Live Systems Handbuch

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

2015-08-23

Live Systems Handbuch
Live Systems Projekt
<debian-live@lists.debian.org>

Live Systems Handbuch a

Über Live Systeme 2

Über dieses Handbuch 3

Über dieses Handbuch 4

About the ${project} 9

About the ${project} 10

Benutzer 12

Installation   **13**

The basics   **17**

Git repositories   85

Git repositories   86

Beispiele   88

Beispiele   89

Examples   90

Anhang   97

Style guide   98

Style guide   99

Über Live Systeme

git

Über dieses Handbuch

Über dieses Handbuch 4

This manual serves as a single access point to all documentation related to the 5 ${project} and in particular applies to the software produced by the project for the Debian 9.0 "${stable}" 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 6 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.

Einige der erwähnten Befehle im Text müssen als Superuser ausgeführt werden. 7 Dies kann entweder dadurch erreicht werden, indem zuerst auf den root Benutzer gewechselt wird mittels `su` oder durch die Benutzung von `sudo`. Um die Befehle welche als unprivilegierter Benutzer ausgeführt werden können und diesen welche Superuser Rechte benötigen, sind den Befehlen $ respektive # vorangestellt. Dieses Symbol ist nicht Teil des Befehls.

Für die Ungeduldigen 8

Obowhl wir denken dass alles in diesem Handbuch mehr oder weniger für die einen 9 oder anderen Benutzer wichtig ist, sind wir uns bewusst, dass es sich um viel Material handelt. Für ein schnelles Erfolgserlebnis in der Anwendung dieser Software schlagen wir die folgende Reihenfolge vor, bevor sie sich mit den Details befassen:

First, read this chapter, About this manual, from the beginning and ending with the 10 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 11 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.

Begriffe 12

Live system: An operating system that can boot without installation to a hard 13 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 (via netboot images, see Building a netboot image), and over the Internet (via the boot parameter `fetch=URL`, see Webbooting).

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. 14

${project}: The project which maintains, among others, the live-boot, live-build, live-config, live-tools and live-manual packages. 15

Host system: The environment used to create the live system. 16

Target system: The environment used to run the live system. 17

live-boot: A collection of scripts used to boot live systems. 18

live-build: A collection of scripts used to build customized live systems. 19

live-config: A collection of scripts used to configure a live system during the boot process. 20

live-tools: A collection of additional scripts used to perform useful tasks within a running live system. 21

live-manual: This document is maintained in a package called live-manual. 22

Debian Installer (d-i): The official installation system for the Debian distribution. 23

Boot parameters: Parameters that can be entered at the bootloader prompt to influence the kernel or live-config. 24

chroot: The *chroot* program, `chroot(8)`, enables us to run different instances of the GNU/Linux environment on a single system simultaneously without rebooting. 25

Binary image: A file containing the live system, such as live-image-i386.hybrid.iso or live-image-i386.img. 26

Target distribution: The distribution upon which your live system will be based. This can differ from the distribution of your host system. 27

stable/testing/unstable: The **stable** distribution, currently codenamed ${stable}, contains the latest officially released distribution of Debian. The **testing** distribution, temporarily codenamed ${testing}, 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 ${testing} or sid, as that is what is supported by the tools themselves. 28

Autoren 29

Liste der Autoren (in alphabetischer Reihenfolge): 30

Ben Armstrong 31

Brendan Sleight 32

Carlos Zuferri   33

Chris Lamb   34

Daniel Baumann   35

Franklin Piat   36

Jonas Stein   37

Kai Hendry   38

Marco Amadori   39

Mathieu Geli   40

Matthias Kirschner   41

Richard Nelson   42

Trent W. Buck   43

Contributing to this document   44

This manual is intended as a community project and all proposals for improvements   45
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.

Applying changes   46

In order to make changes to the English manual you have to edit the right files   47
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:

48

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

You may build the live-manual from the top level directory of your Git checkout by   49
executing:

50

```
$ make build
```

Since it takes a while to build the manual in all supported languages, authors may   51
find it convenient to use one of the fast proofing shortcuts when reviewing the new
documentation they have added to the English manual. Using `PROOF=1` builds live-
manual in html format, but without the segmented html files, and using `PROOF=2`
builds live-manual in pdf format, but only the A4 and letter portraits. That is why
using either of the `PROOF=` possibilities can save up a considerable amount of time,
e.g:

52

```
$ make build PROOF=1
```

When proofing one of the translations it is possible to build only one language by   53
executing, e.g:

54

```
$ make build LANGUAGES=de
```

It is also possible to build by document type, e.g: 55

56

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

Or combine both, e.g: 57

58

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

After revising your work and making sure that everything is fine, do not use `make`   59
`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.

Translation   60

In order to translate live-manual, follow these steps depending on whether you   61
are starting a translation from scratch or continue working on an already existing
one:

Start a new translation from scratch   62

Translate the **about_manual.ssi.pot**, **about_project.ssi.pot** and **in-**   63
dex.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   64
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   65

If your target language has already been added, you can randomly con-   66
tinue 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   67
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 infor- 68
mative 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 untrans- 69
lated 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 trans- 70
lation 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 lo-calization (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 71
compulsory thanks to the .gitignore file but it is a good practice to avoid committing files involuntarily.

About the ${project}

About the ${project}

Motivation

What is wrong with current live systems

When ${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:

76

They are not Debian projects and therefore lack support from within Debian.

77

They mix different distributions, e.g. **testing** and **unstable**.

78

They support i386 only.

79

They modify the behaviour and/or appearance of packages by stripping them down to save space.

80

They include packages from outside of the Debian archive.

81

They ship custom kernels with additional patches that are not part of Debian.

82

They are large and slow due to their sheer size and thus not suitable for rescue issues.

83

They are not available in different flavours, e.g. CDs, DVDs, USB-stick and netboot images.

84

Why create our own live system?

85

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:

86

It is a subproject of Debian.

87

It reflects the (current) state of one distribution.

88

It runs on as many architectures as possible.

89

It consists of unchanged Debian packages only.

90

It does not contain any packages that are not in the Debian archive.

91

It uses an unaltered Debian kernel with no additional patches.

92

Philosophy

93

Only unchanged packages from Debian "main"

94

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.

95

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

96

As an exception, our own packages such as live-boot, live-build or live-config may   97
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.

