Anttila Henri

Continuous Integration and System Test Automation

Case Exertus

Thesis
Spring 2018
Seinäjoki University of Applied Sciences (SeAMK): School of Technology
Information technology

SEINÄJOEN AMMATTIKORKEAKOULU
SEINÄJOKI UNIVERSITY OF APPLIED SCIENCES

SEINÄJOEN AMMATTIKORKEAKOULU

Opinnäytetyön tiivistelmä

Koulutusyksikkö: Tekniikan yksikkö

Tutkinto-ohjelma: Tietotekniikka

Suuntautumisvaihtoehto: Tietoliikennetekniikka

Tekijä: Henri Anttila

Työn nimi: Jatkuva integraatio ja testiautomaatio: case Exertus

Ohjaaja: Alpo Anttonen

Vuosi: 2018 Sivumäärä: 46 Liitteiden lukumäärä: 1

Tässä opinnäytetyössä toteutettiin järjestelmät ohjelmistokoodin jatkuvaa integrointia ja testauksen automatisointia varten Exertus Oy:lle. Työn tarkoituksena oli yhtenäistää ja automatisoida lähdekoodin kääntäminen ajettaviksi binääritiedostoiksi ja näin mahdollistaa testaamisen automatisoiminen. Työssä kuvataan Jenkins CI -käännösautomaatiopalvelimen käyttöönottoa sekä toteutetaan CAN-väylä pohjainen testausautomaatiojärjestelmä ja näiden välinen yhteistoiminta. Työn tuloksena Exertukselle pystytettiin Jenkins-pohjainen lähdekoodin käännösautomaatiojärjestelmä sekä testiautomaatiojärjestelmä, jolla voi ajaa Exertuksen omalla Guitu no code -kehitysympäristöllä toteutettuja testejä.

Avainsanat: automatisointi, jatkuva integrointi, DevOps, CI, järjestelmäintegraatio, testausautomaatio, ohjelmistotestaus, Jenkins, XStudio

SEINÄJOKI UNIVERSITY OF APPLIED SCIENCES

Thesis abstract

Faculty: School of Technology

Degree program: Information Technology

Specialization: Telecommunication technology

Author: Henri Anttila

Title of thesis: Continuous Integration and System Test Automation: Case Exertus

Supervisor: Alpo Anttonen

Year: 2018 Number of pages: 46 Number of appendices: 1

This thesis described the implementation of continuous integration and test automation systems for Exertus Oy. The purpose of the project described in the thesis was to standardize and automate the source code build process, which is a prerequisite for automated testing. To this end, the Jenkins build automation server was taken into use and a custom CAN bus based test automation system was developed. As a result, Exertus' daily source code is automatically built and tested each night on each of their hardware modules. The tests for the test automation system can be created in Exertus' own Guitu no-code development environment.

Keywords: automation, continuous integration, DevOps, CI, test automation, system integration, software testing, Jenkins, XStudio

TABLE OF CONTENTS

Terms and Abbreviations

Apache Tomcat	An open source Java servlet container. Also see: Java servlet.
API	Application Programming Interface
ash	Almquist Shell. A lightweight Unix shell and command language popular in embedded Linux systems. Also see: BusyBox.
bash	Bourne Again Shell. A Unix shell and command language widely used in Linux.
BusyBox	A software that provides a set of Unix tools in a single executable. Common in embedded systems with limited resources.
C	A popular general-purpose computer programming language. Imperative paradigm.
C++	A popular general-purpose computer programming language. Imperative, object-oriented and generic paradigms. Heavily influenced by C.
CAN	Controller Area Network. See chapter 2.7 for more details.
CANOpen	A CAN communication protocol by CiA. See CiA and chapter 2.8.
CD	Continuous delivery, practice of automatically pushing promoted release candidates to end users
CI	Continuous integration, practice of automatically turning new source code into an internally distributable software package.

CiA	CAN in Automation. International users' and manufacturers' group for the CAN network.
CLI	Command Line Interface. A text based user interface. Also see: GUI.
CPU	Central Processing Unit
DRY	Don't repeat yourself. A software development principle to avoid code repetition and data duplication.
DSL	Domain-specific language. A programming language specialized in a particular application domain.
FOSS	Free and open-source software
Framebuffer	A portion of RAM that contains the bitmap to be drawn on the display.
Git	A distributed revision control system. See chapter 2.1
GRT	Guitu Runtime. See chapter 1.5.1.
GUI	Graphical User Interface. What you see is what you get. Also see: CLI.
HTTP	Hypertext Transfer Protocol, a fundamental data communication protocol of the world wide web.
I/O	Input / Output
IaC	Infrastructure as code
Java	A general-purpose, object-oriented computer-programming language by Sun Microsystems / Oracle.
Java bytecode	The instruction set of the JVM. Also see: Java, JVM.
Java servlet	A Java program that extends the capabilities of a server. Purpose similar to PHP and Microsoft's ASP.NET.

JVM	Java Virtual Machine. A virtual computing machine that allows running Java bytecode. Also see: Java, Java bytecode.
Linux	Or GNU/Linux. FOSS operating system built around the Linux kernel. Also see: FOSS.
Make	A build automation tool for creating executable programs and libraries from the source code based on a Makefile. Also see: Makefile and chapter 2.5.
Makefile	A file that directs Make on how to compile and link a program. Also see: Make and chapter 2.5.
Map	An abstract data type. Also known as associative array, symbol table or dictionary. A collection of key-value pairs.
MVP	Minimum viable product. See chapter 1.6.
OD	Object Dictionary. A CANOpen data model. See chapter 2.8.1.
OOP	Object-oriented programming. A programming paradigm based on the concept of objects.
OS	Operating system. Software that manages hardware and software resources and provides services for computer programs.
OSI model	The Open Systems Interconnection model. A conceptual model for describing a communication or computing system or network.
PDO	Process Data Object. A CANOpen communication protocol. See chapter 2.8.2 on PDOs for further details.
SDO	Service Data Object. A CANOpen communication protocol. See chapter 2.8.3 on SDOs for further details.

QA	Quality assurance
SaaS	Software as a Service. A centrally hosted, subscription based software licensing and delivery model.
SCP	Secure Copy Protocol. An SSH based file transfer protocol. Also see: SSH.
SDK	Software development kit. A set of tools provided for creating additional functionality for an existing platform (software or hardware).
Shell	A user interface for access to an OS' services.
SQL	Standard Query Language. A DSL for managing data stored in a relational database. See: DSL.
SSH	Secure Shell. A cryptographic client-server network protocol for secure operation over an unsecure network.
Subversion	Also known as SVN. A revision control system akin to Git.
SUT	Service / Software / Subject / System Under Test. The target on which testing is being done.
UI	User interface. Also see: GUI, CLI.
Userland	Also user space. Code that runs outside an OS kernel, such as applications.
UX	User Experience
VM	Virtual Machine. An emulated computer system.
YAML	YAML Ain't Markup Language or Yet Another Markup Language. A human readable data serialization language akin to JSON and XML.

