Abstract
Software development and testing is one of the most important and time consuming tasks in the creation of practically any embedded system today. This is directly related to the ever increasing amount of software in embedded devices and its relevancy with regard to safety and security aspects. The industry realized long ago that inadequate software integration and testing has terrible cost consequences due to many different factors, e.g. late bugs or redesigns, liability issues, late access to the market, etc. Moreover, industry realized that the best way to deal with the software complexity is to start performing integration and testing as early as possible. Advance software engineering methods, such as e.g. Extreme Programming, advocates using Continuous Integration (CI) as a way to solve the “integration hell” of having many software developers working on different parts of the code base simultaneously. Continuous Integration is intended to be used in combination with some sort of automated testing (and build servers), which allow the ability to automatically run tests periodically or even after every commit action. Continuous Integration works well with common unit testing methods that do not rely on the final physical devices to be available before the testing can get started. However, testing software layers that are closer to the hardware and/or require a physical device is much more complex to scale to the needs of every software developer. This is mainly due to the late and limited availability of physical hardware prototypes or final devices. This whitepaper describes how simulation-based Virtualizer Development Kits are the perfect technology to remove the dependency with hardware and to enable the integration and testing of hardware dependent software (as well as full software stacks) in a continuous manner. A case study will illustrate the integration of VDKs with the most popular CI framework, Jenkins, and the most popular testing framework for Linux and ARM-based SoCs, LAVA, in the context of Linux kernel and device driver development.

Introduction
Software development and testing are some of the most effort consuming tasks in the creation of practically any embedded system today. This is directly related to the ever increasing amount of software in embedded devices, the more severe requirements in terms of quality (including safety and security aspects), plus the ever shortening development times from conception to mass production. These challenges imply that companies performing inadequate or inefficient software integration and testing are running higher risks with terrible cost consequences. For instance, bugs found late that delay a project or the cost of problems found after release of a product (including liability issues).

The embedded industry has realized that the best way to deal with the software complexity is to start performing integration and testing as early as possible. Agile development, Unit Testing, Continuous Integration, Automated Testing are all methods that try to enable software developers to achieve the goal of shortening the duration of projects by shifting left the start timeline. However, for many projects, embedded software has a strong dependency on the availability of the targeted hardware. This dependency implies that the potential benefits of doing early integration and testing cannot be
fully realized. Physical prototyping is a technique used to alleviate this problem, allowing software teams to start working earlier. However, in general physical hardware is cumbersome to work with and it does not scale easily for advance software engineering methods like CI and Automated Testing.

Virtual prototyping is a simulation-based technique that uses models to mimic the digital hardware. Virtual prototypes are accurate enough to allow the same embedded software that executes on the physical hardware to run on the virtual version unmodified. Being simulation-based, only virtual prototyping has the flexibility and ease of use to scale to the needs of software teams who have embarked on such advance software engineering methods, such as CI.

**Continuous Integration and its limitations for embedded software**

Advance software engineering methods, such as e.g. Extreme programming, advocates using CI as a way to solve the “integration hell” of having many software developers working on different parts of the code base simultaneously. Benefits of CI are to detect earlier integration bugs and avoid last minute chaos at release dates, which saves both time and money over the lifespan of a project. The CI lifecycle advocates to:

1. Commit changes regularly to a source version control system
2. Build the software automatically after every commit action
3. Test the software automatically after every build is done
4. Report the results back to every developer.

In order to scale this process to every software developer working on the project, build and test servers are used to provide the required computing power and automation.

![Continuous Integration lifecycle](image)

A CI framework can take care of automatically running tests on the software after it has been successfully built. Testing hardware dependent embedded software (e.g. device drivers) or complete embedded software stacks requires the software to be executed in the target system. When using physical hardware for this, not only is the execution platform required but also special hardware devices. These special hardware devices are required to control, stimulate and monitor the target system in order to carry out the tests in an automated and continuous fashion. This is very cumbersome to setup and maintain, but also very difficult to scale to the needs of every software developer part of the project.

