Reducing friction in testing using QEMU and Labgrid for Yocto-based products

Joschka Seydell, Zühlke Engineering

About… my street credibility and what to expect in this talk

Facts

- Embedded Linux/Yocto/C++ engineer for 7+ years
- Projects in various industries, mostly for Yocto-based products
- Active in the RAUC/meta-rauc communities
- Enjoys building prototypes and 10k runs

Reach out

✉ joschka.seydell@zuehlke.com

jsydll

Outline

- Context and focus
- Technical solution proposal
- Hands-on and findings
- Conclusion & bonus

HIL test infrastructure often comes with long cycles and significant maintenance

Who knows one of these?

„The nighly is still running…“

“I’m back in 2h to check if I can merge.”

“Hi team, I locked this device for testing.”

“Our serial connection seems to be flaky.”

“Just copy and adjust this integration test, then we’re on the safe side.”

Common pain points

Long waiting times
for HIL test results

Side-effects between tests
(incl. exclusive resources)

Maintenance & debugging effort
(for custom infrastructure)

Finding bugs in system software using (not only!) emulation, from commit to release

Goals

Finding bugs

Formal verification

Scopes

Compilation Unit | Package

Image | System

Environments

Host (includes Simulation, Docker, ...)

Emulation

Hardware

Triggers

Commit | Merge Request

Nightly | Weekly

Release

Emulation shines where scalability and usability are more important than hardware accuracy

- ✓ Partitioned systems
- ✓ Reboots involved
- ✓ Configuration / logic permutations
- ✓ Special start states
- ✓ Local, easy development

- ? Connectivity
- ? Distributed systems' communication
- ? Brickable peripherals
- ? Architecture specifics

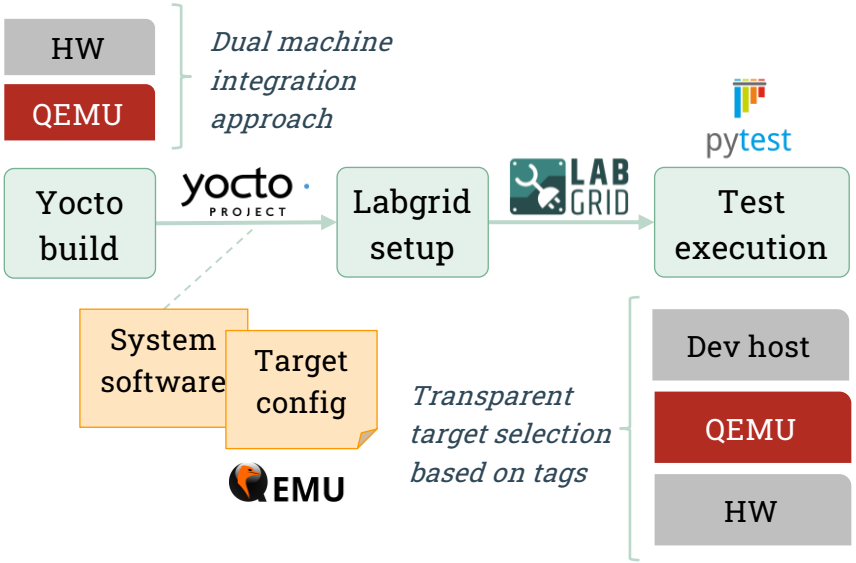
- × Driver / hardware behavior tests
- × Timing dependencies

← ***Close-enough***
Improving dev experience

*Only for **high impact** features*

Inappropriate effort
*Little **meaningfulness*** →

An end-to-end integration for emulated tests using existing tools can speed up feedback time and reduce debugging effort notably



No one-size-fits-all test target!

