Automated Testing of Embedded Systems Using SiL and HiL

An Introduction

Matthias Miehl

2019-11-13 18:00

Testautomatisierung Meetup

TOP-Tagungszentren AG
Emil-Figge-Str. 43, Dortmund

Introduction

- Matthias Miehl
- Studied Electrical Engineering
- Software developer for 4 years

Until October at SOREL GmbH in Wetter.
From Januar at cbb Software GmbH in Lübeck.

makomi.net

- Projects (Python, TDD)
- Blog (Git, Z-Wave)
- Telegram group: embedded_sw_quality

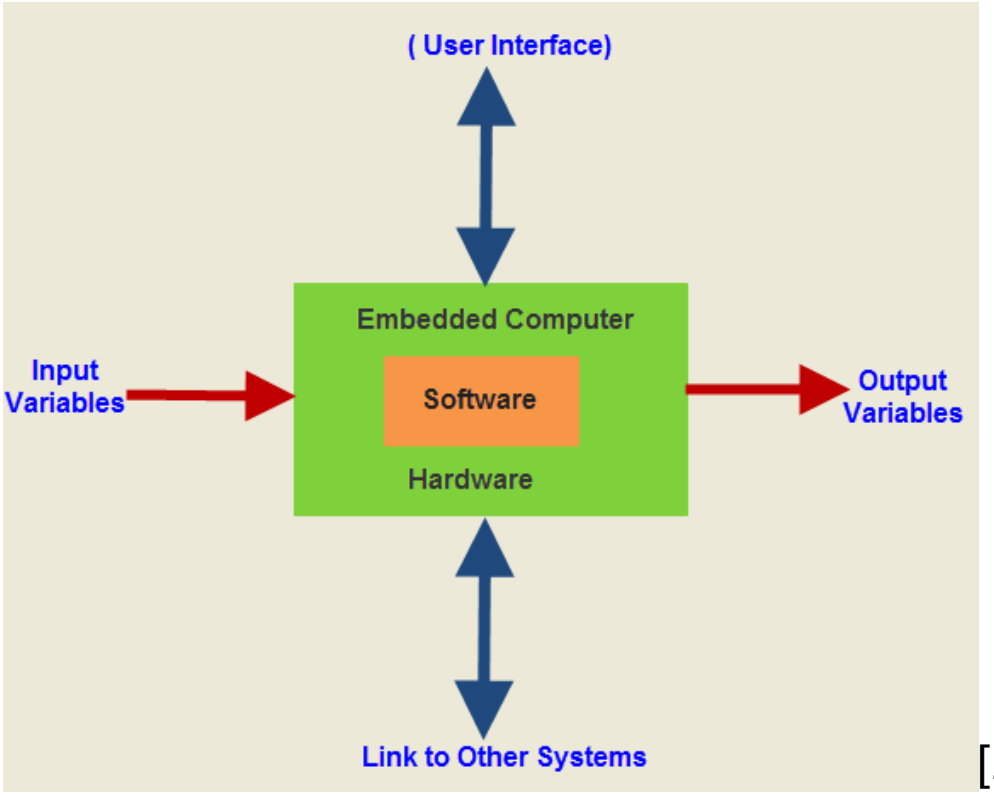
  “Relevant articles, news, and insights regarding all aspects of software quality with a focus on embedded systems.“

Goals

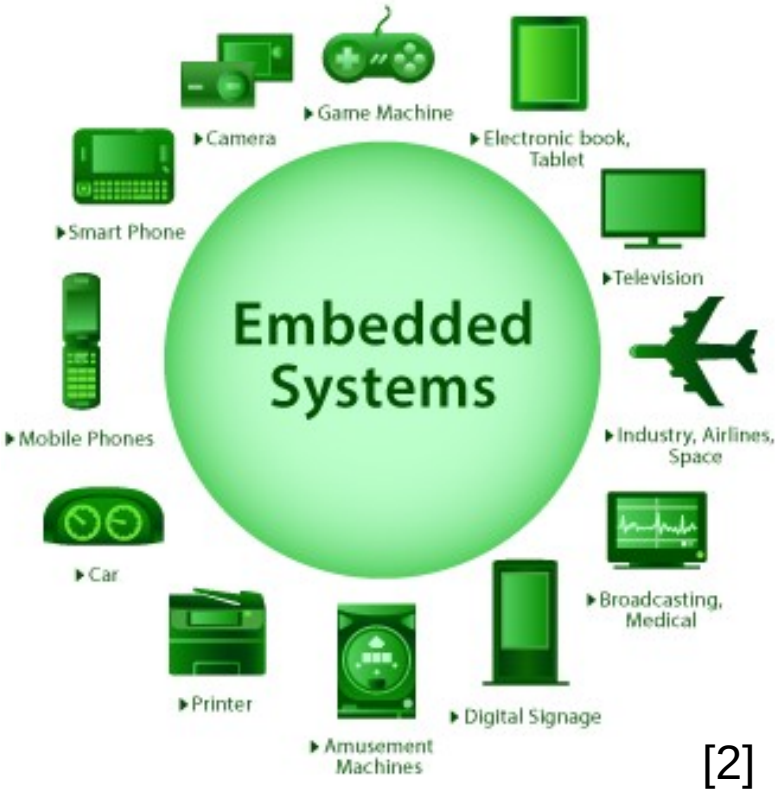
- Give an introduction to automated testing of embedded systems.
- Highlight differences when testing embedded systems.
- Share biggest the challenges and questions that came up.
- An interesting discussion.

What is an embedded system?

Our working definition
A physical device that contains a small computer purpose-build for the task at hand. This computer typically performs measurement and control tasks.

