Yocto or Debian for your embedded system? Yes

Wednesday August 27, 2025 16:20 - 17:00
CEST

About

Alan Martinovic

Northern.tech - makers of Mender

Patch BSPs on different boards to make Mender to work

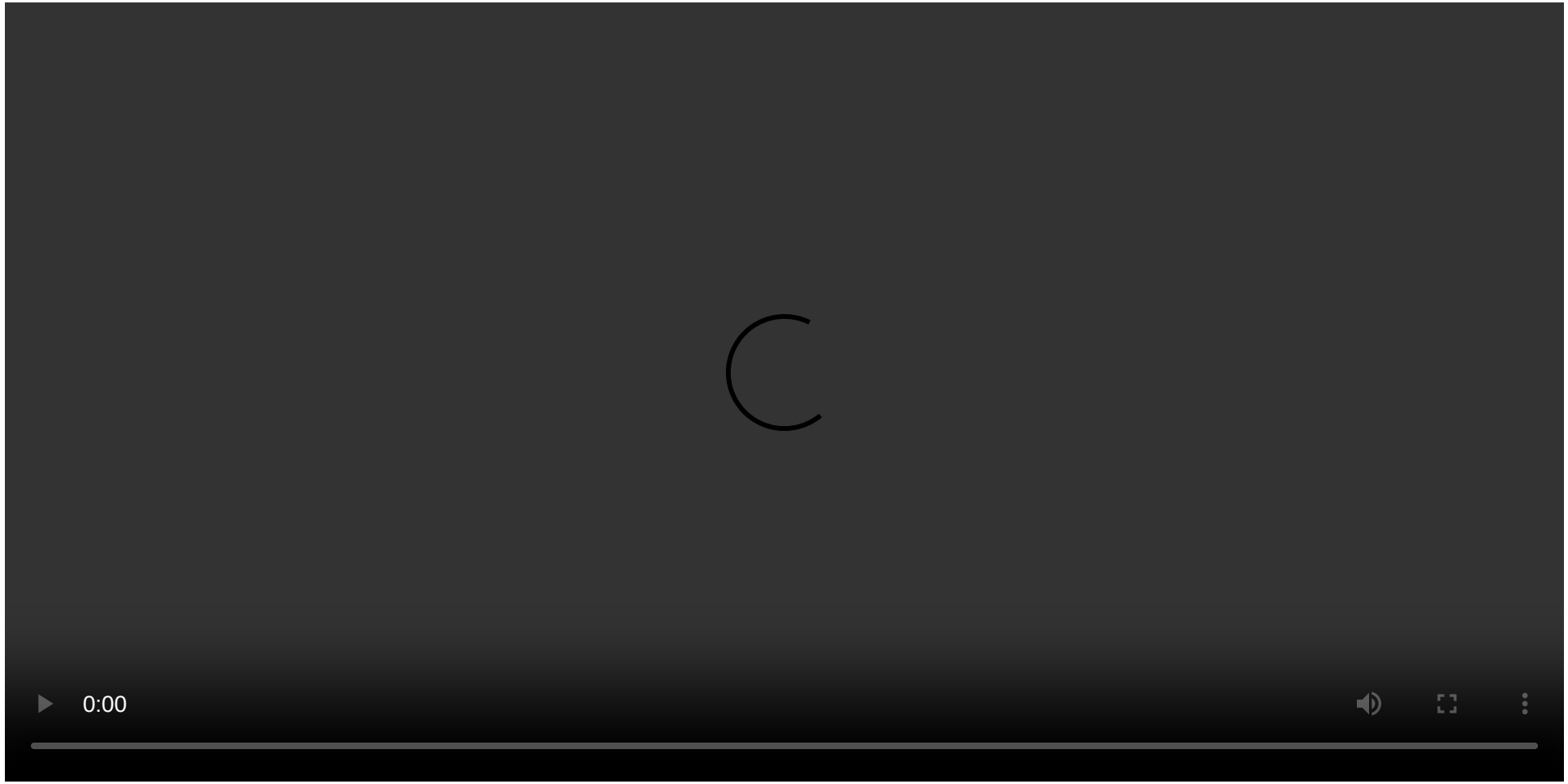
Coach teams to make educated decisions on OS

...so I can integrate Mender into it.

How many have Debian/Ubuntu on your laptop?

How many run a yocto built OS your laptop?

From 0 to get Debian golden image



Take a USB stick with a live disk.

Boot from the live disk OS.

