eficode

GUIDE

Developing embedded software with DevOps

Contents

Introduction

This ebook explains how to improve the development processes for embedded software by adopting basic DevOps practices. You will learn the major challenges to successful adoption and how they can be overcome with tooling and methodology.

eficode

What are embedded systems?

What do we mean by embedded software systems and how do they relate to DevOps?

Defining our terminology

Is DevOps incompatible?

Software development in embedded systems is a unique process and there are good reasons for supposing that DevOps might not be the right approach. However, resistance to DevOps in the embedded world is often due to concerns about practice rather than theory. The benefits are well known, but adoption is slow because of perceived barriers around practicalities.

But this is a misconception. As this ebook will show, there are many aspects of DevOps that can be incorporated into embedded development without difficulty. For example, embedded developers should be able to control what they build and how they build it. They should have proper version control and automated builds. And they should be able to build quality into their embedded system much earlier by testing more effectively. DevOps provides workable solutions to all of these issues and this ebook will explore each of them in turn.

First, a few words on what is commonly understood by DevOps and embedded software.

What do we mean by "DevOps?"

DevOps is an approach to software that seeks to integrate the relationship between development (Dev) and IT operations (Ops). Its methods and practices evolved from the Agile movement in software that was kickstarted by publication of The Agile Manifesto in 2001. The term "DevOps" was coined by Patrick Debois in 2009 and popularized by the first DevOpsDays conference in Ghent later that year.

The aim of DevOps is to accelerate the software development life cycle and produce a continuous delivery of high quality software releases to the production environment.

What makes embedded software different?

Embedded software is used to control machines and devices, but there is more to it than domestic appliances. The modern world runs on embedded software: critical infrastructure, medical devices, nuclear energy, airplanes and cars are all controlled by it. The challenge faced for embedded systems is that the customer is often very difficult to deliver to.

The brief outline given here suggests friction and compatibility between DevOps and embedded software. How is it possible to make "early and continuous delivery of valuable software" to systems that are embedded?

eficode

Unique challenges with embedded software

Embedded software development poses a unique set of challenges for DevOps practitioners

Identifying the barriers to successful implementation

What is it about embedded software that makes its developers so reluctant to adopt DevOps? Let's consider some of the big challenges.

Testing

Automated testing is an essential feature of DevOps. The big challenge with testing for embedded software is the lack of access to production-like environments. How can continuous releases to production be made without rigorous QA? Is there a way to include the hardware in the testing loop?

Deployment

The embedded world is frequently concerned with custom deployments. This is a potential danger because, unlike application software, the live environment is the physical world. Updating a car while it's being driven, or a piece of medical equipment that's being used to keep someone alive carries clear and obvious risks. Even if the device is available for updating, is it connected? If so, how is it connected, and how can a release be securely deployed to it?

Safety

Testing embedded software is especially serious because it is used in systems that are safety critical. Stakeholders in embedded systems for aircraft, missile defence and healthcare, to take only a few examples, are understandably very apprehensive about the potential for any new release to result in failure.

Are embedded and DevOps incompatible?

Embedded software development is concerned with safety issues, custom deployments, and limited access to production-like environments. By contrast, DevOps is a method of releasing software that's agile, collaborative, and automated. This seeming incompatibility is the reason why DevOps has been frequently dismissed as a method for embedded. However, there are a lot of benefits to be had by beginning with the adoption of some basic practices.

Busting the myth of incompatibility

DevOps is based on a set of core principles, tools and methods that can be implemented in any development lifecycle, including those for embedded software. The first thing to consider is versioning and traceability

Adapting DevOps solutions for embedded projects

Source code versioning

Configuration management and versioning are prerequisites to automated builds, CI, and eventually DevOps. Versioning tells the developer everything about how the system works: what the dependencies are, what's been built, and what's been tested. This provides full traceability across the entire SDLC. Getting this right is an essential step to successfully implementing DevOps in an embedded pipeline.

The first and most obvious place to start is with source code versioning, where the industry standard is Git. There are many different approaches to selecting a version control strategy, but for DevOps there are good reasons for keeping it simple. Taking this approach means you can avoid long-lived branches, complex merge situations, and manual error-prone work from consuming your time.

