Live Systems Manual

Live Systems Project <debian-live@lists.debian.org>

Contents

Live Systems Manual

About

1. About this manual

This manual serves as a single access point to all documentation related to the Live Systems project and in particular applies to the software produced by the project for the Debian 8.0 “**jessie** ” release. An up-to-date version can always be found at ‹`http://live-systems.org/`›

While *live-manual* is primarily focused on helping you build a live system and not on end-user topics, an end-user may find some useful information in these sections: ‹The Basics› covers downloading prebuilt images and preparing images to be booted from media or the network, either using the web builder or running *live-build* directly on your system. ‹Customizing run time behaviours› describes some options that may be specified at the boot prompt, such as selecting a keyboard layout and locale, and using persistence.

Some of the commands mentioned in the text must be executed with superuser privileges which can be obtained by becoming the root user via `su` or by using `sudo`. To distinguish between commands which may be executed by an unprivileged user and those requiring superuser privileges, commands are prepended by `$` or `#` respectively. This symbol is not a part of the command.

1.1 For the impatient

While we believe that everything in this manual is important to at least some of our users, we realize it is a lot of material to cover and that you may wish to experience early success using the software before delving into the details. Therefore, we suggest reading in the following order.

First, read this chapter, ‹About this manual›, from the beginning and ending with the ‹Terms› section. Next, skip to the three tutorials at the front of the ‹Examples› section designed to teach you image building and customization basics. Read ‹Using the examples› first, followed by ‹Tutorial 1: A default image›, ‹Tutorial 2: A web browser utility› and finally ‹Tutorial 3: A personalized image›. By the end of these tutorials, you will have a taste of what can be done with live systems.

We encourage you to return to more in-depth study of the manual, perhaps next reading ‹The basics›, skimming or skipping ‹Building a netboot image›, and finishing by reading the ‹Customization overview› and the chapters that follow it. By this point, we hope you are thoroughly excited by what can be done with live systems and motivated to read the rest of the manual, cover-to-cover.

1.2 Terms

- **Live system** : An operating system that can boot without installation to a hard drive. Live systems do not alter local operating system(s) or file(s) already installed on the computer hard drive unless instructed to do so. Live systems are typically booted from media such as CDs, DVDs or USB sticks. Some may also boot over the network.
- **Live medium** : As distinct from live system, the live medium refers to the CD, DVD or USB stick where the binary produced by *live-build* and used to boot the live system is written. More broadly, the term also refers to any place where this binary resides for the purposes of booting the live system, such as the location for the network boot files.
- **Live Systems Project** : The project which maintains, among others, the *live-boot*, *live-build*, *live-config*, *live-tools* and *live-manual* packages.
- **Live system** : A live system that may be booted from CDs, DVDs, USB sticks, over the network (via netboot images), and over the Internet (via boot parameter `fetch=URL`).
- **Host system** : The environment used to create the live system.
- **Target system** : The environment used to run the live system.
- ***live-boot*** : A collection of scripts used to boot live systems.
- ***live-build*** : A collection of scripts used to build customized live systems.
- ***live-config*** : A collection of scripts used to configure a live system during the boot process.
- ***live-tools*** : A collection of additional scripts used to perform useful tasks within a running live system.
- ***live-manual*** : This document is maintained in a package called *live-manual*.
- **Debian Installer (d-i)** : The official installation system for the Debian distribution.
- **Boot parameters** : Parameters that can be entered at the bootloader prompt to influence the kernel or *live-config*.
- **chroot** : The *chroot* program, `chroot(8)`, enables us to run different instances of the GNU/Linux environment on a single system simultaneously without rebooting.
- **Binary image** : A file containing the live system, such as binary.iso or binary.img.
- **Target distribution** : The distribution upon which your live system will be based. This can differ from the distribution of your host system.
- **stable/testing/unstable** : The **stable** distribution, currently codenamed **wheezy** , contains the latest officially released distribution of Debian. The **testing** distribution, temporarily codenamed **jessie** , is the staging area for the next **stable** release. A major

advantage of using this distribution is that it has more recent versions of software relative to the **stable** release. The **unstable** distribution, permanently codenamed **sid** , is where active development of Debian occurs. Generally, this distribution is run by developers and those who like to live on the edge. Throughout the manual, we tend to use codenames for the releases, such as **jessie** or **sid** , as that is what is supported by the tools themselves.

1.3 Authors

A list of authors (in alphabetical order):

- Ben Armstrong
- Brendan Sleight
- Carlos Zuferri
- Chris Lamb
- Daniel Baumann
- Franklin Piat
- Jonas Stein
- Kai Hendry
- Marco Amadori
- Mathieu Geli
- Matthias Kirschner
- Richard Nelson
- Trent W. Buck

1.4 Contributing to this document

This manual is intended as a community project and all proposals for improvements and contributions are extremely welcome. Please see the section ‹Contributing to the project› for detailed information on how to fetch the commit key and make good commits.

1.4.1 Applying changes

In order to make changes to the English manual you have to edit the right files in `manual/-en/` but prior to the submission of your contribution, please preview your work. To preview the *live-manual*, ensure the packages needed for building it are installed by executing:

```
# apt-get install make po4a ruby ruby-nokogiri sisu-complete texlive-generic-recommended
```

You may build the *live-manual* from the top level directory of your Git checkout by executing:

```
$ make build
```

Since it takes a while to build the manual in all supported languages, you may find it convenient when proofing to build for only one language, e.g. by executing:

```
$ make build LANGUAGES=en
```

It is also possible to build by document type, e.g:

```
$ make build FORMATS=pdf
```

Or combine both, e.g:

```
$ make build LANGUAGES=de FORMATS=html
```

After revising your work and making sure that everything is fine, do not use `make commit` unless you are updating translations in the commit, and in that case, do not mix changes to the English manual and translations in the same commit, but use separate commits for each. See the ‹Translation› section for more details.

1.4.2 Translation

In order to translate *live-manual*, follow these steps depending on whether you are starting a translation from scratch or continue working on an already existing one:

- Start a new translation from scratch
  - Translate the **about_manual.ssi.pot** , **about_project.ssi.pot** and **index.html.in.pot** files in `manual/pot/` to your language with your favourite editor (such as *poedit*) and send the translated `.po` files to the mailing list to check their integrity. *live-manual*' s integrity check not only ensures that the `.po` files are 100% translated but it also detects possible errors.
  - Once checked, to enable a new language in the autobuild it is enough to add the initial translated files to `manual/po/${LANGUAGE}/` and run `make commit`. And then, edit `manual/_sisu/home/index.html` adding the name of the language and its name in English between brackets.
- Continue with an already started translation
  - If your target language has already been added, you can randomly continue translating the remaining .po files in `manual/po/${LANGUAGE}/` using your favourite editor (such as *poedit*) .
  - Do not forget that you need to run `make commit` to ensure that the translated manuals are updated from the .po files and then you can review your changes launching `make build` before `git add .`, `git commit -m "Translating..."` and `git push`. Remember that since `make build` can take a considerable amount of time, you can proofread languages individually as explained in ‹Applying changes›

After running `make commit` you will see some text scroll by. These are basically informative messages about the processing status and also some hints about what can be done in order to improve *live-manual*. Unless you see a fatal error, you usually can proceed and submit your contribution.

live-manual comes with two utilities that can greatly help translators to find untranslated and changed strings. The first one is “make translate”. It launches an script that tells you in detail how many untranslated strings there are in each .po file. The second one, the “make fixfuzzy” target, only acts upon changed strings but it helps you to find and fix them one by one.

Keep in mind that even though these utilities might be really helpful to do translation work on the command line, the use of an specialized tool like *poedit* is the recommended way to do the task. It is also a good idea to read the Debian localization (l10n) documentation and, specifically to *live-manual*, the ‹Guidelines for translators›.

Note: You can use `make clean` to clean your git tree before pushing. This step is not

compulsory thanks to the .gitignore file but it is a good practice to avoid committing files involuntarily.

2. About the Live Systems Project

2.1 Motivation

2.1.1 What is wrong with current live systems

When Live Systems Project was initiated, there were already several Debian based live systems available and they are doing a great job. From the Debian perspective most of them have one or more of the following disadvantages:

- They are not Debian projects and therefore lack support from within Debian.
- They mix different distributions, e.g. **testing** and **unstable** .
- They support i386 only.
- They modify the behaviour and/or appearance of packages by stripping them down to save space.
- They include packages from outside of the Debian archive.
- They ship custom kernels with additional patches that are not part of Debian.
- They are large and slow due to their sheer size and thus not suitable for rescue issues.
- They are not available in different flavours, e.g. CDs, DVDs, USB-stick and netboot images.

2.1.2 Why create our own live system?

Debian is the Universal Operating System: Debian has a live system to show around and to accurately represent the Debian system with the following main advantages:

- It is a subproject of Debian.
- It reflects the (current) state of one distribution.
- It runs on as many architectures as possible.
- It consists of unchanged Debian packages only.
- It does not contain any packages that are not in the Debian archive.
- It uses an unaltered Debian kernel with no additional patches.

2.2 Philosophy

2.2.1 Only unchanged packages from Debian “main”

We will only use packages from the Debian repository in the “main” section. The non-free section is not part of Debian and therefore cannot be used for official live system images.

We will not change any packages. Whenever we need to change something, we will do that in coordination with its package maintainer in Debian.

As an exception, our own packages such as *live-boot*, *live-build* or *live-config* may temporarily be used from our own repository for development reasons (e.g. to create development snapshots). They will be uploaded to Debian on a regular basis.

2.2.2 No package configuration of the live system

In this phase we will not ship or install sample or alternative configurations. All packages are used in their default configuration as they are after a regular installation of Debian.

Whenever we need a different default configuration, we will do that in coordination with its package maintainer in Debian.

A system for configuring packages is provided using debconf allowing custom configured packages to be installed in your custom produced live system images, but for the ‹prebuilt live images› we choose to leave packages in their default configuration, unless absolutely necessary in order to work in the live environment. Wherever possible, we prefer to adapt packages within the Debian archive to work better in a live system versus making changes to the live toolchain or ‹prebuilt image configurations›. For more information, please see ‹Customization overview›.

2.3 Contact

- **Mailing list** : The primary contact for the project is the mailing list at ‹`http://lists.debian.org/debian-live/`›. You can email the list directly by addressing your mail to ‹`debian-live@lists.debian.org.`› The list archives are available at ‹`http://lists.debian.org/debian-live/`›.
- **IRC** : A number of users and developers are present in the #debian-live channel on irc.debian.org (OFTC). When asking a question on IRC, please be patient for an answer. If no answer is forthcoming, please email the mailing list.
- **BTS** : The ‹Reporting bugs›.

User

3. Installation

3.1 Requirements

Building live system images has very few system requirements:

- Superuser (root) access
- An up-to-date version of *live-build*
- A POSIX-compliant shell, such as *bash* or *dash*
- *debootstrap* or *cdebootstrap*
- Linux 2.6 or newer.

Note that using Debian or a Debian-derived distribution is not required - *live-build* will run on almost any distribution with the above requirements.

3.2 Installing live-build

You can install *live-build* in a number of different ways:

- From the Debian repository
- From source
- From snapshots

If you are using Debian, the recommended way is to install *live-build* via the Debian repository.

3.2.1 From the Debian repository

Simply install *live-build* like any other package:

```
# apt-get install live-build
```

3.2.2 From source

live-build is developed using the Git version control system. On Debian based systems, this is provided by the *git* package. To check out the latest code, execute:

```
$ git clone git://live-systems.org/git/live-build.git
```

You can build and install your own Debian package by executing:

```
$ cd live-build
$ dpkg-buildpackage -b -uc -us
$ cd ..
```

Now install whichever of the freshly built `.deb` files you were interested in, e.g.

```
# dpkg -i live-build_3.0-1_all.deb
```

You can also install *live-build* directly to your system by executing:

```
# make install
```

and uninstall it with:

```
# make uninstall
```

3.2.3 From `snapshots'

If you do not wish to build or install *live-build* from source, you can use snapshots. These are built automatically from the latest version in Git and are available on ‹`http://live-systems.org/debian/`›.

3.3 Installing live-boot and live-config

Note: You do not need to install *live-boot* or *live-config* on your system to create customized live systems. However, doing so will do no harm and is useful for reference purposes. If you only want the documentation, you may now install the *live-boot-doc* and *live-config-doc* packages separately.

3.3.1 From the Debian repository

Both *live-boot* and *live-config* are available from the Debian repository as per ‹Installing live-build›.

3.3.2 From source

To use the latest source from git, you can follow the process below. Please ensure you are familiar with the terms mentioned in ‹Terms›.

- Checkout the *live-boot* and *live-config* sources

```
$ git clone git://live-systems.org/git/live-boot.git
$ git clone git://live-systems.org/git/live-config.git
```

Consult the *live-boot* and *live-config* man pages for details on customizing if that is your reason for building these packages from source.

- Build *live-boot* and *live-config* .deb files

You must build either on your target distribution or in a chroot containing your target platform: this means if your target is **jessie** then you should build against **jessie** .

Use a personal builder such as *pbuilder* or *sbuild* if you need to build *live-boot* for a target distribution that differs from your build system. For example, for **jessie** live images, build *live-boot* in a **jessie** chroot. If your target distribution happens to match your build system distribution, you may build directly on the build system using `dpkg-buildpackage` (provided by the *dpkg-dev* package):

```
$ cd live-boot
$ dpkg-buildpackage -b -uc -us
$ cd ../live-config
$ dpkg-buildpackage -b -uc -us
```

- Use applicable generated .deb files

As *live-boot* and *live-config* are installed by *live-build* system, installing the packages in the host system is not sufficient: you should treat the generated .deb files like any other custom packages. Since your purpose for building from source is like to test new things over the short term before the official release, follow ‹Installing modified or third-party packages› to temporarily include the relevant files in your configuration. In particular, notice that both packages are divided into a generic part, a documentation part and one or more back-ends. Include the generic part, only one back-end matching your configuration, and optionally the documentation. Assuming you are building a live image in the current directory and have generated all .deb files for a single version of both packages in the directory above, these bash commands would copy all of the relevant packages including default back-ends:

```
$ cp ../live-boot{_,-initramfs-tools,-doc}*.deb  config/packages.chroot/
$ cp ../live-config{_,-sysvinit,-doc}*.deb  config/packages.chroot/
```

3.3.3 From `snapshots'

You can let *live-build* automatically use the latest snapshots of *live-boot* and *live-config* by configuring the package repository on live-systems.org as a third-party repository in your *live-build* configuration directory.

4. The basics

This chapter contains a brief overview of the build process and instructions for using the three most commonly used image types. The most versatile image type, `iso-hybrid`, may be used on a virtual machine, optical medium or USB portable storage device. In certain special cases, as explained later, the `hdd` type may be more suitable. The chapter finishes with instructions for building and using a `netboot` type image, which is a bit more involved due to the setup required on the server. This is an slightly advanced topic for anyone who is not familiar already with netbooting, but it is included here because once the setup is done, it is a very convenient way to test and deploy images for booting on the local network without the hassle of dealing with image media.