- End-to-end automation
- Emulation replaces *some* execution scenarios
- FOSS instead of self-made solutions
- Conscious selection of test placement

Building blocks:
QEMU as lightweight VM for embedded architectures

Capabilities

+ All major architectures
+ Various peripherals (e.g. USB, timers, interrupts, I2C, SPI, GPIO, UART, CAN, …)
+ Hardware acceleration

Limitations

− Imperfect behavior
− Limited support of specialized hardware
− Configuration complexity

Particularly interesting:

Allows full virtualization

Mimics real board performance

Ephemeral execution possible

(-snapshot)

Scalable / parallelizable

Debugging & system control capabilities

Building blocks:
Labgrid to cover key tasks like board control and resource management

Capabilities

+ pytest plugin / cmdline tool → technology-wise not disruptive
+ Common board control features like ssh, serial, bootloader interface
+ Comes with Qemu driver out of the box!
+ Follows the configuration as code approach

Limitations (?)

− Does enforce some design patterns on your tests

State of the project

Quite mature (v25.0)

Actively developed

Flourishing community

Investigating speed gains, ease of use and limitations by emulating a representative embedded setup

From machine definition, over generating target configs to transparent test execution

- → kas-based Yocto setup, using development / testing containers
- → ARM “board” with u-boot and kernel contained in rootfs
- → Main test scenario: A/B **Software Update** with 3 pseudo-versions (manufacturing, LTS, latest)
- → Core image minimal for raspberrypi4-64 and custom qemuarm64 machine

Now, finally: Code!

Follow along here: https://github.com/JSydll/emulated-yocto-linux-testing/

virt-aarch64.wks

```
# Note: QEMU does not come with a bundled u-boot, so the BIOS parameter is used to pass in the
# built binary. The system image does not contain the bootloader.

bootloader --ptable gpt

part --ondisk vda --source bootimg-partition --fstype=vfat --active --fixed-size 16M --label bootfiles --
align 4096

part / --ondisk vda --source rootfs --fstype=ext4 --fixed-size 250M --label root_a --align 4096
part / --ondisk vda --source empty --fstype=ext4 --fixed-size 250M --label root_b --align 4096
```

virt-aarch64.conf

```
require conf/machine/include/arm/armv8a/tune-cortexa72.inc
require conf/machine/include/qemu.inc

MACHINEOVERRIDES =. "qemuarm64:"
UBOOT_MACHINE = "qemu_arm64_defconfig"
KMACHINE:virt-aarch64 = "qemuarm64"

QB_SYSTEM_NAME = "qemu-system-aarch64"
QB_MACHINE = "-machine virt"
QB_CPU_VALUE = "cortex-a72"
QB_CPU = "-cpu ${QB_CPU_VALUE}"
QB_MEM_VALUE = "4G"
QB_MEM = "-m ${QB_MEM_VALUE}"
QB_SMP = "-smp 4"
...
# u-boot support
QB_DEFAULT_BIOS = "u-boot.bin"
QB_ROOTFS_OPT = "-drive id=disk0,file=@ROOTFS@,if=none,format=qcow2 -device virtio-blk-device,drive=disk0"
QB_OPT_APPEND = " -drive if=pflash,format=raw,index=1,readonly=off,file=${TMPDIR}/deploy/images/${MACHINE}/bootenv.img"
# canbus support
QB_EXTRA_OPTS = " \
    -object can-bus,id=canbus0 \
    -device kvaser_pci,canbus=canbus0 \
"
QB_OPT_APPEND:append = " ${QB_EXTRA_OPTS}"
```

u-boot_%.bbappend

```
FILESEXTRAPATHS:prepend := "${THISDIR}/files:"

SRC_URI += " \
    file://0001-qemu-arm-prefer-virtio-as-boot-source.patch \
    file://boot.cmd.in \
"

RDEPENDS:${PN} = " devicetree-qemuarm"
DEVICETREE_NAME = "${MACHINE}.dtb"
...
UBOOT_BOOTENV_FILE = "bootenv.img"

do_configure:prepend () {
    sed -e 's/@@DEVICETREE_NAME@@/${DEVICETREE_NAME}/' \
        -e 's/@@KERNEL_IMAGETYPE@@/${KERNEL_IMAGETYPE}/' \
        -e 's/@@KERNEL_BOOTCMD@@/${KERNEL_BOOTCMD}/' \
        "${UNPACKDIR}/boot.cmd.in" > "${UNPACKDIR}/boot.cmd"
}

do_deploy:append (){
    # Note: The flash size is predefined by qemu
    qemu-img create -f raw ${UBOOT_BOOTENV_FILE} 64M
}

do_deploy[depends] += "qemu-system-native:do_populate_sysroot"
```