Binary artifact versioning

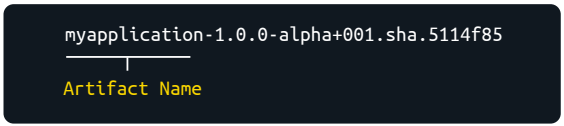
Looking beyond the source code, it is also wise to have a strategy for versioning binary artifacts. There are lots of different approaches to binary versioning, but Semantic Versioning is fairly standard across the industry. Here is an example:

First, there is the Artifact Name. In this case it is "myapplication", but it could be something like "controller" or whatever type of hardware is being working on.

The next three numbers document Major.Minor.Patch information. This tells developers everything they need to know about compatibility. A major bump tells them that backwards compatibility is broken and they should check that all their dependencies still work. A minor bump tells them there are new features, but backwards compatibility is still maintained. The patch simply records the number of bug fixes/ patches.

Then there is the Prerelease Version. In this case it's alpha, but it could be beta, RC1, etc.

Everything after the plus sign is the build metadata. It's advisable to make this as simple as possible. In this example the build number is (001) followed by the git short SHA (5114f85), so the developer can check out the source code for the binary any time it is required.



build.h

```
#ifndef _BUILD_H_
#define  _BUILD_H_

#define  BUILD_NUMBER 0
#define  BUILD_SHA 0000000
#define  BUILD_JOB “DEVELOPER”

#endif
```

version.h

```
#ifndef _VERSION_H_
#define  _VERSION_H_

#define  PRODUCT_NAME “demo_app”
#define  MAJOR_VERSION 1
#define  MAJOR_VERSON 0
#define  PATCH_VERSON 9
#define  PRE_RELEASE_VERSION “-SNAPSHOT”

#define  xstr(s) str(s)
#define  str (s) #s

#define VERSION_STRING xstr (MAJOR_VERSION) \
                        “ . “ xstr(MINOR_VERSION) \
                        “ . “ xstr(PATCH_VERSION) \ PRE_RELEASE_VERSION
#include “build.h”
#define BUILD_STRING VERSION_STRING \
                “+” xstr(BUILD_NUMBER)  \
                “.sha.” xstr(BUILD_SHA)

#define PRODUCT_STRING PRODUCT_NAME “-” BUILDING_STRING
```

Contrast this approach to versioning with something like myapplication-v10. This tells a developer nothing about the dependencies. Semantic Versioning is a simple, straightforward naming convention that provides all the basic information about the artifacts.

Next, create a second file called version.h to contain the rest of the semantic information and metadata. The binaries can now report their own build, version, how many patches have been applied, and so on.

Now that versioning is under control, let’s turn attention to the build.

Automate the build process

The first step is to take the build process out of the IDE and into a build script. Establishing the build process in living documentation means there will always be a single source of truth for how your software is built. Check the build script into version control along with the files and database needed to build it.

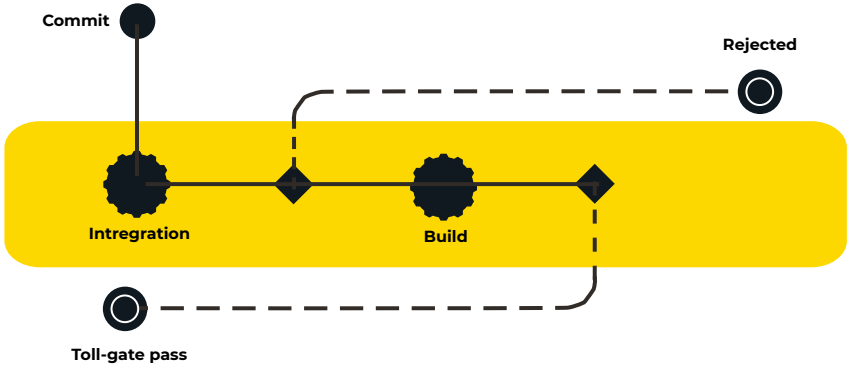
A build script is particularly important for embedded software because it gives us traceability and reproducibility. With this system, the environment for any build can be quickly reproduced with the same results.

Now that the build process is out of the IDE and into a build script it is possible to automate the builds on a neutral build server. A centralized build server retrieves everything from version control and builds on a machine free from developer dependencies. Now it works on everyone's laptop.

The build server can act as a gatekeeper for a certain level of quality in our central repository. If changes can pass through "integration" and "build" the code is good enough to share with the rest of the team.