Images and Figures

1 INTRODUCTION

Quality assurance is an umbrella term for the practice of preventing mistakes and defects in manufactured products and provided services. The process of quality assurance can look very different depending on its target. In other words, one would not assure the quality of a 10-story skyscraper with the same methods one uses to assure the quality of a set of Lego bricks. Yet, the end goal of the process remains the same. This thesis concerns implementing automated quality assurance in the realm of embedded computer software and electronics design.

Even inside the realm of software testing the testing requirements are directly correlated with the purpose of the software. The on-board computer of the latest airplane model, where safety and reliability are of utmost importance, requires far more comprehensive testing than a web browser based game of chess does. The methods of and approaches to testing are numerous and testing can be done on multiple levels of software abstraction. Some of these methods include peer review of the produced source code, testing the units of the source code, such as functions and classes, the interfaces or the system as a whole. (Kautto 1996, 1.1.)

Quality assurance can be done by manually operating and manipulating the SUT in different ways, using different inputs and comparing the actual outcome with the expectations and making sure that they align. Manual testing is by its very nature time-consuming and prone to errors. If testing is carried out in this way, for example by the same software developer who wrote the software, there’s also a risk that the test cases may be chosen based on how the code is expected to work. In the worst-case scenario, if a formal testing protocol is not defined or properly followed, there is also a chance of parts or all the quality assurance process being neglected. (Kautto 1996, 1.2.)

It is also not enough to test just the new functionality, because in certain situations new functionality or changes made to an existing one can have unexpected and unintended side-effects and consequences in unexpected places, a phenomenon known as regression (Kautto 1996, 2.2). Thus, it is also necessary to make sure that the already verified functionality remains free of defects. The amount of testing that needs to be carried out increases with the new functionality. It naturally follows, that manual testing will therefore consume an ever-increasing amount of time.

To overcome the issue of time consumption, testing needs to happen automatically with every new piece of functionality that has been introduced. Automation enables the application of all the available tests on every change introduced and also effortless reapplication of the tests after the errors have been fixed. This makes sure that everything that is being tested still retains its original integrity and intended functionality. Being able to run the whole suite of tests on every change increases the reliability of the subject under test.

This increased reliability opens further automation opportunities. As the code is being tested rigorously in a short period of time and in an automatic, formally defined and codified fashion and on smaller batches of changes, new software versions can be taken into use more frequently with less need to worry about them being unreliable compared to the older ones. This further enables the catching of possible defects earlier through internal user experience thus reducing their spread and impact.

A process like this is promoted by the DevOps movement and referred to as Continuous Integration. It may not be plausible to do all types of testing on all the changes and some of the testing might be difficult to automate. A more rigorous testing process can be carried out less frequently, for example before a new public release.

The type of testing described in this thesis is a type of black box system testing, where tests are run against the underlying hardware and software during runtime without considering the underlying code. This testing is used to determine that the features being tested produce results that are both predictable and conform to the formal requirement documentation.

1.1 Outline of the Thesis

The previous chapter provided background information for this thesis. The next introductory chapters will familiarize the reader with the company Exertus Oy and some of its hardware and software products that are relevant for understanding the rest of this thesis. After that, the aim and scope of this paper is discussed providing the reader an overall view of the project that is described herein.

The thesis continues by describing the set of tools used to reach the goal of this paper. The reader will be introduced to Phabricator, Jenkins, Docker, Make and Makefiles, XStudio Test Management Suite, the CAN bus, the CANOpen specification and some of its functionality, Exertus' own Guitu and the Wall of Test, an in-house testing configuration running the Exertus Guitu control system applications.

After these chapters, the reader should have an elementary understanding of these concepts and be able to follow through the next chapters which discuss the actual implementation details. The implementation chapters first discuss the implementation of the build automation and, after that, proceed to test automation. Both chapters have an internal structure that goes through the business requirements, challenges that arise from them and how those requirements affect the implementation decisions. The last chapter provides a summary of the end results. It also discusses some of the challenges encountered along the way and finally what the future could hold for this project.

The thesis will not provide minute details on how all the parts of the system are installed, implemented or configured where such information is not relevant for the aim and scope of this paper or doesn't considerably deviate from the documentation and intended usage of the software in question. There are a lot of smaller tools and utilities used and not all of them can be described in detail within the scope of this thesis. Describing any such tools that are considered basic professional knowledge and as auxiliary or means-to-an-end is outside the scope of this thesis. However, a rather comprehensive Terms and Abbreviations section is provided. The reader is highly suggested to look up any new concepts there. If there is a more comprehensive explanation available elsewhere in the paper, the Terms and Abbreviations will have a reference to the relevant chapter.

During the course of this thesis any source code or data will be presented with the following formatting:

```
#!/bin/bash
NUM1=2
```

```
NUM2=1
if [ $NUM1 -eq $NUM2 ]; then
    echo "Equal"
else
    echo "NOT equal"
fi
```

Where hexadecimal numbers are used instead of decimal numbers, the prefix 0x is prepended to the number.

1.2 Company Introduction: Exertus Oy

Exertus Oy, from now on referred to as just Exertus, was founded in the year 2003 in Seinäjoki, Finland. Their core competence is in developing CAN bus based, distributed machine control systems. To achieve this, Exertus manufactures its own control system hardware products based on the CAN bus technology. They also develop software for programming and diagnosing these systems and provide a comprehensive solution development service based on these products. Solution development encompasses the system's whole life-cycle, including services such as system design, prototyping, testing, series production, customer support and service. Exertus aims to provide a full technology stack for building complete control systems, both small and large. (Exertus Oy, Yritys. Accessed on 23rd Jan 2018)

For marketing purposes, Exertus' hardware products are grouped into four different product categories: displays, controllers, sensors and other products such as CAN hubs. In addition to their own hardware, Exertus is also a retailer for the IKUSI brand remote controls. (Exertus Ltd, Products. [Accessed on 23rd Jan 2018]). Exertus also has a lineup of software products that are introduced in more detail below.

2 EXERTUS PRODUCTS

During the course of this thesis references will be made to Exertus products. While the project described in this thesis is meant to cover the testing of all Exertus hardware products, only two representative examples are required from a technological point of view.

This chapter will familiarize the reader with a subset of Exertus' hardware and software products. Instead of the marketing categories mentioned in the company introduction, this thesis uses a high-level technological division for the hardware products; bare metal products and MIC Platform products. The term bare metal product or system is used to refer to a system that runs without an intervening operating system and instead runs a monolithic, single-purpose software that interacts directly with the underlying hardware. A bare metal product used as an example in this paper is the HCM2010S.

MIC Platform system is used to refer to the Exertus line of products that run the Linux operating system. A MIC Platform product used for the purposes of this paper is the MIC2000S.This division is made because it affects the way in which these devices can be interacted with and the actual implementation details.