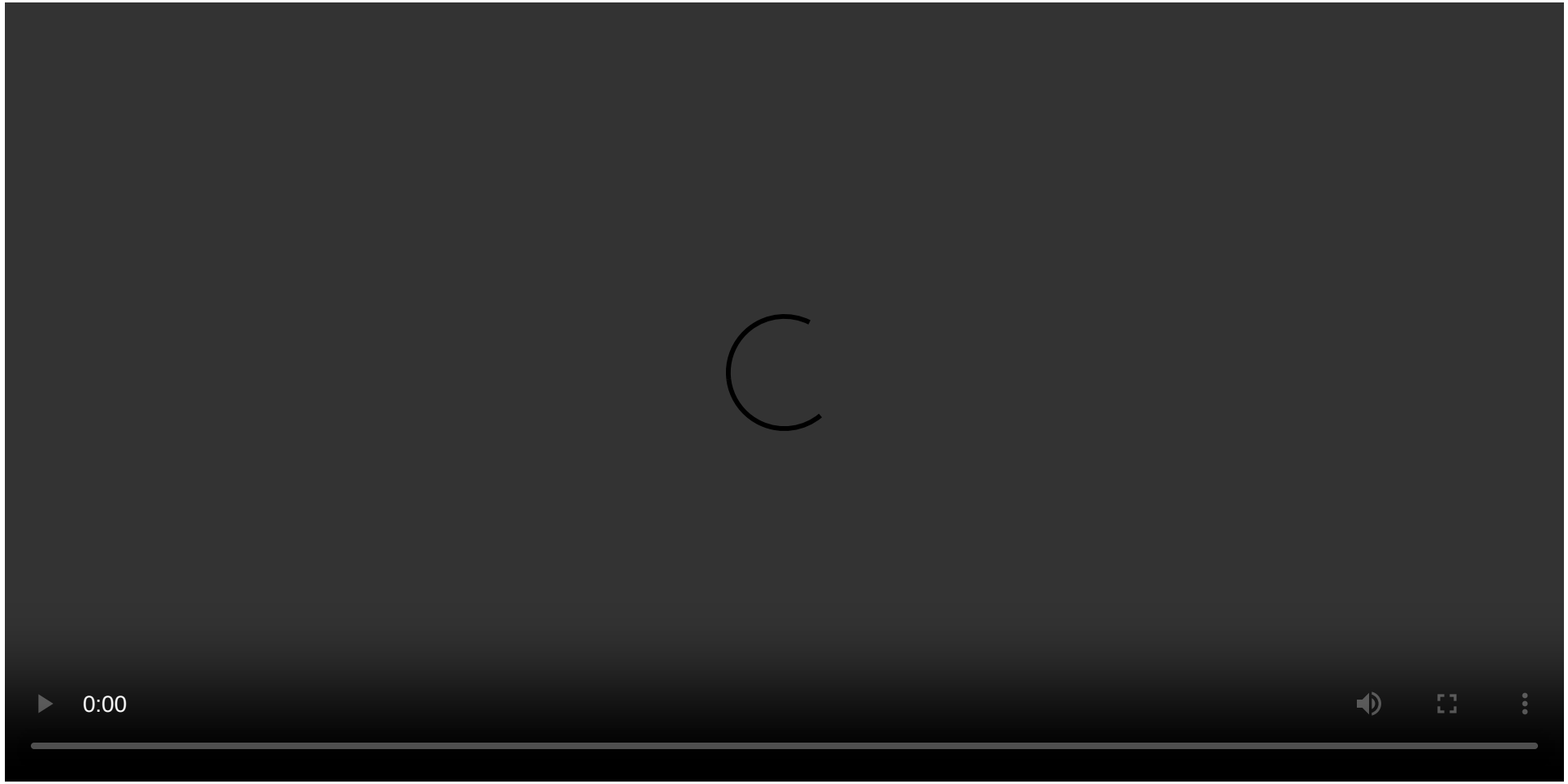
Copy the OS from your device.

Result:

```
user@debian:~$ ls
disk-debian-13-golden.img
```

One file!

From 0 to get a yocto build



Reality check

That was not a clean build.

A ~3h screen recording crashed my recording app...

```
sudo apt install build-essential chrpath cpio debianutils \
    diffstat file gawk gcc git iputils-ping libacl1 liblz4-tool \
    locales python3 python3-git python3-jinja2 python3-pexpect \
    python3-pip python3-subunit socat texinfo unzip wget xz-utils \
    zstd

git clone -b scarthgap git://git.yoctoproject.org/poky
cd poky
source oe-init-build-env

cat >> conf/local.conf << EOF
IMAGE_FSTYPES += "wic"
EOF

bitbake core-image-minimal
```

Result:

```
user@debian:~/poky/build$ ls $(bitbake-getvar --value  DEPLOY_DIR_IMAGE)
addonx64.efi.stub
bzImage
bzImage--6.6.96+git0+533e805705_cfb2162b88-r0-qemux86-64-20250822141717.bin
bzImage-qemux86-64.bin
core-image-minimal-qemux86-64.rootfs-20250822141717.ext4
core-image-minimal-qemux86-64.rootfs-20250822141717.manifest
core-image-minimal-qemux86-64.rootfs-20250822141717.qemuboot.conf
core-image-minimal-qemux86-64.rootfs-20250822141717.spdx.tar.zst
core-image-minimal-qemux86-64.rootfs-20250822141717.tar.bz2
core-image-minimal-qemux86-64.rootfs-20250822141717.testdata.json
core-image-minimal-qemux86-64.rootfs-20250822141717.wic
core-image-minimal-qemux86-64.rootfs.ext4                       # I'll come up again
core-image-minimal-qemux86-64.rootfs.manifest
core-image-minimal-qemux86-64.rootfs.qemuboot.conf
core-image-minimal-qemux86-64.rootfs.spdx.tar.zst
core-image-minimal-qemux86-64.rootfs.tar.bz2                    # Me too
core-image-minimal-qemux86-64.rootfs.testdata.json
core-image-minimal-qemux86-64.rootfs.wic                        # Me too
core-image-minimal.env
grub-efi-bootx64.efi
linuxx64.efi.stub
modules--6.6.96+git0+533e805705_cfb2162b88-r0-qemux86-64-20250822141717.tgz
modules-qemux86-64.tgz
```

Too much stuff...

