB-supported architectures. The hosted bundle is also available as an installable, ready-to-run package.

When the package is being extended beyond the base, there's a tradeoff: The system becomes more usable, but includes various non-LSB commands and files, and therefore is a little less "pure" in a testing sense.

12.7 Downloading the Sample Implementation

To download the LSB Sample Implementation go to the LSB Download Web site.[6] Table 12.1 represents the information that can be accessed via the World Wide Web.

`lsbsi-boot` LSB Sample Implementation bootable system containing a kernel and other support to make it possible to boot the LSB-si in a standalone mode. Requires installation of `lsbsi-core` and post installation configuration.

`lsbsi-core` LSB Sample Implementation base system containing only what is required by the LSB Written Specification. Can be entered via `chroot`.

`lsbsi-graphics` LSB Sample Implementation graphics system containing the files necessary to support graphical applications within the LSB-si. Requires installation of `lsbsi-core`.

Table 12.1: Sample Implementation Download Page

Specification Version	*Package*	*Version*	*Architecture*
2.0.0	`lsbsi-boot`	2.0.2	IA32, IA64, PPC32, PPC64, S390, S390X, x86_64
2.0.0	`lsbsi-core`	2.0.2	IA32, IA64, PPC32, PPC64, S390, S390X, x86_64
2.0.0	`lsb-graphics`	2.0.2	IA32, IA64, PPC32, PPC64, S390, S390X, x86_64
2.0.0	`lsb-test`	2.0.2	IA32, IA64, PPC32, PPC64, S390, S390X, x86_64
2.0.0	`lsb-uml`	2.0.2	IA32, IA64, PPC32, PPC64, S390, S390X, x86_64

6. `http://www.linuxbase.org/download/`

`lsbsi-test` LSB Sample Implementation testable system containing additional files and programs needed to enable the LSB-si as a testing environment. Requires installation of `lsbsi-core`.

`lsbsi-uml` LSB Sample Implementation hosted system containing support to run User-Mode Linux, a kernel virtual machine that runs as a user-level application on the host system. UML may not be available for all LSB-supported architectures. Requires installation of `lsbsi-core`.

`lsb` This package provides the base LSB package. Note that this package should only ever be installed on the LSB Sample Implementation. Installing this package on any other distribution may damage the system.

`lsb-umlsi` LSB Sample Implementation User-Mode Linux environment.