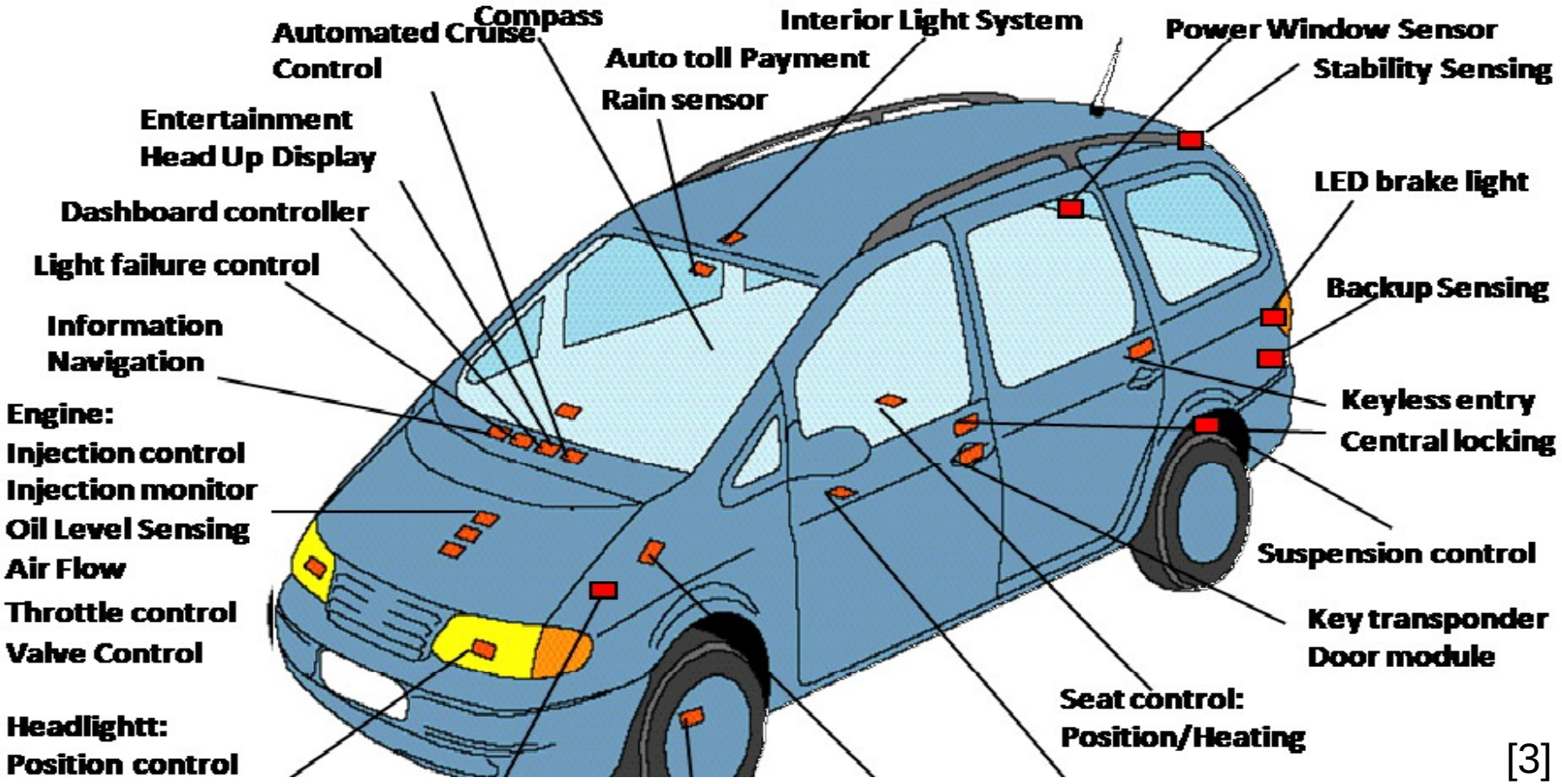


[1]



[2]

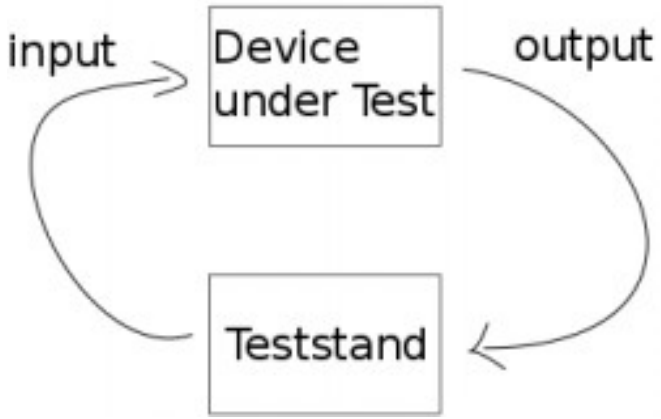
What is an embedded system? (cont‘d)



[3]

What are SiL and HiL tests?

Software in the Loop, Hardware in the Loop



The SW or HW is made to believe that it is built into a real system.

A test stand adapts all IOs for this purpose.

Test type: functional black box test.
Act as acceptance tests in our concept.

Example: HiL test stand

Sends electrical signals to the DUT, in order to emulate external HW.
Interprets the outputs of the DUT, in order to capture its reaction.

Parameterizes the DUT (e.g. via UART). Typically done by user or other ES.

- Setup: Load factory settings, select program, set parameters, ...
- Teardown: ...

Adapter boards are required for the different IO types of the DUT.

About SOREL

- Founded in 1991.
- Originally: Analog heating controllers
- Currently: Digital heating controllers and room thermostats

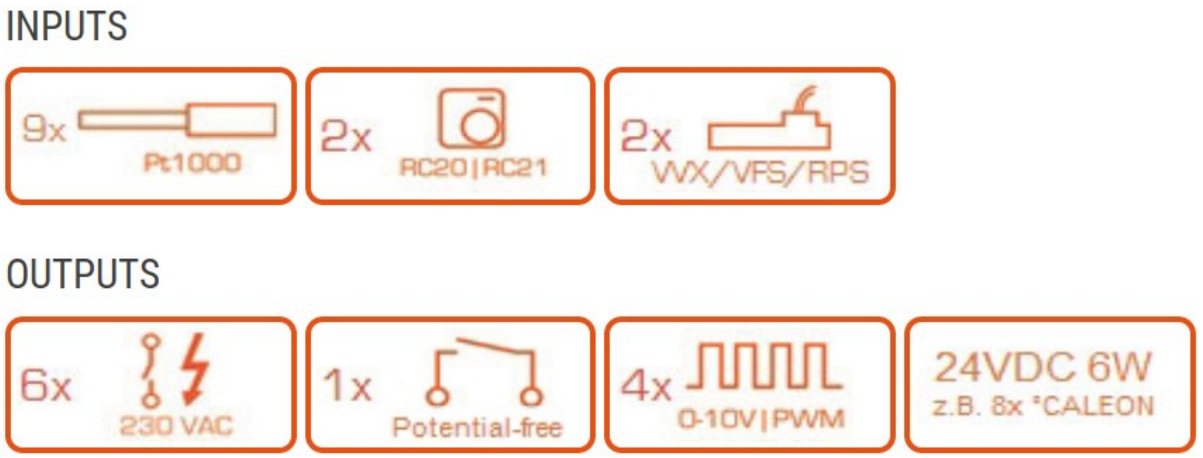
SOREL® Mikroelektronik GMBH

Initial situation

Development resources

- SW devs: 3 internal, 4 external + 1 to 2 students
- Testers: 1 key person + 2 to 3 Students