Result:

```
ls $(bitbake-getvar --value  DEPLOY_DIR_IMAGE)
...
core-image-minimal-qemux86-64.rootfs.ext4
core-image-minimal-qemux86-64.rootfs.tar.bz2
core-image-minimal-qemux86-64.rootfs.wic
...
```

Three files!

Let's get philosophical.

What is an OS when it's running?

A bunch of files being executed.

A bunch of other files telling the executing ones how to behave.

What is an OS when it isn't running?

Still a bunch of files.

Reality check

That's not entirely true.

Can I see an OS when it isn't running?

Debian Golden image	Yocto
disk-debian-13-golden.img	core-image-minimal-qemux86-64.rootfs.wic core-image-minimal-qemux86-64.rootfs.ext4 core-image-minimal-qemux86-64.rootfs.tar

Why does Debian have 1 file and yocto has 3?

An "OS that isn't running" can be in many formats.

And it's not just files.

It's files put together in a way they can work together (dependencies).
And some things that aren't a file, keep them together.

Disk image (`disk-debian-13-golden.img` , `core-image-minimal-qemux86-64.rootfs.wic`)

Rootfs partition content (`core-image-minimal-qemux86-64.rootfs.ext4`)

Rootfs as directory (`core-image-minimal-qemux86-64.rootfs.tar`)

Disk image

```
cd ~/poky/build/tmp/deploy/images/qemux86-64
file -L core-image-minimal-qemux86-64.rootfs.wic
# core-image-minimal-qemux86-64.rootfs.wic:
# DOS/MBR boot sector;
# partition 1 : ID=0xc, active, start-CHS (0x10,0,1), end-CHS (0x1d0,3,20), startsector 2048, 57460 sectors;
# partition 2 : ID=0x83, start-CHS (0x1e0,0,1), end-CHS (0x3ff,3,32), startsector 61440, 99558 sectors
```

You can flash to the storage of your device directly.

Has all the raw data you need to boot the device.

You can attach to a loop device.

```
LOOPDEV=$(sudo losetup --show -f -P core-image-minimal-qemux86-64.rootfs.wic)
lsblk $LOOPDEV
# NAME      MAJ:MIN RM  SIZE RO TYPE MOUNTPOINTS
# loop0       7:0    0 78.6M  0 loop
# |-loop0p1 259:0    0 28.1M  0 part
# `-loop0p2 259:1    0 48.6M  0 part
```

Reality check

A disk image doesn't always have everything you need to boot.

Rootfs partition content

```
cd ~/poky/build/tmp/deploy/images/qemux86-64
file -L core-image-minimal-qemux86-64.rootfs.ext4
# core-image-minimal-qemux86-64.rootfs.ext4:
# Linux rev 1.0 ext4 filesystem data,
# UUID=d500904f-afca-4747-a47a-6b9c843a375c (extents) (64bit) (large files) (huge files)
```

You can flash to the partition of your device directly.

You can mount this.

```
sudo mount core-image-minimal-qemux86-64.rootfs.ext4 mounted/
ls mounted/
bin  boot  dev  etc  home  lib  lost+found  media  mnt  proc  run  sbin  sys  tmp  usr  var
```

Reality check

There are partition PARTUUIDs and file system UUIDs.

Rootfs as directory

```
user@debian:~/poky/build/tmp/deploy/images/qemux86-64$ file -L core-image-minimal-qemux86-64.rootfs.tar.bz2
core-image-minimal-qemux86-64.rootfs.tar.bz2:
bzip2 compressed data, block size = 900k
```

You can uncompress and explore as a directory structure.

```
mkdir untared
tar -xjf core-image-minimal-qemux86-64.rootfs.tar.bz2  -C untared
ls untared/
bin  boot  dev  etc  home  lib  media  mnt  proc  run  sbin  sys  tmp  usr  var
```

I built the vanilla image.

How do I add my app to this?

Source (counter.py)

```
mkdir ~/counter-1.0
cat > ~/counter-1.0/counter.py << EOF
#!/usr/bin/env python3
import time

seconds = 0
while True:
    print(f"Seconds: {seconds}")
    time.sleep(1)
    seconds += 1
EOF
```

Endless counter of seconds.

Unit file (counter.service)

```
cat > ~/counter-1.0/counter.service << EOF
[Unit]
Description=Second Counter Application

[Service]
Environment=PYTHONUNBUFFERED=1
Type=simple
ExecStart=/usr/bin/counter.py

[Install]
WantedBy=multi-user.target
EOF
```

The enabler of the endless counter of seconds.

Debian Golden image install

```
# pseudo code
scp counter.py root@<IP>:/usr/bin/
scp counter.service root@<IP>:/etc/systemd/system/

ssh root@<IP>
sudo apt-get install python
sudo systemctl daemon-reload
sudo systemctl enable counter.service
sudo systemctl start counter.service
```

Turn off device.

Make a `disk-debian-13-golden-v2.img`.

