Supporting Continuous Integration in Embedded Software

Andrew T. Graham

Tektronix, Inc. | andrew.graham@tektronix.com

Abstract

Applying continuous integration practices to embedded software development is an endeavor filled with challenges unique to a hardware-centric ecosphere. One such challenge, imperative to software quality, is overcoming a pure development mindset. In a hardware project, the ability to use a project's automation infrastructure to test a customer experience ahead of a traditional test phase is invaluable.

Continuous integration practices for an embedded software application enables: abstract and modular test pipelines, greater coverage over possible customer-facing permutations, and a flexible yet maintainable infrastructure.

This paper covers three key concepts that are core to continuous integration in embedded software: minimum accessibility, abstract communication, and dynamically reconfigurable hardware.

Biography

Andrew Graham is a Junior Software Engineer at Tektronix. His current work emphasizes improvement of build deployment and continuous integration architecture. In 2014, he graduated from Portland State University with a Bachelor's of Science in Computer Science and a double minor in Mathematics and Philosophy.

1 Introduction

Development of embedded software is heavily influenced by the hardware. Key project milestones often describe the progress of hardware. Software milestones, in this situation, can be mapped to those already defined by hardware. This leads to a situation where software can feel like it's in an unnatural gait. The demand for continuous integration practices and systems to ease these pressures is high, but, often it is challenging to begin implementation.

Continuous integration is a software development paradigm that encourages the frequent integration of code into a common repository. This simple action is frequently coupled with automated sequences that build, test, and qualify code with the goal of reducing risk (Duvall, Matyas, & Glover, 2007). Continuous integration systems need to include the hardware for embedded software. In embedded software, risk is reduced by verifying quality and behavior on the hardware it was intended to be distributed on.

This paper introduces three concepts that can be used to help begin continuous integration efforts in the field of embedded software. Those three concepts can be summarized as:

1. Minimum Accessibility
2. Abstract Communication
3. Dynamically Reconfigurable Hardware

In a typical software development scenario, there are many tools and best practices that are used to generate dynamic and flexible pipelines in continuous integration systems. Being able to use container images for just-in-time testing of small applications makes sense; however, measuring the quality of the software when physical hardware is involved adds complexity. We must test as closely to the physical end-user configuration as possible. That means, at minimum, testing the software on the hardware. In the best case, it means testing the software and hardware in conjunction while the product is in a customer facing state. As much testing as is possible should be:

1. On actual hardware (with the hardware optimally in an as-delivered state)
2. A component of an automated continuous integration pipeline

Software should be tested on prototype hardware as soon as it is available. Hardware, especially when proprietary, has an added challenge of being difficult to abstract into nice, organized containers or virtual machines. Even if concepts such as containers or virtual machines are used for simulating the hardware there is still a wide gap for testing the complete customer-facing package. This gap reinforces the need to test on the actual devices being developed. It also underscores the concept that the hardware needs to be accessible to the continuous integration infrastructure. The product and its interfaces spend time in infancy while being developed. The interfaces that will used to automate and test may not exist during the initial stages of development. When they are developed it's likely that they have never been automated before. Implementing continuous integration onto embedded software on developing hardware is not usually trivial. Initially, it can be a complicated form of parallel development, vying to take advantage of functionality as it becomes available.

When the number of possible customer configurations is high, integrating hardware with an automated build and test system becomes increasingly difficult. Often, each new physical product requires custom work to integrate into an automated system. It can be difficult to re-use automated functionality from product to product since much of it can be sensitive to the physical device itself. The significant cost of producing many prototypes to support development must be considered. Ideally, automation should be able to:

1. Accept an arbitrary physical device
2. Deploy applicable embedded software to the device
3. Execute relevant tests
4. Re-configure the device in preparation for the next iteration without the need for manual intervention

1.1 Example Infrastructure

The principles outlined in this paper may not be applicable to all embedded software applications. To demonstrate why some of the techniques work, it is first necessary to describe the supporting stack used during the evolution of these ideas.

This paper attempts to generalize as much as reasonable. Past this introductory section, reference to these specific tools will be limited unless it is necessary in the illustration of the core concepts.