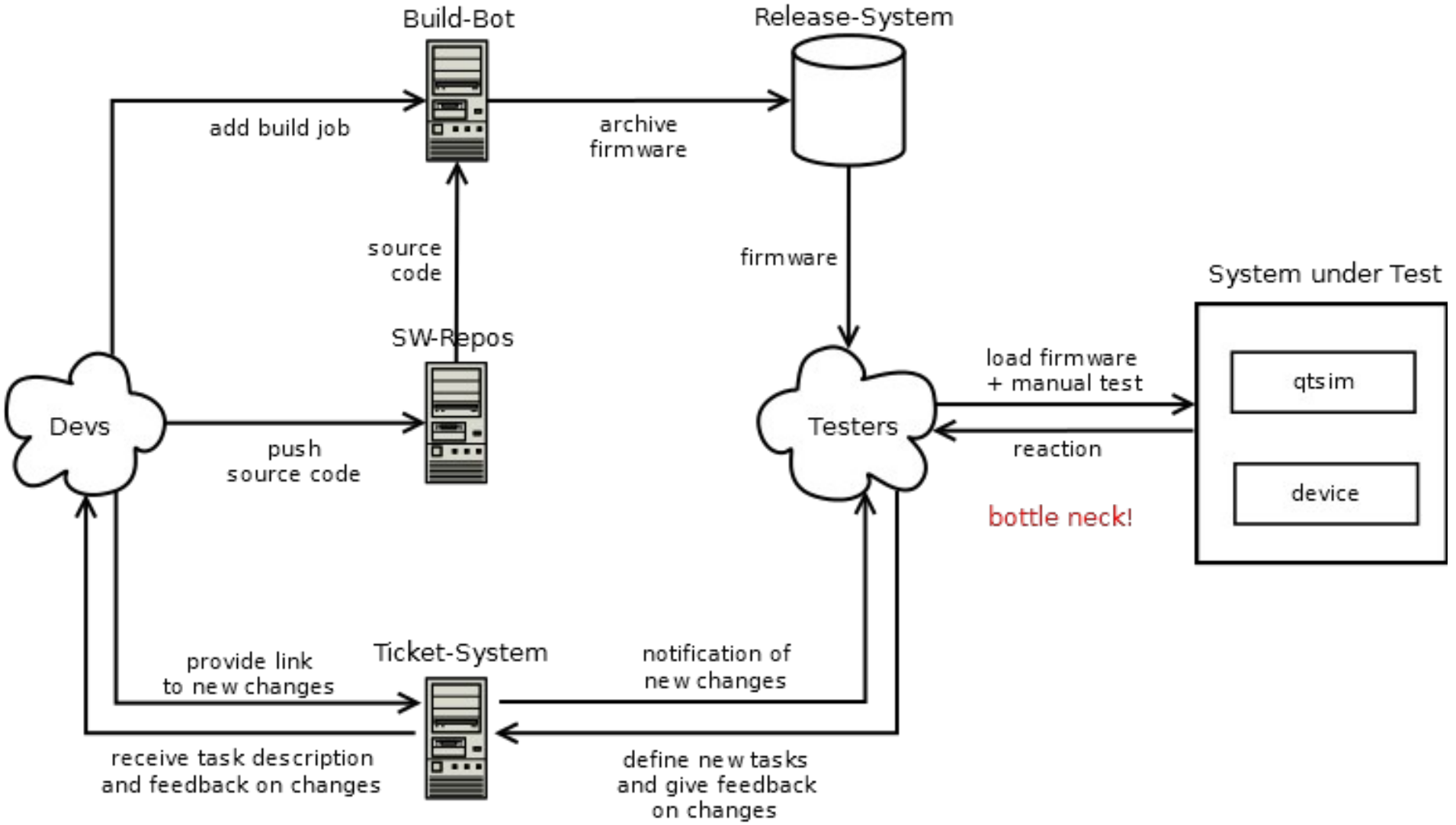
Product to be tested



Embedded-specific challenge in automated testing:
Establish connections to the HW. Every embedded system is different and makes its own demands on the test infrastructure. Therefore often DIY solutions.

Current development process

Interactions.



Typical issues

What is the root cause and what has to change?