No package configuration of the live system   98

In this phase we will not ship or install sample or alternative configurations. All pack-   99
ages 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   100
with its package maintainer in Debian.

A system for configuring packages is provided using debconf allowing custom con-   101
figured 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 configura-
tion, 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 configu-
rations. For more information, please see Customization overview.

Contact   102

Mailing list: The primary contact for the project is the mailing list at 「https://lists.debian.org/debia...   103
. You can email the list directly by addressing your mail to debian-live@lists.debian.org.
The list archives are available at 「https://lists.debian.org/debian-live/」 .

IRC: A number of users and developers are present in the #debian-live channel   104
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   105
「Debian Bug Tracking System」 (BTS) contains details of bugs reported by users
and developers. Each bug is given a number, and is kept on file until it is marked
as having been dealt with. For more information, please see Reporting bugs.

Benutzer

Installation

git

Installation 108

Requirements 109

Building live system images has very few system requirements: 110

Superuser (root) access 111

An up-to-date version of live-build 112

A POSIX-compliant shell, such as *bash* or *dash* 113

debootstrap 114

Linux 2.6 or newer. 115

Note that using Debian or a Debian-derived distribution is not required - live-build 116
will run on almost any distribution with the above requirements.

Installing live-build 117

You can install live-build in a number of different ways: 118

From the Debian repository 119

From source 120

From snapshots 121

If you are using Debian, the recommended way is to install live-build via the Debian 122
repository.

From the Debian repository 123

Simply install live-build like any other package: 124

125

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

From source 126

live-build is developed using the Git version control system. On Debian based sys- 127
tems, this is provided by the *git* package. To check out the latest code, execute:

128

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

You can build and install your own Debian package by executing: 129

130

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

Now install whichever of the freshly built .deb files you were interested in, e.g.   131

132

```
# dpkg -i live-build_4.0-1_all.deb
```

You can also install live-build directly to your system by executing: 133

134

```
# make install
```

and uninstall it with: 135

136

```
# make uninstall
```

From 'snapshots'   137

If you do not wish to build or install live-build from source, you can use snapshots.   138
These are built automatically from the latest version in Git and are available on
「http://live-systems.org/debian/」.

Installing live-boot and live-config   139

Note: You do not need to install live-boot or live-config on your system to create   140
customized live systems. However, doing so will do no harm and is useful for ref-
erence purposes. If you only want the documentation, you may now install the
live-boot-doc and *live-config-doc* packages separately.

From the Debian repository   141

Both live-boot and live-config are available from the Debian repository as per In-   142
stalling live-build.

From source   143

To use the latest source from git, you can follow the process below. Please ensure   144
you are familiar with the terms mentioned in Terms.

Checkout the live-boot and live-config sources   145

146

```
$ 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   147
your reason for building these packages from source.

Build live-boot and live-config .deb files   148

You must build either on your target distribution or in a chroot containing your target platform: this means if your target is ${testing} then you should build against ${testing}. 149

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 ${testing} live images, build live-boot in a ${testing} 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): 150

151

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

Use applicable generated .deb files 152

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 likely 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: 153

154

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

From 'snapshots' 155

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. 156

The basics

git

The basics   158

This chapter contains a brief overview of the build process and instructions for us-   159
ing 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 stor-
age device. In certain special cases, as explained later, the `hdd` type may be more
suitable. The chapter includes detailed 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 already familiar with netboot-
ing, 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.

The section finishes with a quick introduction to webbooting which is, perhaps, the   160
easiest way of using different images for different purposes, switching from one to
the other as needed using the internet as a means.

Throughout the chapter, we will often refer to the default filenames produced by   161
live-build. If you are downloading a prebuilt image instead, the actual filenames
may vary.

What is a live system?   162

A live system usually means an operating system booted on a computer from a   163
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 architec-   164
tures (currently amd64 and i386). It is made from the following parts:

> **Linux kernel image**, usually named `vmlinuz*`   165

> **Initial RAM disk image (initrd)**: a RAM disk set up for the Linux boot, containing   166
> 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   167
> 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' mecha-
> nism 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,   168
> 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   169

Linux kernel, its initrd, and a bootloader to run them, all in one medium-dependant format (ISO9660 image, disk image, etc.).

Downloading prebuilt images 170

While the focus of this manual is developing and building your own live images, you 171
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 ⌈`https://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/`⌋.

Using the web live image builder 172

As a service to the community, we run a web-based live image builder service at 173
⌈`http://live-systems.org/build/`⌋. 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.

Web builder usage and caveats 174

The web interface currently makes no provision to prevent the use of invalid com- 175
binations 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 `586` 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 176
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 177
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.

First steps: building an ISO hybrid image   178

Regardless of the image type, you will need to perform the same basic steps to   179
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   180
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.

181

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

Then, run the `lb config` command. This will create a "config/" hierarchy in the current   182
directory for use by other commands:

183

```
$ lb config
```

No parameters are passed to these commands, so defaults for all of their various   184
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` com-   185
mand:

186

```
# lb build
```

This process can take a while, depending on the speed of your computer and your   187
network connection. When it is complete, there should be a `live-image-i386.hybrid.iso`
image file, ready to use, in the current directory.

Note: If you are building on an amd64 system the name of the resulting image will   188
be `live-image-amd64.hybrid.iso`. Keep in mind this naming convention throughout the
manual.

Using an ISO hybrid live image   189

After either building or downloading an ISO hybrid image, which can be obtained   190
at 「https://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.

Burning an ISO image to a physical medium   191

Burning an ISO image is easy. Just install *xorriso* and use it from the command-line   192
to burn the image. For instance:

193

```
# apt-get install xorriso
$ xorriso -as cdrecord -v dev=/dev/sr0 blank=as_needed live-image-i386.hybrid.iso
```

Copying an ISO hybrid image to a USB stick

194

ISO images prepared with `xorriso`, can be simply copied to a USB stick with the `cp` 195 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 196 the image to the stick. **This will definitely overwrite any previous contents on your stick!**

197

