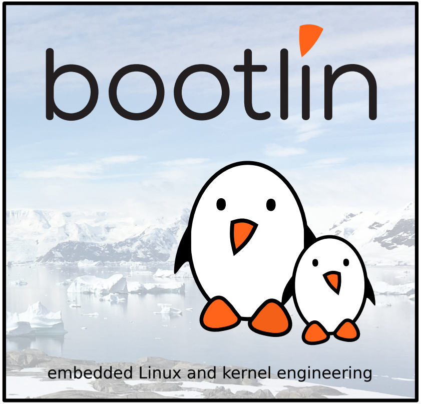
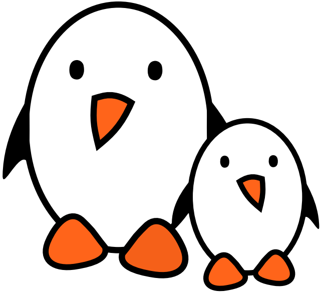
Building embedded Debian / Ubuntu systems with ELBE

Köry Maincent
kory.maincent@bootlin.com



Köry Maincent

- Embedded Linux engineer at Bootlin
  - Embedded Linux, Linux kernel, Yocto, Buildroot **expertise**
  - **Development**, consulting and training
  - Strong open-source focus
- Open-source contributor
  - Contributed Ubuntu support to ELBE
  - Used ELBE to build Ubuntu systems for an ARM32 i.MX6 platform and an ARM64 Rockchip RK3399 platform
- Living in **Toulouse**, France



Agenda

- System integration: available options
- Overview of ELBE
- Building simple Debian/Ubuntu images with ELBE
- Customizing the images contents

System integration: several possibilities

	Pros	**Cons**
Building everything manually	Full flexibility Learning experience	Dependency hell Need to understand a lot of details Version compatibility Lack of reproducibility
Binary distribution Debian, Ubuntu, Fedora, etc.	Easy to create and extend Large set of existing packages Well-known tools for non-embedded experts Robust and regular security updates	Hard to customize Hard to optimize (boot time, size) Hard to rebuild the full system from source Large system Uses native compilation (slow) No well-defined mechanism to generate an image Lots of mandatory dependencies Not available for all architectures
Build systems Buildroot, Yocto, PTXdist, etc.	Nearly full flexibility Built from source: customization and optimization are easy Fully reproducible Uses cross-compilation Have embedded specific packages not necessarily in desktop distros Make more features optional	Not as easy as a binary distribution Build time

Debian build systems

Several projects have been created to automate the process of building and customizing a Debian image:

- Hand-made scripts
- ELBE
- Debos
- Isar

Debian build systems

Several projects have been created to automate the process of building and customizing a Debian image:

- Hand-made scripts
  - Hardly reproducible and maintable
  - Everybody rolls his own
- ELBE
- Debos
- Isar

Debian build systems

Several projects have been created to automate the process of building and customizing a Debian image:

- Hand-made scripts
- ELBE
  - First release in 2015
  - Python code to use generic Debian tools
  - Only supported Debian, but we (Bootlin) contributed Ubuntu support
  - `https://elbe-rfs.org/`
  - **The focus of this talk**
- Debos
- Isar

Debian build systems

Several projects have been created to automate the process of building and customizing a Debian image:

- Hand-made scripts
- ELBE
- Debos
  - Image and partition customizable
  - Possibility to tune the rootfs
  - Can not build custom packages from source
  - Written in Go
  - `https://github.com/go-debos/debos`
- Isar