Throughout the chapter, we will often refer to the default filenames produced by *live-build*. If you are ‹downloading a prebuilt image› instead, the actual filenames may vary.

4.1 What is a live system?

A live system usually means an operating system booted on a computer from a removable

medium, such as a CD-ROM or USB stick, or from a network, ready to use without any installation on the usual drive(s), with auto-configuration done at run time (see ‹Terms›).

With live systems, it's an operating system, built for one of the supported architectures (currently amd64 and i386). It is made from the following parts:

- **Linux kernel image** , usually named `vmlinuz*`
- **Initial RAM disk image (initrd)** : a RAM disk set up for the Linux boot, containing modules possibly needed to mount the System image and some scripts to do it.
- **System image** : The operating system's filesystem image. Usually, a SquashFS compressed filesystem is used to minimize the live system image size. Note that it is read-only. So, during boot the live system will use a RAM disk and `union' mechanism to enable writing files within the running system. However, all modifications will be lost upon shutdown unless optional persistence is used (see ‹Persistence›).
- **Bootloader** : A small piece of code crafted to boot from the chosen medium, possibly presenting a prompt or menu to allow selection of options/configuration. It loads the Linux kernel and its initrd to run with an associated system filesystem. Different solutions can be used, depending on the target medium and format of the filesystem containing the previously mentioned components: isolinux to boot from a CD or DVD in ISO9660 format, syslinux for HDD or USB drive booting from a VFAT partition, extlinux for ext2/3/4 and btrfs partitions, pxelinux for PXE netboot, GRUB for ext2/3/4 partitions, etc.

You can use *live-build* to build the system image from your specifications, set up a Linux kernel, its initrd, and a bootloader to run them, all in one medium-dependant format (ISO9660 image, disk image, etc.).

4.2 Downloading prebuilt images

While the focus of this manual is developing and building your own live images, you may simply wish to try one of our prebuilt images, either as an introduction to their use or instead of building your own. These images are built using our ‹*live-images* git repository› and official stable releases are published at ‹`http://www.debian.org/CD/live/`›. In addition, older and upcoming releases, and unofficial images containing non-free firmware and drivers are available at ‹`http://live-systems.org/cdimage/release/`›.

4.3 Using the web live image builder

As a service to the community, we run a web-based live image builder service at ‹`http://live-build.debian.net/`›. This site is maintained on a best effort basis. That is, although we strive to keep it up-to-date and operational at all times, and do issue notices for significant

operational outages, we cannot guarantee 100% availability or fast image building, and the service may occasionally have issues that take some time to resolve. If you have problems or questions about the service, please ‹contact us›, providing us with the link to your build.

4.3.1 Web builder usage and caveats

The web interface currently makes no provision to prevent the use of invalid combinations of options, and in particular, where changing an option would normally (i.e. using *live-build* directly) change defaults of other options listed in the web form, the web builder does not change these defaults. Most notably, if you change `--architectures` from the default `i386` to `amd64`, you must change the corresponding option `--linux-flavours` from the default `486` to `amd64`. See the `lb_config` man page for the version of *live-build* installed on the web builder for more details. The version number of *live-build* is listed at the bottom of the web builder page.

The time estimate given by the web builder is a crude estimate only and may not reflect how long your build actually takes. Nor is the estimate updated once it is displayed. Please be patient. Do not refresh the page you land on after submitting the build, as this will resubmit a new build with the same parameters. You should ‹contact us› if you don't receive notification of your build only once you are certain you've waited long enough and verified the notification e-mail did not get caught by your own e-mail spam filter.

The web builder is limited in the kinds of images it can build. This keeps it simple and efficient to use and maintain. If you would like to make customizations that are not provided for by the web interface, the rest of this manual explains how to build your own images using *live-build*.

4.4 First steps: building an ISO hybrid image

Regardless of the image type, you will need to perform the same basic steps to build an image each time. As a first example, create a build directory, change to that directory and then execute the following sequence of *live-build* commands to create a basic ISO hybrid image containing a default live system without X.org. It is suitable for burning to CD or DVD media, and also to copy onto a USB stick.

The name of the working directory is absolutely up to you, but if you take a look at the examples used throughout *live-manual*, it is a good idea to use a name that helps you identify the image you are working with in each directory, especially if you are working or experimenting with different image types. In this case you are going to build a default system so let's call it, for example, live-default.

```
$ mkdir live-default && cd live-default
```

Then, run the `lb config` command. This will create a “config/” hierarchy in the current directory for use by other commands:

```
$ lb config
```

No parameters are passed to `lb config`, so defaults for all of its various options will be used. See ‹The lb config command› for more details.

Now that the “config/” hierarchy exists, build the image with the `lb build` command:

```
# lb build
```

This process can take a while, depending on the speed of your computer and your network connection. When it is complete, there should be a `binary.hybrid.iso` image file, ready to use, in the current directory.

4.5 Using an ISO hybrid live image

After either building or downloading an ISO hybrid image, which can be obtained at ‹`http://www.debian.org/CD/live/`›, the usual next step is to prepare your medium for booting, either CD-R(W) or DVD-R(W) optical media or a USB stick.

4.5.1 Burning an ISO image to a physical medium

Burning an ISO image is easy. Just install *xorriso* and use it from the command-line to burn the image. For instance:

```
# apt-get install xorriso

$ xorriso -as cdrecord -v dev=/dev/sr0 blank=as_needed binary.hybrid.iso
```

4.5.2 Copying an ISO hybrid image to a USB stick

ISO images prepared with `xorriso`, can be simply copied to a USB stick with the `cp` program or an equivalent. Plug in a USB stick with a size large enough for your image file and determine which device it is, which we hereafter refer to as `${USBSTICK}`. This is the device file of your key, such as `/dev/sdb`, not a partition, such as `/dev/sdb1`! You can find the right device name by looking in `dmesg`'s output after plugging in the stick, or better yet, `ls -l /dev/disk/by-id`.

Once you are certain you have the correct device name, use the `cp` command to copy the image to the stick. **This will definitely overwrite any previous contents on your stick!**

```
$ cp binary.hybrid.iso ${USBSTICK}

$ sync
```

4.5.3 Using the space left on a USB stick

To use the remaining free space after copying `binary.hybrid.iso` to a USB stick, use a partitioning tool such as *gparted* or *parted* to create a new partition on the stick. The first partition will be used by the live system.

```
# gparted ${USBSTICK}
```

After the partition is created, where `${PARTITION}` is the name of the partition, such as `/dev/sdb2`, you have to create a filesystem on it. One possible choice would be ext4.

```
# mkfs.ext4 ${PARTITION}
```

Note: If you want to use the extra space with Windows, apparently that OS cannot normally access any partitions but the first. Some solutions to this problem have been discussed on our ‹mailing list›, but it seems there are no easy answers.

Remember: Every time you install a new binary.hybrid.iso on the stick, all data on the stick will be lost because the partition table is overwritten by the contents of the image, so back up your extra partition first to restore again after updating the live image.

4.5.4 Booting the live medium

The first time you boot your live medium, whether CD, DVD, USB key, or PXE boot, some setup in your computer's BIOS may be needed first. Since BIOSes vary greatly in features and key bindings, we cannot get into the topic in depth here. Some BIOSes provide a key to bring up a menu of boot devices at boot time, which is the easiest way if it is available on your system. Otherwise, you need to enter the BIOS configuration menu and change the boot order to place the boot device for the live system before your normal boot device.

Once you've booted the medium, you are presented with a boot menu. If you just press enter here, the system will boot using the default entry, `Live` and default options. For more information about boot options, see the “help” entry in the menu and also the *live-boot* and *live-config* man pages found within the live system.

Assuming you've selected `Live` and booted a default desktop live image, after the boot messages scroll by, you should be automatically logged into the `user` account and see a desktop, ready to use. If you have booted a console-only image, such as `standard` or `rescue` flavour ‹prebuilt images›, you should be automatically logged in on the console to the `user` account and see a shell prompt, ready to use.

4.6 Using a virtual machine for testing

It can be a great time-saver for the development of live images to run them in a virtual machine (VM). This is not without its caveats:

- Running a VM requires enough RAM for both the guest OS and the host and a CPU with hardware support for virtualization is recommended.
- There are some inherent limitations to running on a VM, e.g. poor video performance, limited choice of emulated hardware.
- When developing for specific hardware, there is no substitute for running on the hardware itself.
- Occasionally there are bugs that relate only to running in a VM. When in doubt, test your image directly on the hardware.

Provided you can work within these constraints, survey the available VM software and choose one that is suitable for your needs.

4.6.1 Testing an ISO image with QEMU

The most versatile VM in Debian is QEMU. If your processor has hardware support for virtualization, use the *qemu-kvm* package; the *qemu-kvm* package description briefly lists the requirements.

First, install *qemu-kvm* if your processor supports it. If not, install *qemu*, in which case the program name is `qemu` instead of `kvm` in the following examples. The *qemu-utils* package is also valuable for creating virtual disk images with `qemu-img`.

```
# apt-get install qemu-kvm qemu-utils
```

Booting an ISO image is simple:

```
$ kvm -cdrom binary.hybrid.iso
```

See the man pages for more details.

4.6.2 Testing an ISO image with VirtualBox

In order to test the ISO with *virtualbox*:

```
# apt-get install virtualbox virtualbox-qt virtualbox-dkms

$ virtualbox
```

Create a new virtual machine, change the storage settings to use `binary.hybrid.iso` as the CD/DVD device, and start the machine.

Note: For live systems containing X.org that you want to test with *virtualbox*, you may wish to include the VirtualBox X.org driver package, *virtualbox-guest-dkms* and *virtualbox-guest-x11*, in your *live-build* configuration. Otherwise, the resolution is limited to 800x600.

```
$ echo "virtualbox-guest-dkms virtualbox-guest-x11" >> config/package-lists/my.list.chroot
```

In order to make the dkms package work, also the kernel headers for the kernel flavour used in your image need to be installed. Instead of manually listing the correct *linux-headers* package in above created package list, the selection of the right package can be done automatically by *live-build*.

```
  $ lb config --linux-packages "linux-image linux-headers"
```

4.7 Building and using an HDD image

Building an HDD image is similar to an ISO hybrid one in all respects except you specify `-b hdd` and the resulting filename is `binary.img` which cannot be burnt to optical media. It is suitable for booting from USB sticks, USB hard drives, and various other portable storage devices. Normally, an ISO hybrid image can be used for this purpose instead, but if you have a BIOS which does not handle hybrid images properly, you need an HDD image.

Note: if you created an ISO hybrid image with the previous example, you will need to clean up your working directory with the `lb clean` command (see ‹The lb clean command›):

```
# lb clean --binary
```

Run the `lb config` command as before, except this time specifying the HDD image type:

```
$ lb config -b hdd
```

Now build the image with the `lb build` command:

```
# lb build
```

When the build finishes, a `binary.img` file should be present in the current directory.

The generated binary image contains a VFAT partition and the syslinux bootloader, ready to be directly written on a USB device. Once again, using an HDD image is just like using an ISO hybrid one on USB. Follow the instructions in ‹Using an ISO hybrid live image›, except use the filename `binary.img` instead of `binary.hybrid.iso`.

Likewise, to test an HDD image with Qemu, install *qemu* as described above in ‹Testing an ISO image with QEMU›. Then run `kvm` or `qemu`, depending on which version your host system needs, specifying `binary.img` as the first hard drive.

```
$ kvm -hda binary.img
```

4.8 Building a netboot image

The following sequence of commands will create a basic netboot image containing a default live system without X.org. It is suitable for booting over the network.

Note: if you performed any previous examples, you will need to clean up your working directory with the `lb clean` command:

```
# lb clean
```

In this specific case, a `lb clean --binary` would not be enough to clean up the necessary stages. The cause for this is that in netboot setups, a different initramfs configuration needs to be used which *live-build* performs automatically when building netboot images. Since the initramfs creation belongs to the chroot stage, switching to netboot in an existing build directory means to rebuild the chroot stage too. Therefore, `lb clean` (which will remove the chroot stage, too) needs to be used.

Run the `lb config` command as follows to configure your image for netbooting:

```
$ lb config -b netboot --net-root-path "/srv/debian-live" --net-root-server "192.168.0.2"
```

In contrast with the ISO and HDD images, netbooting does not, itself, serve the filesystem image to the client, so the files must be served via NFS. Different network filesystems can be chosen through lb config. The `--net-root-path` and `--net-root-server` options specify the location and server, respectively, of the NFS server where the filesytem image will be located at boot time. Make sure these are set to suitable values for your network and server.

Now build the image with the `lb build` command:

```
# lb build
```

In a network boot, the client runs a small piece of software which usually resides on the EPROM of the Ethernet card. This program sends a DHCP request to get an IP address and information about what to do next. Typically, the next step is getting a higher level bootloader via the TFTP protocol. That could be pxelinux, GRUB, or even boot directly to an operating system like Linux.

For example, if you unpack the generated `binary.netboot.tar` archive in the `/srv/-`

`debian-live` directory, you'll find the filesystem image in `live/filesystem.squashfs` and the kernel, initrd and pxelinux bootloader in `tftpboot/`.

We must now configure three services on the server to enable netbooting: the DHCP server, the TFTP server and the NFS server.

4.8.1 DHCP server

We must configure our network's DHCP server to be sure to give an IP address to the netbooting client system, and to advertise the location of the PXE bootloader.

Here is an example for inspiration, written for the ISC DHCP server `isc-dhcp-server` in the `/etc/dhcp/dhcpd.conf` configuration file:

```
# /etc/dhcp/dhcpd.conf - configuration file for isc-dhcp-server

ddns-update-style none;

option domain-name "example.org";
option domain-name-servers ns1.example.org, ns2.example.org;

default-lease-time 600;
max-lease-time 7200;

log-facility local7;

subnet 192.168.0.0 netmask 255.255.255.0 {
   range 192.168.0.1 192.168.0.254;
   filename "pxelinux.0";
   next-server 192.168.0.2;
   option subnet-mask 255.255.255.0;
   option broadcast-address 192.168.0.255;
   option routers 192.168.0.1;
}
```

4.8.2 TFTP server

This serves the kernel and initial ramdisk to the system at run time.

You should install the *tftpd-hpa* package. It can serve all files contained inside a root directory, usually `/srv/tftp`. To let it serve files inside `/srv/debian-live/tftpboot`, run as root the following command:

```
# dpkg-reconfigure -plow tftpd-hpa
```

and fill in the new tftp server directory when being asked about it.

4.8.3 NFS server

Once the guest computer has downloaded and booted a Linux kernel and loaded its initrd, it will try to mount the Live filesystem image through a NFS server.

You need to install the *nfs-kernel-server* package.