```
$ cp live-image-i386.hybrid.iso ${USBSTICK}
$ sync
```

Note: The *sync* command is useful to ensure that all the data, which is stored in 198 memory by the kernel while copying the image, is written to the USB stick.

Using the space left on a USB stick

199

After copying the `live-image-i386.hybrid.iso` to a USB stick, the first partition on 200 the device will be filled up by the live system. To use the remaining free space, use a partitioning tool such as *gparted* or *parted* to create a new partition on the stick.

201

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

After the partition is created, where `${PARTITION}` is the name of the partition, such 202 as `/dev/sdb2`, you have to create a filesystem on it. One possible choice would be ext4.

203

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

Note: If you want to use the extra space with Windows, apparently that OS cannot 204 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 live-image-i386.hybrid.iso on the 205 **stick, all data on the stick will be lost because the partition table is over-written by the contents of the image, so back up your extra partition first to restore again after updating the live image.**

Booting the live medium 206

The first time you boot your live medium, whether CD, DVD, USB key, or PXE boot, 207
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 208
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 209
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 a `standard` flavour prebuilt image, you should be automatically logged in on the
console to the `user` account and see a shell prompt, ready to use.

Using a virtual machine for testing 210

It can be a great time-saver for the development of live images to run them in a 211
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 212
with hardware support for virtualization is recommended.

There are some inherent limitations to running on a VM, e.g. poor video perfor- 213
mance, limited choice of emulated hardware.

When developing for specific hardware, there is no substitute for running on the 214
hardware itself.

Occasionally there are bugs that relate only to running in a VM. When in doubt, 215
test your image directly on the hardware.

Provided you can work within these constraints, survey the available VM software 216
and choose one that is suitable for your needs.

Testing an ISO image with QEMU 217

The most versatile VM in Debian is QEMU. If your processor has hardware support 218
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 219
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`.

220

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

Booting an ISO image is simple: 221

                                                                                222

```
$ kvm -cdrom live-image-i386.hybrid.iso
```

See the man pages for more details.   223

Testing an ISO image with VirtualBox   224

In order to test the ISO with *virtualbox*:   225

                                                                                226

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

Create a new virtual machine, change the storage settings to use `live-image-i386.hybrid.iso` 227
as the CD/DVD device, and start the machine.

Note: For live systems containing X.org that you want to test with *virtualbox*, you 228
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.

                                                                                229

```
$ 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 230
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.

                                                                                231

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

Building and using an HDD image   232

Building an HDD image is similar to an ISO hybrid one in all respects except you 233
specify `-b hdd` and the resulting filename is `live-image-i386.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 234
to clean up your working directory with the `lb clean` command (see The lb clean
command):

                                                                                235

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

Run the `lb config` command as before, except this time specifying the HDD image   236
type:

237

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

Now build the image with the `lb build` command: 238

239

```
# lb build
```

When the build finishes, a `live-image-i386.img` file should be present in the current   240
directory.

The generated binary image contains a VFAT partition and the syslinux bootloader,   241
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 `live-image-i386.img` instead of `live-image-i386.hybrid.iso`.

Likewise, to test an HDD image with Qemu, install *qemu* as described above in   242
Testing an ISO image with QEMU. Then run `kvm` or `qemu`, depending on which version
your host system needs, specifying `live-image-i386.img` as the first hard drive.

243

```
$ kvm -hda live-image-i386.img
```

Building a netboot image   244

The following sequence of commands will create a basic netboot image containing   245
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   246
working directory with the `lb clean` command:

247

```
# lb clean
```

In this specific case, a `lb clean --binary` would not be enough to clean up the neces-   248
sary stages. The cause for this is that in netboot setups, a different initramfs con-
figuration needs to be used which live-build performs automatically when building
netboot images. Since the initramfs creation belongs to the chroot stage, switch-
ing 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: 249

250

```
$ 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   251
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 filesystem 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: 252

253

```
# lb build
```

In a network boot, the client runs a small piece of software which usually resides on   254
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 `live-image-i386.netboot.tar` archive in the   255
`/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   256
server, the TFTP server and the NFS server.

DHCP server   257

We must configure our network's DHCP server to be sure to give an IP address to the   258
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`   259
in the `/etc/dhcp/dhcpd.conf` configuration file:

260

```
# /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;
}
```

TFTP server   261

This serves the kernel and initial ramdisk to the system at run time.   262

You should install the *tftpd-hpa* package. It can serve all files contained inside a root   263

directory, usually `/srv/tftp`. To let it serve files inside `/srv/debian-live/tftpboot`, run as root the following command:

264

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

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

265

NFS server

266

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.

267

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

268

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

269

270

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

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

271

272

```
# 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.

273

Netboot testing HowTo

274

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

275

To make our life easier, we can use virtualization.

276

Qemu

277

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

278

Edit `/etc/qemu-ifup`:

279

280

```
#!/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`.   281

Launch `qemu` with "`-net nic,vlan=0 -net tap,vlan=0,ifname=tun0`"   282

Webbooting   283

Webbooting is a convenient way of retrieving and booting live systems using the in-   284
ternet as a means. The requirements for webbooting are very few. On the one hand,
you need a medium with a bootloader, an initial ramdisk and a kernel. On the other
hand, a web server to store the squashfs files which contain the filesystem.

Getting the webboot files   285

As usual, you can build the images yourself or use the prebuilt files, which are   286
available on the project's homepage at 「`http://live-systems.org/`」. Using prebuilt im-
ages would be handy for doing initial testing until one can fine tune their own
needs. If you have built a live image you will find the files needed for webboot-
ing in the build directory under `binary/live/`. The files are called `vmlinuz`, `initrd.img`
and `filesystem.squashfs`.

It is also possible to extract those files from an already existing iso image. In order   287
to achieve that, loopback mount the image as follows:

                                                                                                         288

```
# mount -o loop image.iso /mnt
```

The files are to be found under the `live/` directory. In this specific case, it would be   289
`/mnt/live/`. This method has the disadvantage that you need to be root to be able
to mount the image. However, it has the advantage that it is easily scriptable and
thus, easily automatized.

But undoubtedly, the easiest way of extracting the files from an iso image and   290
uploading it to the web server at the same time, is using the midnight commander
or *mc*. If you have the *genisoimage* package installed, the two-pane file manager
allows you to browse the contents of an iso file in one pane and upload the files via
ftp in the other pane. Even though this method requires manual work, it does not
require root privileges.

Booting webboot images   291

While some users will prefer virtualization to test webbooting, we refer to real hard-   292
ware here to match the following possible use case which should only be considered
as an example.

In order to boot a webboot image it is enough to have the components mentioned   293
above, i.e. `vmlinuz` and `initrd.img` in a usb stick inside a directory named `live/` and in-
stall syslinux as bootloader. Then boot from the usb stick and type `fetch=URL/PATH/TO/FILE`
at the boot options. live-boot will retrieve the squashfs file and store it into ram. This

way, it is possible to use the downloaded compressed filesystem as a regular live system. For example:

```
append boot=live components fetch=http://192.168.2.50/images/webboot/filesystem.squashfs
```

Use case: You have a web server in which you have stored two squashfs files, one which contains a full desktop, like for example gnome, and a standard one. If you need a graphical environment for one machine, you can plug your usb stick in and webboot the gnome image. If you need one of the tools included in the second type of image, perhaps for another machine, you can webboot the standard one.

Overview of tools

git

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.

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, among 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 provides the tools to generate a skeleton configuration directory. This could be considered to be similar to tools such as *dh-make*. For more information about these tools, read on, since the remainder of this section discuses the four most important commands. Note that the preceding `lb` is a generic wrapper for live-build 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.

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 the initial skeleton configuration tree.

Issuing `lb config` without any arguments creates the `config/` subdirectory which is populated with some default settings in configuration files, and two skeleton trees named `auto/` and `local/`.

```
$ lb config
[2015-01-06 19:25:58] lb config
P: Creating config tree for a debian/stretch/i386 system
P: Symlinking hooks...
```

git

Using `lb config` without any arguments would be suitable for users who need a very basic image, or who intend to provide a more complete configuration via `auto/config` later (see Managing a configuration for details). 313

Normally, you will want to specify some options. For example, to specify which package manager to use while building the image: 314

315

```
$ lb config --apt aptitude
```

It is possible to specify many options, such as: 316

317

```
$ 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. 318

The `lb build` command 319

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. 320

The `lb clean` command 321

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. 322

The live-boot package 323

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. 324

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. 325

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. 326

The live-config package 327

live-config consists of the scripts that run at boot time after live-boot to configure 328
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.

Managing a configuration

Managing a configuration   330

This chapter explains how to manage a live configuration from initial creation, through   331
successive revisions and successive releases of both the live-build software and the
live image itself.

Dealing with configuration changes   332

Live configurations rarely are perfect on the first try. It may be fine to pass `lb`   333
`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.

Why use auto scripts? What do they do?   334

The `lb config` command stores the options you pass to it in `config/*` files along with   335
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 `–binary-images`, any dependent options that
were defaulted for the old image type may no longer work with the new ones. 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 wrap-   336
pers 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 config-
uration 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.

Use example auto scripts   337

For your convenience, live-build comes with example auto shell scripts to copy and   338
edit. Start a new, default configuration, then copy the examples into it:

                                                                         339

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

Edit `auto/config`, adding any options as you see fit. For instance: 340

341

```
#!/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`.

342

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.

343

Clone a configuration published via Git

344

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 ${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.

345

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

346

347

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

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".

348

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

349

350

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

This enables you to use `lso:` 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:

351

352

```
$ lb config --config lso: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. 353

In any case, remember that every time you will have to build the image as superuser: 354
`lb build`

Customizing contents

Customization overview

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

Build time vs. boot time configuration

Live system configuration options are divided into build-time options which are op-
tions 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.

Stages of the build

The build process is divided into stages, with various customizations applied in se-
quence 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 sys-
tem. This is followed by the **chroot** stage, which completes the construction of ch-
root 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.

Supplement lb config with files 364

Although `lb config` creates a skeletal configuration in the `config/` directory, to ac- 365
complish 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.

Customization tasks 366

The following chapters are organized by the kinds of customization task users typ- 367
ically perform: Customizing package installation, Customizing contents and Cus-
tomizing locale and language cover just a few of the things you might want to
do.

Customizing package installation

Customizing package installation

369

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.

Package sources

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 ${testing} for the ${testing} 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 ${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.

Distribution mirrors 379

The Debian archive is replicated across a large network of mirrors around the world 380
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.

Distribution mirrors used at build time 381

To set the distribution mirrors used at build time to point at a local mirror, it is 382
sufficient to set –`mirror-bootstrap` and –`mirror-chroot-security` as follows.

383

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

The chroot mirror, specified by –`mirror-chroot`, defaults to the –`mirror-bootstrap` 384
value.

Distribution mirrors used at run time 385

The –`mirror-binary*` options govern the distribution mirrors placed in the binary im- 386
age. These may be used to install additional packages while running the live system.
The defaults employ `httpredir.debian.org`, 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.

387

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

Additional repositories 388

You may add more repositories, broadening your package choices beyond what is 389
available in your target distribution. These may be, for example, for backports, ex-
perimental or custom packages. To configure additional repositories, create `config/archives/you`
and/or `config/archives/your-repository.list.binary` files. As with the –`mirror-*` op-
tions, 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   390
the debian-live snapshot repository at live system build time.

391

```
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   392
added to your live system's `/etc/apt/sources.list.d/` directory.

If such files exist, they will be picked up automatically.   393

You should also put the GPG key used to sign the repository into `config/archives/your-repository.`   394
files.

Should you need custom APT pinning, such APT preferences snippets can be placed   395
in `config/archives/your-repository.pref.{binary,chroot}` files and will be automatically
added to your live system's `/etc/apt/preferences.d/` directory.

Choosing packages to install   396

There are a number of ways to choose which packages live-build will install in your   397
image, covering a variety of different needs. You can simply name individual pack-
ages 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 pack-
ages before they are available from a repository.

Package lists   398

Package lists are a powerful way of expressing which packages should be installed.   399
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   400
is determined by your choice of APT utility. See Choosing apt or aptitude for more
details.

Using metapackages   401

The simplest way to populate your package list is to use a task metapackage main-   402
tained by your distribution. For example:

403

```
$ 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   404
predefined lists, task metapackages are not specific to the Live System project. In-
stead, 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: 405

406

```
$ 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: 407

408

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

Local package lists 409

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. 410

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. 411

 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. 412

Local binary package lists 413

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. 414

Generated package lists

<div align="right">415</div>

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`.

<div align="right">416</div>

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:

<div align="right">417</div>

<div align="right">418</div>

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

Using conditionals inside package lists

<div align="right">419</div>

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`.

<div align="right">420</div>

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

<div align="right">421</div>

<div align="right">422</div>

```
#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:

<div align="right">423</div>

<div align="right">424</div>

```
#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`:

<div align="right">425</div>

<div align="right">426</div>

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

The nesting of conditionals is not supported.

<div align="right">427</div>

Removing packages at install time

<div align="right">428</div>

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.

<div align="right">429</div>

Desktop and language tasks

<div align="right">430</div>

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.

<div align="right">431</div>

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.

<div align="right">432</div>

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:

<div align="right">433</div>

<div align="right">434</div>