Yocto

Start the env.

Create the layer.

Create the recipe directory structure.

```
cd poky
source oe-init-build-env
bitbake-layers create-layer ../meta-counter
bitbake-layers add-layer ../meta-counter
mkdir -p ../meta-counter/recipes-apps/counter/files
```

Move the source files to the correct location.

```
cp ~/counter-1.0/counter.py \
    ../meta-counter/recipes-apps/counter/files/counter.py

cp ~/counter-1.0/counter.service \
    ../meta-counter/recipes-apps/counter/files/counter.service
```

Create a recipe that describes the installation

```
cat > ../meta-counter/recipes-apps/counter/counter_1.0.bb << 'EOF'
SUMMARY = "Endless counter"
DESCRIPTION = "Keep those seconds coming!"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"

SRC_URI = "file://counter.py \
           file://counter.service"

inherit systemd

RDEPENDS:${PN} = "python3-core"

SYSTEMD_SERVICE:${PN} = "counter.service"
SYSTEMD_AUTO_ENABLE = "enable"

do_install() {
    install -d ${D}${bindir}
    install -m 0755 ${WORKDIR}/counter.py ${D}${bindir}/counter.py

    install -d ${D}${systemd_system_unitdir}
    install -m 0644 ${WORKDIR}/counter.service ${D}${systemd_system_unitdir}/counter.service
}
EOF
```

Add the package to the image and rebuild

```
echo 'IMAGE_INSTALL:append = " counter"' >> conf/local.conf
bitbake core-image-minimal
```

Differences in the two approaches

In Golden image Debian we made changes on a running image.
Losing installation steps along the way.

But ad-hoc installation steps are familiar (scp/apt).

In yocto the steps for everything are defined.

We trigger a build based on those.

If we need something ad-hoc tool, we need to build it.

Golden image Debian:

```
Strong opinion on what your OS consists of.
Easy run time installation of new stuff.
Familiar environment.
Fine tuning of system apps/kernel is hard.
Rebuilding is hard.
Parallel development is hard.
```

Yocto:

```
Reproducible and versioned.
Fine tune the OS to your needs.
Dev collaboration as if it's software.
No runtime before a successful build.
A Domain Specific Language (DSL) you have to learn.
Ad-hoc chages in runtime are hard.
```

Expectations:

```
I only want to worry about my app [Bussiness value].
    (But sell a device :))
I want ad-hoc changes in runtime [Famililar env].
I don't want to care about OS apps/kernel [Trust in Debian]
I want multiple people to be able to introduce changes. [Faster features]
```

Alternatives to Debian golden images and Yocto exist.

These tools build custom Debian images from 0.

Tools I'll go through:

- manual approach (`mmdebstrap` + `guestfish`)
- elbe
- edi

There are more:

- mkosi
- debos
- different manual combinations (ie. `debootstrap` + `genimage`)

I'm not covering.
No specific reason, too many to explore.

Manual approach

(`mmdebstrap` + `guestfish`)

`mmdebstrap`

Website: mmdebstrap github

License: expat

`guestfish`

Website: online man

License: GPL-2.0-or-later

High level overview

Run a CLI app to build a directory structure.

Run a CLI app to set up the partitions (and a bit more).

You have the OS to flash to the device.

Run a CLI app to build a directory structure.

```
sudo apt-get install guestfish mmdebstrap
mkdir mmdebstrap
cd mmdebstrap

sudo mmdebstrap --mode=root --variant=minbase \
  --include=linux-image-amd64,grub-efi-amd64,systemd-sysv \
  --customize-hook='chroot "$1" sh -ec "echo root:root | /usr/sbin/chpasswd"' \
  --customize-hook='chroot "$1" sh -ec "mkdir -p /efi"' \
  stable fs.tar
```

`stable fs.tar` - directory structure

No bootloader.

Yes bootloader `grub-efi-amd64`.

Schrodinger's bootloader.

Goal: `directory structure-> disk image`

Run a CLI app to set up the partitions and stuff...

```
# Fixed GPT type GUIDs
ESP_TYPE_GUID=c12a7328-f81f-11d2-ba4b-00a0c93ec93b
LINUX_ROOT_GUID=0fc63daf-8483-4772-8e79-3d69d8477de4

# Your own unique PARTUUID
EFI_PART_UUID=$(uuidgen)
ROOTFS_PART_UUID=$(uuidgen)

guestfish -x -N debian.img=disk:600M -- \
  part-init /dev/sda gpt : \
  part-add /dev/sda p 2048 133119   : \
  part-add /dev/sda p 133120 -2048  : \
  part-set-gpt-type /dev/sda 1 "$ESP_TYPE_GUID" : \
  part-set-gpt-type /dev/sda 2 "$LINUX_ROOT_GUID" : \
  part-set-gpt-guid /dev/sda 1 "$EFI_PART_UUID" : \
  part-set-gpt-guid /dev/sda 2 "$ROOTFS_PART_UUID" : \
  mkfs vfat /dev/sda1 : \
  mkfs ext4 /dev/sda2 : \
  mount /dev/sda2 / : mkdir-p /boot/efi : mount /dev/sda1 /boot/efi : \
  tar-in fs.tar / xattrs:true : \
  mkdir-p /etc/default/grub.d : \
  write /etc/default/grub.d/10-rootpart.cfg "GRUB_CMDLINE_LINUX=\"root=PARTUUID=$ROOTFS_PART_UUID console=ttyS0,115200n8\"" : \
  write /etc/default/grub.d/20-status.cfg  'GRUB_CMDLINE_LINUX_DEFAULT="systemd.show_status=1 rd.systemd.show_status=1"' : \
  command 'chroot / grub-install --target=x86_64-efi --efi-directory=/boot/efi --bootloader-id=debian --removable' : \
  command 'chroot / update-grub' : \
  command 'chroot / grub-editenv /boot/grub/grubenv create' : \
  write /etc/fstab "PARTUUID=$ROOTFS_PART_UUID / ext4 defaults 0 1" : \
  sync : umount-all : shutdown
```