Commit changes by merging them in Git and then perform a build – for embedded a "build' might mean something like compile, link, and run all the unit tests – and if the build is successful the code is ready to be shipped to the central repository. This is how to practice Continuous Integration.
But why stop there? If the commit is good enough for version control, it might

Continuous Integration

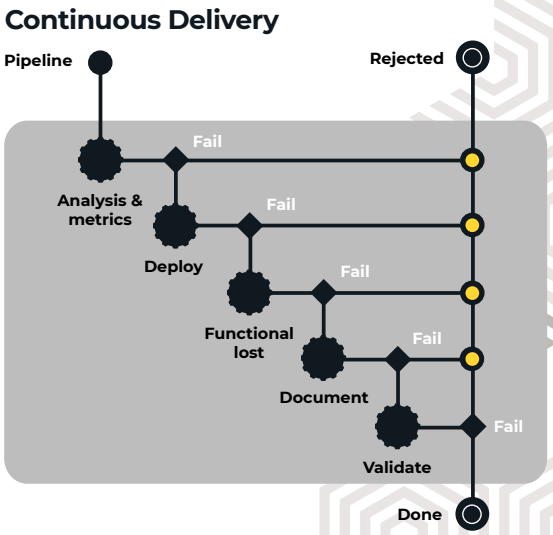


be of sufficient quality for production. There's no reason to hold back quality software, so run the commit through the Continuous Delivery pipeline to see if it's ready for the live environment.

For embedded systems the Continuous Delivery pipeline will contain elements like static analysis and functional tests. The code might also be deployed by flashing a device to see if it boots, or by introducing hardware to the loop. The key thing to remember is that the pipeline can be customized for any number of use cases and if the code fails at any of the predefined steps it means it isn't release quality. However, if it passes, we can ship it straight to production.

It's important to recognize the need for separate CI and CD pipelines because "good enough to share with the team" and "good enough for production" are different definitions of "done". Some qualification steps can take days to complete, so if the code has to pass through the entire Continuous Delivery pipeline just to get to version control it will slow the team down significantly.

Conversely, while developers should be able to push commits to the central repository as often as possible, stricter criteria should be in place for deploying to the production environment.

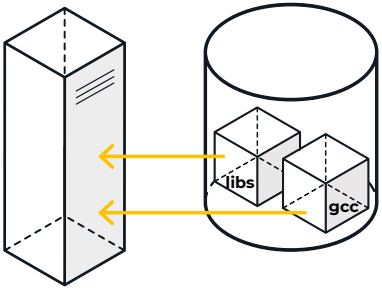


Manage the binaries

The next step is to take these new processes and tools for versioning, version control and automated builds, and put them into a system to maintain control over how we build. Storing all of the binaries, tools, and dependencies in an artifact management system means they can be easily deposited and shared with the rest of the team.

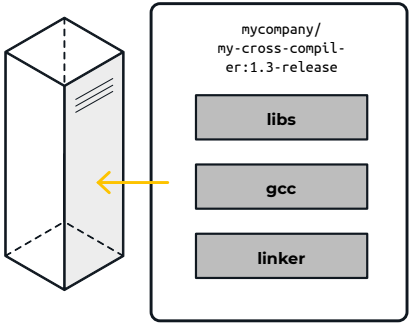
The artifact management system provides easy access to all of the build artifacts. This means that the CI server, test systems, and development environments can share exactly the same dependencies.

Next, why not capture the entire build and test environments in Docker images, defined as code, and then build automatically? This can form the basis of binary reproducible builds and shared development environments.



JFrog ARTIFACTORY

Sonatype Nexus



docker

```
#Dockerfile FROM
busybox:1.26.2-glibe
ADD tools.tar.gz
RUN make all
```

Use the same tools everywhere, indevelopment, CI, and release. Defined as code and version controlled.

Configuration as code

Tools like Git, Jenkins, Docker, and Artifactory can be used in combination to maintain a full traceability ecosystem where everything needed for a build can be defined as code and stored in version control. Once this in place new build environments can be built on demand. This is particularly useful for embedded because it frequently involves complex builds, so to be able to reproduce them quickly and easily is a huge benefit.

Automate the testing

The first step is to take the build process out of the IDE and into a build script. Establishing the build process in living documentation means there will always be a single source of truth for how your software is built. Check the build script into version control along with the files and database needed to build it.

A build script is particularly important for embedded software because it gives us traceability and reproducibility. With this system, the environment for any build can be quickly reproduced with the same results.

Now that the build process is out of the IDE and into a build script it is possible to automate the builds on a neutral build server. A centralized build server retrieves everything from version control and builds on a machine free from developer dependencies. Now it works on everyone's laptop.