**Continuous Delivery and its limitations for embedded software**

Beyond CI, Continuous Deployment or Continuous Delivery is a process adopted by many companies. In a Continuous Delivery process, the software can be released in production at any moment (after a certain maturity is reached). Continuous Delivery has more rigorous requirements in terms of quality and hence requires more testing to be carried out automatically in order to prevent defects escaping to the released
software. This includes new bugs introduced by new software, but also old bugs (regressions). This level of automated testing is extremely complex and expensive when performed with physical hardware close to the final target system. System testing typically happens in very specialized and complex hardware labs, which are difficult to scale and automate. Running system tests may take days or even weeks in a hardware lab, which makes the concept of Continuous Delivery practically impossible in those circumstances.

Continuous Integration and Automated Testing Using VDKs

A Virtualizer Development Kit (VDK) is a “software development kit” that uses a virtual prototype as a target. A virtual prototype is a fully functional representation of the digital hardware of a system. This representation is detailed enough to let the same embedded software binary execute unmodified on a virtual prototype as in the real hardware, but also abstract enough to simulate close to real time. Hence, virtual prototyping enables early integration and testing of hardware dependent software and complete software stacks.

Being simulation-based, VDKs can easily scale to serve every software developer working with a CI system. No extra hardware is required to enable physical prototypes, which by itself simplifies the setup, maintenance and reduces the cost. Moreover, VDKs can also be used in a Continuous Delivery process to allow executing as many test as needed during a nightly run, through parallelization. As an example, a VDK customer was able to run 700,000 tests (including fault injection tests used for functional safety validation) in one night using 600 parallel simulations executed in a cloud-based server farm[1].

Integration with testing frameworks

As mentioned in the previous section, VDKs replace the target hardware with a simulation model, but it does not replace the testing framework required to create, schedule, dispatch and monitor the tests. Instead VDKs aim to seamlessly integrate with any testing framework and tools used in existing flows. Hence, being able to reuse the same tests already available.

A typical test framework executes tests that apply stimuli (or control stimuli sources) to the device under test and gather results to decide whether the test passes or fails. In order to link a VDK with an existing testing framework, the VDK provides a set of interfaces that can be used to remotely control and communicate with the VDK. This interface provides a TCP/IP socket-based connection to control the built-in scripting layer. More information about this interface can be found in a previous white paper[2]. The VDK scripting layer is used to control the simulation (run/stop, pause/continue), as well as apply stimuli and gather responses from external interfaces like with physical hardware.

---


However, a VDK is not limited to just these external interfaces, but any internal part of the hardware model can be stimulated, faulted and monitored. These extra capabilities can be used to extend the test scenarios to cover corner cases difficult to recreate with hardware, as well as fault injection scenarios that mimic faulty hardware or transient errors.

**Fault injection capabilities**

VDKs have built-in fault injection capabilities through the scripting layer. Through this layer a user (interactive) or a test (automated) have access to all internal modeled hardware elements, e.g. memory content, registers and signals, which can be used to inject a fault anywhere on the hardware or the loaded embedded software. The major benefit of fault injection with VDKs is that it is completely non-intrusive, i.e. unlike other techniques the embedded software does not need to be modified. On top of this, the scripting layer provides full control to define complex scenarios and corner-cases difficult to produce in real hardware. This is achieved by concatenating triggers to describe complex situations. Trigger actions can be based on time, hardware and software events. Finally, VDKs are completely deterministic and, unlike with hardware, fault scenarios can be repeated and put on regression testing to run after any change on the embedded software. More information regarding fault-injection with VDKs can be found in a previous white paper on Fault Mode and Effect Analysis (FMEA)[3].

**Code coverage capabilities**

Another important capability of VDKs, in the context of CI and testing, is code coverage. Code coverage is a key metric to measure the degree to which the source code is tested. Higher code coverage means lower chance to contain bugs in your software. VDKs provide an on-target code coverage measurement solution, which does not require the software to be adapted or instrumented. Unlike using real hardware, measuring code coverage in a VDK does not need or consume any hardware resource. All this together improves the quality of the measured coverage, making VDKs a perfect tool to guide the continuous improvement of the test suite. VDKs offers good support for coverage criteria including: statement and function coverage, call coverage and branch coverage. VDKs provide an XML-based flow that can gather and unify the results of multiple (parallel) test runs. This flow can also be integrated as part of a CI system, for example Jenkins through the usage of plugins like Cobertura[4].