```
$ 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
```

Kernel flavour and version

<div align="right">435</div>

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.

<div align="right">436</div>

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-586` metapackage.

<div align="right">437</div>

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-3.18.0-trunk-amd64` kernel. You would configure that image as follows:

438

439

```
$ lb config --linux-packages linux-image-3.18.0-trunk
$ echo "deb http://ftp.debian.org/debian/ experimental main" > config/archives/experimental↩
    .list.chroot
```

Custom kernels
440

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.
441

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 repository.
442

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.
443

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:
444

Use an initial ramdisk.
445

Include the union filesystem module (i.e. usually `aufs`).
446

Include any other filesystem modules required by your configuration (i.e. usually `squashfs`).
447

Installing modified or third-party packages
448

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.
449

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` may be of interest, however. The creation of bespoke packages is covered in the Debian New Maintainers' Guide at ⌜`https://www.debian.org/doc/maint-guide/`⌟ and elsewhere.
450

There are two ways of installing modified custom packages: 451

    `packages.chroot`   452

    Using a custom APT repository   453

Using `packages.chroot` is simpler to achieve and useful for "one-off" customizations   454
but has a number of drawbacks, while using a custom APT repository is more time-
consuming to set up.

Using `packages.chroot` to install custom packages   455

To install a custom package, simply copy it to the `config/packages.chroot/` directory.   456
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   457
use `dpkg-name`.

Using `packages.chroot` for installation of custom packages has disadvantages: 458

    It is not possible to use secure APT.   459

    You must install all appropriate packages in the `config/packages.chroot/` directory. 460

    It does not lend itself to storing live system configurations in revision control. 461

Using an APT repository to install custom packages   462

Unlike using `packages.chroot`, when using a custom APT repository you must ensure   463
that you specify the packages elsewhere. See Choosing packages to install for de-
tails.

While it may seem unnecessary effort to create an APT repository to install custom   464
packages, the infrastructure can be easily re-used at a later date to offer updates
of the modified packages.

Custom packages and APT   465

live-build uses APT to install all packages into the live system so will therefore inherit   466
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 num-
ber.

Because of this, you may wish to increment the version number in your custom   467
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.

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.

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.

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/
```

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.   484

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.   485

Passing options to apt or aptitude   486

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`:   487

488

```
$ 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.   489

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.   490

APT pinning   491

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   492

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 ${testing} 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, ${testing}. The following will accomplish this: 493

494

```
$ 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.lis but don't want the user prompted to store wifi passwords in the keyring. This meta-package 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: 495

496

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

Customizing contents

git

Customizing contents

498

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.

499

Includes

500

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:

501

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.

502

Binary local includes: These allow you to add or replace files in the binary image. Please see Binary local includes for more information.

503

Please see Terms for more information about the distinction between the "Live" and "binary" images.

504

Live/chroot local includes

505

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.

506

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:

507

508

```
$ 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:

509

510

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

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

511

Binary local includes

512

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:

513

514

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

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

515

Hooks

516

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

517

Live/chroot local hooks

518

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.

519

Boot-time hooks

520

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.

521

Binary local hooks

522

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

523

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.

Preseeding Debconf questions 524

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. 525

For more information about debconf, please see `debconf(7)` in the *debconf* package. 526

Customizing run time behaviours

Customizing run time behaviours 528

All configuration that is done during run time is done by live-config. Here are some 529
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.

Customizing the live user 530

One important consideration is that the live user is created by live-boot at boot time, 531
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 532
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`

533

```
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,powerdev,sca` 534
as a boot parameter.

It is also possible to change the default username "user" and the default password 535
"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: 536

537

```
$ 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 de- 538
scribed 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/`

Customizing locale and language 539

When the live system boots, language is involved in two steps: 540

the locale generation 541

setting the keyboard configuration 542

The default locale when building a Live system is `locales=en_US.UTF-8`. To define the 543
locale that should be generated, use the `locales` parameter in the `--bootappend-live`
option of `lb config`, e.g.

544

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

Multiple locales may be specified as a comma-delimited list. 545

This parameter, as well as the keyboard configuration parameters indicated be- 546
low, 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 547
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 lay-
outs 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:

548

```
$ 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. 549

Often, only the layout needs to be configured. For example, to get the locale files 550
for German and Swiss German keyboard layout in X use:

551

```
$ 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. 552
For example, to set up a French system with a French-Dvorak layout (called Bepo)
on a TypeMatrix EZ-Reach 2030 USB keyboard, use:

553

```
$ 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-*` 554
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:

555

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

Persistence 556

A live cd paradigm is a pre-installed system which runs from read-only media, like a 557
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 558
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 559
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 560
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 561
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 specify-
ing 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. 562

    a filesystem, identified by its filesystem label. 563

    an image file located on the root of any readable filesystem (even an NTFS parti- 564
    tion of a foreign OS), identified by its filename.

The volume label for overlays must be `persistence` but it will be ignored unless it 565
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 566
can be, for instance, an ext4 partition on a hard disk or on a usb key created with,
e.g.:

567

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

See also Using the space left on a USB stick. 568

If you already have a partition on your device, you could just change the label with 569
one of the following:

570

```
# 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 persis- 571
tence:

572

```
$ 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 573
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:

574

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

Then, create the persistence.conf file adding content and unmount the image file. 575

576

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

Now, reboot into your live medium with the boot parameter "persistence". 577

The persistence.conf file 578

A volume with the label persistence must be configured by means of the persistence.conf 579
file to make arbitrary directories persistent. That file, located on the volume's filesys-
tem root, controls which directories it makes persistent, and in which way.