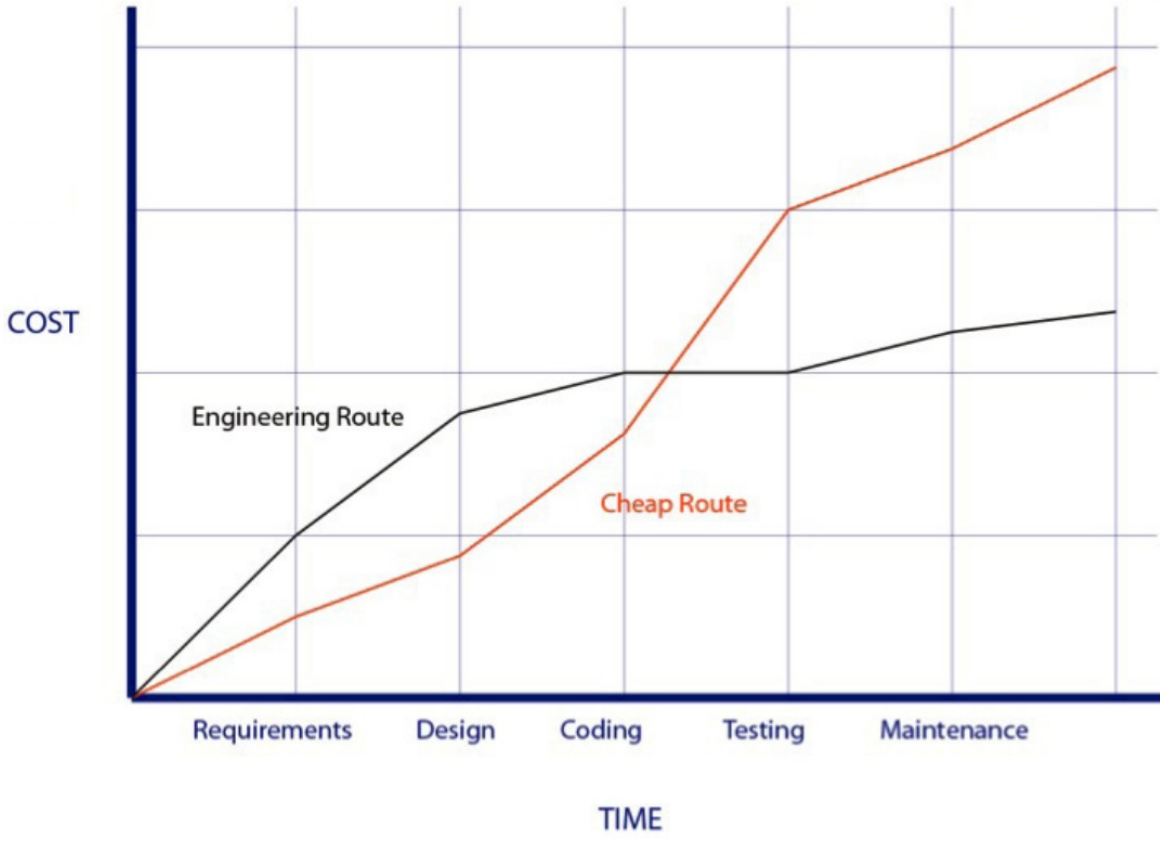
- Tasks are not completed within the scheduled time.
  The more complex the task, the more it is misjudged.
- The software fails to start over a long period of time or crashes repeatedly and the reason is not found.
- There are more and more defects and it takes ever longer until even simple defects are found and corrected.
- < 50% of the time remains for refactoring, planning, and new features. With a declining tendency.
- Software developers too often receive incomplete and/or incorrect specifications. This leads them to think up plausible user and technical requirements instead of programming – which of course does not work, especially with complex tasks.
- Things that used to work, suddenly don‘t work anymore.
- ...

Consequence: The stress level for all participants increases continuously.

Realizations regarding SW dev process

Not limited to embedded systems or automated testing.

„The Economics of Software Quality“ [4]



- The more complex the task, the more important becomes proper engineering (spec, design, TDD, Clean Code, ...).
- Every test type catches different defects. Combine several different test types to find as many defects as possible.
- It is impossible to optimize everything equally (dev time, code space, code execution speed, code cleanlyness). One has to prioritize and whatever comes last, suffers the most (typically testing).
- Automation is required if one wants to work in small task increments, i.e. fast feedback loops a.k.a. Agile.
- For software, the software dev process determines the resulting software quality.
- Only fix a bug if you have a failing test.
- Use best practices when starting a new project. -> Anhang B.

Solution approach

i.e. the interesting part of the presentation.

1. Prevent new damage (technical debt)
2. Repair existing damage (technical debt)
3. Automate high ROI tasks
4. Metrics
5. Use acceptance and system tests (SiL, HiL)

-> New development process

Solution approach

i.e. the interesting part of the presentation.

1. Prevent new damage (technical debt)

(a) Only make changes to code, if the code is in a good condition, i.e. it is easy to add a feature and obvious how to fix a bug. Otherwise, refactor the code prior to making any change.

(b) Fix defects as early as possible in the development process by using quality gates after major steps. This will allow us to expect a certain level of quality in the following step.

Solution approach

i.e. the interesting part of the presentation.

2. Repair existing damage (technical debt)

Problematic parts of the software that are large and central to the system, are refactored proactively to reduce the amount of new technical debt being created in their vicinity, by being the basis for new code.

Furthermore, since they are large, refactoring them takes some time and must be done before a feature needs to be added or a defect needs to be fixed in them. (cp. 1a)

Solution approach

i.e. the interesting part of the presentation.

3. Automate high ROI tasks

Create faster feedback loops by automating what currently makes the most sense to be automated:
- Has good ratio of being useful (task takes a lot of time + performed often)
- Easy/ quick to automate
- Makes sense in the bigger picture of where we want to be in the end

Example:
1. Automate integration
2. Pre-commit tests (Git best practices, coding guidelines, ...)
3. Unit tests

We don't need to fully implement each step, prior to taking the next. Once the ground work is done for one step, it is relatively easy to add more steps and improve existing ones. In the end, previously separate steps can be combined to one highly automated big step.

Solution approach

i.e. the interesting part of the presentation.

4. Metriken

We define metrics, collect data, and interpret it to understand what is happening/ whether we are improving, and what can be optimized.
Measure your current situation to determine whether you improve.

Interesting metrics:

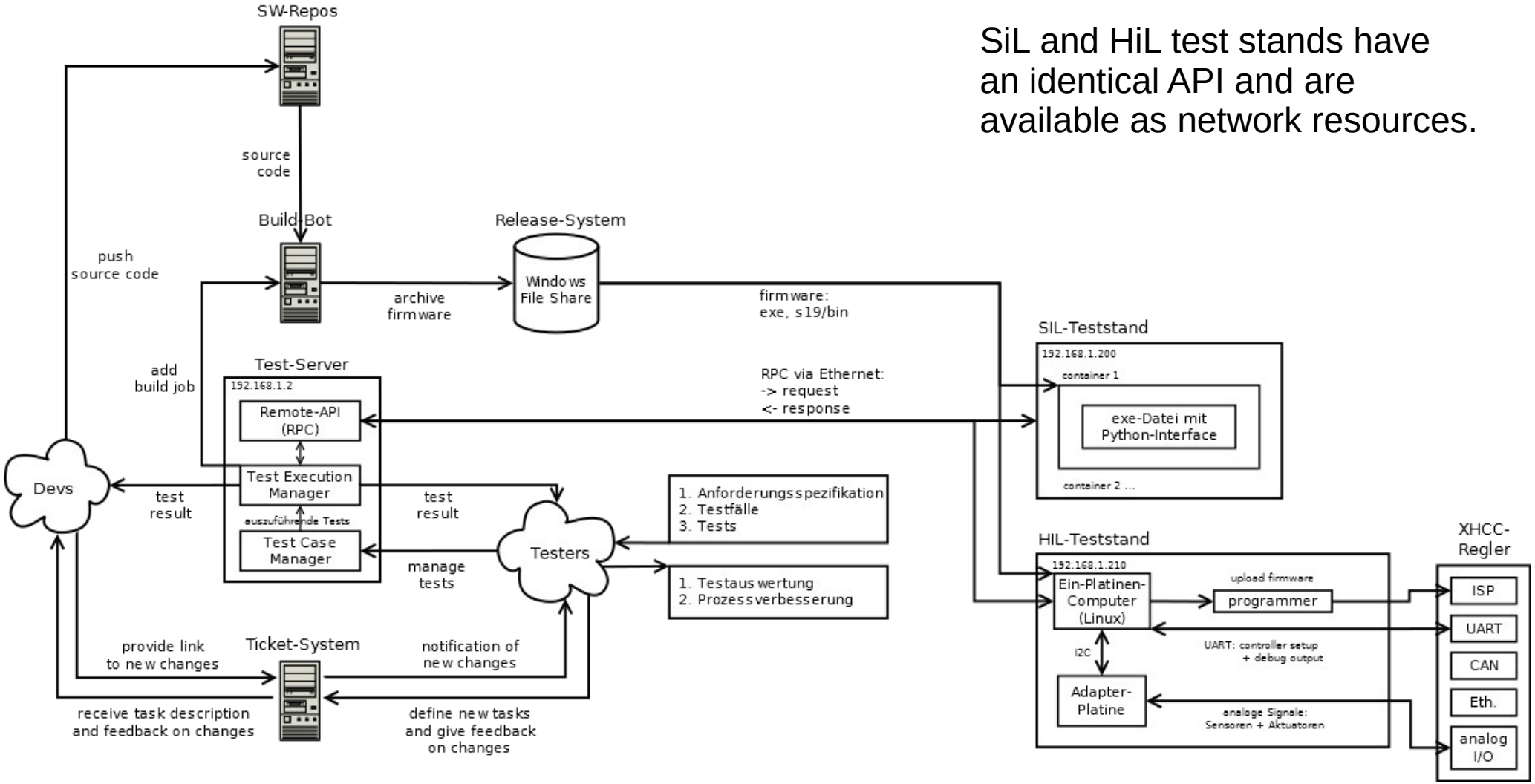
- Static analysis: compiler, linters, code complexity, ...
- Ratio between time spent implementing features vs. finding&fixing defects.
- Defect removal efficiency.
  Adopt the technologies that will get us above 95% in total defect removal efficiency before delivering the software.
- Cost of delivering a feature of certain complexity.
- Cost of fixing a defect at a certain stage.

Also: Keep an eye out for the path towards automated SiL and HiL testing.
One important prerequisite: written requirements/ functional specification.

New development process

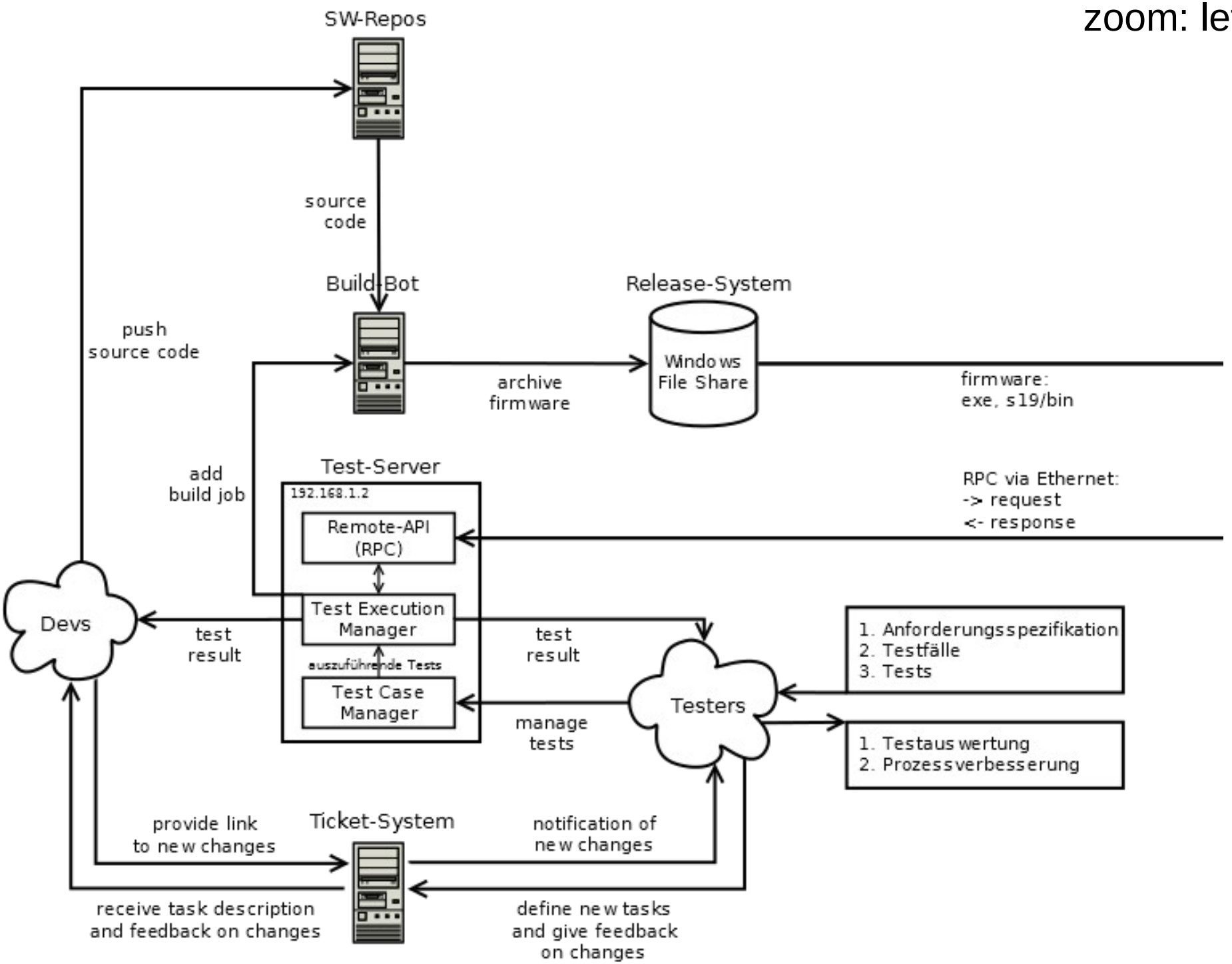
i.e. the interesting part of the presentation.