The build server can act as a gatekeeper for a certain level of quality in our central repository. If changes can pass through "integration" and "build" the code is good enough to share with the rest of the team.

Test Driven Development (TDD)

Quality Assurance for embedded tends to be a phased approach that only happens after changes have been made to the copy of the codebase. But what if the tests could be written before development? Starting with the tests means it's possible to decide in advance what the code will need to do to meet the requirements. This approach is called Test Driven Development (TDD) and it specifies and validates what the code will do before the developers start writing it.

Ordinarily, code is written and then tested. Then, when the tests fail, the development team has to go back to refactor the code. The basic concept behind TDD is to write and correct failing tests before developing anything new. Developers only need to write a small amount of code each time to pass. The tests function as the requirement conditions for the new code, so as soon as it passes it's good to ship.

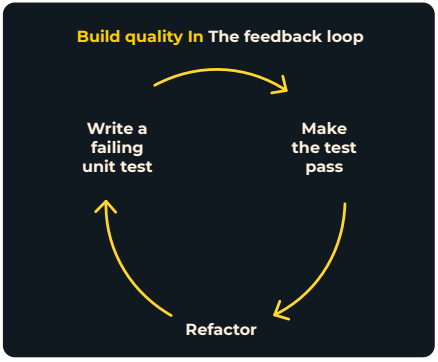
Introduce the hardware

Eventually, it will become necessary to bring hardware into the testing loop.

Ideally, the embedded system should be connected to the actual system, but this is rarely possible for reasons of safety/access/time/cost. In place of the real thing it is possible to introduce hardware-in-the-loop (HIL) as a substitute.

HIL provides something very close to a live environment by simulating the physical properties in the machine or system. It includes the complexity of the real hardware system under control in the testing loop by adding mathematical representations for all the relevant dynamical systems. HIL works by using electrical emulation of sensors and actuators as an interface between the plant simulation and the software under test.

With HIL it's possible to run all sorts of tests that would destroy the actual plant. That enables testers to know the limitations of the hardware in a controlled, safe environment. For this reason HIL is a great technique for building quality and safety into an embedded system. The simulation can be stress tested to limits way beyond what is physically required by the real plant. And being able to establish the safety and security of software earlier means it can be shipped much faster.



Prioritize your testing

Prioritizing relevant tests is one way to save time and keep pace with development. Embedded developers might not want to waste time with UI testing, even if there is one. There is always an end-to-end system though, and system tests should be right at the top of the testing pyramid.

System tests

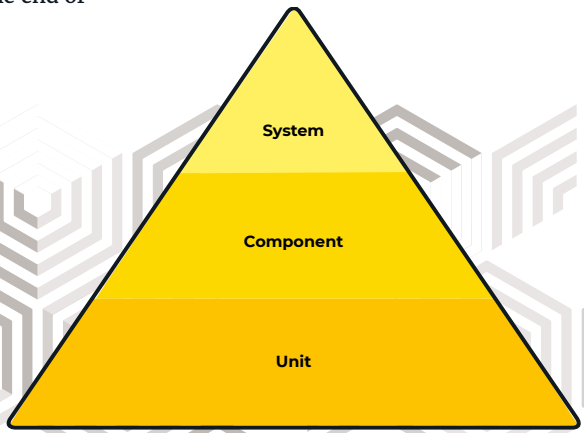
Feedback on high level tests is extremely valuable, so running full, end-to-end system tests on the hardware is a clear priority. However, this comes with challenges that are unique to embedded. Controlling the power source, programming the devices, creating input stimulus, and sensing the results are all obstacles to a successful system test. Some of these problems, like input stimulus, will be context specific and can be tackled with HIL. But there are also tools that can help with generic problems like power control and sensing results.

Component tests

There will probably be lots of components and they should all be tested individually, ideally in their own pipeline. But, let's say there is something like a big signal block in the system. Making sure that each filter and detector is working properly is essential, but why not create a whole test system around it to save time? Chain everything together and give it a real life input that can be detected at the end of the signal processing chain.

Unit tests

Ideally, every line of code and every branch outcome in our software should be unit tested. Why are unit tests needed if there's already a suite of successful system tests? If it works at macro level, why bother with the small stuff?



In Growing Object Oriented Software, Guided by Tests, Freeman and Pryce develop the idea of internal vs external quality. Internal quality is concerned with how quickly changes can be made, how safe it is to make them, and how ambitious developers can be without refactoring. Unit testing provides high internal quality.