2.1 HCM2010S Hybrid Controller Module

The HCM2010S Hybrid Controller Module pictured in image 1 is a Guitu-programmable I/O controller with 60 configurable I/O channels and two CAN interfaces. The HCM2010S runs on an ARM Cortex M4 168MHz CPU. (Exertus Ltd, HCM2010S Tech Doc. [Accessed on 25th February 2018].)

Image 1. HCM2010S Hybrid Controller Module
(Exertus Oy, Products. [Accessed on 30th March 2018]).

2.2 MIC2000S Multi Information Controller

MIC2000S Multi Information Controller in image 2 is an embedded Linux based controller that combines a traditional I/O controller, display controller and data logger with USB, Ethernet, RS232 and CAN interfaces. It runs on an ARM Cortex A9 Dual Core 800 MHz main CPU and also includes an ARM Cortex M4 168MHz auxiliary CPU for dedicated handling of the I/O. The MIC2000S has a total of 40 Guitu-programmable I/Os and three CAN interfaces. (Exertus, MIC2000S Tech Doc. [Accessed on 25th February 2018].)

Image 2. MIC2000S Multi Information Controller
(Exertus Ltd, Products. [Accessed on 30th March 2018]).

2.3 Guitu Graphical Programming Environment

Guitu is Exertus' own graphical programming environment provided for Microsoft Windows operating systems. It is a multi-user tool used to create GUIs and I/O configurations for machine control systems. It also contains function block diagram based scripting tools and a debugger. Guitu creates the low-level CAN communications between the different controllers on the background without the programmer needing to understand the details of the CAN bus and the CANOpen protocol. (Exertus Ltd: Guitu Programming Environment). Guitu consists of two different parts: The Guitu software for programming the controllers and Guitu Runtime (GRT for short) that is used to run the control system configurations created with Guitu and to handle the CAN traffic and other lower level details of the CANOpen implementation. Guitu is programmed in C++. GRT is programmed in C.

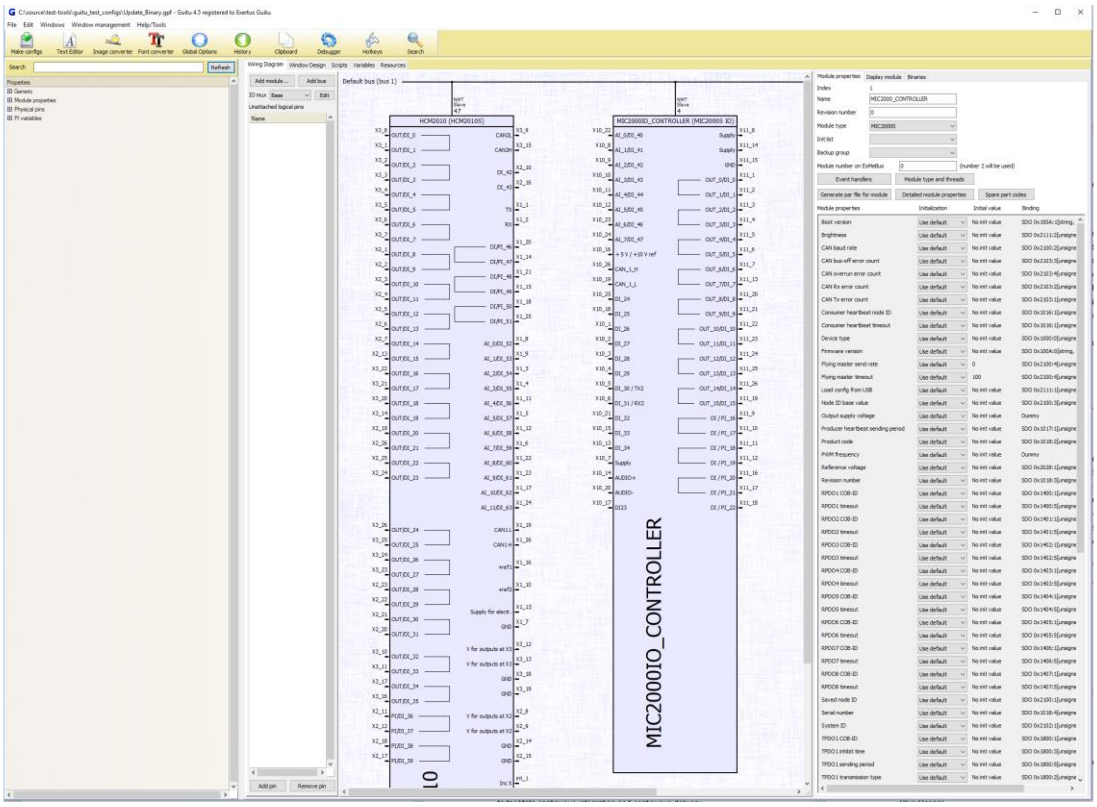
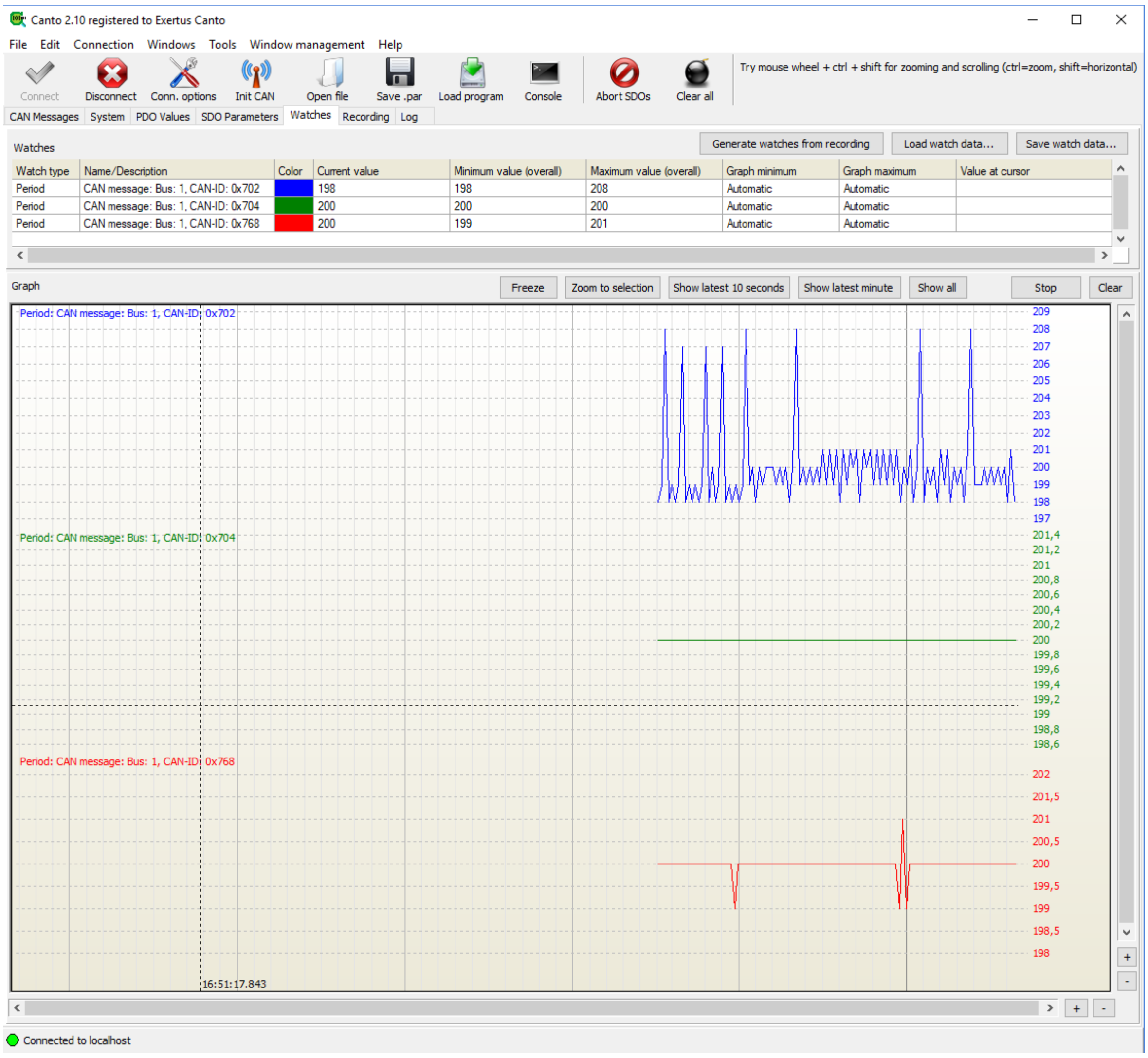