(1) Interactions

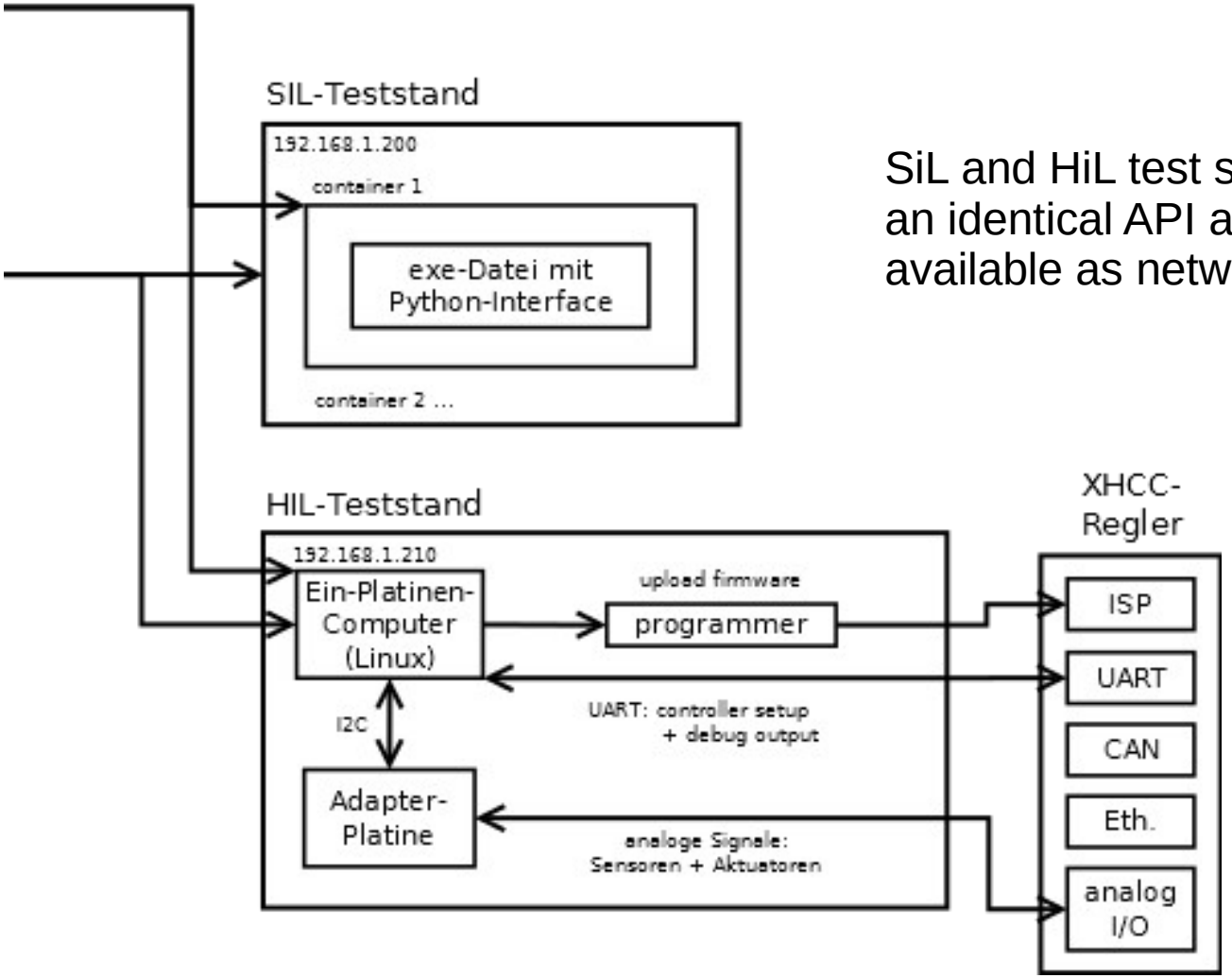


SiL and HiL test stands have an identical API and are available as network resources.

zoom: left


SW-Repos
source code
Build-Bot
Release-System
push source code
Windows File Share
archive firmware
firmware: exe, s19/bin
add build job
Test-Server
192.168.1.2
Remote-API (RPC)
RPC via Ethernet:
-> request
<- response
Devs
Test Execution Manager
test result
test result
auszuführende Tests
Test Case Manager
manage tests
Testers
1. Anforderungsspezifikation
2. Testfälle
3. Tests
1. Testauswertung
2. Prozessverbesserung
provide link to new changes
Ticket-System
notification of new changes
receive task description and feedback on changes
define new tasks and give feedback on changes