Debian build systems

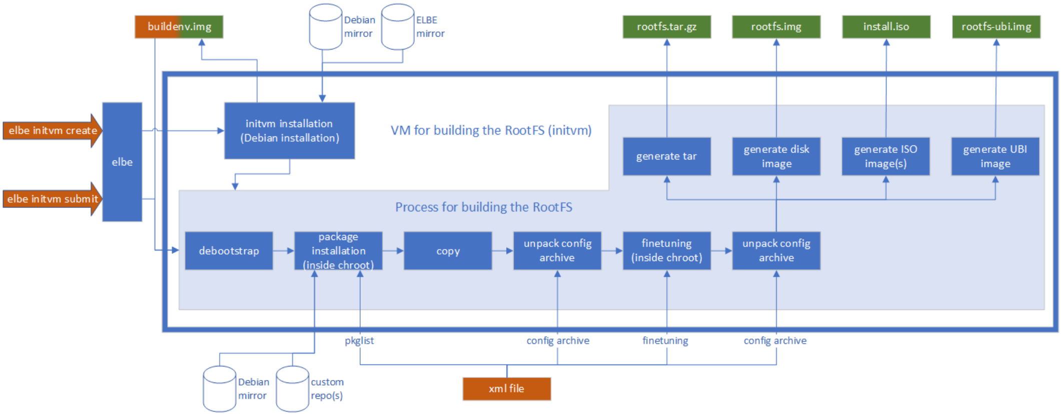
Several projects have been created to automate the process of building and customizing a Debian image:

- Hand-made scripts
- ELBE
- Debos
- Isar
  - Uses bitbake, needs Yocto knowledge
  - Not tested (less active than ELBE?)
  - `https://github.com/ilbers/isar`

ELBE advantages

- Builds a Debian distribution
  - Powerful package management
  - Huge amount of packages
  - Let the Debian/Ubuntu maintainers do all the work on packages
  - Have reliable and regular security updates
- Build your own packages
- Manage licences
- Several architectures, several image generation options
- Tune your rootfs

Overall ELBE process



Source: *https://wiki.dh-electronics.com/index.php/ELBE_Overview*

ELBE: getting started

1. Download ELBE from its Git repository
2. Create the *initvm*, a Debian virtual machine that includes the ELBE daemon.

```
$ ./elbe initvm create
```

3. Then, after each reboot, you need to make sure the *initvm* is started:

```
$ ./elbe initvm start
```

ELBE: build a basic Debian or Ubuntu image

- In ELBE, the system to generate is described by an XML file
- To build a Debian system for the BeagleBone Black, including bootloader and Linux kernel:

```
$ ./elbe initvm submit examples/armhf-ti-beaglebone-black.xml
```

  The build takes approximately 50 min on my laptop

- To build a basic Ubuntu system, with no bootloader or kernel:

```
$ ./elbe initvm submit examples/armhf-ubuntu.xml
```

  The build takes approximately 30 minutes

ELBE: result directory

Contents of the result directory, with the `--build-sdk` option enabled:

- `bin-cdrom.iso`
- `image.tgz`
- `license-*`
- `setup-elbe...sh`
- `source.xml`
- `src-cdrom.iso`
- `sysroot.tar.xz`

```
$ ls elbe-build-20200903-113635
bin-cdrom.iso
elbe-report.txt
image.tgz
licence-chroot.txt
licence-chroot.xml
licence-sysroot-host.txt
licence-sysroot-host.xml
licence-sysroot-target.txt
licence-sysroot-target.xml
licence-target.txt
licence-target.xml
log.txt
setup-elbe-sdk-arm-[...]-armhf-ubuntu-1.0.sh
source.xml
src-cdrom.iso
sysroot.tar.xz
validation.txt
```

ELBE: contents of the XML file

- Global node:
- Project node:
- Target node:

ELBE: contents of the XML file

- Global node:

```
<ns0:RootFileSystem ... >
...
</ns0:RootFileSystem>
```

- Project node:
- Target node:

ELBE: contents of the XML file

- Global node:
- Project node:

```
<project>
  <name>Image name</name>
  <version>1.0</version>
  <description>
    Image description
  </description>
  <buildtype>armhf</buildtype>
  <mirror>
    <primary_host>ftp.de.debian.org</primary_host>
    <primary_path>/debian</primary_path>
    <primary_proto>http</primary_proto>
  </mirror>
  <suite>buster</suite>
</project>
```

- Target node:

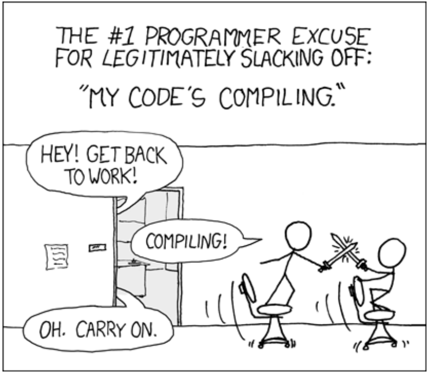
ELBE: contents of the XML file

- Global node:
- Project node:
- Target node:

```
<target>
  <hostname>myImage</hostname>
  <domain>tec.linutronix.de</domain>
  <passwd>foo</passwd>
  <console>ttyS0,115200</console>
  <images> ...  </images>
  <fstab> ...  </fstab>
  <package> ... </package>
  <finetuning> ... </finetuning>
  <pkg-list> ... </pkg-list>
</target>
```

ELBE: day to day work

- The ELBE `submit` command allows to build an image from scratch
  - Builds all parts described in the XML file in one command
  - Good for releases/deliveries
  - But rebuilds everything!
- The ELBE `control` command allows to work in a more-fine grained way
  - Doesn't build all parts described in the XML file
  - Good for day-to-day work, image adjustement and customization



ELBE: using the control command (1/2)

- Create a project

```
$ ./elbe control create_project
/var/cache/elbe/0a7b1788-b2ab-4b53-9319-0a810dab30d9
$ PRJ="/var/cache/elbe/0a7b1788-b2ab-4b53-9319-0a810dab30d9"
```

- Define the image/system to build based on its XML file

```
$ ./elbe control set_xml $PRJ armhf-ti-beaglebone-black.xml
```

- Start the build and wait until it completes

```
$ ./elbe control build $PRJ
$ ./elbe control wait_busy $PRJ
```

ELBE: using the control command (2/2)

- Now you can update/tweak your XML file, and restart the build

```
$ ./elbe control set_xml $PRJ armhf-ti-beaglebone-black.xml
$ ./elbe control build $PRJ
$ ./elbe control wait_busy $PRJ
```

- And retrieve the build results

```
$ ./elbe control get_files $PRJ
$ ./elbe control get_file $PRJ sdcard.img.tar.gz
```

Image customization

ELBE allows to

- Do various tweaks on the resulting filesystem from the XML file
- Add more files/directories to your rootfs with an overlay
- Add Debian packages to the image
- Build your own packages
- Add your packages to the delivery XML image file

Customize: tune your rootfs/image

```
<finetuning>
```

- Copy or move files: bootloader and kernel images in `/boot`
- Use shell commands
- Remove useless files/directories to shrink the image size
- Extract file from chroot in the initvm to the output build directory

```
</finetuning>
```

- `https://elbe-rfs.org/docs/sphinx/article-elbe-schema-reference.html#type-finetuning`

Customize: tune your rootfs/image

```
<finetuning>
```

- Copy or move files: bootloader and kernel images in `/boot`

```
<cp path="/usr/lib/u-boot/am335x_boneblack/MLO">/boot/MLO</cp>
<cp path="/usr/lib/u-boot/am335x_boneblack/u-boot.img">/boot/u-boot.img</cp>
<mv path="/usr/lib/linux-image-*-armmp/am335x-boneblack.dtb">/boot/am335x-boneblack.dtb</mv>
<mv path="/boot/initrd.img-*-armmp">/boot/initrd.img-armmp</mv>
<mv path="/boot/vmlinuz-*-armmp">/boot/vmlinuz-armmp</mv>
```

- Use shell commands
- Remove useless files/directories to shrink the image size
- Extract file from chroot in the initvm to the output build directory

```
</finetuning>
```

- `https://elbe-rfs.org/docs/sphinx/article-elbe-schema-reference.html#type-finetuning`

Customize: tune your rootfs/image

```
<finetuning>
```

- Copy or move files: bootloader and kernel images in /boot
- Use shell commands

```
  <command>echo "uenvcmd=setenv bootargs 'console=ttyO0,115200 root=/dev/mmcblk0p2';
load mmc 0:1 0x84000000 vmlinuz-armmp;load mmc 0:1 0x82000000 am335x-boneblack.dtb;
load mmc 0:1 0x88000000 initrd.img-armmp;bootz 0x84000000 0x88000000:\${filesize} 0x82000000" >
/boot/uEnv.txt</command>
```