Partitioning

```
guestfish -x -N debian.img=disk:600M -- \
  part-init /dev/sda gpt : \
  part-add /dev/sda p 2048 133119   : \
  part-add /dev/sda p 133120 -2048  : \
  part-set-gpt-type /dev/sda 1 "$ESP_TYPE_GUID" : \
  part-set-gpt-type /dev/sda 2 "$LINUX_ROOT_GUID" : \
  part-set-gpt-guid /dev/sda 1 "$EFI_PART_UUID" : \
  part-set-gpt-guid /dev/sda 2 "$ROOTFS_PART_UUID" : \
  write /etc/fstab "PARTUUID=$ROOTFS_PART_UUID / ext4 defaults 0 1" : \
  write /etc/default/grub.d/10-rootpart.cfg \
    "GRUB_CMDLINE_LINUX=\"root=PARTUUID=$ROOTFS_PART_UUID console=ttyS0,115200n8\"" : \
  ...
  ...
```

Bootloader installation

```
guestfish -x -N debian.img=disk:600M -- \
  ...
  ...

  mkdir-p /etc/default/grub.d : \
  command 'chroot / grub-install --target=x86_64-efi --efi-directory=/boot/efi --bootloader-id=debian --removable' : \
  command 'chroot / update-grub' : \
  command 'chroot / grub-editenv /boot/grub/grubenv create' : \
```

We now have both:

```
$ ls
debian.img        "I'm a real disk image!" exclaimed the OS happily.
fs.tar
```

Test

```
qemu-system-x86_64 \
  -m 2G  \
  -cpu host \
  -enable-kvm \
  -bios /usr/share/OVMF/OVMF_CODE.fd \
  -drive file=debian.img,if=virtio,format=raw \
  -netdev user,id=net0,hostfwd=tcp::8822-:22 \
  -nographic \
  -device e1000,netdev=net0
```

How do I add my app?

1. Add a manually install as part of `mmdebstrap`
2. Manually install as a `guestfish` command
3. Deploy it as an overlay with `guestfish`
4. Modify the image after creation

   …

When in Debian, do as the Debianers/Debianites? do.

And build Debian packages.

Reality check

Building a Debian package with all the metadata is as complicated as a yocto recipe.

Python source

```
mkdir ~/package
mv ~/counter-1.0 ~/package/
```

Debian package metadata

```
cd ~/package/counter-1.0
mkdir -p debian/source
echo "3.0 (native)" > debian/source/format

cat > debian/control << EOF
Source: counter
Maintainer: Your Name <you@example.com>
Build-Depends: debhelper-compat (= 13)

Package: counter
Architecture: all
Depends: python3
Description: Counter service
EOF

cat > debian/changelog << EOF
counter (1.0) unstable; urgency=low
  * First version
 -- You <you@example.com>  $(date -R)
EOF

cat > debian/counter.install << EOF
counter.py usr/bin/
counter.service lib/systemd/system/
EOF

printf '#!/usr/bin/make -f\n%%:\n\tdh $@\n' > debian/rules
chmod +x debian/rules
```

```
tree .
# .
# ├── counter.py
# ├── counter.service
# └── debian
#     ├── changelog
#     ├── control
#     ├── counter.install
#     ├── rules
#     └── source
#         └── format
```

Build

```
dpkg-buildpackage -us -uc -b

ls ../
counter-1.0           counter_1.0_amd64.buildinfo
counter_1.0_all.deb   counter_1.0_amd64.changes
```