**Case Study: VDK Integration with Jenkins and LAVA**

In this case study we use an embedded Linux software port for an ARMv8 Cortex-A multi-cluster reference platform as an example. Figure 3 provides an overview of the process and the different tools used. The source code of the Linux kernel and device drivers are added to a Subversion[5] source version control system. Jenkins is used as our CI system and LAVA is used as the automation framework to test the software. LAVA uses a VDK as a target system. This VDK models the ARMv8 Cortex-A multi-cluster platform.

![Figure 3. VDK integration with Jenkins and LAVA](Image)


Jenkins configuration

Jenkins[6] is the most popular open source CI tool. Jenkins is written in Java and brings by default a lot of support for Java based projects. Jenkins is extensible through plugins to support projects written in languages other than Java and to integrate with many different version control and database systems.

In our case study, we create a new Jenkins project (or item) as a “Freestyle project”. We configure the new Jenkins item to use the Subversion repository link to the embedded Linux sources. This is shown in Figure 4. For the next step, we configure the Jenkins project to rebuild the software every time a commit action happens in the repository. For that we select ‘Poll SCM’ in the ‘Build Triggers’ section of the project and add a “post-commit” hook in the Subversion repository. This “post-commit” hook will inform Jenkins that it should trigger a new build every time a software developer commits a change to the main repository.

![Figure 4. Source code management configuration in Jenkins](image)

The next step is to configure the build. For this we use regular shell scripts that can be called from Jenkins, as illustrated in Figure 5. A first script will be added to call the Linux cross-compile process. This build process relies on ‘make’ and the ARM GNU cross compiler chain. The verbose output of the build is recorded by Jenkins and any errors will be reported as a failed build by Jenkins. All information can be visualized from within the Jenkins web interface. If the build is successful a new set of binaries are created for the Linux Image, filesystem and bootloader.

![Figure 5. Build and test configuration in Jenkins](image)

As a subsequent build step, the recently built software is tested. For this, we use a second shell script that controls the testing. This shell script relies on LAVA to run the tests, see Figure 5. More information about LAVA and the existing integration with VDKs can be found in a previous white paper[7].


In the shell script, a LAVA job is passed as an argument to the LAVA dispatcher. This LAVA job points to the software recently built, the target device (in this case the ARMv8 VDK) and the tests to run. Both the LAVA dispatcher and the VDK simulation will send their output to Jenkins for reporting. In case anything goes wrong during the execution of the tests, Jenkins will label the build as failed. The figure below shows a brief extract of the LAVA and VDK logs as recorded by Jenkins.

Summary

In this whitepaper we discussed how advanced software methods like Continuous Integration and Automated Testing are key to cope with the challenges and risks of embedded software projects. We also pointed out how the dependency on hardware availability and its limitations in terms of scalability are a blocking factor to the successful adoption of techniques such as Continuous Integration and Continuous Delivery. We introduced virtual prototyping as the technology that can truly scale to serve the testing needs of both, (a) embedded software developers in a team using a Continuous Integration system and (b) a quality assurance department trying to adopt a Continuous Delivery process for its embedded products. We showed, through a case study, how Virtualizer Development Kits can seamlessly integrate with the most popular Continuous Integration tool, Jenkins, and the automation framework, LAVA, to easily complement and enhance existing testing flows.

About the Author

Victor Reyes is currently a Technical Marketing Manager in the System Level Solutions group at Synopsys. His responsibilities are in the area of Virtual Prototype technology and tools with special focus on Automotive. Victor Reyes received his MsC and PhD in Electronics and Telecommunication from University of Las Palmas, Spain, in 2002 and 2008 respectively. Before joining Synopsys, he held positions at CoWare, NXP Semiconductors and Philips Research.

Additional Resources