zoom: right

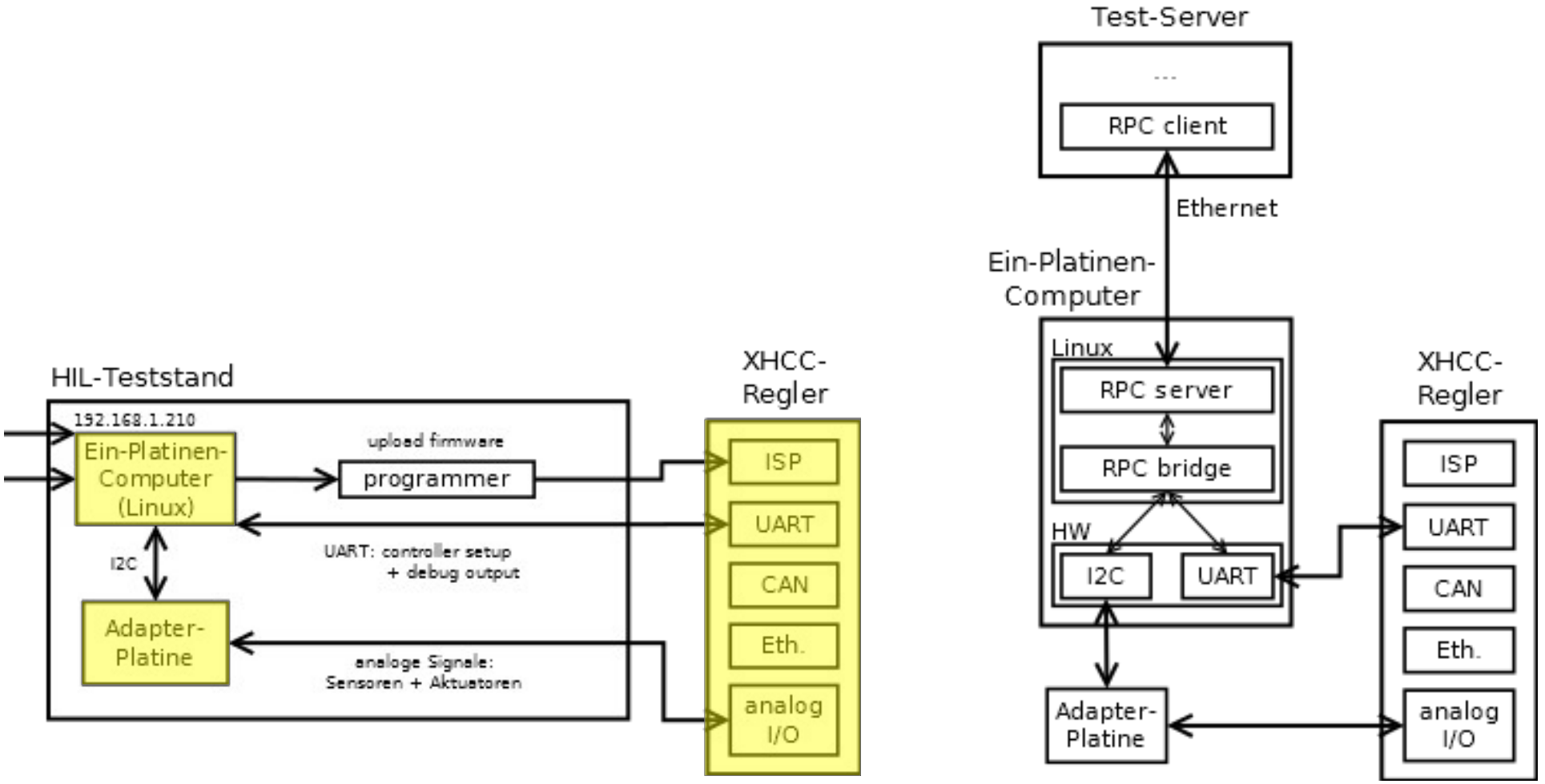


SiL and HiL test stands have an identical API and are available as network resources.

New development process

i.e. the interesting part of the presentation.

(1) Interactions: HiL test stand



New development process

i.e. the interesting part of the presentation.

(2) Process

1. Idea

2. Specification

2.1 User requirements aka requirements spec (Lastenheft)
2.2 QG1: Validation with customer
2.3 Technical requirements aka functional spec (Pflichtenheft)
2.4 QG2: Make sure the tech reqs are complete, clear, consistent, and correct.

3. Implementation (by oneself) (-> details p23)

3.1 SW design
3.2 Code implementation
Trace code back to tech reqs by linking tests and commits to lines/ items of tech reqs.
3.3 QG3
(a) Pre-commit tests (engineering/ tech reqs for developing the SW) (-> details p24)
(b) Acceptance tests (SiL)
3.4 Request pre-commit peer code review

4. Implementation (group)

4.1 QG4: Manual pre-commit peer code review
4.2 Automated integration into dev branch (where all other devs base their work on)

New development process

i.e. the interesting part of the presentation.

(2) Process (cont‘d)

5. In depth tests

5.1 QG5

- (a) Production and debug build
  All following stages use this binary -> *Deploy what you test!*
  Generate trend data: warnings, (more in-depth/ better) static analysis, unit tests, ...
- (b) Acceptance tests (HiL)
  Generate trend data: memory usage, timing data, … [a]
  Feeds it to the CI server for visualization.
- (c) (in parallel/ continuously done) manual/ exploratory tests
- (d) (in parallel/ continuously done) stress tests (time intensive)
  Ex.: capacity/ throughput, long running, fuzzing, property testing

5.2 Queue Release Candidate (RC) for release
+ tag respective commit in VCS with RC ID to connect it to all test results

6. Deployment

6.1 Automatically upload RC to update server once manually authorized
6.2 Receive device telemetry for field tests
Ex.: warnings, errors, kernel panics, sensor/ actuator anomalies
6.3 QG6: Customer ;-)

New development process

i.e. the interesting part of the presentation.

(2) Process detail: devs submitting a new increment of work

1. Rebase on development branch

2. Compile code (personal build)

3. Run pre-commit tests

4. Run SiL acceptance tests

5. Make changes and return to #2 until done

6. Once AOK: Request peer Code Review

New development process

i.e. the interesting part of the presentation.

(2) Process detail: pre-commit tests

Keep them fast and reliable!

Example: Let devs to run parts of the pre-commit tests during implementation.

- Git Best Practices
- No translation file issues?
- No new static analyzer issues (in the lines/ files touched)?
  (linters, code complexity)
- Does it compile?
- No new warnings (in the lines/ files touched)?
  `-Wall -Wextra -pedantic …`
- All unit tests green?
- Smoke test 1: Does the application run?
- Smoke test 2: Does the application perform its most fundamental function?
- Dynamic code analysis

New development process

i.e. the interesting part of the presentation.