- Remove useless files/directories to shrink the image size
- Extract file from chroot in the initvm to the output build directory

```
</finetuning>
```

- https://elbe-rfs.org/docs/sphinx/article-elbe-schema-reference.html#type-finetuning

Customize: tune your rootfs/image

```
<finetuning>
```

- Copy or move files: bootloader and kernel images in /boot
- Use shell commands
- Remove useless files/directories to shrink the image size

```
  <rm>/var/cache/apt/archives/*.deb</rm>
  <rm>/var/cache/apt/*.bin</rm>
  <rm>/var/lib/apt/lists/ftp*</rm>
```

- Extract file from chroot in the initvm to the output build directory

```
</finetuning>
```

- https://elbe-rfs.org/docs/sphinx/article-elbe-schema-reference.html#type-finetuning

Customize: tune your rootfs/image

```
<finetuning>
```

- Copy or move files: bootloader and kernel images in `/boot`
- Use shell commands
- Remove useless files/directories to shrink the image size
- Extract file from chroot in the initvm to the output build directory

```
  <artifact>/usr/lib/u-boot/am335x_boneblack/MLO</artifact>
  <artifact>/boot/am335x-boneblack.dtb</artifact>
```

```
</finetuning>
```

- https://elbe-rfs.org/docs/sphinx/article-elbe-schema-reference.html#type-finetuning

Customize: add an overlay to the image

- An **overlay** is a set of files/directories to copy over the root filesystem, at end of the build process
- Create the contents of the overlay

```
$ mdkir -p overlay/etc/ssh/
$ cp ssh_config overlay/etc/ssh/
```

- Load the overlay contents in the project. They will be stored base64-encoded into the XML file.

```
$ ./elbe chg_archive project.xml overlay
$ cat project.xml
...
<archive>QlpoOTFBWSZTWcCETrAAASl/hciQAEBKd//wf+9d0f/v/+EAAIAIAAhQA9vTnIjbbt3GnQSimBCe
o00elJ6T8pMQ/VPKPUND1DQZAD1GNQaaJo0jU2qNE2o2iAZAPRBgmmgMIaASIkJiJk2qb0hqPQg0
...
gQQryzKUut03vhovrNrCuxRapzudUWmgdIumf09YPKi0a0FJL/i7kinChIYEInWA
</archive></ns0:RootFileSystem>
```

Customize: add a Debian package

- Adding a Debian package from official repository is as easy as listing it in the `<pkg-list>` XML node.

project.xml

```xml
<pkg-list>
  <pkg>openssh-server</pkg>
</pkg-list>
```

Customize: build your packages

- In addition to packages from the official Debian repository, one will often want to build custom packages
  - For a bootloader or kernel image configured specifically for the platform
  - For a customized variant of packages available in the official repositories
  - For in-house/custom applications and libraries
- The following steps must be followed
  1. Follow the Debian packaging procedure by *debianizing* the source code.
  2. Add your *debianized* package to the image.
  3. Build your package with ELBE

Build your packages: debianize the source

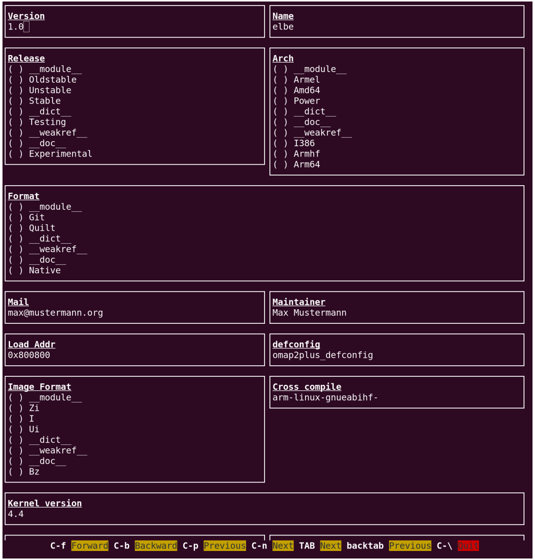
- For some well-known packages (U-Boot, Barebox, Linux), use the `debianize` command to generates some sane default providing a complete and usable `debian/` folder
- This command will show an UI that allows to set the configuration.
- The basics items are the version, the name, the Release state, the architecture, the configurations and some information relative to the owner.