How custom overlay mounts are configured is described in full detail in the persis- 580
tence.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:

581

```
# 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 582
be copied into the persistence volume, and from then on all changes to these di-
rectories 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.

Using more than one persistence store 583

There are different methods of using multiple persistence store for different use 584
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 585
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 586
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 loca- 587
tions 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 par-
tition 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 588
a `persistence.conf` file in its root. The live-boot man page contains more information
about how to use these labels with legacy names.

Using persistence with encryption 589

Using the persistence feature means that some sensible data might get exposed to 590
risk. Especially if the persistent data is stored on a portable device such as a usb
stick or an external hard drive. That is when encryption comes in handy. Even if
the entire procedure might seem complicated because of the number of steps to be
taken, it is really easy to handle encrypted partitions with live-boot. In order to use
luks, which is the supported encryption type, you need to install *cryptsetup* both
on the machine you are creating the encrypted partition with and also in the live
system you are going to use the encrypted persistent partition with.

To install *cryptsetup* on your machine: 591

592

```
# apt-get install cryptsetup
```

To install *cryptsetup* in your live system, add it to your package-lists: 593

594

```
$ lb config
$ echo "cryptsetup" > config/package-lists/encryption.list.chroot
```

Once you have your live system with *cryptsetup*, you basically only need to create a
new partition, encrypt it and boot with the `persistence` and `persistence-encryption=luks`
parameters. We could have already anticipated this step and added the boot param-
eters following the usual procedure:

595

596

```
$ lb config --bootappend-live "boot=live components persistence persistence-encryption=luks↩
    "
```

Let's go into the details for all of those who are not familiar with encryption. In the
following example we are going to use a partition on a usb stick which corresponds
to `/dev/sdc2`. Please be warned that you need to determine which partition is the
one you are going to use in your specific case.

597

The first step is plugging in your usb stick and determine which device it is. The rec-
ommended method of listing devices in live-manual is using `ls -l /dev/disk/by-id`.
After that, create a new partition and then, encrypt it with a passphrase as fol-
lows:

598

599

```
# cryptsetup --verify-passphrase luksFormat /dev/sdc2
```

Then open the luks partition in the virtual device mapper. Use any name you like.
We use **live** here as an example:

600

601

```
# cryptsetup luksOpen /dev/sdc2 live
```

The next step is filling the device with zeros before creating the filesystem:

602

603

```
# dd if=/dev/zero of=/dev/mapper/live
```

Now, we are ready to create the filesystem. Notice that we are adding the label
`persistence` so that the device is mounted as persistence store at boot time.

604

605

```
# mkfs.ext4 -L persistence /dev/mapper/live
```

To continue with our setup, we need to mount the device, for example in `/mnt`.

606

607

```
# mount /dev/mapper/live /mnt
```

And create the `persistence.conf` file in the root of the partition. This is, as explained
before, strictly necessary. See The persistence.conf file.

608

609

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

Then unmount the mount point: 610

                                                                                   611

```
# umount /mnt
```

And optionally, although it might be a good way of securing the data we have just   612
added to the partition, we can close the device:

                                                                                   613

```
# cryptsetup luksClose live
```

Let's summarize the process. So far, we have created an encryption capable live   614
system, which can be copied to a usb stick as explained in Copying an ISO hybrid
image to a USB stick. We have also created an encrypted partition, which can be lo-
cated in the same usb stick to carry it around and we have configured the encrypted
partition to be used as persistence store. So now, we only need to boot the live sys-
tem. At boot time, live-boot will prompt us for the passphrase and will mount the
encrypted partition to be used for persistence.

Customizing the binary image

Customizing the binary image 616

Bootloaders 617

live-build uses *syslinux* and some of its derivatives (depending on the image type) 618
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` 619
and edit the files in there. If you do not want to bother modifying all supported boot-
loader configurations, only providing a local customized copy of one of the bootload-
ers, e.g. **isolinux** in `config/bootloaders/isolinux` is enough too, depending on your
use case.

When modifying one of the default themes, if you want to use a personalized back- 620
ground image that will be displayed together with the boot menu, add a splash.png
picture of 640x480 pixels. Then, remove the splash.svg file.

There are many possibilities when it comes to making changes. For instance, sys- 621
linux 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** 622
file specifying the timeout in units of 1/10 seconds. A modified **isolinux.cfg** to boot
after five seconds would be similar to this:

623

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

ISO metadata 624

When creating an ISO9660 binary image, you can use the following options to add 625
various textual metadata for your image. This can help you easily identify the ver-
sion or configuration of an image without booting it.

`LB_ISO_APPLICATION/-iso-application NAME`: This should describe the application that 626
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, 627
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 628
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. 629
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.

Customizing Debian Installer

Customizing Debian Installer

631

Live system images can be integrated with Debian Installer. There are a number   632
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"   633
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".

Types of Debian Installer

634

The three main types of installer are: 635

"Normal" Debian Installer: This is a normal live system image with a separate   636
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   637
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   638
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   639
initrd which (when selected from the appropriate bootloader) launches into an in-
stance of the Debian Installer.

Installation will proceed in an identical fashion to the "normal" installation described   640
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   641
items such as bootloaders and local users, etc.

Note: to support both normal and live installer entries in the bootloader of the same   642
live medium, you must disable live-installer by preseeding `live-installer/enable=false`.

"Desktop" Debian Installer: Regardless of the type of Debian Installer included,   643
`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 im-   644
ages, 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 586 \
        --debian-installer live
$ echo debian-installer-launcher >> config/package-lists/my.list.chroot
```

Customizing Debian Installer by preseeding

As described in the Debian Installer Manual, Appendix B at ⌜https://www.debian.org/releases/stable/i386/apb
, "Preseeding provides a way to set answers to questions asked during the instal-
lation 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
```

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/`.

Projekt

Contributing to the project

Contributing to the project

653

When submitting a contribution, please clearly identify its copyright holder and in- 654
clude 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. 655
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 ${project} uses Git as version control system and source code management. As 656
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: 657

   Non fast-forward pushes. 658

   Merge commits. 659

   Adding or removing tags or branches. 660

Even though all commits might be revised, we ask you to use your common sense 661
and make good commits with good commit messages.

   Write commit messages that consist of complete, meaningful sentences in English, 662
   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 663
   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. 664
   Make one different commit for each change you make.

Making changes

665

In order to push to the repositories, you must follow the following procedure. Here 666
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: 667

668

```
$ 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:

669

670

```
$ 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:

671

672

```
$ 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:

673

674

```
 $ 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.

675

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:

676

677

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

Push the commit to the server:

678

679