(3) Expansion stages/ design ideas for the infrastructure

a. Semi-automate integration process (until it is finally fully automated)

b. TDD or at least unit tests

c1. Create CI/CD infrastructure design
c2. Set up and document new CI/CD infrastructure
c3. Integrate unit tests into the new infrastructure
c4. Integrate more: static analysis, code size monitoring, ...

d1. Build HiL test stands
d2. Write acceptance tests and integrate them

e. Build SiL test stands and integrate them

f1. SiL + process simulation
f2. HiL + process simulation

g1. Simulataneous testing of multiple controllers connected via CAN
g2. Virtual CAN controllers

Current challenges

Software quality (testing, CI/CD, embedded) is a big field.
- Difficult to grasp as a newbie.
- Difficult to recognize what defines the situation one is in.
  - What is the root cause?
  - What are our requirements and which are easy to implement?

  Hence, difficult to formulate a well adapted step-by-step instruction from scratch for building a CI/CD infrastructure.

Poor conditions
Lots of new stuff: Testing, CI/CD, embedded. Little time and people.

Result
Slow progress and great uncertainty about what helps the most and in which order test types should be implemented.

Iterative and explorative approach necessary.
- Con: Cost and time intensive.
- Pro: Internal know-how and an individual (optimal?) solution.

Specific uncertainties and questions

Relatively clear; Does somebody have any practical experience?

1) SiL/ HiL: Building your own solution vs. buying one?

2) Design
   How detailed?
   Which abstractions to choose?
   How much modularity is enough?

Less clear

3) Which test types first? (unit, integration, system)

4) Open source the core and develop it publicly?

5) Requirements engineering for such a small company
   5.1) Which degree of traceability makes sense?
   5.2) Which software to use for managing the traceability matrix and tests?

6) How to best approach the process simulation?
   Does somebody have any practical experience?

Specific uncertainties and questions (cont‘d)

7) How to determine "defect removal efficiency" in practice?

8) How well does our current approach match our situation?
   Do we focus on what is most important?
   How detailed should the analysis and plan be?

9) How to best deal with a lack of knowledge and
   experience in order to achieve good and fast progress?

10) Which abstractions/ architecture diagrams are best suited
    to support one when automating a process?
    (infrastructure, interactions, decision flow graphs, ...)

Appendix A: Useful tools and topics

Toolchain in a docker container
- dockcross: https://github.com/dockcross/dockcross
  Explanation: CppCon 2018: Michael Caisse
  “Modern C++ in Embedded Systems - The Saga Continues”
  https://www.youtube.com/watch?v=LfRLQ7IChtg&t=14m39s

Automated code formatting
- clang-format: clangformat.com

Compile time reduction
- Combining SCons and Ninja builds: https://el-tramo.be/blog/scons2ninja/
- Activate LTO (Link Time Optimization): `-Flto` (youtube.com/watch?v=dOfucXtyEsU&t=56m)

Git Best Practices
- git-cop: https://github.com/bkuhlmann/git-cop
- pre-commit: https://pre-commit.com/
- git-style-guide: https://github.com/agis/git-style-guide

Complexity metrics
- lizard: https://github.com/terryyin/lizard
- ravioli: https://github.com/ElectronVector/ravioli

Appendix A: Useful tools and topics (cont‘d)

Static code analysis
- Frama-C: http://frama-c.com/download.html
- clang static analyzer: https://clang-analyzer.llvm.org/
- clang-tidy: https://clang.llvm.org/extra/clang-tidy/
- infer: https://github.com/facebook/infer
- cppcheck: https://github.com/danmar/cppcheck
- PVS-Studio: https://www.viva64.com/en/pvs-studio/
- gcc-poison: https://github.com/leafsr/gcc-poison (w/o `strlen` and `strcmp`)

Dynamic code analysis
- Google AddressSanatizer: https://github.com/google/sanitizers/wiki/AddressSanitizer
- LLVM Clang: https://clang.llvm.org/
- gcc: `-fsanitize=address -fsanitize=undefined -std=c11 -fdiagnostics-color`
  Instrumentation options: https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html

Symbolic execution
- History of symbolic execution: https://github.com/enzet/symbolic-execution

Property testing
- Fuzzing vs. Property Testing: https://news.ycombinator.com/item?id=20279500
  HackerNews discussion and explanation
- rapidcheck: https://github.com/emil-e/rapidcheck

Appendix B: Best practices for new projects

Lightweight processes for improving code quality and identifying problems early:

1) Fix all of your warnings
2) Set up a static analysis tool for your project (linter, code complexity, ...)
3) Measure and tackle complexity in your software
4) Create automated code formatting rules
5) Have your code reviewed
6) „s/“ directory for common actions (https://chadaustin.me/2017/01/s/)
7) Setup Doxygen
8) Setup a code formatter
9) Be reasonable
   - Use a VCS like Git
   - Use a sensible directory structure (bin, build, doc, tools, src, …)
   - Automate the use of static analysis tools
   - ...

Based on an article by Embedded Artistry:
https://embeddedartistry.com/newsletter-archive/2018/3/5/march-2018-lightweight-processes-to-improve-quality

Hint: Subscribe to “Embedded Artistry“ newsletter.

Image references

[1] https://www.aldec.com/images/content/TySOM_Embedded-Systems.png
[2] https://www.elprocus.com/wp-content/uploads/2016/10/Embedded-System.png
[3] http://www.theengineeringprojects.com/wp-content/uploads/2016/11/automotive.png
[4] http://www.informit.com/store/economics-of-software-quality-9780132582209

Text references

[a] Jacob Beningo, „5 Embedded System Characteristics Every Engineer Should Monitor“, https://www.beningo.com/5-embedded-system-characteristics-every-engineer-should-monitor/, 2019-02-14, Zugriff: 2019-11-12

Questions?

Thank you!

E-Mail: say-hi@makomi.net

Slides: makomi.net/posts/automated-testing-of-embedded-systems/

Telegram group: embedded_sw_quality

“Relevant articles, news, and insights regarding all aspects of software quality with a focus on embedded systems.“