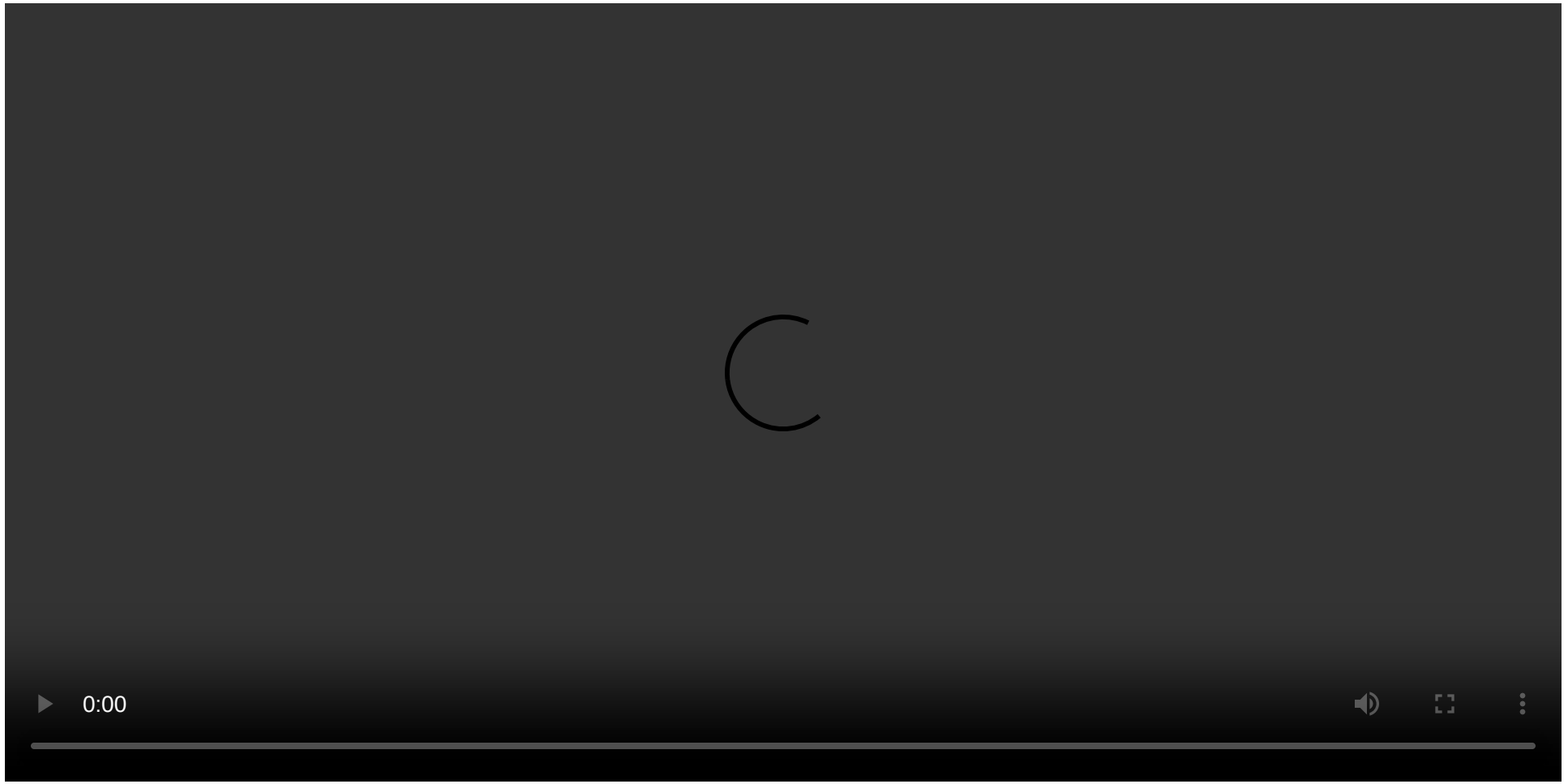
Include the app

```
cd mmdebstrap

sudo mmdebstrap --mode=root --variant=minbase \
  --include=linux-image-amd64,grub-efi-amd64,systemd-sysv \
  --customize-hook='chroot "$1" sh -ec "echo root:root | /usr/sbin/chpasswd"' \
  --customize-hook='chroot "$1" sh -ec "mkdir -p /efi"' \
  --customize-hook='cp ../package/counter_1.0_all.deb  "$1/tmp/"' \
  --customize-hook='chroot "$1" apt update' \
  --customize-hook='chroot "$1" apt install -y /tmp/counter_1.0_all.deb' \
  --customize-hook='chroot "$1" dpkg -i /tmp/counter_1.0_all.deb' \
  stable fs.tar

ESP_TYPE_GUID=c12a7328-f81f-11d2-ba4b-00a0c93ec93b
LINUX_ROOT_GUID=0fc63daf-8483-4772-8e79-3d69d8477de4
EFI_PART_UUID=$(uuidgen)
ROOTFS_PART_UUID=$(uuidgen)

guestfish -x -N debian.img=disk:600M -- \
  part-init /dev/sda gpt : \
  part-add /dev/sda p 2048 133119   : \
  part-add /dev/sda p 133120 -2048  : \
  part-set-gpt-type /dev/sda 1 "$ESP_TYPE_GUID" : \
  part-set-gpt-type /dev/sda 2 "$LINUX_ROOT_GUID" : \
  part-set-gpt-guid /dev/sda 1 "$EFI_PART_UUID" : \
  part-set-gpt-guid /dev/sda 2 "$ROOTFS_PART_UUID" : \
  mkfs vfat /dev/sda1 : \
  mkfs ext4 /dev/sda2 : \
  mount /dev/sda2 / : mkdir-p /boot/efi : mount /dev/sda1 /boot/efi : \
  tar-in fs.tar / xattrs:true : \
  mkdir-p /etc/default/grub.d : \
  write /etc/default/grub.d/10-rootpart.cfg "GRUB_CMDLINE_LINUX=\"root=PARTUUID=$ROOTFS_PART_UUID console=ttyS0,115200n8\"" : \
  write /etc/default/grub.d/20-status.cfg  'GRUB_CMDLINE_LINUX_DEFAULT="systemd.show_status=1 rd.systemd.show_status=1"' : \
  command 'chroot / grub-install --target=x86_64-efi --efi-directory=/boot/efi --bootloader-id=debian --removable' : \
  command 'chroot / update-grub' : \
  command 'chroot / grub-editenv /boot/grub/grubenv create' : \
  write /etc/fstab "PARTUUID=$ROOTFS_PART_UUID / ext4 defaults 0 1" : \
  sync : umount-all : shutdown
```

```
sudo mmdebstrap --mode=root --variant=minbase \
  ...
  --customize-hook='cp counter_1.0-1_all.deb "$1/tmp/"' \
  --customize-hook='chroot "$1" apt update' \
  --customize-hook='chroot "$1" apt install -y /tmp/counter_1.0-1_all.deb' \
  ...
  stable fs.tar
```