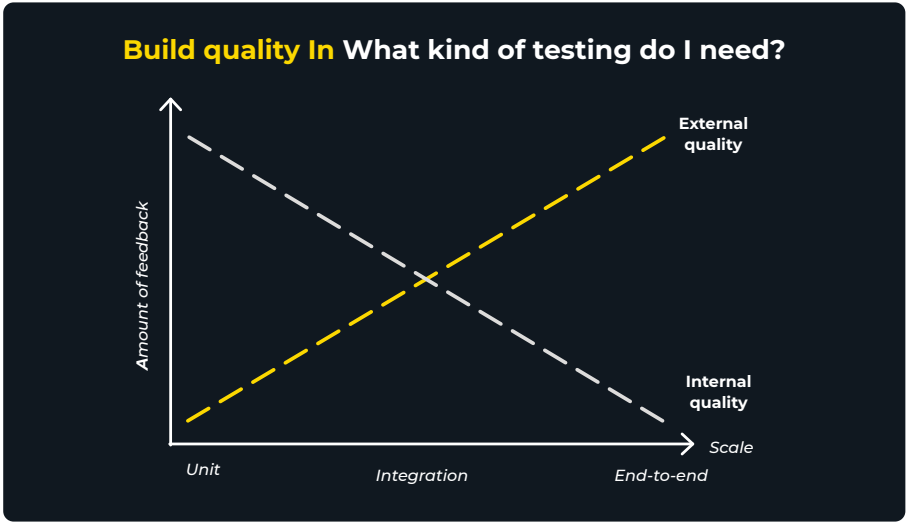
External quality is seen from the customer's point of view. Does the software do what it is supposed to do? Does it satisfy our end user? This is why performing unit tests and systems tests is essential. The code should be quick and easy to change, but it also has to meet the expectations of customers.

Use an analytics tool

Static Analysis is the best way to identify bugs and errors early in our SDLC. In embedded software this is important because once bugs are lodged in the system they are costly to mitigate.

Runtime errors are obviously an important consideration for embedded software, so it's important to run Dynamic Analysis too. This will alert the team to memory leaks, pointer arithmetic errors, and time dependencies as the software is running.

There are lots of analytics tools to choose from, but in embedded software it's wise to be conservative. Embedded systems tend to have long lifespans, so go with longstanding vendors and get the source code whenever possible.



Conclusion

There are good reasons for adopting DevOps as an effective approach to embedded. The key lies in mitigating risk through continuous improvement

Overcoming the obstacles one by one

Continuous improvement

Embedded software development will always pose big challenges due to the nature of production environments. Custom deploys, safety issues, regulatory concerns, and lack of access to adequate testing environments means that the pipelines used for developing application software cannot be simply repurposed for embedded.

At the same time, working in embedded does not provide justification for ad hoc versioning or building in the IDE. Those are aspects that can be improved upon without delay, and once they are under control it's not a big leap to running a successful CI server.

Testing will remain the biggest challenge in embedded, but there are solutions available there too. Test Driven Development enables developers to build quality and efficiency into the code from the first line. Adopting HIL means they can ship code in confidence, safe in the knowledge that it is safe. And by combining system and unit tests, high quality software can be achieved both internally and externally.

In conclusion, the essential principles of DevOps are perfectly compatible with embedded systems with the right tools and the right approach. By starting with version control and working towards automated testing, it's possible to continuously improve on an embedded software delivery pipeline without the need for large scale reorganization, or even risk.

To learn more about DevOps for embedded systems contact us at www.eficode.com

Developing embedded software with DevOps

Author

Michael Long is the Techninal Director at Eficode Praqma and has 10 years of experience in the oil and gas sector as a software engineer and project architect.

What is Eficode?

Eficode is driving the DevOps movement across seven countries with ideas that put customer value and team satisfaction on center stage. Eficode was doing DevOps before the term even existed by advising global brands on how to make software more effectively. Today, Eficode transforms companies with unmatched DevOps expertise and solutions like the Eficode ROOT DevOps Platform, a DevOps Toolchain as a service. Eficode's community of more than 300 professionals is building the future of software development together.

If you want to know more, contact us

embedded@eficode.com

Finland	+358 207 40 11 22
Denmark	+45 31 68 98 75
Norway	+47 48 67 63 60
Sweden	+46 76 340 30 50
Germany	+49 172 4 15 16 17
The Netherlands	+31 20 280 41 18

You can find all our contact information here:
www.eficode.com/about/find-us


eficode