The following applications comprise the continuous integration system referenced in this paper:

1. Jenkins (Cloud Bees, Inc., 2017)
2. Bitbucket (Atlassian, Inc., 2017)
3. VSphere (VMware, 2017)
4. Vagrant (HashiCorp, 2017)
5. Chef (Chef Software, Inc., 2017)

The automation server that will be referenced in this paper is Jenkins.

Bitbucket is used for version control, though that is a small aspect of the specific topics covered by this paper.

Virtual Machines were used extensively. The VSphere hypervisor was used to contain the virtual machines in conjunction with Vagrant for controlling availability and Chef defining the configuration of the Virtual Machines.

1.2 Testing the Software on the Hardware

Before getting into the three techniques that are the focus of this paper, a moment should be taken to emphasize the importance of testing on actual hardware when developing embedded software.

There are many techniques for simulating or sandboxing software applications in various containers, abstractions, and virtual machines. These methods work when you need to quickly verify the behavior of software that runs within the confines of other software. Embedded software, on the other hand, is meant to run in the confines of the hardware it's being written for. Mock interfaces and pure development environments should be used for quick prototyping. Gaining confidence in the quality of the software as a customer is supposed to receive it, however, requires a concerted effort in testing on the hardware it was intended to run on.

We gain in two critical areas with this approach: test coverage of the real hardware interfaces and confidence in the complete end-user package.

Mock interfaces are good for verifying ideal and logical behavior of software on hardware. They are not so good at identifying physical issues (timing for instance) that manifest when integrating the software with the hardware.

It should also be understood that development confidence in the customer facing solution is greatly enhanced by testing the salable product.

2 Minimum Accessibility

A typical product has a few existing customer-facing interfaces. Where possible, automated testing should use these interfaces. There are times it is not possible for automation to use the existing interfaces and special ones must be created for accessibility to functionality of the embedded software. The addition of these special interfaces is what constitutes minimum accessibility.

Minimum accessibility is the amount of additional access, on top of customer-facing interfaces, to the physical product that is tolerable to accommodate continuous integration of the embedded software on hardware.

Accessibility exceptions must be made, but, to the smallest extent necessary. Utilizing customer-facing interfaces is essential to gaining broad confidence in the quality of functionality in the embedded software. Abstracting those connections in automation is imperative. In an ideal world, we would always test the exact configuration a customer is going to buy. There would not be any of the assumptions, shortcuts, or band-aids that are often applied in a development environment.

Creating exceptions for accessibility is the pragmatic solution in the development of embedded software with continuous integration. Engineers should not be hindered any more than necessary in their pursuit of finding and fixing issues in a continuous integration system. If engineers are going to quickly and accurately debug an issue then they need more than the polite prompt or warning dialogue a customer might see. They are going to want access to consoles, logs, and the underlying system.

Minimum accessibility is practical because it means that the difference between a salable environment and an environment intended for testing is minimum, by definition. Given some customer-facing device then only explicit exceptions should be made available for continuous integration. Any exceptions should be well known and understood by the development team.

2.1 Deciding on Minimum Accessibility

The spectrum of embedded projects is vast and there is no one-size-fits-all rule for the correct amount of minimum accessibility in continuous integration testing. Making accessibility exceptions should be a matter of determining acceptable risk. Any modification to a customer-facing device introduces some risk. That risk should be well understood before committing to any sort of accessibility exceptions. Once the risks have been assessed, then the next step is determining how much accessibility is necessary and reasonable.

Minimum accessibility is achieved by starting at a strict customer state and then adding explicit exceptions for the continuous integration system. The system should collect applicable logs and gather as much information as is requested by the development team. This should not expand into a full debug environment. It may be the case that your customer-facing functionality already provides an adequate interface for automation.