Then, make the filesystem image available through NFS by adding a line like the following to `/etc/exports`:

```
/srv/debian-live *(ro,async,no_root_squash,no_subtree_check)
```

and tell the NFS server about this new export with the following command:

```
# exportfs -rv
```

Setting up these three services can be a little tricky. You might need some patience to get all of them working together. For more information, see the syslinux wiki at ‹`http://www.syslinux.org/wiki/index.php/PXELINUX`› or the Debian Installer Manual's TFTP Net Booting section at ‹`http://d-i.alioth.debian.org/manual/en.i386/ch04s05.html`›. They might help, as their processes are very similar.

4.8.4 Netboot testing HowTo

Netboot image creation is made easy with *live-build*, but testing the images on physical machines can be really time consuming.

To make our life easier, we can use virtualization.

4.8.5 Qemu

- Install *qemu*, *bridge-utils*, *sudo*.

Edit `/etc/qemu-ifup`:

```
#!/bin/sh
sudo -p "Password for $0:" /sbin/ifconfig $1 172.20.0.1
echo "Executing /etc/qemu-ifup"
echo "Bringing up $1 for bridged mode..."
sudo /sbin/ifconfig $1 0.0.0.0 promisc up
```

```
echo "Adding $1 to br0..."
sudo /usr/sbin/brctl addif br0 $1
sleep 2
```

Get, or build a `grub-floppy-netboot`.

Launch `qemu` with "`-net nic,vlan=0 -net tap,vlan=0,ifname=tun0`"

5. Overview of tools

This chapter contains an overview of the three main tools used in building live systems: *live-build*, *live-boot* and *live-config*.

5.1 The live-build package

live-build is a collection of scripts to build live systems. These scripts are also referred to as "commands".

The idea behind *live-build* is to be a framework that uses a configuration directory to completely automate and customize all aspects of building a Live image.

Many concepts are similar to those used to build Debian packages with *debhelper*:

- The scripts have a central location for configuring their operation. In *debhelper*, this is the `debian/` subdirectory of a package tree. For example, dh_install will look, amongst others, for a file called `debian/install` to determine which files should exist in a particular binary package. In much the same way, *live-build* stores its configuration entirely under a `config/` subdirectory.
- The scripts are independent - that is to say, it is always safe to run each command.

Unlike *debhelper*, *live-build* contains a tool to generate a skeleton configuration directory, `lb config`. This could be considered to be similar to tools such as *dh-make*. For more information about `lb config`, please see ‹The lb config command›.

The remainder of this section discusses the three most important commands:

- **lb config** : Responsible for initializing a Live system configuration directory. See ‹The lb config command› for more information.
- **lb build** : Responsible for starting a Live system build. See ‹The lb build command› for more information.
- **lb clean** : Responsible for removing parts of a Live system build. See ‹The lb clean command› for more information.

5.1.1 The `lb config` command

As discussed in ‹live-build›, the scripts that make up *live-build* read their configuration with the `source` command from a single directory named `config/`. As constructing this directory by hand would be time-consuming and error-prone, the `lb config` command can be used to create skeleton configuration folders.

Issuing `lb config` without any arguments creates a `config/` subdirectory which it populates with some default settings, and a skeleton `auto/` subdirectory tree.

```
$ lb config
[2013-01-01 09:14:22] lb_config
P: Considering defaults defined in /etc/live/build.conf
P: Creating config tree for a debian/i386 system
```

Using `lb config` without any arguments would be suitable for users who need a very basic image, or who intend to later provide a more complete configuration via `auto/config` (see ‹Managing a configuration› for details).

Normally, you will want to specify some options. For example, to specify which distribution you want to build using its codename:

```
$ lb config --distribution sid
```

It is possible to specify many options, such as:

```
$ lb config --binary-images netboot --bootappend-live "boot=live components hostname=live-host username=live-user↩
    " ...
```

A full list of options is available in the `lb_config` man page.

5.1.2 The `lb build` command

The `lb build` command reads in your configuration from the `config/` directory. It then runs the lower level commands needed to build your Live system.

5.1.3 The `lb clean` command

It is the job of the `lb clean` command to remove various parts of a build so subsequent

builds can start from a clean state. By default, `chroot`, `binary` and `source` stages are cleaned, but the cache is left intact. Also, individual stages can be cleaned. For example, if you have made changes that only affect the binary stage, use `lb clean --binary` prior to building a new binary. If your changes invalidate the bootstrap and/or package caches, e.g. changes to `--mode`, `--architecture`, or `--bootstrap`, you must use `lb clean --purge`. See the `lb_clean` man page for a full list of options.

5.2 The live-boot package

live-boot is a collection of scripts providing hooks for the *initramfs-tools*, used to generate an initramfs capable of booting live systems, such as those created by *live-build*. This includes the live system ISOs, netboot tarballs, and USB stick images.

At boot time it will look for read-only media containing a `/live/` directory where a root filesystem (often a compressed filesystem image like squashfs) is stored. If found, it will create a writable environment, using aufs, for Debian like systems to boot from.

More information on initial ramfs in Debian can be found in the Debian Linux Kernel Handbook at ‹`http://kernel-handbook.alioth.debian.org/`› in the chapter on initramfs.

5.3 The live-config package

live-config consists of the scripts that run at boot time after *live-boot* to configure the live system automatically. It handles such tasks as setting the hostname, locales and timezone, creating the live user, inhibiting cron jobs and performing autologin of the live user.

6. Managing a configuration

This chapter explains how to manage a live configuration from initial creation, through successive revisions and successive releases of both the *live-build* software and the live image itself.

6.1 Dealing with configuration changes

Live configurations rarely are perfect on the first try. It may be fine to pass `lb config` options from the command-line to perform a single build, but it is more typical to revise those options and build again until you are satisfied. To support these changes, you will need auto scripts which ensure your configuration is kept in a consistent state.

6.1.1 Why use auto scripts? What do they do?

The `lb config` command stores the options you pass to it in `config/*` files along with many other options set to default values. If you run `lb config` again, it will not reset any option that was defaulted based on your initial options. So, for example, if you run `lb config` again with a new value for `--distribution`, any dependent options that were defaulted for the old distribution may no longer work with the new. Nor are these files intended to be read or edited. They store values for over a hundred options, so nobody, let alone yourself, will be able to see in these which options you actually specified. And finally, if you run `lb config`, then upgrade *live-build* and it happens to rename an option, `config/*` would still contain variables named after the old option that are no longer valid.

For all these reasons, `auto/*` scripts will make your life easier. They are simple wrappers to the `lb config`, `lb build` and `lb clean` commands that are designed to help you manage your configuration. The `auto/config` script stores your `lb config` command with all desired options, the `auto/clean` script removes the files containing configuration variable values, and the `auto/build` script keeps a `build.log` of each build. Each of these scripts is run automatically every time you run the corresponding `lb` command. By using these scripts, your configuration is easier to read and is kept internally consistent from one revision to the next. Also, it will be much easier for you identify and fix options which need to change when you upgrade *live-build* after reading the updated documentation.

6.1.2 Use example auto scripts

For your convenience, *live-build* comes with example auto shell scripts to copy and edit. Start a new, default configuration, then copy the examples into it:

```
$ mkdir mylive && cd mylive && lb config
$ cp /usr/share/doc/live-build/examples/auto/* auto/
```

Edit `auto/config`, adding any options as you see fit. For instance:

```
#!/bin/sh
lb config noauto \
     --architectures i386 \
     --linux-flavours 686-pae \
     --binary-images hdd \
     --mirror-bootstrap http://ftp.ch.debian.org/debian/ \
     --mirror-binary http://ftp.ch.debian.org/debian/ \
     "${@}"
```

Now, each time you use `lb config`, `auto/config` will reset the configuration based on these

options. When you want to make changes to them, edit the options in this file instead of passing them to `lb config`. When you use `lb clean`, `auto/clean` will clean up the `config/*` files along with any other build products. And finally, when you use `lb build`, a log of the build will be written by `auto/build` in `build.log`.

Note: A special `noauto` parameter is used here to suppress another call to `auto/config`, thereby preventing infinite recursion. Make sure you don't accidentally remove it when making edits. Also, take care to ensure when you split the `lb config` command across multiple lines for readability, as shown in the example above, that you don't forget the backslash (at the end of each line that continues to the next.

6.2 Clone a configuration published via Git

Use the `lb config --config` option to clone a Git repository that contains a live system configuration. If you would like to base your configuration on one maintained by the Live Systems project, look at ‹`http://live-systems.org/gitweb/`› for the repository named `live-images` in the category `Packages`. This repository contains the configurations for the live systems ‹prebuilt images›.

For example, to build a rescue image, use the `live-images` repository as follows:

```
$ mkdir live-images && cd live-images
$ lb config --config git://live-systems.org/git/live-images.git
$ cd images/rescue
```

Edit `auto/config` and any other things you need in the `config` tree to suit your needs. For example, the unofficial non-free prebuilt images are made by simply adding `--archive-areas “main contrib non-free”`.

You may optionally define a shortcut in your Git configuration by adding the following to your `${HOME}/.gitconfig`:

```
[url "git://live-systems.org/git/"]
        insteadOf = ldn:
```

This enables you to use `ldn:` anywhere you need to specify the address of a `live-systems.org` git repository. If you also drop the optional `.git` suffix, starting a new image using this configuration is as easy as:

```
$ lb config --config ldn:live-images
```

Cloning the entire `live-images` repository pulls the configurations used for several images. If you feel like building a different image after you have finished with the first one, change to another directory and again and optionally, make any changes to suit your needs.

In any case, remember that every time you will have to build the image as superuser: `lb build`

7. Customization overview

This chapter gives an overview of the various ways in which you may customize a live system.

7.1 Build time vs. boot time configuration

Live system configuration options are divided into build-time options which are options that are applied at build time and boot-time options which are applied at boot time. Boot-time options are further divided into those occurring early in the boot, applied by the *live-boot* package, and those that happen later in the boot, applied by *live-config*. Any boot-time option may be modified by the user by specifying it at the boot prompt. The image may also be built with default boot parameters so users can normally just boot directly to the live system without specifying any options when all of the defaults are suitable. In particular, the argument to `lb --bootappend-live` consists of any default kernel command line options for the Live system, such as persistence, keyboard layouts, or timezone. See ‹Customizing locale and language›, for example.

Build-time configuration options are described in the `lb config` man page. Boot-time options are described in the man pages for *live-boot* and *live-config*. Although the *live-boot* and *live-config* packages are installed within the live system you are building, it is recommended that you also install them on your build system for easy reference when you are working on your configuration. It is safe to do so, as none of the scripts contained within them are executed unless the system is configured as a live system.

7.2 Stages of the build

The build process is divided into stages, with various customizations applied in sequence in each. The first stage to run is the **bootstrap** stage. This is the initial phase of populating the chroot directory with packages to make a barebones Debian system. This is followed by the **chroot** stage, which completes the construction of chroot directory, populating it with all of the packages listed in the configuration, along with any other materials. Most

customization of content occurs in this stage. The final stage of preparing the live image is the **binary** stage, which builds a bootable image, using the contents of the chroot directory to construct the root filesystem for the Live system, and including the installer and any other additional material on the target medium outside of the Live system's filesystem. After the live image is built, if enabled, the source tarball is built in the **source** stage.

Within each of these stages, there is a particular sequence in which commands are applied. These are arranged in such a way as to ensure customizations can be layered in a reasonable fashion. For example, within the **chroot** stage, preseeds are applied before any packages are installed, packages are installed before any locally included files are copied, and hooks are run later, after all of the materials are in place.

7.3 Supplement lb config with files

Although `lb config` creates a skeletal configuration in the config/ directory, to accomplish your goals, you may need to provide additional files in subdirectories of config/. Depending on where the files are stored in the configuration, they may be copied into the live system's filesystem or into the binary image filesystem, or may provide build-time configurations of the system that would be cumbersome to pass as command-line options. You may include things such as custom lists of packages, custom artwork, or hook scripts to run either at build time or at boot time, boosting the already considerable flexibility of debian-live with code of your own.

7.4 Customization tasks

The following chapters are organized by the kinds of customization task users typically perform: ‹Customizing package installation›, ‹Customizing contents› and ‹Customizing locale and language› cover just a few of the things you might want to do.

8. Customizing package installation

Perhaps the most basic customization of a live system is the selection of packages to be included in the image. This chapter guides you through the various build-time options to customize *live-build*' s installation of packages. The broadest choices influencing which packages are available to install in the image are the distribution and archive areas. To ensure decent download speeds, you should choose a nearby distribution mirror. You can also add your own repositories for backports, experimental or custom packages, or include packages directly as files. You can define lists of packages, including metapackages which will install many related packages at once, such as packages for a particular desktop or language. Finally, a number of options give some control over *apt*, or if you prefer, *aptitude*, at build time when packages are installed. You may find these handy if you use a proxy,

want to disable installation of recommended packages to save space, or need to control which versions of packages are installed via APT pinning, to name a few possibilities.

8.1 Package sources

8.1.1 Distribution, archive areas and mode

The distribution you choose has the broadest impact on which packages are available to include in your live image. Specify the codename, which defaults to **jessie** for the **jessie** version of *live-build*. Any current distribution carried in the archive may be specified by its codename here. (See ‹Terms› for more details.) The `--distribution` option not only influences the source of packages within the archive, but also instructs *live-build* to behave as needed to build each supported distribution. For example, to build against the **unstable** release, **sid** , specify:

```
$ lb config --distribution sid
```

Within the distribution archive, archive areas are major divisions of the archive. In Debian, these are `main`, `contrib` and `non-free`. Only `main` contains software that is part of the Debian distribution, hence that is the default. One or more values may be specified, e.g.

```
$ lb config --archive-areas "main contrib non-free"
```

Experimental support is available for some Debian derivatives through a `--mode` option. By default, this option is set to `debian` only if you are building on a Debian or on an unknown system. If `lb config` is invoked on any of the supported derivatives, it will default to create an image of that derivative. If `lb config` is run in e.g. `ubuntu` mode, the distribution names and archive areas for the specified derivative are supported instead of the ones for Debian. The mode also modifies *live-build* behaviour to suit the derivatives.

Note: The projects for whom these modes were added are primarily responsible for supporting users of these options. The Live Systems project, in turn, provides development support on a best-effort basis only, based on feedback from the derivative projects as we do not develop or support these derivatives ourselves.

8.1.2 Distribution mirrors

The Debian archive is replicated across a large network of mirrors around the world so

that people in each region can choose a nearby mirror for best download speed. Each of the `--mirror-*` options governs which distribution mirror is used at various stages of the build. Recall from ‹Stages of the build› that the **bootstrap** stage is when the chroot is initially populated by *debootstrap* with a minimal system, and the **chroot** stage is when the chroot used to construct the live system's filesystem is built. Thus, the corresponding mirror switches are used for those stages, and later, in the **binary** stage, the `--mirror-binary` and `--mirror-binary-security` values are used, superseding any mirrors used in an earlier stage.

8.1.3 Distribution mirrors used at build time

To set the distribution mirrors used at build time to point at a local mirror, it is sufficient to set `--mirror-bootstrap`, `--mirror-chroot-security` and `--mirror-chroot-backports` as follows.

```
$ lb config --mirror-bootstrap http://localhost/debian/ \
          --mirror-chroot-security http://localhost/debian-security/ \
          --mirror-chroot-backports http://localhost/debian-backports/
```

The chroot mirror, specified by `--mirror-chroot`, defaults to the `--mirror-bootstrap` value.