Build your packages: debianize the source

```
$ export PATH=$PATH:`pwd`
$ cd ../linux
$ elbe debianize
```

Build your packages: debianize the source

```
$ export PATH=$PATH:`pwd`
$ cd ../linux
$ elbe debianize
```

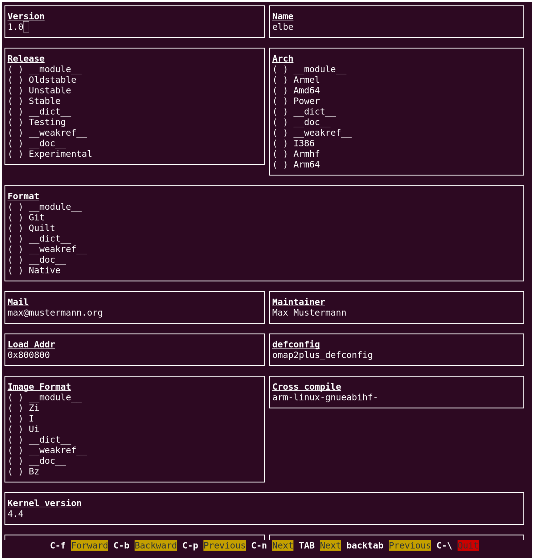


```
Version                                  Name
1.0                                      elbe

Release                                  Arch
( ) __module__                           ( ) __module__
( ) Oldstable                            ( ) Armel
( ) Unstable                             ( ) Amd64
( ) Stable                               ( ) Power
( ) __dict__                             ( ) __dict__
( ) Testing                              ( ) __doc__
( ) __weakref__                          ( ) __weakref__
( ) __doc__                              ( ) I386
( ) Experimental                         ( ) Armhf
                                         ( ) Arm64

Format
( ) __module__
( ) Git
( ) Quilt
( ) __dict__
( ) __weakref__
( ) __doc__
( ) Native

Mail                                     Maintainer
max@mustermann.org                       Max Mustermann

Load Addr                                defconfig
0x800800                                 omap2plus_defconfig

Image Format                             Cross compile
( ) __module__                           arm-linux-gnueabihf-
( ) Zi
( ) I
( ) Ui
( ) __dict__
( ) __weakref__
( ) __doc__
( ) Bz

Kernel version
4.4

C-f Forward C-b Backward C-p Previous C-n Next TAB Next backtab Previous C-\ Quit
```

Build your packages: debianize the source

```
$ export PATH=$PATH:`pwd`
$ cd ../linux
$ elbe debianize
```

```
$ ls debian
changelog
compat
control
copyright
linux-headers-4.14-kernel.install
linux-image-4.14-kernel.install
linux-libc-dev-4.14-kernel.install
postinst
postrm
preinst
prerm
rules
source
```



```
Version                                  Name
1.0                                      elbe

Release                                  Arch
( ) __module__                           ( ) __module__
( ) Oldstable                            ( ) Armel
( ) Unstable                             ( ) Amd64
( ) Stable                               ( ) Power
( ) __dict__                             ( ) __dict__
( ) Testing                              ( ) __doc__
( ) __weakref__                          ( ) __weakref__
( ) __doc__                              ( ) I386
( ) Experimental                         ( ) Armhf
                                         ( ) Arm64

Format
( ) __module__
( ) Git
( ) Quilt
( ) __dict__
( ) __weakref__
( ) __doc__
( ) Native

Mail                                     Maintainer
max@mustermann.org                       Max Mustermann

Load Addr                                defconfig
0x800800                                 omap2plus_defconfig

Image Format                             Cross compile
( ) __module__                           arm-linux-gnueabihf-
( ) Zi
( ) I
( ) Ui
( ) __dict__
( ) __weakref__
( ) __doc__
( ) Bz

Kernel version
4.4

      C-f Forward C-b Backward C-p Previous C-n Next TAB Next backtab Previous C-\ Quit
```