We used the following questions can be used to determine what level of accessibility is necessary:

1. What interfaces are available to a customer? How expressive are they?
    a. Graphical User Interface
    b. Physical buttons/Display
    c. Programmatic Interface
2. What cannot be achieved through an existing customer interface? Is it accessible through a backdoor (think customer service/repair technician access)? Can automation access the device this way? Is that okay?
    a. Collecting or viewing logs
    b. Analyzing hardware state/internal diagnostics
    c. Testing manufacturing steps
3. What things cannot be acquired through a backdoor or interface (think engineering/development debug access)? Can external mechanisms simulate the action?
    a. Debug console
    b. Error case simulation
    c. Power operations (i.e. physically pressing the power button to turn the instrument on/off)
    d. Connecting/Disconnecting external media
    e. Connecting/Disconnecting fixtures, accessories, attachments

The amount of access that automation requires will depend on the breadth and depth of testing being done and the amount of information the software engineering team needs to be successful in development.

2.2 The Tradeoffs of Testing a Minimally Accessible System

When talking about testing software there is often a weighing of the pros and cons of release versus debug builds. Embedded software is no different apart from being embedded on some device. In fact, the same arguments can be made in the embedded world for testing both with and without some accessibility exceptions.

Some potential risks testing embedded software with minimal accessibility, depending on the exception, are:

1. Timing issues
2. Unintended reliance on debug mechanisms that should not exist in the release product
3. Access to functionality that does not accurately represent scenarios customers could find themselves in
4. Unintended security loopholes that become undetectable when special interfaces are enabled

On the other hand, testing embedded software with minimal accessibility has its upsides as well:

1. Access for debugging
2. Quick prototyping
3. Tolerant environment

3 Abstract Communication

Continuous integration systems have many positive qualities for workload and task balancing. Ideally, we would be able to have our hardware take advantage of those capabilities. When doing so, we have to be careful to respect minimum accessibility. Finding the compromise between the benefits of the continuous integration system and maintaining minimum accessibility brings us to abstract communication.

Abstract communication is how a continuous integration system communicates with a device under test while respecting minimum accessibility. This is achieved through intermediary applications, virtual machines, and/or containers that act as representatives to the continuous integration system for the hardware.

For Jenkins, there is the concept of slaves or nodes. These are the assets available to the continuous integration system for jobs to accomplish various tasks on. A traditional example of a node would be a build server. It is often possible to add the hardware that's being developed as a node in a continuous integration system. At face value, this has some benefits:

1. Streamlined build deployment
2. Easy to manage testing
3. Leverage the power of a continuous integration system

However, adding hardware as nodes in the system has the drawback of potentially over-exposing interfaces. This directly violates minimum accessibility and is undesirable. There is a compromise in abstracting the communication between the continuous integration system and the hardware.

Continuous integration systems are already decent at load balancing and managing pipelines. Given those positive qualities, the appeal of using the instruments as nodes for testing is highly desirable. The catch is that it is very undesirable to have Jenkins infrastructure installed on the customer-facing physical devices. If the salable product is not going to be enabled for use as a node in an arbitrary continuous integration server, then it does not make sense to allow for any of it to be present.

Yet, if the instruments are not nodes then it becomes an awkward balancing act. Jenkins already balances workload and testing in a transparent and sensible way. It is possible to reconcile the usage of customer-facing devices as test nodes, while practicing minimum accessibility principles by using abstract communication.

An abstract and modular pipeline is essential when deploying software builds to a dynamic range of hardware. Abstracted communication enables flexible pipelines to be created between the physical device and the automation infrastructure. Abstract communication is all about leveraging the inherent abilities of a continuous integration system while simulating physical instruments as testing nodes

3.1 Rationalizing Abstraction

Our first attempts at integrating instruments into continuous integration was to put the client code on the instrument. This created a scenario where the instrument itself was a node of automation. At face value, this achieved the basic benefits outlined earlier in this section, but it had the distinct disadvantage of not being in a customer-facing configuration. The modifications that were made to make the device a node made it difficult to maintain good minimum accessibility. This posed an unacceptable risk. For example, if the instrument crashed then it also crashed as a node. The situation required much manual intervention that was untenable in the long term.

Abstracting the relationship with a virtual machine, container, or other device achieves the same perceived benefits as directly using the product as a node without exacerbating the minimum accessibility problem. In fact, abstracting the communication demonstrates good discipline when assessing what is required to be minimally accessible by automation.