8.1.4 Distribution mirrors used at run time

The `--mirror-binary*` options govern the distribution mirrors placed in the binary image. These may be used to install additional packages while running the live system. The defaults employ `http.debian.net`, a service that chooses a geographically close mirror based, among other things, on the user's IP family and the availability of the mirrors. This is a suitable choice when you cannot predict which mirror will be best for all of your users. Or you may specify your own values as shown in the example below. An image built from this configuration would only be suitable for users on a network where “`mirror`” is reachable.

```
$ lb config --mirror-binary http://mirror/debian/ \
          --mirror-binary-security http://mirror/debian-security/ \
          --mirror-binary-backports http://mirror/debian-backports/
```

8.1.5 Additional repositories

You may add more repositories, broadening your package choices beyond what is available in your target distribution. These may be, for example, for backports, experimental or custom packages. To configure additional repositories, create `config/archives/your-repository.list.chroot`, and/or `config/archives/your-repository.list.binary` files. As with the `--mirror-*` options, these govern the repositories used in the **chroot** stage when building the image, and in the **binary** stage, i.e. for use when running the live system.

For example, `config/archives/live.list.chroot` allows you to install packages from the debian-live snapshot repository at live system build time.

```
deb http://live-systems.org/ sid-snapshots main contrib non-free
```

If you add the same line to `config/archives/live.list.binary`, the repository will be added to your live system's `/etc/apt/sources.list.d/` directory.

If such files exist, they will be picked up automatically.

You should also put the GPG key used to sign the repository into `config/archives/your-repository.key.{binary,chroot}` files.

Should you need custom APT pinning, such APT preferences snippets can be placed in `config/archives/your-repository.pref.{binary,chroot}` files and will be automatically added to your live system's `/etc/apt/preferences.d/` directory.

8.2 Choosing packages to install

There are a number of ways to choose which packages *live-build* will install in your image, covering a variety of different needs. You can simply name individual packages to install in a package list. You can also use metapackages in those lists, or select them using package control file fields. And finally, you may place package files in your `config/` tree, which is well suited to testing of new or experimental packages before they are available from a repository.

8.2.1 Package lists

Package lists are a powerful way of expressing which packages should be installed. The list syntax supports conditional sections which makes it easy to build lists and adapt them for use in multiple configurations. Package names may also be injected into the list using shell helpers at build time.

Note: The behaviour of *live-build* when specifying a package that does not exist is determined by your choice of APT utility. See ‹Choosing apt or aptitude› for more details.

8.2.2 Using metapackages

The simplest way to populate your package list is to use a task metapackage maintained by your distribution. For example:

```
$ lb config
$ echo task-gnome-desktop > config/package-lists/desktop.list.chroot
```

This supercedes the older predefined list method supported in `live-build` 2.x. Unlike predefined lists, task metapackages are not specific to the Live System project. Instead, they are maintained by specialist working groups within the distribution and therefore reflect the consensus of each group about which packages best serve the needs of the intended users. They also cover a much broader range of use cases than the predefined lists they replace.

All task metapackages are prefixed `task-`, so a quick way to determine which are available (though it may contain a handful of false hits that match the name but aren't metapackages) is to match on the package name with:

```
$ apt-cache search --names-only ^task-
```

In addition to these, you will find other metapackages with various purposes. Some are subsets of broader task packages, like `gnome-core`, while others are individual specialized parts of a Debian Pure Blend, such as the `education-*` metapackages. To list all metapackages in the archive, install the `debtags` package and list all packages with the `role::metapackage` tag as follows:

```
$ debtags search role::metapackage
```

8.2.3 Local package lists

Whether you list metapackages, individual packages, or a combination of both, all local package lists are stored in `config/package-lists/`. Since more than one list can be used,

this lends itself well to modular designs. For example, you may decide to devote one list to a particular choice of desktop, another to a collection of related packages that might as easily be used on top of a different desktop. This allows you to experiment with different combinations of sets of packages with a minimum of fuss, sharing common lists between different live image projects.

Package lists that exist in this directory need to have a `.list` suffix in order to be processed, and then an additional stage suffix, `.chroot` or `.binary` to indicate which stage the list is for.

Note: If you don't specify the stage suffix, the list will be used for both stages. Normally, you want to specify `.list.chroot` so that the packages will only be installed in the live filesystem and not have an extra copy of the `.deb` placed on the medium.

8.2.4 Local binary package lists

To make a binary stage list, place a file suffixed with `.list.binary` in `config/package-lists/`. These packages are not installed in the live filesystem, but are included on the live medium under `pool/`. You would typically use such a list with one of the non-live installer variants. As mentioned above, if you want this list to be the same as your chroot stage list, simply use the `.list` suffix by itself.

8.2.5 Generated package lists

It sometimes happens that the best way to compose a list is to generate it with a script. Any line starting with an exclamation point indicates a command to be executed within the chroot when the image is built. For example, one might include the line `! grep-aptavail -n -sPackage -FPriority standard |sort` in a package list to produce a sorted list of available packages with `Priority: standard`.

In fact, selecting packages with the `grep-aptavail` command (from the `dctrl-tools` package) is so useful that `live-build` provides a `Packages` helper script as a convenience. This script takes two arguments: `field` and `pattern`. Thus, you can create a list with the following contents:

```
$ lb config
$ echo '! Packages Priority standard' > config/package-lists/standard.list.chroot
```

8.2.6 Using conditionals inside package lists

Any of the *live-build* configuration variables stored in `config/*` (minus the `LB_` prefix) may

be used in conditional statements in package lists. Generally, this means any `lb config` option uppercased and with dashes changed to underscores. But in practice, it is only the ones that influence package selection that make sense, such as `DISTRIBUTION`, `ARCHITECTURES` or `ARCHIVE_AREAS`.

For example, to install `ia32-libs` if the `--architectures amd64` is specified:

```
#if ARCHITECTURES amd64
ia32-libs
#endif
```

You may test for any one of a number of values, e.g. to install *memtest86+* if either `--architectures i386` or `--architectures amd64` is specified:

```
#if ARCHITECTURES i386 amd64
memtest86+
#endif
```

You may also test against variables that may contain more than one value, e.g. to install *vrms* if either `contrib` or `non-free` is specified via `--archive-areas`:

```
#if ARCHIVE_AREAS contrib non-free
vrms
#endif
```

The nesting of conditionals is not supported.

8.2.7 Removing packages at install time

You can list packages in files with `.list.chroot_live` and `.list.chroot_install` suffixes inside the `config/package-lists` directory. If both a live and an install list exist, the packages in the `.list.chroot_live` list are removed with a hook after the installation (if the user uses the installer). The packages in the `.list.chroot_install` list are present both in the live system and in the installed system. This is a special tweak for the installer and may be useful if you have `--debian-installer live` set in your config, and wish to remove live system-specific packages at install time.

8.2.8 Desktop and language tasks

Desktop and language tasks are special cases that need some extra planning and configuration. Live images are different from Debian Installer images in this respect. In the Debian Installer, if the medium was prepared for a particular desktop environment flavour, the corresponding task will be automatically installed. Thus, there are internal `gnome-desktop`, `kde-desktop`, `lxde-desktop` and `xfce-desktop` tasks, none of which are offered in `tasksel`'s menu. Likewise, there are no menu entries for tasks for languages, but the user's language choice during the install influences the selection of corresponding language tasks.

When developing a desktop live image, the image typically boots directly to a working desktop, the choices of both desktop and default language having been made at build time, not at run time as in the case of the Debian Installer. That's not to say that a live image couldn't be built to support multiple desktops or multiple languages and offer the user a choice, but that is not *live-build*' s default behaviour.

Because there is no provision made automatically for language tasks, which include such things as language-specific fonts and input-method packages, if you want them, you need to specify them in your configuration. For example, a GNOME desktop image containing support for German might include these task metapackages:

```
$ lb config
$ echo "task-gnome-desktop task-laptop" >> config/package-lists/my.list.chroot
$ echo "task-german task-german-desktop task-german-gnome-desktop" >> config/package-lists/my.list.chroot
```

8.2.9 Kernel flavour and version

One or more kernel flavours will be included in your image by default, depending on the architecture. You can choose different flavours via the `--linux-flavours` option. Each flavour is suffixed to the default stub `linux-image` to form each metapackage name which in turn depends on an exact kernel package to be included in your image.

Thus by default, an `amd64` architecture image will include the `linux-image-amd64` flavour metapackage, and an `i386` architecture image will include the `linux-image-486` and `linux-image-686-pae` metapackages. At time of writing, these packages depend on `linux-image-3.2.0-4-amd64`, `linux-image-3.2.0-4-486` and `linux-image-3.2.0-4-686-pae`, respectively.

When more than one kernel package version is available in your configured archives, you can specify a different kernel package name stub with the `--linux-packages` option. For

example, supposing you are building an `amd64` architecture image and add the experimental archive for testing purposes so you can install the `linux-image-3.7-trunk-amd64` kernel. You would configure that image as follows:

```
$ lb config --linux-packages linux-image-3.7-trunk
$ echo "deb http://ftp.debian.org/debian/ experimental main" > config/archives/experimental.list.chroot
```

8.2.10 Custom kernels

You can build and include your own custom kernels, so long as they are integrated within the Debian package management system. The *live-build* system does not support kernels not built as `.deb` packages.

The proper and recommended way to deploy your own kernel packages is to follow the instructions in the `kernel-handbook`. Remember to modify the ABI and flavour suffixes appropriately, then include a complete build of the `linux` and matching `linux-latest` packages in your reposistory.

If you opt to build the kernel packages without the matching metapackages, you need to specify an appropriate `--linux-packages` stub as discussed in ‹Kernel flavour and version›. As we explain in ‹Installing modified or third-party packages›, it is best if you include your custom kernel packages in your own repository, though the alternatives discussed in that section work as well.

It is beyond the scope of this document to give advice on how to customize your kernel. However, you must at least ensure your configuration satisfies these minimum requirements:

- Use an initial ramdisk.
- Include the union filesystem module (i.e. usually `aufs`).
- Include any other filesystem modules required by your configuration (i.e. usually `squashfs`).

8.3 Installing modified or third-party packages

While it is against the philosophy of a live system, it may sometimes be necessary to build a live system with modified versions of packages that are in the Debian repository. This may be to modify or support additional features, languages and branding, or even to remove elements of existing packages that are undesirable. Similarly, “third-party” packages may be used to add bespoke and/or proprietary functionality.

This section does not cover advice regarding building or maintaining modified packages.

Joachim Breitner's `How to fork privately' method from ‹http://www.joachim-breitner.de/blog/archives/282-How-to-fork-privately.html› may be of interest, however. The creation of bespoke packages is covered in the Debian New Maintainers' Guide at ‹http://www.debian.org/doc/maint-guide/› and elsewhere.

There are two ways of installing modified custom packages:

- `packages.chroot`
- Using a custom APT repository

Using `packages.chroot` is simpler to achieve and useful for “one-off” customizations but has a number of drawbacks, while using a custom APT repository is more time-consuming to set up.

8.3.1 Using `packages.chroot` to install custom packages

To install a custom package, simply copy it to the `config/packages.chroot/` directory. Packages that are inside this directory will be automatically installed into the live system during build - you do not need to specify them elsewhere.

Packages **must** be named in the prescribed way. One simple way to do this is to use `dpkg-name`.

Using `packages.chroot` for installation of custom packages has disadvantages:

- It is not possible to use secure APT.
- You must install all appropriate packages in the `config/packages.chroot/` directory.
- It does not lend itself to storing live system configurations in revision control.

8.3.2 Using an APT repository to install custom packages

Unlike using `packages.chroot`, when using a custom APT repository you must ensure that you specify the packages elsewhere. See ‹Choosing packages to install› for details.

While it may seem unnecessary effort to create an APT repository to install custom packages, the infrastructure can be easily re-used at a later date to offer updates of the modified packages.

8.3.3 Custom packages and APT

live-build uses APT to install all packages into the live system so will therefore inherit behaviours from this program. One relevant example is that (assuming a default configuration) given a package available in two different repositories with different version numbers, APT will elect to install the package with the higher version number.

Because of this, you may wish to increment the version number in your custom packages' `debian/changelog` files to ensure that your modified version is installed over one in the official Debian repositories. This may also be achieved by altering the live system's APT pinning preferences - see ‹APT pinning› for more information.

8.4 Configuring APT at build time

You can configure APT through a number of options applied only at build time. (APT configuration used in the running live system may be configured in the normal way for live system contents, that is, by including the appropriate configurations through `config/includes.chroot/`.) For a complete list, look for options starting with `apt` in the `lb_config` man page.

8.4.1 Choosing apt or aptitude

You can elect to use either *apt* or *aptitude* when installing packages at build time. Which utility is used is governed by the `--apt` argument to `lb config`. Choose the method implementing the preferred behaviour for package installation, the notable difference being how missing packages are handled.

- `apt`: With this method, if a missing package is specified, the package installation will fail. This is the default setting.
- `aptitude`: With this method, if a missing package is specified, the package installation will succeed.

8.4.2 Using a proxy with APT

One commonly required APT configuration is to deal with building an image behind a proxy. You may specify your APT proxy with the `--apt-ftp-proxy` or `--apt-http-proxy` options as needed, e.g.

```
$ lb config --apt-http-proxy http://proxy/
```

8.4.3 Tweaking APT to save space

You may find yourself needing to save some space on the image medium, in which case one or the other or both of the following options may be of interest.

If you don't want to include APT indices in the image, you can omit those with:

```
$ lb config --apt-indices false
```

This will not influence the entries in `/etc/apt/sources.list`, but merely whether `/var/-lib/apt` contains the indices files or not. The tradeoff is that APT needs those indices in order to operate in the live system, so before performing `apt-cache search` or `apt-get install`, for instance, the user must `apt-get update` first to create those indices.

If you find the installation of recommended packages bloats your image too much, provided you are prepared to deal with the consequences discussed below, you may disable that default option of APT with:

```
$ lb config --apt-recommends false
```

The most important consequence of turning off recommends is that `live-boot` and `live-config` themselves recommend some packages that provide important functionality used by most Live configurations, such as `user-setup` which `live-config` recommends and is used to create the live user. In all but the most exceptional circumstances you need to add back at least some of these recommends to your package lists or else your image will not work as expected, if at all. Look at the recommended packages for each of the `live-*` packages included in your build and if you are not certain you can omit them, add them back into your package lists.

The more general consequence is that if you don't install recommended packages for any given package, that is, “packages that would be found together with this one in all but unusual installations” (Debian Policy Manual, section 7.2), some packages that users of your Live system actually need may be omitted. Therefore, we suggest you review the difference turning off recommends makes to your packages list (see the `binary.packages` file generated by `lb build`) and re-include in your list any missing packages that you still want installed. Alternatively, if you find you only want a small number of recommended packages left out, leave recommends enabled and set a negative APT pin priority on selected packages to prevent them from being installed, as explained in ‹APT pinning›.