linux-yocto_%.bbappend

```
FILESEXTRAPATHS:prepend := "${THISDIR}/files:"

SRC_URI:append = " file://mtd_flash.cfg“
```

devicetree-qemuarm_1.0.bb

```
inherit devicetree

SRC_URI:qemuarm = "file://qemuarm.dts"
SRC_URI:qemuarm64 = "file://qemuarm64.dts"

COMPATIBLE_MACHINE = "^(qemuarm|qemuarm64)$"
```

There are examples out there, but getting *everything* right (machine tune, bootloader, DTB, …) needs some expertise

Should I start with a QEMU board model or `-machine virt`?

→ In short, it's technically possible to fully configure a custom board based on `virt` – but it probably comes with significant effort (and potentially custom QEMU device model extensions)

How to get the bootflow right?

→ Setup of dedicated "partition" for bootenv on special pflash device

→ Bootloader (script) needs to be adjusted adjusted accordingly (deviating from real setup)

→ Dumping the DTB and passing it to the kernel

```
runqemu wic.qcow2 nographic qemuparams="-machine dumpdtb=qemuarm64.dtb"
dtc -I dtb -O dts qemuarm64.dtb > qemuarm64.dts
```

lg-env-config.yml

```yaml
targets:
  main:
    resources: &common-resources
      - NetworkService:
          address: ""
          port: 22
          username: root
      - UpdateBundles:
          latest: "@@LG_ARTIFACT_MOUNTPOINT@@/latest.@@UPDATE_BUNDLE_ARTIFACT@@"
          lts: "@@LG_ARTIFACT_MOUNTPOINT@@/lts.@@UPDATE_BUNDLE_ARTIFACT@@"
          manufacturing: "@@LG_ARTIFACT_MOUNTPOINT@@/manufacturing.@@UPDATE_BUNDLE_ARTIFACT@@"

    drivers:
      - QEMUDriver:
          qemu_bin: qemu_aarch64
          machine: virt
          cpu: @@QB_CPU_VALUE@@
          memory: @@QB_MEM_VALUE@@
          nic: user,model=virtio-net-pci
          bios: uboot_latest
          disk: rootfs_latest
          extra_args: " @@QB_SMP@@ @@QB_EXTRA_OPTS@@ -snapshot -drive if=pflash,format=raw,index=1,readonly=off,
                      file=@@LG_ARTIFACT_MOUNTPOINT@@/latest.bootenv.img"
      - ShellDriver: &shell-config
          login_prompt: "@@MACHINE@@ login: "
          username: "root"
          prompt: 'root@@@MACHINE@@+:[^ ]+ '
          login_timeout: 200
      - SSHDriver: {}
      - QEMUSetupStrategy: {}
```

Minimizing workflow disruptions by building a seamless testing experience through config generation and target abstractions

Concepts applied in the proof of concept:

→ Single source of (machine) truth: **Generate the labgrid target configs** in Yocto

→ Write **target independent tests** that only differ in setup using pytest fixtures

→ Use **tags and labgrid features** to automate test selection and placement

(mapping tests to designated targets and execution scenarios decentrally, i.e. per test case)

→ Make QEMU instances **ephemeral**, to always start at a well defined state

More ideas:

? Tests could use the QEMU Machine Protocol (QMP) to set system in dedicated state

update_flow_test.py

```python
pytestmark = pytest.mark.update

@pytest.mark.release
@pytest.mark.hardware_only
@pytest.mark.parametrize('version_to_install',
        [SoftwareVersion.lts, SoftwareVersion.latest]
    )
def test_update_from_manufacturing_succeeds(update_manufacturing: update.UpdateFlow, version_to_install: SoftwareVersion) -> None:
    """
    Test that the happy path update from manufacturing to the specified version succeeds.
    """
    update_manufacturing.execute_all_steps(version_to_install)


...

@pytest.mark.smoketest
@pytest.mark.parametrize('version_to_install',
        [SoftwareVersion.latest]
    )
def test_update_from_latest_succeeds(update_latest: update.UpdateFlow, version_to_install: SoftwareVersion) -> None:
    """
    Test that the happy path update from latest to the specified version succeeds.
    """
    update_latest.execute_all_steps(version_to_install)
```

conftest.py

```
def get_update_flow(env: Environment, preset_software_version: versions.SoftwareVersion) -> update.UpdateFlow:
    target = None

    if preset_software_version != versions.SoftwareVersion.latest:
        # Allow skipping setup steps by choosing “preset” images
        target = env.get_target(preset_software_version.name)

    if target is None:
        target = env.get_target()

    strategy = target.get_strategy()
    strategy.transition("shell")

    shell = target.get_driver("ShellDriver")
    target.activate(shell)

    ssh = target.get_driver("SSHDriver")
    target.activate(ssh)

    update_flow = update.UpdateFlow(target, shell, ssh)

    current_version = versions.get_current_software_version(ssh)
    if current_version != preset_software_version:
        # Establish clean state (only required for hardware targets)
        update_flow.enable_force_install()
        update_flow.execute_all_steps(preset_software_version)

    return update_flow
```

Benchmark: Sequential execution on Raspi4 vs. parallel execution in QEMU

```
$ time test-run-pytest --durations=5 -m update
=== slowest 5 durations ===
103.93s setup    test_update/update_flow_test.py::test_update_from_manufacturing_succeeds[SoftwareVersion.latest]
92.70s call     test_update/update_flow_test.py::test_update_from_manufacturing_succeeds[SoftwareVersion.latest]
87.21s call     test_update/update_flow_test.py::test_update_from_manufacturing_succeeds[SoftwareVersion.lts]
78.03s call     test_update/update_flow_test.py::test_update_from_latest_succeeds[SoftwareVersion.latest]
69.31s call     test_update/update_flow_test.py::test_update_from_lts_succeeds[SoftwareVersion.lts]
=== 5 passed, 2 deselected in 561.19s (0:09:21) ===

real    9m22,318s
```

```
$ time test-run-pytest -n auto --durations=5 -m update
8 workers [5 items]
=== slowest 5 durations ===
86.72s call     test_update/update_flow_test.py::test_update_from_manufacturing_succeeds[SoftwareVersion.lts]
83.69s call     test_update/update_flow_test.py::test_update_from_lts_succeeds[SoftwareVersion.lts]
82.80s call     test_update/update_flow_test.py::test_update_from_lts_succeeds[SoftwareVersion.latest]
82.55s call     test_update/update_flow_test.py::test_update_from_latest_succeeds[SoftwareVersion.latest]
80.90s call     test_update/update_flow_test.py::test_update_from_manufacturing_succeeds[SoftwareVersion.latest]
=== 5 passed in 125.16s (0:02:05) ===
real    2m6.806s
```

Preliminary results show strong reuse of build artifacts and significant speed gains for parallelizable test cases

~4.5 times faster execution

Unchanged: Installs / reboots take time

Depends on host performance

Independence of tests required

No external / blockable resources

No dependencies on side-effects

Reuse of build artifacts

14% differences (by package count),

mostly due to images/bundles,

bootloader & kernel packages

Improvements besides emulation

Identifying suitable tags

Avoiding unnecessary tests

Emulation can pay off fast for straight-forward systems and when testing on image/system level is a productivity bottleneck

low — **Costs and Complexity** — *high* →

"Off-the-shelf" integration	Adding simple peripherals	Optimizing performance / behavior	Getting close to hardware

So does it pay off?