Build your packages: debianize the source

- For other packages, you have to do it manually by creating the required files for debianizing your package.
- The information about these files are in the following link
- `https://www.debian.org/doc/manuals/maint-guide/dreq.en.html`
- Use or inspire yourself from already debianized packages if you can

Build your packages: build process

- Packages are built using the Debian *pbuilder* tool, which builds inside a *chroot*. This *chroot* needs to be created once:

```
$ elbe pbuilder create --xmlfile=project.xml --writeproject=project.prj --cross
$ PRJ=$(cat project.prj)
```

- Go to the source directory of the package to build, create the output directory

```
$ cd ../linux
$ mkdir ../out
```

- Start the build. By default, uses native build with Qemu, `--cross` enables cross-building.

```
$ elbe pbuilder build --cross --project $PRJ --out ../out
```

- Grab the results from the `out` folder

```
$ ls ../out
linux-headers-4.14-kernel_1.0_armhf.deb
linux-image-4.14-kernel_1.0_armhf.deb
linux-libc-dev-4.14-kernel_1.0_armhf.deb
```

Build your packages: add your packages to the image

- When the `elbe pbuilder` command completes, the package is automatically added to the local repository in the *initvm* project directory ($PRJ).
- You only need to add your package to the `<pkg-list>` node in the XML file to bring it into the image.

project.xml

```
<pkg-list>
  <pkg>linux-image-4.14-kernel</pkg>
  <pkg>linux-headers-4.14-kernel</pkg>
</pkg-list>
```

Build your package: automatically build the package

- The procedure describes so far, which uses `elbe pbuilder` manually is perfect during development
- Allows to quickly rebuild just the package that needs to be rebuilt
- For a final release, one will want a procedure that rebuilds everything: all packages, and the image.
- This can be done by adding a `<pbuilder>` node to the XML file:

project.xml

```xml
<pbuilder>
  <git revision="xxx">git@github.com:kmaincent/linux.git</git>
</pbuilder>
```

- Currently, the build of packages described in the `<pbuilder>` node are built natively. There are patches on the mailing list to enable cross-compilation, which we have successfully used.

Tip: avoid rebuilding packages

- When creating a new project, you may not want to build all your packages if you already have them compiled.
- The `prjrepo upload` command allows to add existing `.deb` packages to the local repository of the project, saving build time.

```
$ ./elbe control create_project
/var/cache/elbe/0a7b1788-b2ab-4b53-9319-0a810dab30d9
$ PRJ="/var/cache/elbe/0a7b1788-b2ab-4b53-9319-0a810dab30d9"
$ ./elbe control set_xml $PRJ project.xml
$ cd ../out
$ find . -name "*.changes" | xargs -I '{}' elbe prjrepo upload_pkg $PRJ {}
$ cd -
$ ./elbe control build $PRJ
$ ./elbe control wait_busy $PRJ
```

SDK

- ELBE can generate a SDK, which provides a cross-compiler and libraries to build code for the target.
- Provided as a self-extractible shell script, much like the Yocto Project SDK.
- `setup-elbe-sdk-arm-linux-gnueabihf-armhf-ubuntu-1.0.sh`
- Sometimes, it is necessary to add more packages to the SDK, for example Qt tools if the target system contains Qt:

project.xml

```xml
<hostsdk-pkg-list>
  <pkg>qt5-qmake-bin</pkg>
  <pkg>qtbase5-dev-tools</pkg>
</hostsdk-pkg-list>
```

Conclusion and references

- ELBE is an interesting and friendly build System
- A small xml file describe all your distribution
- The Distribution is customizable with your own packages
- References
  - https://elbe-rfs.org/
  - https://elinux.org/images/e/e5/Using_ELBE_to_Build_Debian_Based_Embedded_Systems.pdf
  - https://wiki.dh-electronics.com/index.php/ELBE_Overview

Questions? Suggestions? Comments?

Köry Maincent

kory.maincent@bootlin.com

`https://bootlin.com/pub/conferences/2020/elce/maincent-building-embedded-debian-ubuntu-systems-elbe/`