8.4.4 Passing options to apt or aptitude

If there is not a `lb config` option to alter APT's behaviour in the way you need, use `--apt-options` or `--aptitude-options` to pass any options through to your configured APT tool. See the man pages for `apt` and `aptitude` for details. Note that both options have default values that you will need to retain in addition to any overrides you may provide. So, for example, suppose you have included something from `snapshot.debian.org` for testing

purposes and want to specify `Acquire::Check-Valid-Until=false` to make APT happy with the stale `Release` file, you would do so as per the following example, appending the new option after the default value `--yes`:

```
$ lb config --apt-options "--yes -oAcquire::Check-Valid-Until=false"
```

Please check the man pages to fully understand these options and when to use them. This is an example only and should not be construed as advice to configure your image this way. This option would not be appropriate for, say, a final release of a live image.

For more complicated APT configurations involving `apt.conf` options you might want to create a `config/apt/apt.conf` file instead. See also the other `apt-*` options for a few convenient shortcuts for frequently needed options.

8.4.5 APT pinning

For background, please first read the `apt_preferences(5)` man page. APT pinning can be configured either for build time, or else for run time. For the former, create `config/archives/*.pref`, `config/archives/*.pref.chroot`, and `config/apt/preferences`. For the latter, create `config/includes.chroot/etc/apt/preferences`.

Let's say you are building a **jessie** live system but need all the live packages that end up in the binary image to be installed from **sid** at build time. You need to add **sid** to your APT sources and pin the live packages from it higher, but all other packages from it lower, than the default priority. Thus, only the packages you want are installed from **sid** at build time and all others are taken from the target system distribution, **jessie** . The following will accomplish this:

```
$ echo "deb http://mirror/debian/ sid main" > config/archives/sid.list.chroot
$ cat >> config/archives/sid.pref.chroot << EOF
Package: live-*
Pin: release n=sid
Pin-Priority: 600

Package: *
Pin: release n=sid
Pin-Priority: 1
EOF
```

Negative pin priorities will prevent a package from being installed, as in the case where you do not want a package that is recommended by another package. Suppose you are building an LXDE image using `task-lxde-desktop` in `config/package-lists/desktop.list.chroot`, but don't want the user prompted to store wifi passwords in the keyring. This

metapackage depends on *lxde-core*, which recommends *gksu*, which in turn recommends *gnome-keyring*. So you want to omit the recommended *gnome-keyring* package. This can be done by adding the following stanza to `config/apt/preferences`:

```
Package: gnome-keyring
Pin: version *
Pin-Priority: -1
```

9. Customizing contents

This chapter discusses fine-tuning customization of the live system contents beyond merely choosing which packages to include. Includes allow you to add or replace arbitrary files in your live system image, hooks allow you to execute arbitrary commands at different stages of the build and at boot time, and preseeding allows you to configure packages when they are installed by supplying answers to debconf questions.

9.1 Includes

While ideally a live system would include files entirely provided by unmodified packages, it is sometimes convenient to provide or modify some content by means of files. Using includes, it is possible to add (or replace) arbitrary files in your live system image. *live-build* provides two mechanisms for using them:

- Chroot local includes: These allow you to add or replace files to the chroot/Live filesystem. Please see ‹Live/chroot local includes› for more information.
- Binary local includes: These allow you to add or replace files in the binary image. Please see ‹Binary local includes› for more information.

Please see ‹Terms› for more information about the distinction between the “Live” and “binary” images.

9.1.1 Live/chroot local includes

Chroot local includes can be used to add or replace files in the chroot/Live filesystem so that they may be used in the Live system. A typical use is to populate the skeleton user directory (`/etc/skel`) used by the Live system to create the live user's home directory. Another is to supply configuration files that can be simply added or replaced in the image without processing; see ‹Live/chroot local hooks› if processing is needed.

To include files, simply add them to your `config/includes.chroot` directory. This directory

corresponds to the root directory `/` of the live system. For example, to add a file `/var/www/index.html` in the live system, use:

```
$ mkdir -p config/includes.chroot/var/www
$ cp /path/to/my/index.html config/includes.chroot/var/www
```

Your configuration will then have the following layout:

```
-- config
    [...]
     |-- includes.chroot
     |   `-- var
     |       `-- www
     |           `-- index.html
    [...]
```

Chroot local includes are installed after package installation so that files installed by packages are overwritten.

9.1.2 Binary local includes

To include material such as documentation or videos on the medium filesystem so that it is accessible immediately upon insertion of the medium without booting the Live system, you can use binary local includes. This works in a similar fashion to chroot local includes. For example, suppose the files `~/video_demo.*` are demo videos of the live system described by and linked to by an HTML index page. Simply copy the material to `config/includes.binary/` as follows:

```
$ cp ~/video_demo.* config/includes.binary/
```

These files will now appear in the root directory of the live medium.

9.2 Hooks

Hooks allow commands to be performed in the chroot and binary stages of the build in order to customize the image.

9.2.1 Live/chroot local hooks

To run commands in the chroot stage, create a hook script with a `.hook.chroot` suffix containing the commands in the `config/hooks/` directory. The hook will run in the chroot after the rest of your chroot configuration has been applied, so remember to ensure your configuration includes all packages and files your hook needs in order to run. See the example chroot hook scripts for various common chroot customization tasks provided in `/usr/share/doc/live-build/examples/hooks` which you can copy or symlink to use them in your own configuration.

9.2.2 Boot-time hooks

To execute commands at boot time, you can supply *live-config* hooks as explained in the “Customization” section of its man page. Examine *live-config*' s own hooks provided in `/lib/live/config/`, noting the sequence numbers. Then provide your own hook prefixed with an appropriate sequence number, either as a chroot local include in `config/includes.chroot/lib/live/config/`, or as a custom package as discussed in ‹Installing modified or third-party packages›.

9.2.3 Binary local hooks

To run commands in the binary stage, create a hook script with a `.hook.binary` suffix containing the commands in the `config/hooks/` directory. The hook will run after all other binary commands are run, but before binary_checksums, the very last binary command. The commands in your hook do not run in the chroot, so take care to not modify any files outside of the build tree, or you may damage your build system! See the example binary hook scripts for various common binary customization tasks provided in `/usr/share/doc/live-build/examples/hooks` which you can copy or symlink to use them in your own configuration.

9.3 Preseeding Debconf questions

Files in the `config/preseed/` directory suffixed with `.cfg` followed by the stage (`.chroot` or `.binary`) are considered to be debconf preseed files and are installed by *live-build* using `debconf-set-selections` during the corresponding stage.

For more information about debconf, please see `debconf(7)` in the *debconf* package.

10. Customizing run time behaviours

All configuration that is done during run time is done by *live-config*. Here are some of the

most common options of *live-config* that users are interested in. A full list of all possibilities can be found in the man page of *live-config*.

10.1 Customizing the live user

One important consideration is that the live user is created by *live-boot* at boot time, not by *live-build* at build time. This not only influences where materials relating to the live user are introduced in your build, as discussed in ‹Live/chroot local includes›, but also any groups and permissions associated with the live user.

You can specify additional groups that the live user will belong to by using any of the possibilities to configure *live-config*. For example, to add the live user to the `fuse` group, you can either add the following file in `config/includes.chroot/etc/live/config/user-setup.conf`:

```
LIVE_USER_DEFAULT_GROUPS="audio cdrom dip floppy video plugdev netdev powerdev scanner bluetooth fuse"
```

or use `live-config.user-default-groups=audio,cdrom,dip,floppy,video,plugdev,netdev,powerde` as a boot parameter.

It is also possible to change the default username “user” and the default password “live”. If you want to do that for any reason, you can easily achieve it as follows:

To change the default username you can simply specify it in your config:

```
$ lb config --bootappend-live "boot=live components username=live-user"
```

One possible way of changing the default password is by means of a hook as described in ‹Boot-time hooks›. In order to do that you can use the “passwd” hook from `/usr/share/doc/live-config/examples/hooks`, prefix it accordingly (e.g. 2000-passwd) and add it to `config/includes.chroot/lib/live/config/`

10.2 Customizing locale and language

When the live system boots, language is involved in two steps:

- the locale generation
- setting the keyboard configuration

The default locale when building a Live system is `locales=en_US.UTF-8`. To define the locale that should be generated, use the `locales` parameter in the `--bootappend-live` option of `lb config`, e.g.

```
$ lb config --bootappend-live "boot=live components locales=de_CH.UTF-8"
```

Multiple locales may be specified as a comma-delimited list.

This parameter, as well as the keyboard configuration parameters indicated below, can also be used at the kernel command line. You can specify a locale by `language_country` (in which case the default encoding is used) or the full `language_country.encoding` word. A list of supported locales and the encoding for each can be found in `/usr/share/i18n/SUPPORTED`.

Both the console and X keyboard configuration are performed by `live-config` using the `console-setup` package. To configure them, use the `keyboard-layouts`, `keyboard-variants`, `keyboard-options` and `keyboard-model` boot parameters via the `--bootappend-live` option. Valid options for these can be found in `/usr/share/X11/xkb/rules/base.lst`. To find layouts and variants for a given language, try searching for the English name of the language and/or the country where the language is spoken, e.g:

```
$ egrep -i '(^!|german.*switzerland)' /usr/share/X11/xkb/rules/base.lst
! model
! layout
   ch              German (Switzerland)
! variant
   legacy          ch: German (Switzerland, legacy)
   de_nodeadkeys   ch: German (Switzerland, eliminate dead keys)
   de_sundeadkeys  ch: German (Switzerland, Sun dead keys)
   de_mac          ch: German (Switzerland, Macintosh)
! option
```

Note that each variant lists the layout to which it applies in the description.

Often, only the layout needs to be configured. For example, to get the locale files for German and Swiss German keyboard layout in X use:

```
$ lb config --bootappend-live "boot=live components locales=de_CH.UTF-8 keyboard-layouts=ch"
```

However, for very specific use cases, you may wish to include other parameters. For example, to set up a French system with a French-Dvorak layout (called Bepo) on a TypeMatrix EZ-Reach 2030 USB keyboard, use:

```
$ lb config --bootappend-live \
     "boot=live components locales=fr_FR.UTF-8 keyboard-layouts=fr keyboard-variants=bepo keyboard-model=↩
          tm2030usb"
```

Multiple values may be specified as comma-delimited lists for each of the `keyboard-*` options, with the exception of `keyboard-model`, which accepts only one value. Please see the `keyboard(5)` man page for details and examples of `XKBMODEL`, `XKBLAYOUT`, `XKBVARIANT` and `XKBOPTIONS` variables. If multiple `keyboard-variants` values are given, they will be matched one-to-one with `keyboard-layouts` values (see `setxkbmap(1) -variant` option). Empty values are allowed; e.g. to define two layouts, the default being US QWERTY and the other being US Dvorak, use:

```
$ lb config --bootappend-live \
     "boot=live components keyboard-layouts=us,us keyboard-variants=,dvorak"
```

10.3 Persistence

A live cd paradigm is a pre-installed system which runs from read-only media, like a cdrom, where writes and modifications do not survive reboots of the host hardware which runs it.

A live system is a generalization of this paradigm and thus supports other media in addition to CDs; but still, in its default behaviour, it should be considered read-only and all the run-time evolutions of the system are lost at shutdown.

`Persistence' is a common name for different kinds of solutions for saving across reboots some, or all, of this run-time evolution of the system. To understand how it works it would be handy to know that even if the system is booted and run from read-only media, modifications to the files and directories are written on writable media, typically a ram disk (tmpfs) and ram disks' data do not survive reboots.

The data stored on this ramdisk should be saved on a writable persistent medium like local storage media, a network share or even a session of a multisession (re)writable CD/DVD. All these media are supported in live systems in different ways, and all but the last one require a special boot parameter to be specified at boot time: `persistence`.

If the boot parameter `persistence` is set (and `nopersistence` is not set), local storage media (e.g. hard disks, USB drives) will be probed for persistence volumes during boot. It is possible to restrict which types of persistence volumes to use by specifying certain boot parameters described in the *live-boot*(7) man page. A persistence volume is any of the following:

- a partition, identified by its GPT name.
- a filesystem, identified by its filesystem label.
- an image file located on the root of any readable filesystem (even an NTFS partition of a foreign OS), identified by its filename.

The volume label for overlays must be `persistence` but it will be ignored unless it contains in its root a file named `persistence.conf` which is used to fully customize the volume's persistence, this is to say, specifying the directories that you want to save in your persistence volume after a reboot. See ‹The persistence.conf file› for more details.

Here are some examples of how to prepare a volume to be used for persistence. It can be, for instance, an ext4 partition on a hard disk or on a usb key created with, e.g.:

```
# mkfs.ext4 -L persistence /dev/sdb1
```

See also ‹Using the space left on a USB stick›.

If you already have a partition on your device, you could just change the label with one of the following:

```
# tune2fs -L persistence /dev/sdb1 # for ext2,3,4 filesystems
```

Here's an example of how to create an ext4-based image file to be used for persistence:

```
$ dd if=/dev/null of=persistence bs=1 count=0 seek=1G # for a 1GB sized image file
$ /sbin/mkfs.ext4 -F persistence
```

Once the image file is created, as an example, to make `/usr` persistent but only saving the changes you make to that directory and not all the contents of `/usr`, you can use the “union” option. If the image file is located in your home directory, copy it to the root of your hard drive's filesystem and mount it in `/mnt` as follows:

```
# cp persistence /
# mount -t ext4 /persistence /mnt
```

Then, create the `persistence.conf` file adding content and unmount the image file.

```
# echo "/usr union" >> /mnt/persistence.conf
# umount /mnt
```

Now, reboot into your live medium with the boot parameter “persistence”.

10.3.1 The persistence.conf file

A volume with the label `persistence` must be configured by means of the `persistence.conf` file to make arbitrary directories persistent. That file, located on the volume's filesystem root, controls which directories it makes persistent, and in which way.

How custom overlay mounts are configured is described in full detail in the persistence.conf(5) man page, but a simple example should be sufficient for most uses. Let's say we want to make our home directory and APT cache persistent in an ext4 filesystem on the /dev/sdb1 partition:

```
# mkfs.ext4 -L persistence /dev/sdb1
# mount -t ext4 /dev/sdb1 /mnt
# echo "/home" >> /mnt/persistence.conf
# echo "/var/cache/apt" >> /mnt/persistence.conf
# umount /mnt
```

Then we reboot. During the first boot the contents of `/home` and `/var/cache/apt` will be copied into the persistence volume, and from then on all changes to these directories will live in the persistence volume. Please note that any paths listed in the `persistence.conf` file cannot contain white spaces or the special `.` and `..` path components. Also, neither `/lib`, `/lib/live` (or any of their sub-directories) nor `/` can be made persistent using custom mounts. As a workaround for this limitation you can add `/ union` to your `persistence.conf` file to achieve full persistence.

10.3.2 Using more than one persistence store

There are different methods of using multiple persistence store for different use cases. For instance, using several volumes at the same time or selecting only one, among various, for very specific purposes.