```
$ git push
```

Reporting bugs

git

Reporting bugs

681

Live systems are far from being perfect, but we want to make it as close as possible   682
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: 683

Always check first the image status updates on our homepage at ⌜`http://live-systems.org/`⌟   684
for known issues.

Before submitting a bug report always try to reproduce the bug with the **most**   685
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   686
(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.

Known issues   687

Since Debian **testing** and Debian **unstable** distributions are moving targets, when   688
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**   689
or **unstable**, but rather, use **stable**. live-build always defaults to the **stable** re-
lease.

Currently known issues are listed under the section 'status' on our homepage at   690
⌜`http://live-systems.org/`⌟ .

It is out of the scope of this manual to train you to correctly identify and fix problems   691
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).

Rebuild from scratch   692

To ensure that a particular bug is not caused by an uncleanly built system, please al-   693
ways rebuild the whole live system from scratch to see if the bug is reproducible.

Use up-to-date packages   694

Using outdated packages can cause significant problems when trying to reproduce   695
(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.

Collect information

696

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.

697

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

698

Architecture of the host system

699

Distribution of the host system

700

Version of live-build on the host system

701

Version of *debootstrap* on the host system

702

Architecture of the live system

703

Distribution of the live system

704

Version of live-boot on the live system

705

Version of live-config on the live system

706

Version of live-tools on the live system

707

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).

708

709

```
# 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.

710

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).

711

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`.

712

Isolate the failing case if possible

713

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

714

broke, it may be that you are including too much in each change set and should develop in smaller increments.

Use the correct package to report the bug against 715

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. 716

However, we would appreciate it if you try to narrow it down according to where the bug appears. 717

At build time while bootstrapping 718

live-build first bootstraps a basic Debian system with *debootstrap*. 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. 719

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. 720

At build time while installing packages 721

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. 722

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. 723

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. 724

At boot time 725

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. 726

At run time 727

If a package was successfully installed, but fails while actually running the Live system, this is probably a bug in the live system. However: 728

Do the research 729

Before filing the bug, please search the web for the particular error message or 730
symptom you are getting. As it is highly unlikely that you are the only person expe-
riencing 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 731
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 732
and live-tools to see whether something similar has already been reported.

Where to report bugs 733

The ${project} keeps track of all bugs in the Bug Tracking System (BTS). For infor- 734
mation on how to use the system, please see ⌜`https://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 735
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 736
and others) should **not** be reported to the Debian BTS unless they can be also re-
produced on a Debian system using official Debian packages.

Coding Style

git

Coding Style 738

This chapter documents the coding style used in live systems. 739

Compatibility 740

Don't use syntax or semantics that are unique to the Bash shell. For example, 741
the use of array constructs.

Only use the POSIX subset - for example, use $(foo) over 'foo'. 742

You can check your scripts with 'sh -n' and 'checkbashisms'. 743

Make sure all shell code runs with 'set -e'. 744

Indenting 745

Always use tabs over spaces. 746

Wrapping 747

Generally, lines are 80 chars at maximum. 748

Use the "Linux style" of line breaks: 749

Bad: 750

751

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

Good: 752

753

```
if foo
then
        bar
fi
```

The same holds for functions: 754

Bad: 755

756

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

Good: 757

758

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

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 `_LB_` prefix.

Local variables start with live-build

_
`_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
```

Miscellaneous

Use ”|” (without the surround quotes) as a separator in calls to sed, e.g. ”`sed -e`
`'s|foo|bar|'`” (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.

Projektabläufe

git

Procedures

This chapter documents the procedures within the ${project} for various tasks that need cooperation with other teams in Debian.

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.

Point Releases

Again, we need updated mirrors of debian and debian-security.

Images are built and mirrored on cdimage.debian.org.

Send announcement mail.

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.

Point release announcement template

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

```
$ sed \
    -e 's|@MAJOR@|9.0|g' \
    -e 's|@MINOR@|9.0.1|g' \
    -e 's|@CODENAME@|stretch|g' \
    -e 's|@ANNOUNCE@|2017/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:

  <https://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>.
```

Git repositories

Git repositories 804

The list of all the available repositories of the ${project} can be found at ⌜http://live-systems.org/gitwe 805
. The project's git URLs have the form: `protocol://live-systems.org/git/repository`.
Thus, in order to clone live-manual read-only, launch:

806

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

Or, 807

808

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

Or, 809

810

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

The cloning addresses with write permission have the form: `git@live-systems.org:/repository`. 811

So, again, to clone live-manual over ssh you must type: 812

813

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

The git tree is made up of several different branches. The **debian** and the **debian-** 814
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 815
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:

816

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

The **debian-next** branch, which is not always fast-forward, is where all the changes 817
are committed first before being merged into the **debian** branch. To make an anal-
ogy, 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.

Handling multiple repositories 818

If you intend to clone several of the live systems repositories and want to switch to 819
the **debian-next** branch right away to check the latest code, write a patch or con-
tribute 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:

820

```
$  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

Beispiele

git

Beispiele

Examples   824

This chapter covers example builds for specific use cases with live systems. If you   825
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.

Using the examples   826

To use these examples you need a system to build them on that meets the require-   827
ments 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   828
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 Dis-
tribution 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:

829

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

Tutorial 1: A default image   830

Use case: Create a simple first image, learning the basics of live-build.   831

In this tutorial, we will build a default ISO hybrid live system image containing only   832
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: 833

834

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

Examine the contents of the `config/` directory if you wish. You will see stored here a   835
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`.   836

837

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

Assuming all goes well, after a while, the current directory will contain `live-image-i386.hybrid.is`   838
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.

Tutorial 2: A web browser utility

839

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

840

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.

841

842

```
$ mkdir tutorial2
$ cd tutorial2
$ lb config
$ 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.

843

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

844

845

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

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

846

Tutorial 3: A personalized image

847

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.

848

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.

849

First revision

850

851

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

Edit `auto/config` to read as follows: 852

853

```
#!/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   854
created:

855

```
$ lb config
```

Now populate your local package list: 856

857

```
$ 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   858
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: 859

860

```
# lb build
```

Note that unlike in the first two tutorials, we no longer have to type `2>&1 | tee`   861
`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   862
initialize our `git` repository, adding only the auto scripts we just created, and then
make the first commit:

863

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

Second revision   864

In this revision, we're going to clean up from the first build, add the *vlc* package to   865
our configuration, rebuild, test and commit.

The `lb clean` command will clean up all generated files from the previous build   866
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.

867

```
# lb clean
```

Now append the *vlc* package to our local package list in `config/package-lists/my.list.chroot`: 868

869

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

Build again: 870

871

```
# lb build
```

Test, and when you're satisfied, commit the next revision: 872

873

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

Of course, more complicated changes to the configuration are possible, perhaps 874
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 customiza- 875
tion are possible, even just using the few features explored in these simple exam-
ples, 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.

A VNC Kiosk Client 876

Use case: Create an image with live-build to boot directly to a VNC server. 877

Make a build directory and create an skeletal configuration inside it, disabling rec- 878
ommends 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.

879

```
$ 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 recom- 880
mended packages to make your image work properly.