Image 3. Sample image of Guitu GUI

2.4 Canto2: CAN Monitoring and Commissioning Tool

Canto2 is Exertus' CAN bus monitoring and commissioning tool that conforms with the standards specified in CANOpen. It can be used to follow and record the traffic on the CAN bus, show basic information about the status of the different nodes on the bus and to set node parameters through the use of the CANOpen Service Data Object (SDO) protocol. (Exertus Ltd: Canto2 Monitoring and Commissioning Tool. [accessed on 25th Feb].)


Canto 2.10 registered to Exertus Canto
File Edit Connection Windows Tools Window management Help
Connect Disconnect Conn. options Init CAN Open file Save .par Load program Console Abort SDOs Clear all
CAN Messages System PDO Values SDO Parameters Watches Recording Log
Generate watches from recording
Load watch data...
Save watch data...
Graph
Freeze
Zoom to selection
Show latest 10 seconds
Show latest minute
Show all
Stop
Clear
Period: CAN message: Bus: 1, CAN-ID: 0x702
Period: CAN message: Bus: 1, CAN-ID: 0x704
Period: CAN message: Bus: 1, CAN-ID: 0x768
16:51:17.843

Try mouse wheel + ctrl + shift for zooming and scrolling (ctrl=zoom, shift=horizontal)

Watches

Watch type	Name/Description	Color	Current value	Minimum value (overall)	Maximum value (overall)	Graph minimum	Graph maximum	Value at cursor
Period	CAN message: Bus: 1, CAN-ID: 0x702	🟦	198	198	208	Automatic	Automatic	
Period	CAN message: Bus: 1, CAN-ID: 0x704	🟩	200	200	200	Automatic	Automatic	
Period	CAN message: Bus: 1, CAN-ID: 0x768	🟥	200	199	201	Automatic	Automatic	

Connected to localhost

Image 4. Sample image of Canto2 GUI
(Exertus Ltd, Products. [Accessed on 30th March 2018]).

2.5 ExGUI Graphics Abstraction Layer

Guitu Runtime uses an in-house graphics abstraction layer called ExGUI to draw its user interface. ExGUI handles the hardware details of drawing graphics, such as maintaining the frame buffer. It provides a software interface for GRT to draw its UI on the built-in or external display.

3 AIM AND SCOPE

The aim of the project described in this paper is to reinforce Exertus' existing testing architecture and processes by setting up a framework for a build and test automation system. Benefits that can be expected from such a system when correctly implemented are:

1. Single common storage for source code
2. A unified way and environment for building the software from source code.
3. A single source for the developers to get the build products, such as software libraries and executables that they need without having to rely on other developers' help or set up own build environments.
4. A unified process of testing the source code that is produced
5. The build and testing process is run automatically each night. The whole set of tests is applied to ensure that the software remains as defect free as possible and that there are no unexpected regressions.
6. Every binary product has gone through a set of these automated tests that verify the functionality and enhance reliability of the released binary products

The scope of the project described in this paper is to establish a minimum viable product (MVP) of the build and test automation system. A minimum viable product is a concept that emphasizes the concept of learning. Eric Ries, author of the popular entrepreneurship blog Startup Lessons Learned and the author of Lean Startup Methodology, who coined the term MVP defined it as "that version of a new product which allows a team to collect the maximum amount of validated learning about customers with the least effort" (Ries, 2008). Customers in this context refers to users of the CI system, mainly Exertus employees.

For the MVP it was decided that the following functionalities are required:

1. Specimen of bare metal monolithic firmware and MIC platform applications are built automatically, in a repeatable and reproducible manner every night.

2. The build products are made available to Exertus' employees via a web interface
3. The build products are automatically deployed to the test bench automatically every night
4. The test bench hardware modules are updated with the latest software in an automatic fashion
5. Multiple tests on Guitu scripting block functionality are carried out on multiple SUTs in an automatic fashion
6. Detailed results of these tests are stored in the test management suite
7. Overall success / failure status of these tests is stored on a per SUT basis in the build automation system
8. The tests need to be implemented in Guitu to allow the test engineers, who might not be familiar with programming with for example C, to create these tests

Having completed these steps, all the necessary basic functionalities have been built and capability of automating the compiling and testing has been fully demonstrated. Future development is then just a matter of collecting usage experience and adapting the design based on that experience and adding more and different types of tests and adding more SUTs to the Wall of Test.

The scope of this thesis lies more in building a framework for executing tests and logging their results than it does on the actual implementation of the tests and test cases. As the project described herein includes a great deal of intercommunication between multiple different systems, this thesis also describes implementation details in the domain of system integration.

4 TOOLS

The CI pipeline is comprised of four main parts: Phabricator, Jenkins, Docker, Make, XStudio and what is colloquially referred to as The Wall of Test. A brief introduction to the relevant concepts of the CAN bus and the CANOpen protocol is also provided.

4.1 Phabricator

Phabricator is a platform for software development collaboration created by Phacility. It comprises multiple applications that each serve a different purpose. Phabricator is available in two delivery models: as a SaaS by Phacility or hosted locally on an in-house server. (Phacility Inc: Phabricator. [Accessed on 27th Jan 2018]). Exertus has chosen to host Phabricator themselves. Phabricator is a FOSS written in PHP (Phabricator, GitHub, accessed on 1st Feb 2018]).