Several different custom overlay volumes (with their own `persistence.conf` files) can be used at the same time, but if several volumes make the same directory persistent, only one of them will be used. If any two mounts are “nested” (i.e. one is a sub-directory of the

other) the parent will be mounted before the child so no mount will be hidden by the other. Nested custom mounts are problematic if they are listed in the same `persistence.conf` file. See the persistence.conf(5) man page for how to handle that case if you really need it (hint: you usually don't).

One possible use case: If you wish to store the user data i.e. `/home` and the superuser data i.e. `/root` in different partitions, create two partitions with the `persistence` label and add a `persistence.conf` file in each one like this, `# echo "/home" > persistence.conf` for the first partition that will save the user's files and `# echo "/root" > persistence.conf` for the second partition which will store the superuser's files. Finally, use the `persistence` boot parameter.

If a user would need multiple persistence store of the same type for different locations or testing, such as `private` and `work`, the boot parameter `persistence-label` used in conjunction with the boot parameter `persistence` will allow for multiple but unique persistence media. An example would be if a user wanted to use a persistence partition labeled `private` for personal data like browser bookmarks or other types, they would use the boot parameters: `persistence persistence-label=private`. And to store work related data, like documents, research projects or other types, they would use the boot parameters: `persistence persistence-label=work`.

It is important to remember that each of these volumes, `private` and `work`, also needs a `persistence.conf` file in its root. The *live-boot* man page contains more information about how to use these labels with legacy names.

11. Customizing the binary image

11.1 Bootloaders

live-build uses *syslinux* and some of its derivatives (depending on the image type) as bootloaders by default. They can be easily customized to suit your needs.

In order to use a full theme, copy `/usr/share/live/build/bootloaders` into `config/bootloaders` and edit the files in there. If you do not want to bother modifying all supported bootloader configurations, only providing a local customized copy of one of the bootloaders, e.g. **isolinux** in `config/bootloaders/isolinux` is enough too, depending on your use case.

There are many possibilities when it comes to making changes. For instance, syslinux derivatives are configured by default with a timeout of 0 (zero) which means that they will pause indefinitely at their splash screen until you press a key.

To modify the boot timeout of a default `iso-hybrid` image just edit a default **isolinux.cfg** file specifying the timeout in units of 1/10 seconds. A modified **isolinux.cfg** to boot after five seconds would be similar to this:

```
include menu.cfg
default vesamenu.c32
prompt 0
timeout 50
```

If you want to use a personalized background image that will be displayed together with the boot menu, add a splash.png picture of 640x480 pixels.

11.2 ISO metadata

When creating an ISO9660 binary image, you can use the following options to add various textual metadata for your image. This can help you easily identify the version or configuration of an image without booting it.

- `LB_ISO_APPLICATION/--iso-application NAME`: This should describe the application that will be on the image. The maximum length for this field is 128 characters.
- `LB_ISO_PREPARER/--iso-preparer NAME`: This should describe the preparer of the image, usually with some contact details. The default for this option is the *live-build* version you are using, which may help with debugging later. The maximum length for this field is 128 characters.
- `LB_ISO_PUBLISHER/--iso-publisher NAME`: This should describe the publisher of the image, usually with some contact details. The maximum length for this field is 128 characters.
- `LB_ISO_VOLUME/--iso-volume NAME`: This should specify the volume ID of the image. This is used as a user-visible label on some platforms such as Windows and Apple Mac OS. The maximum length for this field is 32 characters.

12. Customizing Debian Installer

Live system images can be integrated with Debian Installer. There are a number of different types of installation, varying in what is included and how the installer operates.

Please note the careful use of capital letters when referring to the “Debian Installer” in this section - when used like this we refer explicitly to the official installer for the Debian system, not anything else. It is often seen abbreviated to “d-i”.

12.1 Types of Debian Installer

The three main types of installer are:

“Normal” Debian Installer : This is a normal live system image with a separate kernel and initrd which (when selected from the appropriate bootloader) launches into a standard Debian Installer instance, just as if you had downloaded a CD image of Debian and booted it. Images containing a live system and such an otherwise independent installer are often referred to as “combined images”.

On such images, Debian is installed by fetching and installing .deb packages using *debootstrap*, from local media or some network-based network, resulting in a default Debian system being installed to the hard disk.

This whole process can be preseeded and customized in a number of ways; see the relevant pages in the Debian Installer manual for more information. Once you have a working preseeding file, *live-build* can automatically put it in the image and enable it for you.

“Live” Debian Installer : This is a live system image with a separate kernel and initrd which (when selected from the appropriate bootloader) launches into an instance of the Debian Installer.

Installation will proceed in an identical fashion to the “normal” installation described above, but at the actual package installation stage, instead of using *debootstrap* to fetch and install packages, the live filesystem image is copied to the target. This is achieved with a special udeb called *live-installer*.

After this stage, the Debian Installer continues as normal, installing and configuring items such as bootloaders and local users, etc.

Note: to support both normal and live installer entries in the bootloader of the same live medium, you must disable *live-installer* by preseeding `live-installer/enable=false`.

“Desktop” Debian Installer : Regardless of the type of Debian Installer included, `d-i` can be launched from the Desktop by clicking on an icon. This is user friendlier in some situations. In order to make use of this, the *debian-installer-launcher* package needs to be included.

Note that by default, *live-build* does not include Debian Installer images in the images, it needs to be specifically enabled with `lb config`. Also, please note that for the “Desktop” installer to work, the kernel of the live system must match the kernel `d-i` uses for the specified architecture. For example:

```
$ lb config --architectures i386 --linux-flavours 486 \
         --debian-installer live
$ echo debian-installer-launcher >> config/package-lists/my.list.chroot
```

12.2 Customizing Debian Installer by preseeding

As described in the Debian Installer Manual, Appendix B at ‹http://www.debian.org/releases/stable/i386/apb.html›, “Preseeding provides a way to set answers to questions asked during the installation process, without having to manually enter the answers while the installation is running. This makes it possible to fully automate most types of installation and even offers some features not available during normal installations.” This kind of customization is best accomplished with *live-build* by placing the configuration in a `preseed.cfg` file included in `config/includes.installer/`. For example, to preseed setting the locale to `en_US`:

```
$ echo "d-i debian-installer/locale string en_US" \
        >> config/includes.installer/preseed.cfg
```

12.3 Customizing Debian Installer content

For experimental or debugging purposes, you might want to include locally built `d-i` component udeb packages. Place these in `config/packages.binary/` to include them in the image. Additional or replacement files and directories may be included in the installer initrd as well, in a similar fashion to ‹Live/chroot local includes›, by placing the material in `config/includes.installer/`.

Project

13. Contributing to the project

When submitting a contribution, please clearly identify its copyright holder and include any applicable licensing statement. Note that to be accepted, the contribution must be licensed under the same license as the rest of the documents, namely, GPL version 3 or later.

Contributions to the project, such as translations and patches, are greatly welcome. Anyone can directly commit to the repositories, however, we ask you to send bigger changes to the mailing list to discuss them first. See the section ‹Contact› for more information.

The Live Systems Project uses Git as version control system and source code management. As explained in ‹Git repositories› there are two main development branches: **debian** and **debian-next** . Everybody can commit to the debian-next branches of the *live-boot*, *live-build*, *live-config*, *live-images*, *live-manual* and *live-tools* repositories.

However, there are certain restrictions. The server will reject:

- Non fast-forward pushes.

- Merge commits.
- Adding or removing tags or branches.

Even though all commits might be revised, we ask you to use your common sense and make good commits with good commit messages.

- Write commit messages that consist of complete, meaningful sentences in English, starting with a capital letter and ending with a full stop. Usually, these will start with the form “Fixing/Adding/Removing/Correcting/Translating/...”.
- Write good commit messages. The first line must be an accurate summary of the contents of the commit which will be included in the changelog. If you need to make some further explanations, write them below leaving a blank line after the first one and then another blank line after each paragraph. Lines of paragraphs should not exceed 80 characters in length.
- Commit atomically, this is to say, do not mix unrelated things in the same commit. Make one different commit for each change you make.

13.1 Making changes

In order to push to the repositories, you must follow the following procedure. Here we use *live-manual* as an example so replace it with the name of the repository you want to work with. For detailed information on how to edit *live-manual* see ‹Contributing to this document›.

- Fetch the public commit key:

```
$ mkdir -p ~/.ssh/keys
$ wget http://live-systems.org/other/keys/git@live-systems.org -O ~/.ssh/keys/git@live-systems.org
$ wget http://live-systems.org/other/keys/git@live-systems.org.pub -O ~/.ssh/keys/git@live-systems.org.pub
$ chmod 0600 ~/.ssh/keys/git@live-systems.org*
```

- Add the following section to your openssh-client config:

```
$ cat >> ~/.ssh/config << EOF
Host live-systems.org
     Hostname live-systems.org
     User git
     IdentitiesOnly yes
     IdentityFile ~/.ssh/keys/git@live-systems.org
EOF
```

- Check out a clone of *live-manual* through ssh:

```
$ git clone git@live-systems.org:/live-manual.git
$ cd live-manual && git checkout debian-next
```

- Make sure you have Git author and email set:

```
  $ git config user.name "John Doe"
  $ git config user.email john@example.org
```

Important: Remember that you should commit any changes on the **debian-next** branch.

- Make your changes. In this example you would first write a new section dealing with applying patches and then prepare to commit adding the files and writing your commit message like this:

```
$ git commit -a -m "Adding a section on applying patches."
```

- Push the commit to the server:

```
$ git push
```

14. Reporting bugs

Live systems are far from being perfect, but we want to make it as close as possible to perfect - with your help. Do not hesitate to report a bug. It is better to fill a report twice than never. However, this chapter includes recommendations on how to file good bug reports.

For the impatient:

- Always check first the image status updates on our homepage at <http://live-systems.org/> for known issues.
- Before submitting a bug report always try to reproduce the bug with the **most recent versions** of the branch of *live-build*, *live-boot*, *live-config* and *live-tools* that you're using (like the newest 4.x version of *live-build* if you're using *live-build* 4).

- Try to give **as specific information as possible** about the bug. This includes (at least) the version of *live-build*, *live-boot*, *live-config*, and *live-tools* used and the distribution of the live system you are building.

14.1 Known issues

Since Debian **testing** and Debian **unstable** distributions are moving targets, when you specify either of them as the target system distribution, a successful build may not always be possible.

If this causes too much difficulty for you, do not build a system based on **testing** or **unstable** , but rather, use **stable** . *live-build* always defaults to the **stable** release.

Currently known issues are listed under the section `status' on our homepage at ‹`http://live-systems.org/`›.

It is out of the scope of this manual to train you to correctly identify and fix problems in packages of the development distributions, however, there are two things you can always try: If a build fails when the target distribution is **testing** , try **unstable** . If **unstable** does not work either, revert to **testing** and pin the newer version of the failing package from **unstable** (see ‹APT pinning› for details).

14.2 Rebuild from scratch

To ensure that a particular bug is not caused by an uncleanly built system, please always rebuild the whole live system from scratch to see if the bug is reproducible.

14.3 Use up-to-date packages

Using outdated packages can cause significant problems when trying to reproduce (and ultimately fix) your problem. Make sure your build system is up-to-date and any packages included in your image are up-to-date as well.

14.4 Collect information

Please provide enough information with your report. Include, at least, the exact version of *live-build* where the bug is encountered and the steps to reproduce it. Please use your common sense and provide any other relevant information if you think that it might help in solving the problem.

To make the most out of your bug report, we require at least the following information:

- Architecture of the host system

- Distribution of the host system
- Version of *live-build* on the host system
- Version of Python on the host system
- Version of *debootstrap* and/or *cdebootstrap* on the host system
- Architecture of the live system
- Distribution of the live system
- Version of *live-boot* on the live system
- Version of *live-config* on the live system
- Version of *live-tools* on the live system

You can generate a log of the build process by using the `tee` command. We recommend doing this automatically with an `auto/build` script (see ‹Managing a configuration› for details).

```
# lb build 2>&1 | tee build.log
```

At boot time, *live-boot* and *live-config* store their logfiles in `/var/log/live/`. Check them for error messages.

Additionally, to rule out other errors, it is always a good idea to tar up your `config/` directory and upload it somewhere (do **not** send it as an attachment to the mailing list), so that we can try to reproduce the errors you encountered. If this is difficult (e.g. due to size) you can use the output of `lb config --dump` which produces a summary of your config tree (i.e. lists files in subdirectories of `config/` but does not include them).

Remember to send in any logs that were produced with English locale settings, e.g. run your *live-build* commands with a leading `LC_ALL=C` or `LC_ALL=en_US`.

14.5 Isolate the failing case if possible

If possible, isolate the failing case to the smallest possible change that breaks. It is not always easy to do this so if you cannot manage it for your report, do not worry. However, if you plan your development cycle well, using small enough change sets per iteration, you may be able to isolate the problem by constructing a simpler `base' configuration that closely matches your actual configuration plus just the broken change set added to it. If you have a hard time sorting out which of your changes broke, it may be that you are including too much in each change set and should develop in smaller increments.

14.6 Use the correct package to report the bug against

If you do not know what component is responsible for the bug or if the bug is a general bug concerning live systems, you can fill a bug against the debian-live pseudo-package.

However, we would appreciate it if you try to narrow it down according to where the bug appears.

14.6.1 At build time while bootstrapping

live-build first bootstraps a basic Debian system with *debootstrap* or *cdebootstrap*. Depending on the bootstrapping tool used and the Debian distribution it is bootstrapping, it may fail. If a bug appears here, check if the error is related to a specific Debian package (most likely), or if it is related to the bootstrapping tool itself.

In both cases, this is not a bug in the live system, but rather in Debian itself and probably we cannot fix it directly. Please report such a bug against the bootstrapping tool or the failing package.

14.6.2 At build time while installing packages

live-build installs additional packages from the Debian archive and depending on the Debian distribution used and the daily archive state, it can fail. If a bug appears here, check if the error is also reproducible on a normal system.

If this is the case, this is not a bug in the live system, but rather in Debian - please report it against the failing package. Running *debootstrap* separately from the Live system build or running `lb bootstrap --debug` will give you more information.

Also, if you are using a local mirror and/or any sort of proxy and you are experiencing a problem, please always reproduce it first by bootstrapping from an official mirror.

14.6.3 At boot time

If your image does not boot, please report it to the mailing list together with the information requested in ‹Collect information›. Do not forget to mention, how/when the image failed exactly, whether using virtualization or real hardware. If you are using a virtualization technology of any kind, please always run it on real hardware before reporting a bug. Providing a screenshot of the failure is also very helpful.

14.6.4 At run time

If a package was successfully installed, but fails while actually running the Live system,

this is probably a bug in the live system. However:

14.7 Do the research

Before filing the bug, please search the web for the particular error message or symptom you are getting. As it is highly unlikely that you are the only person experiencing a particular problem. There is always a chance that it has been discussed elsewhere and a possible solution, patch, or workaround has been proposed.