By abstracting communication additional testable events can be enabled that provide insight and coverage into various customer-facing interactions, such as:

1. Being able to install a new application on the instrument
2. Being able to perform power operations on the instrument (i.e. reboot, power off)
3. Being able to verify physical test configurations (i.e. check cables, sources, probes)

If the device were configured as a node on the continuous integration system then there is additional complexity added to the pipeline if a node needs to power-cycle. This would break the communication between the continuous integration system and the node until it came back alive. When the automation system communicates with an abstraction, then pipeline can be implemented in a transparent and straightforward way. For example, if the device needs to be reset then a test could be written to reset the device. The test will pass or fail depending on whether the device successfully reset and the continuous integration system gets test results from that operation.

Additional testable events and appropriate naming of the abstractions so they look like the actual devices to the automation system provide a marked advantage toward the appearance of a rational development pipeline. The idea of abstract communication can be expanded to more than the literal connection pathway between the device and the continuous integration system. Abstracted communication can also manifest itself as a bridge between developers and the testing of embedded software on some devices; For example, a developer might see the following sequence of events:

1. Build A triggers from a code change in version control
2. Build A deploys to some set of instruments in the test pool device(s)
3. Tests are executed against the device(s) running the embedded software
4. The test results are collected, analyzed, and the build is promoted (or not)

The reality of what is occurring behind the scenes is much more complicated:

1. Build A triggers from a code change in version control
2. Build A deploys to some set of instruments in the test pool; The abstract representation of each of these instruments:

    a. Executes diagnostic tests (Ensure the physical device is responsive and in a ready state for Build A)
    b. Executes an installation test
3. Tests are executed against the devices running the embedded software by the set of abstract representations of said devices
4. The abstract representations aggregate the test results and report back
5. The test results are analyzed and the build is promoted (or not)

This segues very well into changing the conversation about the automation lab, continuous integration, and the discrete instruments. Being able to remotely trigger events on an instrument in a customer facing configuration force developers to think about how a customer would perform actions all throughout the development cycle. This is especially poignant when some operation is expected that requires manual interaction with the physical instrument (i.e. plugging in a thumb drive or pressing the power button to turn on the instrument).

3.2 Abstracting Communication

We solved the problem of abstract communication with virtual machines (VM's). These machines were configured as lightweight headless VM's, whose sole purpose was to act as an intermediary between the physical instrument and the continuous integration system. We then acquired the ability to:

1. Abstract the VM's name in such a way that it appears to be a specific instrument when viewed in Jenkins (i.e. Oscilloscope-1234)
2. Utilize the VMs as a testing environment (The test VM executes applicable tests against a target device)
3. Use tools like Chef and Vagrant to manage a fleet of VMs; as opposed to having to configure many discrete instruments (this is especially ideal to ensure a consistent test environment)
4. Use Jenkins to manage workload and test coverage

Continuous integration systems already perform so many important tasks that it's very convenient to leverage that power against an abstracted test pool of discrete instruments. These abstractions can be configured to present the appearance that the actual device is connected to the build system as a node. They can also be used to convey information about the physical state of a device, which, in turn, can be used to facilitate test coverage over the various device configurations.

4 Dynamically Reconfiguring Hardware

Complex devices have many potential setups. It is not always possible to execute test iterations on all configurations. It should be, however, easy for a continuous integration system to accommodate devices that swap their physical state to some other setup.

Dynamically reconfiguring hardware means taking a device and altering its physical state into another valid configuration. The continuous integration system should tolerate these changes without manual intervention. The testing that the system performs should automatically adjust itself to appropriately test the new structure.

The ideal project, embedded or not, has a small set of finite testable configurations. Given a small enough set, it is reasonable to test all possible configurations. For embedded applications, the benefit is transparent in the cost of prototypes. Each device permutation adds additional cost to a project such that the budget required to test all permutations can quickly become unfeasible. Being able to dynamically reconfigure a smaller pool of resources helps to increase coverage while saving money and materials.