The following Phabricator applications are used in the current workflow:

1. Differential:	peer review tool for source code commits
2. Diffusion:	a git repository browser
3. Maniphest:	for tracking bugs, ideas and enhancement proposals
4. Phriction:	a wiki tool for hosting internal documentation

4.2 Jenkins

Jenkins is an automation server software that can be used to automate virtually anything that can be put into code. Its most prominent use has been in automation of repetitive software development tasks such as compiling source code to executable binaries, running automated test procedures, code analysis, software packaging and making new software versions available to company's internal users such as developers as well as end users. Jenkins is a common tool in a DevOps toolchain to facilitate continuous integration and continuous delivery.

Jenkins is a server based system that runs in a Java servlet container, such as Apache Tomcat (Jenkins.io, Installing Jenkins. Accessed on 1st Feb 2018). Historically, development of Jenkins started as Hudson by Kohsuke Kawaguchi at Sun Microsystems. After Oracle's acquisition of Sun Microsystems, development of Jenkins began as a source code fork of Hudson. Jenkins was made available as a FOSS under the MIT license. (Kawaguchi, 2014).

Jenkins uses the concept of a pipeline to define and codify a process that involves building the software in a reliable and repeatable manner and progressing the built software through stages of testing and deployment. The pipeline is written in a domain-specific language (DSL) into a text file known as a Jenkinsfile. It provides a means to codify the build, test and deployment process and commit it into a source code control system, such as Git. Jenkins also supports creating different types of pipeline jobs through its GUI. The GUI based jobs were decided against as administratively heavy and less transparent. By writing the pipeline as code, Exertus can start putting parts of its infrastructure and processes into code. (Jenkins.io, Pipeline: What is Jenkins Pipeline? [Accessed on 1st Feb 2018].)

The Pipeline DSL comes in two flavors: declarative and scripted. The former is a later development that aims to make writing Pipelines easier and to provide richer syntactical features over the latter. The scripted pipeline allows more flexibility. (Jenkins.io, Pipeline: Declarative versus Scripted Pipeline syntax. [Accessed on 1st Feb 2018].)

Jenkins also includes a concept of shared libraries. In effect, a shared library is a collection of procedures written in Apache Groovy that can be shared by multiple pipelines, avoiding redundancy (Jenkins.io, Extending with Shared Libraries. [Accessed on 1st Feb 2018]). Apache Groovy is an object-oriented programming language for the Java platform that can be compiled to bytecode executable by the JVM.

There iss also a new development in Jenkins known as Blue Ocean. Blue Ocean is an enhancement for Jenkins GUI and UX. In its current form, however, the traditional Jenkins GUI is more mature. The move to Blue Ocean can be done at a later date, when it has reached feature parity with the traditional GUI approach. (Jenkins.io, Blue Ocean [Accessed on 1st Feb 2018].)

Jenkins has a way of distributing its pipeline workloads between multiple nodes, or agents. This feature of Jenkins, however, is not utilized in this project. Every workload is run on the master node.

4.3 Docker and Containerization

Docker is a container platform developed by Docker, Inc. Containerization is a form of operating-system-level virtualization. Instead of creating a complete virtual machine with a separate guest OS, Docker shares the host machine kernel between the containers, but virtualizes the userland. Any state that is created inside a Docker container, is only available inside that container and doesn't affect the state of the host machine or the other containers. Docker is primarily developed for Linux, although a Windows version of Docker is also available. (Docker, Inc., Get Started, Part 1: Orientation and setup. [Accessed on 2nd April 2018].)

A Docker container is launched by running a Docker image. A Docker image is an executable package that includes everything needed to run an application. The code, libraries, environment variables and configuration files. A container is a runtime instance of an image that becomes a process that holds the current state. (Docker, Inc., Get Started, Part 1: Orientation and setup. [Accessed on 2nd April 2018].)

Containers are defined by so called Dockerfiles that are simple text files used to describe what goes into the environment inside a Docker container. (Docker, Inc., Get Started, Part 2: Containers. Accessed on 2nd April 2018. [Accessed on 2nd April 2018].)