You should pay particular attention to the live systems mailing list, as well as the homepage, as these are likely to contain the most up-to-date information. If such information exists, always include the references to it in your bug report.

In addition, you should check the current bug lists for *live-build*, *live-boot*, *live-config* and *live-tools* to see whether something similar has already been reported.

14.8 Where to report bugs

The Live Systems project keeps track of all bugs in the Bug Tracking System (BTS). For information on how to use the system, please see ‹`http://bugs.debian.org/`›. You can also submit the bugs by using the `reportbug` command from the package with the same name.

In general, you should report build time errors against the *live-build* package, boot time errors against *live-boot*, and run time errors against *live-config*. If you are unsure of which package is appropriate or need more help before submitting a bug report, please report it against the debian-live pseudo-package. We will then take care about it and reassign it where appropriate.

Please note that bugs found in distributions derived from Debian (such as Ubuntu and others) should **not** be reported to the Debian BTS unless they can be also reproduced on a Debian system using official Debian packages.

15. Coding Style

This chapter documents the coding style used in live systems.

15.1 Compatibility

- Don't use syntax or semantics that are unique to the Bash shell. For example, the use of array constructs.
- Only use the POSIX subset - for example, use $(foo) over `foo`.
- You can check your scripts with `sh -n' and `checkbashisms'.

- Make sure all shell code runs with `set -e'.

15.2 Indenting

- Always use tabs over spaces.

15.3 Wrapping

- Generally, lines are 80 chars at maximum.
- Use the “Linux style” of line breaks:

Bad:

```
if foo; then
        bar
fi
```

Good:

```
if foo
then
        bar
fi
```

- The same holds for functions:

Bad:

```
Foo () {
        bar
}
```

Good:

```
Foo ()
{
        bar
}
```

15.4 Variables

- Variables are always in capital letters.
- Variables used in *live-build* always start with `LB_` prefix.
- Internal temporary variables in *live-build* should start with the `<=underscore>LB_` prefix.
- Local variables start with *live-build* `<=underscore><=underscore>LB_` prefix.
- Variables in connection to a boot parameter in *live-config* start with `LIVE_`.
- All other variables in *live-config* start with _ prefix.
- Use braces around variables; e.g. write `${FOO}` instead of `$FOO`.
- Always protect variables with quotes to respect potential whitespaces: write `"${FOO}"` not `${FOO}`.
- For consistency reasons, always use quotes when assigning values to variables:

Bad:

```
FOO=bar
```

Good:

```
FOO="bar"
```

- If multiple variables are used, quote the full expression:

Bad:

```
if [ -f "${FOO}"/foo/"${BAR}"/bar ]
then
         foobar
fi
```

Good:

```
if [ -f "${FOO}/foo/${BAR}/bar" ]
then
         foobar
fi
```

15.5 Miscellaneous

- Use "`|`" (without the surround quotes) as a separator in calls to sed, e.g. "`sed -e `s|'`" (without "").
- Don't use the `test` command for comparisons or tests, use "`[`" "`]`" (without ""); e.g. "`if [ -x /bin/foo ]; ...`" and not "`if test -x /bin/foo; ...`".
- Use `case` wherever possible over `test`, as it's easier to read and faster in execution.
- Use capitalized names for functions to limit messing with the users environment.

16. Procedures

This chapter documents the procedures within the Live Systems project for various tasks that need cooperation with other teams in Debian.

16.1 Major Releases

Releasing a new stable major version of Debian includes a lot of different teams working together to make it happen. At some point, the Live team comes in and builds live system images. The requirements to do this are:

- A mirror containing the released versions for the debian and debian-security archives which the debian-live buildd can access.
- The names of the image need to be known (e.g. debian-live-VERSION-ARCH-FLAVOUR.iso).
- The data from debian-cd needs to be synced (udeb exclude lists).
- Images are built and mirrored on cdimage.debian.org.

16.2 Point Releases

- Again, we need updated mirrors of debian and debian-security.
- Images are built and mirrored on cdimage.debian.org.
- Send announcement mail.

16.2.1 Last Point Release of a Debian Release

Remember to adjust both chroot and binary mirrors when building the last set of images for a Debian release after it has been moved away from ftp.debian.org to archive.debian.org. That way, old prebuilt live images are still useful without user modifications.

16.2.2 Point release announcement template

An annoucement mail for point releases can be generated using the template below and the following command:

```
$ sed \
     -e 's|@MAJOR@|7.0|g' \
     -e 's|@MINOR@|7.0.1|g' \
     -e 's|@CODENAME@|wheezy|g' \
     -e 's|@ANNOUNCE@|2013/msgXXXXX.html|g'
```

Please check the mail carefully before sending and pass it to others for proof-reading.

```
Updated Live @MAJOR@: @MINOR@ released

The Live Systems project is pleased to announce the @MINOR@ update of the
Live images for the stable distribution Debian @MAJOR@ (codename "@CODENAME@").

The images are available for download at:

   <http://live-systems.org/cdimage/release/current/>

and later at:

   <http://cdimage.debian.org/cdimage/release/current-live/>

This update includes the changes of the Debian @MINOR@ release:

   <http://lists.debian.org/debian-announce/@ANNOUNCE@>

Additionally it includes the following Live-specific changes:

  * [INSERT LIVE-SPECIFIC CHANGE HERE]
  * [INSERT LIVE-SPECIFIC CHANGE HERE]
  * [LARGER ISSUES MAY DESERVE THEIR OWN SECTION]

About Live Systems
------------------
The Live Systems project produces the tools used to build official
live systems and the official live images themselves for Debian.

About Debian
------------
The Debian Project is an association of Free Software developers who
volunteer their time and effort in order to produce the completely free
operating system Debian.

Contact Information
-------------------
For further information, please visit the Live Systems web pages at
<http://live-systems.org/>, or contact the Live Systems team at
<debian-live@lists.debian.org>.
```

17. Git repositories

The list of all the available repositories of the Live Systems Project can be found at <http://live-systems.org/gitweb/>. The project's git URLs have the form: `protocol://live-systems.org/git/repository`. Thus, in order to clone *live-manual* read-only, launch:

```
$ git clone git://live-systems.org/git/live-manual.git
```

Or,

```
$ git clone https://live-systems.org/git/live-manual.git
```

Or,

```
$ git clone http://live-systems.org/git/live-manual.git
```

The cloning addresses with write permission have the form: `git@live-systems.org:/repository`.

So, again, to clone *live-manual* over ssh you must type:

```
$ git clone git@live-systems.org:live-manual.git
```

The git tree is made up of several different branches. The **debian** and the **debian-next** branches are particularly noteworthy because they contain the actual work that will eventually be included in each new release.

After cloning any of the existing repositories, you will be on the **debian** branch. This is appropriate to take a look at the state of the project's latest release but before starting work it is crucial to switch to the **debian-next** branch. To do so:

```
$ git checkout debian-next
```

The **debian-next** branch, which is not always fast-forward, is where all the changes are committed first before being merged into the **debian** branch. To make an analogy, it is like a testing ground. If you are working on this branch and need to pull, you will have to

do a `git pull --rebase` so that your local modifications are staged while pulling from the server and then your changes will be put on top of it all.

17.1 Handling multiple repositories

If you intend to clone several of the live systems repositories and want to switch to the **debian-next** branch right away to check the latest code, write a patch or contribute with a translation you ought to know that the git server provides a `mrconfig` file to ease the handling of multiple repositories. In order to use it you need to install the *mr* package and after that, launch:

```
$ mr bootstrap http://live-systems.org/other/mr/mrconfig
```

This command will automatically clone and checkout to the **debian-next** branch the development repositories of the Debian packages produced by the project. These include, among others, the *live-images* repository, which contains the configurations used for the prebuilt images that the project publishes for general use. For more information on how to use this repository, see ‹Clone a configuration published via Git›

Examples

18. Examples

This chapter covers example builds for specific use cases with live systems. If you are new to building your own live system images, we recommend you first look at the three tutorials in sequence, as each one teaches new techniques that will help you use and understand the remaining examples.

18.1 Using the examples

To use these examples you need a system to build them on that meets the requirements listed in ‹Requirements› and has *live-build* installed as described in ‹Installing live-build›.

Note that, for the sake of brevity, in these examples we do not specify a local mirror to use for the build. You can speed up downloads considerably if you use a local mirror. You may specify the options when you use `lb config`, as described in ‹Distribution mirrors used at build time›, or for more convenience, set the default for your build system in `/etc/live/build.conf`. Simply create this file and in it, set the corresponding `LB_MIRROR_*` variables

to your preferred mirror. All other mirrors used in the build will be defaulted from these values. For example:

```
LB_MIRROR_BOOTSTRAP="http://mirror/debian/"
LB_MIRROR_CHROOT_SECURITY="http://mirror/debian-security/"
LB_MIRROR_CHROOT_BACKPORTS="http://mirror/debian-backports/"
```

18.2 Tutorial 1: A default image

Use case: Create a simple first image, learning the basics of *live-build*.

In this tutorial, we will build a default ISO hybrid live system image containing only base packages (no Xorg) and some live system support packages, as a first exercise in using *live-build*.

You can't get much simpler than this:

```
$ mkdir tutorial1 ; cd tutorial1 ; lb config
```

Examine the contents of the `config/` directory if you wish. You will see stored here a skeletal configuration, ready to customize or, in this case, use immediately to build a default image.

Now, as superuser, build the image, saving a log as you build with `tee`.

```
# lb build 2>&1 | tee build.log
```

Assuming all goes well, after a while, the current directory will contain `binary.hybrid.iso`. This ISO hybrid image can be booted directly in a virtual machine as described in ‹Testing an ISO image with Qemu› and ‹Testing an ISO image with VirtualBox›, or else imaged onto optical media or a USB flash device as described in ‹Burning an ISO image to a physical medium› and ‹Copying an ISO hybrid image to a USB stick›, respectively.

18.3 Tutorial 2: A web browser utility

Use case: Create a web browser utility image, learning how to apply customizations.

In this tutorial, we will create an image suitable for use as a web browser utility, serving as an introduction to customizing live system images.

```
$ mkdir tutorial2
$ cd tutorial2
$ echo "task-lxde-desktop iceweasel" >> config/package-lists/my.list.chroot
```

Our choice of LXDE for this example reflects our desire to provide a minimal desktop environment, since the focus of the image is the single use we have in mind, the web browser. We could go even further and provide a default configuration for the web browser in `config/includes.chroot/etc/iceweasel/profile/`, or additional support packages for viewing various kinds of web content, but we leave this as an exercise for the reader.

Build the image, again as superuser, keeping a log as in ‹Tutorial 1›:

```
# lb build 2>&1 | tee build.log
```

Again, verify the image is OK and test, as in ‹Tutorial 1›.

18.4 Tutorial 3: A personalized image

Use case: Create a project to build a personalized image, containing your favourite software to take with you on a USB stick wherever you go, and evolving in successive revisions as your needs and preferences change.

Since we will be changing our personalized image over a number of revisions, and we want to track those changes, trying things experimentally and possibly reverting them if things don't work out, we will keep our configuration in the popular `git` version control system. We will also use the best practice of autoconfiguration via `auto` scripts as described in ‹Managing a configuration›.

18.4.1 First revision

```
$ mkdir -p tutorial3/auto
$ cp /usr/share/doc/live-build/examples/auto/* tutorial3/auto/
$ cd tutorial3
```

Edit `auto/config` to read as follows:

```
#!/bin/sh

lb config noauto \
     --architectures i386 \
     --linux-flavours 686-pae \
     "${@}"
```

Perform `lb config` to generate the config tree, using the `auto/config` script you just created:

```
$ lb config
```

Now populate your local package list:

```
$ echo "task-lxde-desktop iceweasel xchat" >> config/package-lists/my.list.chroot
```

First, `--architectures i386` ensures that on our `amd64` build system, we build a 32-bit version suitable for use on most machines. Second, we use `--linux-flavours 686-pae` because we don't anticipate using this image on much older systems. Third, we have chosen the *lxde* task metapackage to give us a minimal desktop. And finally, we have added two initial favourite packages: *iceweasel* and *xchat*.

Now, build the image:

```
# lb build
```

Note that unlike in the first two tutorials, we no longer have to type `2>&1 |tee build.log` as that is now included in `auto/build`.

Once you've tested the image (as in ‹Tutorial 1›) and are satisfied it works, it's time to initialize our `git` repository, adding only the auto scripts we just created, and then make the first commit:

```
$ git init
$ cp /usr/share/doc/live-build/examples/gitignore .gitignore
$ git add .
$ git commit -m "Initial import."
```

18.4.2 Second revision

In this revision, we're going to clean up from the first build, add the *vlc* package to our configuration, rebuild, test and commit.

The `lb clean` command will clean up all generated files from the previous build except for the cache, which saves having to re-download packages. This ensures that the subsequent `lb build` will re-run all stages to regenerate the files from our new configuration.

```
# lb clean
```

Now append the *vlc* package to our local package list in `config/package-lists/my.list.chroot`:

```
$ echo vlc >> config/package-lists/my.list.chroot
```

Build again:

```
# lb build
```

Test, and when you're satisfied, commit the next revision:

```
$ git commit -a -m "Adding vlc media player."
```

Of course, more complicated changes to the configuration are possible, perhaps adding files in subdirectories of `config/`. When you commit new revisions, just take care not to hand edit or commit the top-level files in `config` containing `LB_*` variables, as these are build products, too, and are always cleaned up by `lb clean` and re-created with `lb config` via their respective `auto` scripts.

We've come to the end of our tutorial series. While many more kinds of customization are possible, even just using the few features explored in these simple examples, an almost infinite variety of different images can be created. The remaining examples in this section cover several other use cases drawn from the collected experiences of users of live systems.

18.5 A VNC Kiosk Client

Use case: Create an image with *live-build* to boot directly to a VNC server.

Make a build directory and create an skeletal configuration inside it, disabling recommends to make a minimal system. And then create two initial package lists: the first one generated with a script provided by *live-build* named `Packages` (see ‹Generated package lists›), and the second one including *xorg*, *gdm3*, *metacity* and *xvnc4viewer*.

```
$ mkdir vnc-kiosk-client
$ cd vnc-kiosk-client
$ lb config -a i386 -k 686-pae --apt-recommends false
$ echo '! Packages Priority standard' > config/package-lists/standard.list.chroot
$ echo "xorg gdm3 metacity xvnc4viewer" > config/package-lists/my.list.chroot
```

As explained in ‹Tweaking APT to save space› you may need to re-add some recommended packages to make your image work properly.