Test

```
qemu-system-x86_64 \
  -m 2G  \
  -cpu host \
  -enable-kvm \
  -bios /usr/share/OVMF/OVMF_CODE.fd \
  -drive file=debian.img,if=virtio,format=raw \
  -netdev user,id=net0,hostfwd=tcp::8822-:22 \
  -nographic \
  -device e1000,netdev=net0


journalctl -fu counter
```


0:00

Manual approach:

- once you know what's going on you can change toolsets if the need comes
- full flexibility in choosing your approach

elbe

Website: https://elbe-rfs.org/index.html
Conference (Köry Maincent, Bootlin): https://www.youtube.com/watch?v=BwHzyCGB7As
License: GNU General Public License v3.0

Special thanks to Thomas Weißschuh for sharing virtual coffee with me to talk about elbe.

High level overview

Set up a virtual machine (VM).

Use that as your background build environment.

Send it xml files.

It give you back build artifacts.

Prep for VM building

```
sudo apt install python3 python3-debian python3-mako python3-lxml python3-apt \
    python3-gpg python3-suds python3-libvirt qemu-utils qemu-kvm p7zip-full \
    make python3-passlib libvirt-clients libvirt-daemon-system debian-archive-keyring git

git clone -b v15.6 https://github.com/Linutronix/elbe.git
cd elbe/

sudo adduser $USER libvirt
newgrp libvirt
sudo adduser $USER kvm
newgrp kvm

# Check if kvm works - much faster build
virt-host-validate
# QEMU: Checking for hardware virtualization                                 : PASS
# QEMU: Checking if device /dev/kvm exists                                   : PASS
# QEMU: Checking if device /dev/kvm is accessible                            : PASS
```

Reality check

Renting cloud resources for your build server?

Very high chance you're not getting kvm.

VM build

```
time ./elbe initvm create --directory ~/elbe-initvm
# real    13m49.045s
# user    10m34.777s
# sys     3m41.975s

#[Optional] Tracking installer progress in a different terminal
tail -f /home/user/elbe-initvm/installer.log

# [Optional] You can list the VM
virsh --connect qemu:///system list --all
```

Serve the VM an xml

```
./elbe initvm submit examples/amd64-with-grub.xml
# Crunching....
```

It rewards you with build artifacts

```
ls elbe-build-*/
# bin-cdrom.iso       licence-chroot.xml  sda.img.tar.xz       src-cdrom-target.iso
# elbe-report.txt     licence-target.txt  source.xml           validation.txt
# hosts               licence-target.xml  src-cdrom-added.iso
# licence-chroot.txt  log.txt             src-cdrom-main.iso
```

Get the disk image

Compressed.

Untar.

```
cd elbe-build-*/
tar xf sda.img.tar.xz
```

Test

```
# root, root
qemu-system-x86_64 \
  -m 2G  \
  -cpu host \
  -enable-kvm \
  -drive file=sda.img,if=virtio,format=raw \
  -netdev user,id=net0,hostfwd=tcp::8822-:22 \
  -nographic \
  -device e1000,netdev=net0
```

Adding the application

Elbe has a built in builder for debian sources.

Initialize the builder for your xml

```
./elbe pbuilder create --xmlfile examples/amd64-with-grub.xml --writeproject ./pbuilder.prj
```

You say:

"Hey VM, I'll be using `examples/amd64-with-grub.xml`

Kick of the pbuilder project for that"

Elbe:

"*Ok*"

Feed app sources to the project

```
cp ~/package/counter-1.0 elbe
cd counter-1.0
../elbe pbuilder build --project "$(cat ../pbuilder.prj)"
```

This does something fun

It adds the project to a local Debian repository.

```
../elbe prjrepo list_packages "$(cat ../pbuilder.prj)"
cd ..
```

Add the application to the xml

As if it's from a public repo

```
HASH=$(openssl passwd -6 'root')
sed -i "s|<passwd>root</passwd>|<passwd_hashed>$HASH</passwd_hashed>|" examples/amd64-with-grub.xml
sed -i '/<\/pkg-list>/i\                    <pkg>counter</pkg>' examples/amd64-with-grub.xml
```

Reality check

I was recreating a password there as well.

`elbe control` is strict with passwords `elbe initvm submit` wasn't.

Kick off the build for the same project

```
./elbe control set_xml "$PRJ" examples/amd64-with-grub.xml
./elbe control build "$PRJ"
./elbe control wait_busy "$PRJ"
./elbe control get_files "$PRJ" --output ./out
```

Reality check

I started with `elbe initvm submit` and then used `elbe control`.