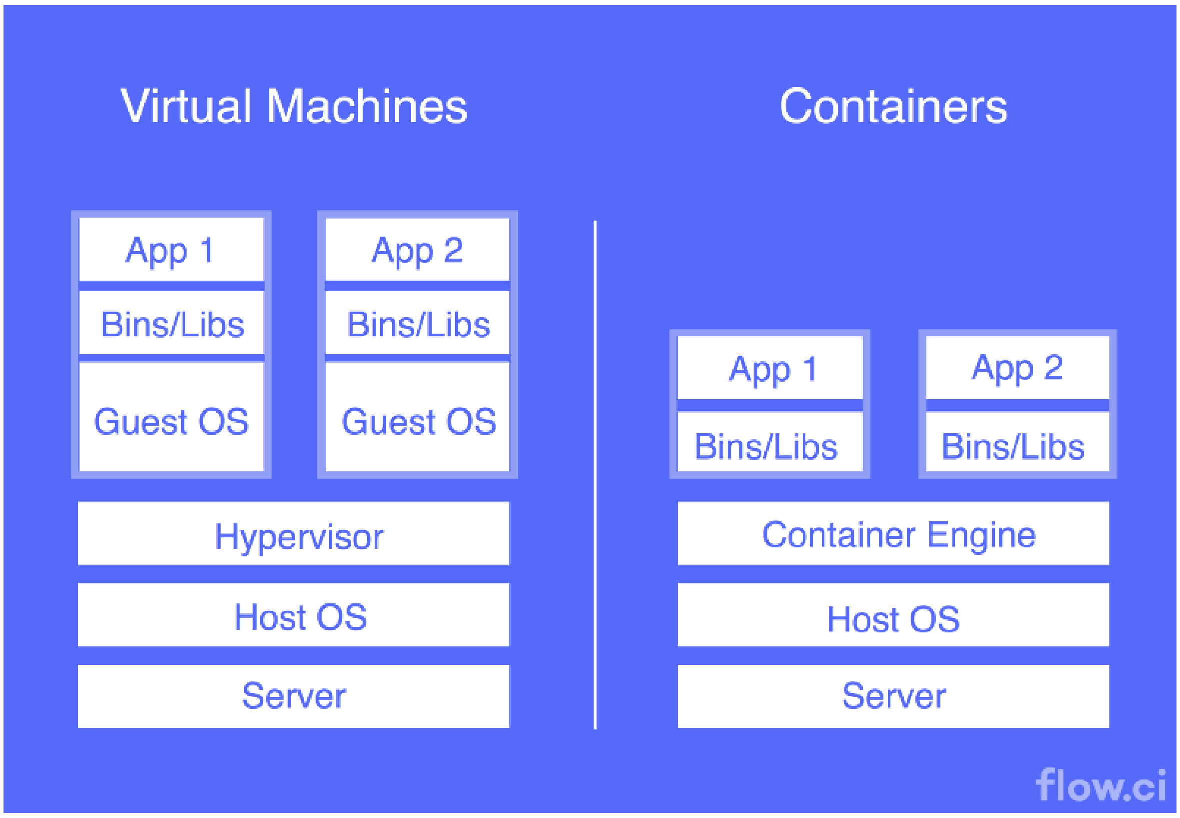
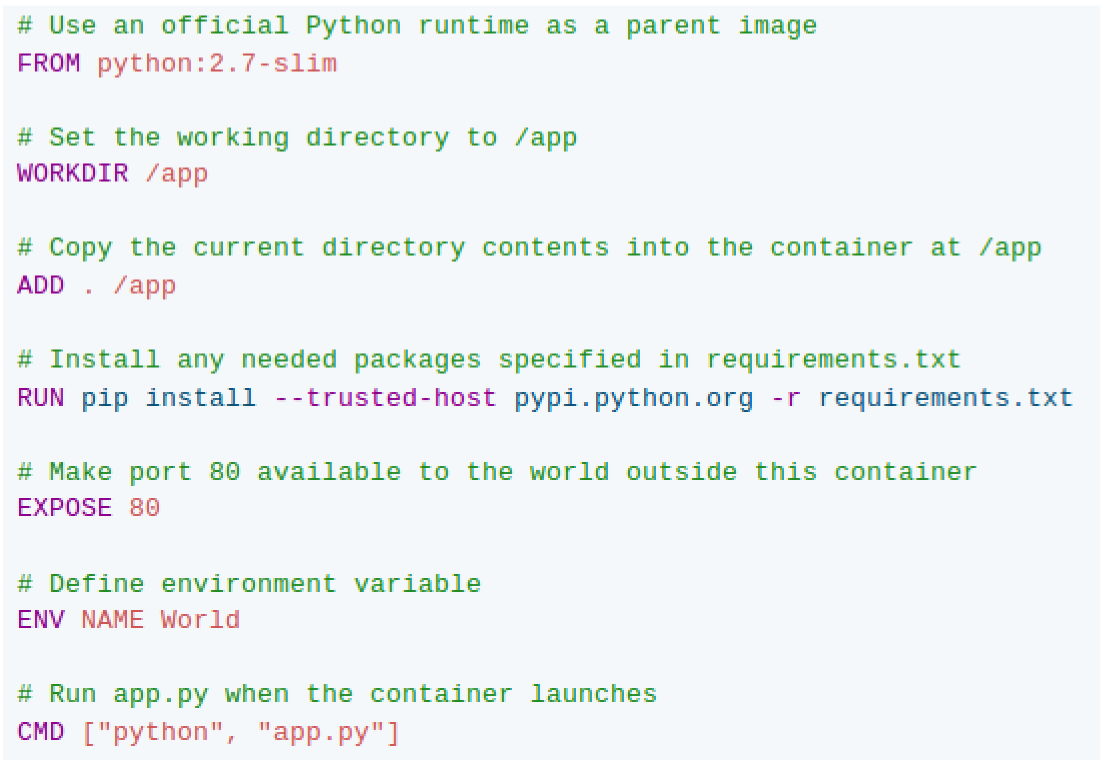


Figure 1. Docker container and VM comparison (medium.com / flow.ci: Introduction to Containers: Concept, Pros and Cons, Orchestration, Docker, and Other Alternatives. [Accessed on 2nd April 2018]).



```
# Use an official Python runtime as a parent image
FROM python:2.7-slim

# Set the working directory to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
ADD . /app

# Install any needed packages specified in requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Make port 80 available to the world outside this container
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

Image 5. A simple example Dockerfile describing a Python container (Docker, Inc., Get Started, Part 2: Containers. Accessed on 2nd April 2018).

flow.ci lists in its Medium.com blog post the following pros and cons for using Docker containers:

Pros of using Docker containers

1. Small file size compared to VMs
2. Containers need less resources to run than VMs
3. Starting new containers takes a very short time
4. Application and its dependencies become portable
5. Simpler deployment
6. Cost effective

Cons of using Docker containers

1. Security. Less isolation than in VMs. Compromised host OS kernel compromises all containers.
2. All containers on the same server must use the same underlying OS
3. Networking containers is tricky

(medium.com / flow.ci: Introduction to Containers: Concept, Pros and Cons, Orchestration, Docker, and Other Alternatives. [Accessed on 2nd April 2018].)

Basically, Docker reliably replicates state and moves it from the host machine and inside the containers, keeping the host machine cleaner and also easier to recreate by reducing the need so setup state and software dependencies.

4.4 GNU Make and Makefiles

GNU Make is a build automation tool that control the generation of executables from the program's source code files. A Makefile contains a list of rules that in turn contain a series of commands to execute to build a target file from source files. It also specifies a list of the target file's dependencies.

The Makefile takes the following pseudo-code shape:

```
target … : prerequisites …
        commands
```

...
...

(Free Software Foundation, Make manual. [Accessed on 2nd April 2018]).

4.5 XStudio Test Management Suite

XStudio is a test management suite created by the French company XQual (XQual, Contact us. [Accessed on 3rd March 2018]). Its structure is based on the following data model:

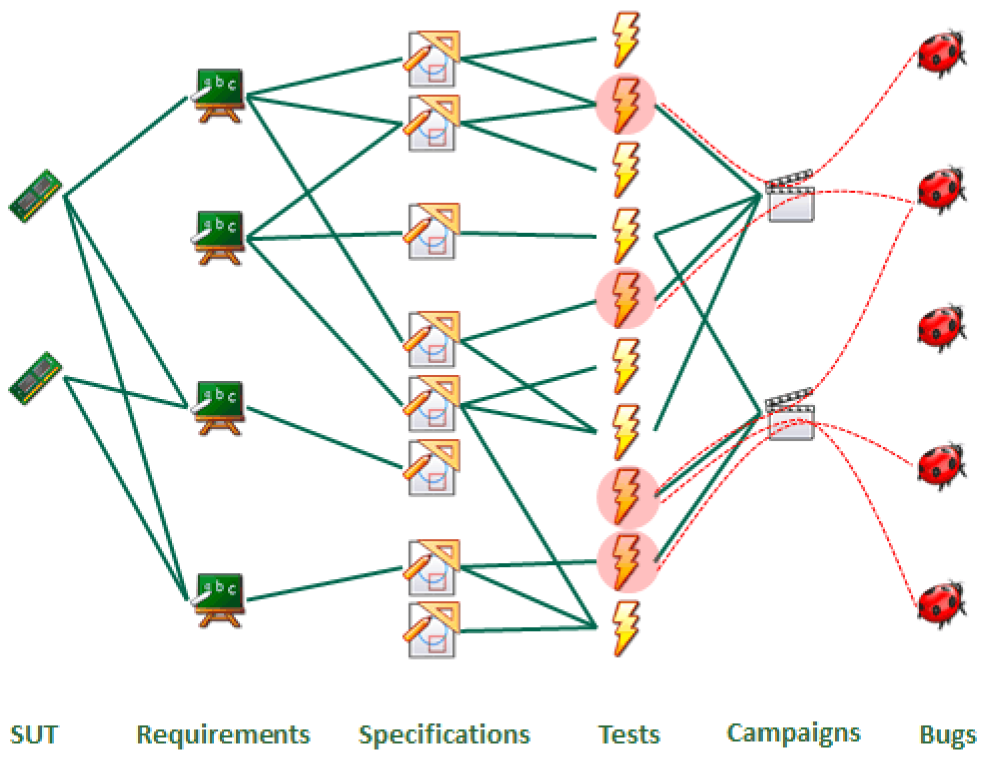


Figure 2. XQual XStudio test management suite data model
(XQual, User Guide. Introduction, Data model. [Accessed on 3rd March 2018]).

The aspects of the XStudio data model that are most relevant for this thesis are the SUTs, tests and test campaigns. Outside of the above diagram, XStudio also has the concept of test cases and campaign sessions, which are of interest to the reader.

An SUT is an abstract object that represents a target that is being tested (XQual, User Guide. The SUTs. [Accessed on 3rd March 2018]). Concretely, these are all the different Exertus products, such as the HCM2010S and the MIC2000S.

The SUTs have attached to them a set of business requirements. For example, a requirement could specify that “The module must implement the bitwise OR-operation for arbitrary signed integers” . (XQual, User Guide. The requirements). The requirements are then fleshed out by more detailed specifications, such as “The bitwise OR-operation must handle negative integers correctly” (XQual, User Guide. The specifications. [Accessed on 3rd March 2018].)

The specifications, in turn, are backed by a set of tests. A test is the smallest executable unit in XStudio. The tests describe the steps that are necessary to make sure that the specified functionality works as expected. The steps can be automated or manual. A test can implement multiple test cases that carry out the testing in different ways and with different parameters, such as implementing different edge cases of the test scenario. (XQual, User Guide. The tests. [Accessed on 3rd March 2018].)

The tests are then collected inside different test campaigns. An example test campaign could be “Verify Guitu scripting block functionality”. This campaign would then include the test described above and lots of other, similar tests. A campaign session is then generated to have XStudio run all the tests and their test cases that belong to the campaign. (XQual, User Guide. The campaigns. [Accessed on 3rd March 2018].)

4.5.1 XAgent

XAgent is a headless executable that can be left running in the background. It polls the XStudio database at specified intervals. If it detects a test campaign that it can run, it will run it and store the results in the XStudio database. (XQual, XAgent. [Accessed on 3rd March 2018].)

4.5.2 XContinuousIntegration, XCI

XContinuousIntegration, from now on referred to as XCI, is a command line utility that can be used to schedule test campaign sessions in XStudio. It provides an integration APIs that can be used together with automation systems such as Jenkins to build a CI pipeline. An XAgent will pick up the test campaign sessions generated by XCI and run them and provide information about the test results that can be fed back to the CI automation system. (XQual, Continuous Integration. [Accessed on 3rd March 2018].)

4.5.3 XStudio Launcher

XQual provides a collection of different launchers for running automated tests in different types of systems. The launchers are open-sourced and provided as an SDK, so that they can be modified as much as necessary. The launchers are built using Java. Their purpose is to run an external script that implements the actual testing functionality and returns information about the test results when it has finished running. (XQual, Open-Source launchers. [Accessed on 3rd March 2018].)

4.6 Controller Area Network (CAN)

A Controller Area Network, or CAN bus, was originally developed for use in passenger cars. It has since seen widespread used in different application domains, such as industrial machine control systems, home and building automation, mobile machines, medical devices as well as many other. (CiA, CAN lower- and higher-layer protocols. [Accessed on 1st April 2018].)

CAN is a multi-master serial bus system that relies on broadcast messages. This means that any node is allowed to access the bus at any time if it is idle. The messages will be received by each node on the bus. Access conflicts are resolved by a bit-wise arbitration of the CAN-ID. The bus work in two levels: recessive and dominant. The dominant level overwrites the recessive level. All nodes that are transmitting a recessive level, but detect a dominant level on the bus will lose bus arbitration and transit into listening mode. The cabling of the CAN bus is usually implemented with a twisted-pair copper cable with a common ground. (CiA, CAN data link layers in some detail. [Accessed on 1st April 2018].)

The CAN bus uses a voltage differential between CAN HIGH (CAN_H) and CAN LOW (CAN_L) conductors to physically encode the CAN messages. Using a voltage differential increases robustness and tolerance to electromagnetic interference. (CiA, CAN high-speed transmission. [Accessed on 1st April 2018].)

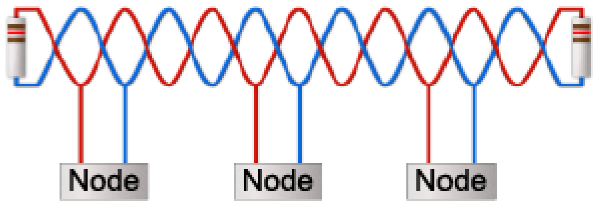


Figure 3. CAN bus cabling
(CiA, CAN data link layers in some detail. [Accessed on 1st April 2018]).

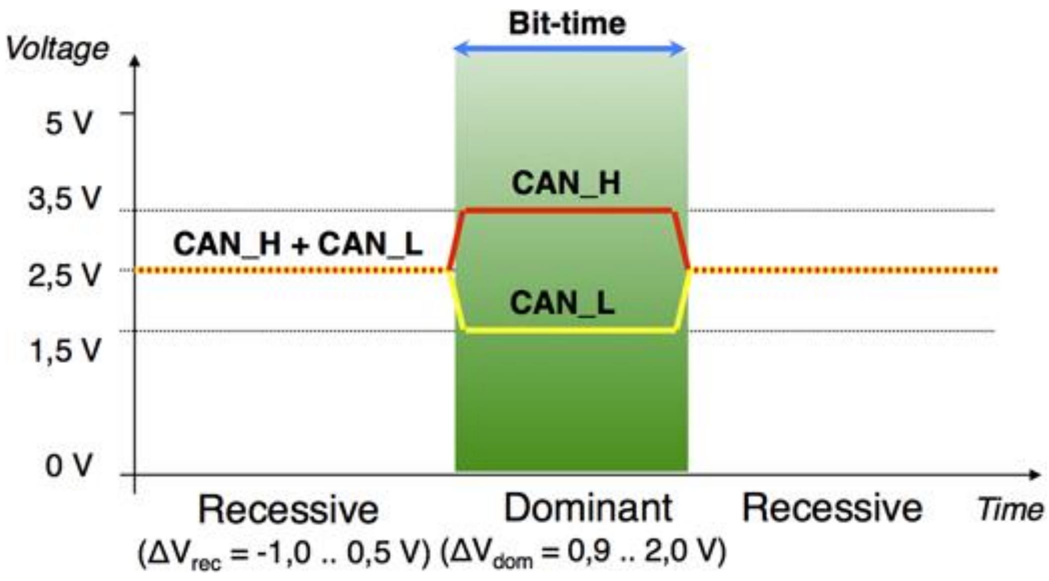


Figure 4. CAN bus encoding
(CiA, CAN high-speed transmission. [Accessed on 1st April 2018]).

4.7 CANopen

CANopen is a communication protocol and device profile specification that implements the OSI model layers from three to seven for CAN. A CANopen compliant device has three logical parts:

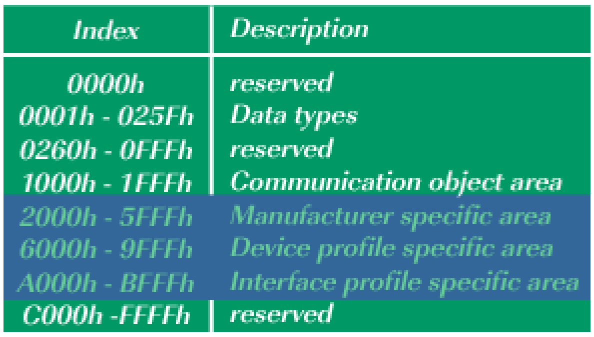
1. The CANopen protocol stack, which handles the communication via the CAN network.
2. The application software, which provides internal functionality and interface to the hardware.