An easy way to list recommends is using *apt-cache*. For example:

```
$ apt-cache depends live-config live-boot
```

In this example we found out that we had to re-include several packages recommended by *live-config* and *live-boot*: `user-setup` to make autologin work and `sudo` as an essential program to shutdown the system. Besides, it could be handy to add `live-tools` to be able to copy the image to RAM and `eject` to eventually eject the live medium. So:

```
$ echo "live-tools user-setup sudo eject" > config/package-lists/recommends.list.chroot
```

After that, create the directory `/etc/skel` in `config/includes.chroot` and put a custom `.xsession` in it for the default user that will launch *metacity* and start *xvncviewer*, connecting to port `5901` on a server at `192.168.1.2`:

```
$ mkdir -p config/includes.chroot/etc/skel
$ cat > config/includes.chroot/etc/skel/.xsession << EOF
#!/bin/sh

/usr/bin/metacity &
/usr/bin/xvncviewer 192.168.1.2:1
```

```
exit
EOF
```

Build the image:

```
# lb build
```

Enjoy.

18.6 A base image for a 128MB USB key

Use case: Create a default image with some components removed in order to fit on a 128MB USB key with a little space left over to use as you see fit.

When optimizing an image to fit a certain media size, you need to understand the tradeoffs you are making between size and functionality. In this example, we trim only so much as to make room for additional material within a 128MB media size, but without doing anything to destroy the integrity of the packages contained within, such as the purging of locale data via the *localepurge* package, or other such “intrusive” optimizations. Of particular note, we use `--debootstrap-options` to create a minimal system from scratch.

```
$ lb config -k 486 --apt-indices false --apt-recommends false --debootstrap-options "--variant=minbase" --↩
     firmware-chroot false --memtest none
```

To make the image work properly, we must re-add, at least, two recommended packages which are left out by the `--apt-recommends false` option. See ‹Tweaking APT to save space›

```
$ echo "user-setup sudo" > config/package-lists/recommends.list.chroot
```

Now, build the image in the usual way:

```
# lb build 2>&1 | tee build.log
```

On the author's system at the time of writing this, the above configuration produced a

77MB image. This compares favourably with the 177MB image produced by the default configuration in ‹Tutorial 1›.

The biggest space-saver here, compared to building a default image on an `i386` architecture system, is to select only the `486` kernel flavour instead of the default `-k "486 686-pae"`. Leaving off APT's indices with `--apt-indices false` also saves a fair amount of space, the tradeoff being that you need to do an `apt-get update` before using *apt* in the live system. Dropping recommended packages with `--apt-recommends false` saves some additional space, at the expense of omitting some packages you might otherwise expect to be there. `--debootstrap-options "--variant=minbase"` bootstraps a minimal system from the start. Not automatically including firmware packages with `--firmware-chroot false` saves some space too. And finally, `--memtest none` prevents the installation of a memory tester.

Note: A minimal system can also be achieved using hooks, like for example the `stripped.hook.chroot` hook found in `/usr/share/doc/live-build/examples/hooks`. It may shave off additional small amounts of space and produce an image of 62MB. However, it does so by removal of documentation and other files from packages installed on the system. This violates the integrity of those packages and that, as the comment header warns, may have unforeseen consequences. That is why using a minimal *debootstrap* is the recommended way of achieving this goal.

18.7 A localized GNOME desktop and installer

Use case: Create a GNOME desktop image, localized for Switzerland and including an installer.

We want to make an iso-hybrid image for i386 architecture using our preferred desktop, in this case GNOME, containing all of the same packages that would be installed by the standard Debian installer for GNOME.

Our initial problem is the discovery of the names of the appropriate language tasks. Currently, *live-build* cannot help with this. While we might get lucky and find this by trial-and-error, there is a tool, `grep-dctrl`, which can be used to dig it out of the task descriptions in tasksel-data, so to prepare, make sure you have both of those things:

```
# apt-get install dctrl-tools tasksel-data
```

Now we can search for the appropriate tasks, first with:

```
$ grep-dctrl -FTest-lang de /usr/share/tasksel/descs/debian-tasks.desc -sTask
Task: german
```

By this command, we discover the task is called, plainly enough, german. Now to find the related tasks:

```
$ grep-dctrl -FEnhances german /usr/share/tasksel/descs/debian-tasks.desc -sTask
Task: german-desktop
Task: german-kde-desktop
```

At boot time we will generate the **de_CH.UTF-8** locale and select the **ch** keyboard layout. Now let's put the pieces together. Recalling from ‹Using metapackages› that task metapackages are prefixed `task-`, we just specify these language boot parameters, then add standard priority packages and all our discovered task metapackages to our package list as follows:

```
$ mkdir live-gnome-ch
$ cd live-gnome-ch
$ lb config \
     -a i386 \
     -k 486 \
     --bootappend-live "boot=live components locales=de_CH.UTF-8 keyboard-layouts=ch" \
     --debian-installer live
$ echo '! Packages Priority standard' > config/package-lists/standard.list.chroot
$ echo task-gnome-desktop task-german task-german-desktop >> config/package-lists/desktop.list.chroot
$ echo debian-installer-launcher >> config/package-lists/installer.list.chroot
```

Note that we have included the *debian-installer-launcher* package to launch the installer from the live desktop, and have also specified the 486 flavour kernel, as it is currently necessary to make the installer and live system kernels match for the launcher to work properly.

Appendix

19. Style guide

19.1 Guidelines for authors

This section deals with some general considerations to be taken into account when writing technical documentation for *live-manual*. They are divided into linguistic features and recommended procedures.

Note: Authors should first read ‹Contributing to this document›

19.1.1 Linguistic features

- *Use plain English*

Keep in mind that a high percentage of your readers are not native speakers of English. So as a general rule try to use short, meaningful sentences, followed by a full stop.

This does not mean that you have to use a simplistic, naive style. It is a suggestion to try to avoid, as much as possible, complex subordinate sentences that make the text difficult to understand for non-native speakers of English.

- *Variety of English*

The most widely spread varieties of English are British and American so it is very likely that most authors will use either one or the other. In a collaborative environment, the ideal variety would be “International English” but it is very difficult, not to say impossible, to decide on which variety among all the existing ones, is the best to use.

We expect that different varieties may mix without creating misunderstandings but in general terms you should try to be coherent and before deciding on using British, American or any other English flavour at your discretion, please take a look at how other people write and try to imitate them.

- *Be balanced*

Do not be biased. Avoid including references to ideologies completely unrelated to *live-manual*. Technical writing should be as neutral as possible. It is in the very nature of scientific writing.

- *Be politically correct*

Try to avoid sexist language as much as possible. If you need to make references to the third person singular preferably use “they” rather than “he” or “she” or awkward inventions such as “s/he”, “s(he)” and the like.

- *Be concise*

Go straight to the point and do not wander around aimlessly. Give as much information as necessary but do not give more information than necessary, this is to say, do not explain unnecessary details. Your readers are intelligent. Presume some previous knowledge on their part.

- *Minimize translation work*

Keep in mind that whatever you write will have to be translated into several other languages. This implies that a number of people will have to do an extra work if you add useless or redundant information.

- *Be coherent*

As suggested before, it is almost impossible to standardize a collaborative document into a perfectly unified whole. However, every effort on your side to write in a coherent way with the rest of the authors will be appreciated.

- *Be cohesive*

Use as many text-forming devices as necessary to make your text cohesive and unambiguous. (Text-forming devices are linguistic markers such as connectors).

- *Be descriptive*

It is preferable to describe the point in one or several paragraphs than merely using a number of sentences in a typical “changelog” style. Describe it! Your readers will appreciate it.

- *Dictionary*

Look up the meaning of words in a dictionary or encyclopedia if you do not know how to express certain concepts in English. But keep in mind that a dictionary can either be your best friend or can turn into your worst enemy if you do not know how to use it correctly.

English has the largest vocabulary that exists (with over one million words). Many of these words are borrowings from other languages. When looking up the meaning of words in a bilingual dictionary the tendency of a non-native speaker of English is to choose the one that sounds more similar in their mother tongue. This often turns into an excessively formal discourse which does not sound quite natural in English.

As a general rule, if a concept can be expressed using different synonyms, it is a good advice to choose the first word proposed by the dictionary. If in doubt, choosing words of Germanic origin (Usually monosyllabic words) is often the right thing to do. Be warned that these two techniques might produce a rather informal discourse but at least your choice of words will be of wide use and generally accepted.

Using a dictionary of collocations is recommended. They are extremely helpful when it comes to know which words usually occur together.

Again it is a good practice to learn from the work of others. Using a search engine to check how other authors use certain expressions may help a lot.

- *False friends, idioms and other idiomatic expressions*

Watch out for false friends. No matter how proficient you are in a foreign language you cannot help falling from time to time in the trap of the so called “false friends”, words that look similar in two languages but whose meanings or uses might be completely different.

Try to avoid idioms as much as possible. “Idioms” are expressions that may convey a completely different meaning from what their individual words seem to mean. Sometimes, idioms might be difficult to understand even for native speakers of English!

- *Avoid slang, abbreviations, contractions...*

Even though you are encouraged to use plain, everyday English, technical writing belongs to the formal register of the language.

Try to avoid slang, unusual abbreviations that are difficult to understand and above all contractions that try to imitate the spoken language. Not to mention typical irc and family friendly expressions.

19.1.2 Procedures

- *Test before write*

It is important that authors test their examples before adding them to *live-manual* to ensure that everything works as described. Testing on a clean chroot or VM can be a good starting point. Besides, it would be ideal if the tests were then carried out on different machines with different hardware to spot possible problems that may arise.

- *Examples*

When providing an example try to be as specific as you can. An example is, after all, just an example.

It is often better to use a line that only applies to a specific case than using abstractions that may confuse your readers. In this case you can provide a brief explanation of the effects of the proposed example.

There may be some exceptions when the example suggests using some potentially dangerous commands that, if misused, may cause data loss or other similar undesirable effects. In this case you should provide a thorough explanation of the possible side effects.

- *External links*

Links to external sites should only be used when the information on those sites is crucial when it comes to understanding a special point. Even so, try to use links to external sites as sparsely as possible. Internet links are likely to change from time to time resulting in broken links and leaving your arguments in an incomplete state.

Besides, people who read the manual offline will not have the chance to follow those links.

- *Avoid branding and things that violate the license under which the manual is published*

Try to avoid branding as much as possible. Keep in mind that other downstream projects might make use of the documentation you write. So you are complicating things for them if you add certain specific material.

live-manual is licensed under the GNU GPL. This has a number of implications that apply to the distribution of the material (of any kind, including copyrighted graphics or logos) that is published with it.

- *Write a first draft, revise, edit, improve, redo if necessary*

- Brainstorm!. You need to organize your ideas first in a logical sequence of events.

- Once you have somehow organized those ideas in your mind write a first draft.

- Revise grammar, syntax and spelling. Keep in mind that the proper names of the releases, such as **jessie** or **sid** , should not be capitalized when referred to as code names.

- Improve your statements and redo any part if necessary.

- *Chapters*

Use the conventional numbering system for chapters and subtitles. e.g. 1, 1.1, 1.1.1, 1.1.2 ... 1.2, 1.2.1, 1.2.2 ... 2, 2.1 ... and so on. See markup below.

If you have to enumerate a series of steps or stages in your description, you can also use ordinal numbers: First, second, third ... or First, Then, After that, Finally ... Alternatively you can use bulleted items.

- *Markup*

And last but not least, *live-manual* uses ‹SiSU› to process the text files and produce a multiple format output. It is recommended to take a look at ‹SiSU's manual› to get familiar with its markup, or else type:

```
$ sisu --help markup
```

Here are some markup examples that may prove useful:

- For emphasis/bold text:

```
*{foo}* or !{foo}!
```

produces: **foo** or **foo** . Use it to emphasize certain key words.

- For italics:

```
/{foo}/
```

produces: *foo*. Use them e.g. for the names of Debian packages.

- For monospace:

```
#{foo}#
```

produces: `foo`. Use it e.g. for the names of commands. And also to highlight some key words or things like paths.

- For code blocks:

```
code{

  $ foo
  # bar

}code
```

produces:

```
$ foo
# bar
```

Use `code{` to open and `}code` to close the tags. It is important to remember to leave a space at the beginning of each line of code.

19.2 Guidelines for translators

This section deals with some general considerations to be taken into account when translating the contents of *live-manual*.

As a general recommendation, translators should have read and understood the translation rules that apply to their specific languages. Usually, translation groups and mailing lists provide information on how to produce translated work that complies with Debian quality standards.

Note: Translators should also read ‹Contributing to this document›. In particular the section ‹Translation›

19.2.1 Translation hints

- *Comments*

The role of the translator is to convey as faithfully as possible the meaning of words, sentences, paragraphs and texts as written by the original authors into their target language.

So they should refrain from adding personal comments or extra bits of information of their own. If they want to add a comment for other translators working on the same documents, they can leave it in the space reserved for that. That is, the header of the strings in the **po** files preceded by a number sign # . Most graphical translation programs can automatically handle those types of comments.

- *TN, Translator's Note*

It is perfectly acceptable however, to include a word or an expression in brackets in the translated text if, and only if, that makes the meaning of a difficult word or expression clearer to the reader. Inside the brackets the translator should make evident that the addition was theirs using the abbreviation “TN” or “Translator's Note”.

- *Impersonal sentences*

Documents written in English make an extensive use of the impersonal form “you”. In some other languages that do not share this characteristic, this might give the false impression that the original texts are directly addressing the reader when they are actually not doing so. Translators must be aware of that fact and reflect it in their language as accurately as possible.

- *False friends*

The trap of “false friends” explained before especially applies to translators. Double check the meaning of suspicious false friends if in doubt.

- *Markup*

Translators working initially with **pot** files and later on with **po** files will find many markup features in the strings. They can translate the text anyway, as long as it is translatable, but it is extremely important that they use exactly the same markup as the original English version.

- *Code blocks*

Even though the code blocks are usually untranslatable, including them in the translation is the only way to score a 100% complete translation. And even though it means more work at first because it might require the intervention of the translators if the code changes, it is the best way, in the long run, to identify what has already been translated and what has not when checking the integrity of the .po files.

- *Newlines*

The translated texts need to have the exact same newlines as the original texts. Be careful to press the “Enter” key or type if they appear in the original files. These newlines often appear, for instance, in the code blocks.

Make no mistake, this does not mean that the translated text needs to have the same length as the English version. That is nearly impossible.

- *Untranslatable strings*

Translators should never translate:

- The code names of releases (which should be written in lowercase)
- The names of programs
- The commands given as examples
- Metadata (often between colons **:metadata:**)
- Links
- Paths

Metadata

SiSU Metadata, document information

Document Manifest @:
‹http://live-systems.org/manual/manifest/live-manual.en.html›
Title: Live Systems Manual
Creator: Live Systems Project <debian-live@lists.debian.org>

Publisher: Live Systems Project <debian-live@lists.debian.org>
Date: 2013-11-25

Version Information
Sourcefile: live-manual.ssm.sst
Filetype: SiSU text 2.0,
Source Digest: SHA256(live-manual.ssm.sst)=7b8ca983a9dbca6f818c3632a2c74e9a5b91b065ce622b736991ab1b8a9be6d1-

Generated
Document (dal) last generated: 2013-11-25 18:29:02 +0000
Generated by: SiSU 4.2.12 of 2013w44/5 (2013-11-08)
Ruby version: ruby 1.9.3p448 (2013-06-27 revision 41675) [i486-linux]