Given thousands of possible option permutations, it is prudent to test only a subset of configurations that represent typical customer usage. These likely customer permutations are known as high-value configurations. These are the configurations that have been identified as the common use cases or most likely applications.

Dynamic configurations are achieved in a two-factor approach. Since the tested item is a physical device, then it may be the case that it needs to be physically reconfigured. Once the device is in a new state then that change needs to be conveyed to the tests and automation. There are not many ways to automatically rewire a sufficiently complex device so, for today, this is largely a manual task. On the other hand, providing updated configuration information to the tests is much more receptive to programmatic methods.

The following is a brief example outlining reconfiguration steps:

1. Create a new test chain that executes the special test(s)
2. Physically re-configure instrument(s)
3. Update configuration settings (this could be a matter of a simple variable that conveys some physical state to the test frameworks)
4. Execute testing at desired interval/intensity

When the special test iteration has garnered enough information then the instrument is converted back into its default high-value state.

4.1 Use Case: Accommodating 'Edge Case' Issues

Even accepting the smaller pool of high-value test configurations it is still a desirable position to be able to reconfigure some, or all, of the lab into other permutations. This is especially the case during a test cycle that begins looking at special configurations or features (for example, characterizing a specific instrument configuration with intermittent failure(s)). These setups are known as edge cases. Even though they may be pointing out a practical and realistic application, it may be the case that they simply are not as likely to be used as other setups.

For example, it may be desirable to verify that certain features behave appropriately when used in stressed conditions. The likelihood of a customer achieving a similar state may be extremely low; therefore, the benefit of maintaining a test configuration that observes this interaction is not considered high-value.

Once the edge case or special test iteration is completed, it should be simple to revert the lab back into its default high-value state.

4.2 Use Case: Experimental Development

Developing a discrete product involves many disciplines including software, electrical and mechanical engineering, industrial design, and manufacturing. On top of software there is electrical, mechanical, industrial design, manufacturing, and many others. Each of these specialized groups approaches problems and tests their implementations in unique ways. Although the continuous integration system is tightly associated with software there are many instances when automation can bridge a gap and help another discipline exercise testing.

Take the example of powering the instrument. Software can have a role in the customer's experience of turning on and off the instrument, but, it's the electrical engineering team who are the key developers of the underlying behavior and design. The implementation can be tested through special configurations and tests that cycle power on the instruments at a high frequency, across many instruments, automatically – all through the comfort and familiar realm of the dynamically configurable lab.

5 Closing Remarks

Deploying continuous integration ideology and infrastructure into the world of embedded software has unique challenges. It is critical for continuous integration to be able to:

1. Make the minimum exceptions needed to adequately access and test the device

2. Communicate in an abstract way that honors minimum accessibility
3. Allow for physical reconfiguration of the discrete product with minimal disruption or refactoring

Those three concepts enable a project in embedded software to be developed using continuous integration workflows much like other non-embedded applications. To see success, there must be a combined effort from the multi-disciplinary teams it takes to produce an embedded application to practice continuous integration. Hardware simply adds an additional, though manageable, challenge that can be overcome by a dedicated and disciplined team.

6 References

Atlassian, Inc. (2017, June 17). *Bitbucket Server Documentation*. Retrieved from Confluence.atlassian.com: https://confluence.atlassian.com/bitbucket-server.documentation-776639749.html

Chef Software, Inc. (2017, June 17). *Learn Chef*. Retrieved from Chef Docs: https://docs.chef.io

Cloud Bees, Inc. (2017, June 17). *Jenkins*. Retrieved from Jenkins Documentation: https://jenkins.io/doc/

Duvall, P. M., Matyas, S., & Glover, A. (2007). *Continuous Integration: Improving Quality and Reducing Risk.* Pearson Education.

HashiCorp. (2017, June 17). *Vagrant by HashiCorp*. Retrieved from Documentation: https://www.vagrantup.com/docs/index.html

VMware. (2017, June 17). *VMware vSphere 6 Documentation*. Retrieved from VMware vSphere Documentation: https://www.vmware.com/support/pubs/vsphere-esxi-vcenter-server-6-pubs.html