One is a quick kick off the build as defined in xml.

The other was incrementally work on the image under a project umbrella.

Test

```
cd out
tar xf sda.img.tar.xz
# root, root
qemu-system-x86_64 \
  -m 2G  \
  -cpu host \
  -enable-kvm \
  -drive file=sda.img,if=virtio,format=raw \
  -netdev user,id=net0,hostfwd=tcp::8822-:22 \
  -nographic \
  -device e1000,netdev=net0
```

No video :(

```
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+7EFCF0E0+7EF0F0E0 CA00

Booting from Hard Disk...
GRUB loading.
Welcome to GRUB!

Debian GNU/Linux 11 amd64-bullseye ttyS0

amd64-bullseye login: root
Password:
Linux amd64-bullseye 5.10.0-32-amd64 #1 SMP Debian 5.10.223-1 (2024-08-104

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
root@amd64-bullseye:~# journalctl -fu counter
-- Journal begins at Tue 2025-08-26 18:58:05 UTC. --
Aug 26 18:58:08 amd64-bullseye counter.py[235]: Seconds: 0
Aug 26 18:58:09 amd64-bullseye counter.py[235]: Seconds: 1
Aug 26 18:58:10 amd64-bullseye counter.py[235]: Seconds: 2
Aug 26 18:58:11 amd64-bullseye counter.py[235]: Seconds: 3
```

elbe:

- xml is master
- built in isolation with the VM
- extra features (compliance iso, testing)

get-edi

Website: https://www.get-edi.io/

Conference (Matthias Lüscher) : https://www.youtube.com/watch?v=BwHzyCGB7As

https://www.youtube.com/watch?v=i8sRrpdn2Hw

License: "edi is licensed under the LGPL license." (from README)

Special thanks to Matthias Lüscher for sharing virtual coffee with me to talk about EDI.

High level overview

Define the whole OS build process in one yml file.

Build your OS

The yml file defines 3 main stages:

1. Set up a minimal image (`preprocessing_commands`)
2. Configure that image with ansible (`playbooks`)
3. Package it up into a disk file (`postprocessing_commands`)

Each stage defines 'plugins'.
Those represents steps of execution.

They are either templated bash scripts (.edi files).
Or ansible playbooks.

Install

```
sudo apt install buildah containers-storage crun \
    curl distrobox dosfstools e2fsprogs fakeroot genimage \
    git mender-artifact mmdebstrap mtools parted \
    python3-sphinx python3-testinfra podman rsync zerofree

curl -s https://packagecloud.io/install/repositories/get-edi/debian/script.deb.sh | sudo bash
sudo apt install edi
ssh-keygen -t ed25519 -C "user@email.com"
```

Clone the example repo

```
git clone --recursive https://github.com/lueschem/edi-pi.git
```

Build

```
cd edi-pi
edi -v project make pi4.yml
```

Results

```
ls artifacts
# pi4_bootstrapped-rootfs.tar  pi4.img                  pi4.seal
# pi4_configured-rootfs.tar    pi4.mender  :)           pi4_timestamp
# pi4_documentation            pi4_prerequisites.log
# pi4_fully_named_artifacts    pi4_rootfs.ext4
```

Reality check

This was rpi4 build, it was the default.

Other tools can assemble using cross compile packages as well.

Wanted to go QEMU for everything but there wasn't an amd64 example.

Testing for real

```
sudo bmaptool copy --nobmap pi4.img /dev/sdc
# Put in the RPi
# Connect serial
sudo picocom -b 112500 /dev/ttyUSB0
# [    0.000000] Booting Linux on physical CPU 0x0000000000 [0x410fd083]
# [    0.000000] Machine model: Raspberry Pi 4 Model B Rev 1.2
# ...
# Debian GNU/Linux 13 raspberry-dca632a86e29 ttyS0
# raspberry-dca632a86e29 login: pi
# Password: <raspberry>
# pi@raspberry-dca632a86e29:~$
```

Ran out of time before getting to a working example with an app :(

edi:

- great choice if you are familiar with ansible
- focus is being full gitops flow for embedded with common tools

Conclusion

Debian vs Yocto for my IoT device?

The answer is still **Yes** to both.

No to runtime golden image.

Yocto:

- more control
- strong focus on embedded Linux
- it can do almost everything
- still my first recommendation

Debian:

- exceptional repository of packages
- battle tested flows
- familiar environment
- offers much better paths than the runtime golden image
- pick a tool that wraps around the core steps if it fits your flow

Golden image Debian:

- unreproducible steps
- unknown runtime artifacts
- bad practice

Reality check

"Golden image" is an overloaded term.
If someone pushes back the full name is:

> "Installing the OS with an installer or byte copying one someone else made, booting it, doing runtime changes then snapshotting it" image

But my board vendor only ships a prebuilt Debian

Ask for build steps.

They are using a variation of what I've shown.

Negotiate it out, you need it for a CI/CD setup.

If that fails, choose a lesser evil and modify the image when it's not running.

Reality check

I didn't cover how to stabilize versions when running `apt-get update; apt-get install`.

Feel free to ask me that after the talk.

Cheers :)

Questions?