An easy way to list recommends is using *apt-cache*. For example: 881

882

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

In this example we found out that we had to re-include several packages recom-
mended 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.

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 "in-
trusive" optimizations. Of particular note, we use `–debootstrap-options` to create a
minimal system from scratch.

```
$ lb config --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

895

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

Now, build the image in the usual way: 896

897

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

On the author's system at the time of writing this, the above configuration produced   898
a 110MB image. This compares favourably with the 192MB image produced by the
default configuration in Tutorial 1.

Leaving off APT's indices with `–apt-indices false` saves a fair amount of space, the   899
tradeoff being that you need to do an `apt-get update` before using *apt* in the live sys-
tem. Dropping recommended packages with `–apt-recommends false` saves some addi-
tional 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   900
`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 91MB. How-
ever, 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 com-
ment header warns, may have unforeseen consequences. That is why using a mini-
mal *debootstrap* is the recommended way of achieving this goal.

A localized GNOME desktop and installer   901

Use case: Create a GNOME desktop image, localized for Switzerland and including   902
an installer.

 We want to make an iso-hybrid image for i386 architecture using our preferred   903
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.   904
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:

905

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

Now we can search for the appropriate tasks, first with: 906

907

```
$ 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 \
    --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. The 586 kernel flavour, which is currently necessary for the launcher to work properly, will be included by default.

Anhang

Style guide

Style guide 915

Guidelines for authors 916

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. 917

Note: Authors should first read Contributing to this document 918

Linguistic features 919

Use plain English 920

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. 921

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. 922

Variety of English 923

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. 924

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. 925

Be balanced 926

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. 927

Be politically correct 928

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. 929

Be concise 930

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. 931

Minimize translation work 932

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. 933

Be coherent 934

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. 935

Be cohesive 936

Use as many text-forming devices as necessary to make your text cohesive and unambiguous. (Text-forming devices are linguistic markers such as connectors). 937

Be descriptive 938

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. 939

Dictionary 940

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. 941

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. 942

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. 943

Using a dictionary of collocations is recommended. They are extremely helpful when it comes to know which words usually occur together. 944

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. 945

False friends, idioms and other idiomatic expressions 946

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. 947

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. 948

Sometimes, idioms might be difficult to understand even for native speakers of English!

Avoid slang, abbreviations, contractions... 949

Even though you are encouraged to use plain, everyday English, technical writing 950
belongs to the formal register of the language.

Try to avoid slang, unusual abbreviations that are difficult to understand and above 951
all contractions that try to imitate the spoken language. Not to mention typical irc
and family friendly expressions.

Procedures 952

Test before write 953

It is important that authors test their examples before adding them to live-manual 954
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 955

When providing an example try to be as specific as you can. An example is, after 956
all, just an example.

It is often better to use a line that only applies to a specific case than using abstrac- 957
tions 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 958
dangerous commands that, if misused, may cause data loss or other similar undesir-
able effects. In this case you should provide a thorough explanation of the possible
side effects.

External links 959

Links to external sites should only be used when the information on those sites is 960
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 961
links.

Avoid branding and things that violate the license under which the manual is 962
published

Try to avoid branding as much as possible. Keep in mind that other downstream 963
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 964

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 965

- Brainstorm!. You need to organize your ideas first in a logical sequence of events. 966

- Once you have somehow organized those ideas in your mind write a first draft. 967

- Revise grammar, syntax and spelling. Keep in mind that the proper names of the 968 releases, such as ${testing} or sid, should not be capitalized when referred to as code names. In order to check the spelling you can run the "spell" target. i.e. `make spell`

- Improve your statements and redo any part if necessary. 969

Chapters 970

Use the conventional numbering system for chapters and subtitles. e.g. 1, 1.1, 971 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 972 also use ordinal numbers: First, second, third ... or First, Then, After that, Finally ... Alternatively you can use bulleted items.

Markup 973

And last but not least, live-manual uses 974 「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:

975

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

Here are some markup examples that may prove useful: 976

- For emphasis/bold text: 977

978

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

produces: **foo** or **foo**. Use it to emphasize certain key words. 979

- For italics: 980

981

```
/{foo}/
```

produces: *foo*. Use them e.g. for the names of Debian packages. 982

- For monospace: 983

984

```
#{foo}#
```

produces: `foo`. Use it e.g. for the names of commands. And also to highlight some key words or things like paths.   985

- For code blocks: 986

                                                                                987

```
code{

 $ foo
 # bar

}code
```

produces: 988

                                                                                989

```
 $ foo
 # bar
```

Use `code{` to open and `}code` to close the tags. It is important to remember to leave   990
a space at the beginning of each line of code.

Guidelines for translators   991

This section deals with some general considerations to be taken into account when   992
translating the contents of live-manual.

As a general recommendation, translators should have read and understood the   993
translation rules that apply to their specific languages. Usually, translation groups
and mailing lists provide information on how to produce translated work that com-
plies with Debian quality standards.

Note: Translators should also read Contributing to this document. In particular the   994
section Translation

Translation hints   995

Comments   996

The role of the translator is to convey as faithfully as possible the meaning of words,   997
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   998
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   999

It is perfectly acceptable however, to include a word or an expression in brackets   1000
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 1001

Documents written in English make an extensive use of the impersonal form "you". 1002
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 1003

The trap of "false friends" explained before especially applies to translators. Double 1004
check the meaning of suspicious false friends if in doubt.

Markup 1005

Translators working initially with **pot** files and later on with **po** files will find many 1006
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 1007

Even though the code blocks are usually untranslatable, including them in the trans- 1008
lation 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 transla- tors 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 1009

The translated texts need to have the exact same newlines as the original texts. Be 1010
careful to press the "Enter" key or type
n 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 1011
same length as the English version. That is nearly impossible.

Untranslatable strings 1012

Translators should never translate: 1013

- The code names of releases (which should be written in lowercase) 1014

- The names of programs 1015

- The commands given as examples 1016

- Metadata (often between colons **:metadata:**) 1017

- Links 1018

- Paths 1019