- ✓ Minimal integration is enough for the test
- ✓ Test target setup/tear down is expensive
- ✓ Hardware is scarce
- × Tests running on hardware anyway
- × Advanced peripheral support required

An end-to-end integration of QEMU and Labgrid can reduce friction in your testing – if your product and test goals match

Take aways

- **Applicability** depends on your product
- Return of (integration) invest depends on test scenario, solution shall be **value-driven**
- If **'close enough'** is sufficient, significant speed and accessibility gains possible
- **Meaningfulness** of test results must be considered in all cases
- Focus of emulated tests is **finding bugs**, not formal verification
- Big lever for reducing friction lies in **test placement and execution triggers**

Thanks for your attention! Questions?

Bonus Material

Digging deeper into peripherals: CAN bus emulation is technically possible but imposes limitations on the host environment

Emulation only:

```
qemu-system-aarch64 ...
  -object can-bus,id=canbus0
  -device kvaser_pci,canbus=canbus0
```

→ only allows to satisfy interface dependencies

Passthrough:

```
qemu-system-aarch64 ...
  -object can-bus,id=canbus0
  -object can-host-socketcan,id=canhost0,\
          if=vcan0,canbus=canbus0
  -device kvaser_pci,canbus=canbus0
```

→ requires can/vcan on host

→ hard to isolate / parallelize

Take away

Integration effort 10% for bare minimum, 20% for more advanced, 70% for close match

- the latter might not be worth it...

Digging deeper into board specifics: Secure boot can only be partially emulated with reasonable effort

Common secure boot flow:

→ ROM code usually proprietary and board specific

→ QEMU does not verify any of the payloads & does not have RPMB emulation (for EFI vars)

→ Some platforms support at least partial emulation (e.g. ARM TF-A, see here)

– involving a significant bit of custom configuration and compilation

Only emulation **from bootloader onwards** feasible or meaningful, though it can help to verify **behavior on failure and corruption** as well as **integration with software updates**.

Overview of test levels, scenarios and execution targets

	Units / Components [test case/suite-based]			**Integration / Services** [full image, scenario-based]			**System Software** [full system, scenario-based]	
Environment	Native	QEMU?	Hardware	Native	QEMU	Hardware	QEMU	Hardware
Trigger	ad hoc	commit, nightly	merge, pre-release	ad hoc	merge, nightly, pre-release	weekly, pre-release	ad hoc, merge, nightly, pre-release	weekly, pre-release
Goal	+ functional correctness + no regressions	+ arch. differences	+ timings etc	+ side effects (on service level) + image configuration	+ generic machine dependencies	+ integration issues + timings etc	+ side effects (on system level) + system configuration + business logic issues	
Cylce time	minutes		minutes-hours	minutes		hours	hours	days
Scenarios	Development tests, interactions between classes/ components			Service APIs & interactions, bootstrapping & configuration, partial architecture/ NFR verification, service failure analysis			User-facing features, update & maintenance scenarios, pentesting, NFR verification	

Alternative approaches and (open source) technologies to consider

For testing applications with hardware dependencies:

- Simulation
- C++ driver mocks w/ bindings to pytest

For test infrastructure:

- https://testinfra.readthedocs.io/en/latest/index.html
  (more focused on server infrastructure though)
- Collabora's Boardswarm?

Useful resources

- Overview article: Speeding Up Embedded Software Testing with QEMU: https://www.codethink.co.uk/articles/2024/qemu-testing-embedded-linux/
- General experiences with QEMU: codethink @OSS Japan 2022: https://www.youtube.com/watch?app=desktop&v=mzrQqB4ZReM
- Guidance on QEMU's capabilities and configuration: https://wiki.archlinux.org/title/QEMU
- How to connect to the QEMU monitor to check the emulation state: https://unix.stackexchange.com/questions/426652/connect-to-running-qemu-instance-with-qemu-monitor
- https://ejaaskel.dev/ for helpful articles on Yocto and QEMU (as well as security)