3. The CANopen object dictionary, that provides an interface between the protocol and application software.

(CiA, CANOpen – The standardized embedded network [Accessed on 1st April 2018].)

4.7.1 CANopen Object Dictionary

Every CANopen node implements an object dictionary, abbreviated as OD. It serves as a data storage that can be read and written by other nodes on the bus. It stores the node’s configuration and state. The Object Dictionary consists of 24-bit addresses that are split into a 16-bit index and an 8-bit sub index. Any OD entry is readable by the CANopen communication services, such as the SDO protocol. (CiA, CANopen internal device architecture. [Accessed on 1st April 2018].)



Index	Description
0000h	reserved
0001h - 025Fh	Data types
0260h - 0FFFh	reserved
1000h - 1FFFh	Communication object area
2000h - 5FFFh	Manufacturer specific area
6000h - 9FFFh	Device profile specific area
A000h - BFFFh	Interface profile specific area
C000h -FFFFh	reserved

Figure 5. The CANopen object dictionary index ranges
(CiA, CANopen internal device architecture. [Accessed on 1st April 2018])

4.7.2 Process Data Object (PDO) Protocol

PDOs are used for broadcasting high priority control and status information. A PDO can be triggered by an event, it can be timer-driven, requested remotely or tied to synchronization messages. A PDO can communicate up to eight bytes of data. (CiA, Process data object (PDO). [Accessed on 1st April 2018].)

4.7.3 Service Data Object (SDO) Protocol

SDOs enable read/write access to the CANopen object dictionary. An SDO consists of two CAN data frames. SDO facilitates peer-to-peer client-server communication between two CANopen devices. The owner of the accessed object dictionary acts as the SDO server and the device that accesses the object dictionary is the SDO client. (CiA, Service data object (SDO). [Accessed on 1st April 2018].)

4.7.4 Network Management (NMT) Protocol

A CANopen device must implement support for the CANopen network management state machine. The NMT state machine consists of following states

1. Initialization
2. Pre-operational
3. Operational
4. Stopped

(CiA, Network management (NMT). [Accessed on 1st April 2018].)

The NMT system is controlled by one node that is chosen as the NMT master. The NMT commands are sent in a single CAN frame with a data length of two bytes. When a node receives an NMT frame containing its own node id, it must perform the commanded state transition. (CiA, Network management (NMT). [Accessed on 2nd April].)

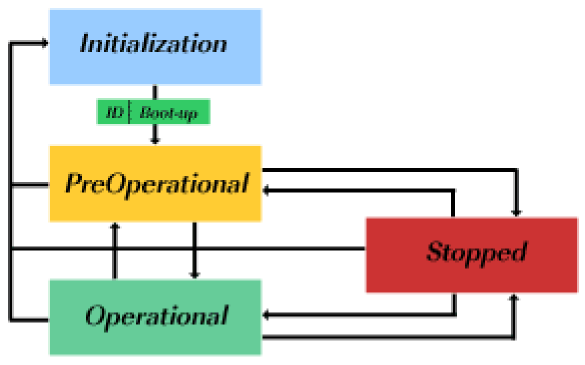


Figure 6. CANopen NMT state diagram
(CiA, Network management (NMT). [Accessed on 2nd April 2018]).

5 IMPLEMENTING THE BUILD AUTOMATION

To be able to automate testing of the latest developments in the source code in the first place, the building of the source code into executable binary needs to be automated. Exertus has already done some experiments with Jenkins, but a more comprehensive implementation is required.

5.1 Requirements for MIC Platform Guitu Runtime

All the MIC Platform devices use a version of the Linux operating system and execute Guitu Runtime as a userland application on top of it. Compiling GRT is dependent on a graphics library, which is a single file generated from the ExGUI’s source codes. GRT has a Makefile available, but the building process still depends on a set of different build flags provided by the user.

An older version of the Linux kernel (from now on “The older system” as opposed to “The new system”) used on one of the MIC Platform devices also needs to be considered. The older system and the new system are built using a different build toolchain. The source code for GRT is stored in a single Git repository which is used for all MIC Platform devices.

5.2 Requirements for MIC Platform ExGUI Library

ExGUI needs to be compiled to compile GRT. Similar to Guitu Runtime, it uses two different build toolchains for the older system and the new system. Both ExGUI and GRT need to be built with the same build toolchain to be compatible with each other. The source code for ExGUI is also stored in a single Git repository.

5.3 Requirements for Bare Metal Devices

Some of the bare metal devices have their source code stored in Subversion repositories. Some have had their source code moved over to Git. It was decided that any source code stored in Subversion repositories would be left out of the build automation scope only to be implemented when they have been moved over to Git. The bare metal devices are dependent on two different Git repositories: the hardware library, which implements the low-level hardware handling and GRT. Some bare metal devices have a built-in display and thus also need ExGUI.

These dependencies are gathered together into a Git super project using Git’s submodule functionality, which allows a Git repository to contain other repositories. These submodules can be frozen to a specific commit to allow controlled updates instead of always using the latest commits.

The hardware library needs to be built with the correct build toolchain and correct flags for make. If ExGUI is needed, it needs to be supplied with the hardware library files. After that, GRT can be built but it needs both the hardware library files and the ExGUI library file.

5.4 Additional Requirements

Exertus wants to also support the concept of different binary flavors. A binary flavor in this context means a binary executable created with a different set of build flags resulting in a different binary product. Candidate flavors could for example be a release version aimed for public release, an internal debug version with debugging symbols left intact or a version that excludes some of the more advanced GRT features in favor of backwards compatibility.

The resultant executable and binary files should also be packaged in a certain format to make them easy to use. It would also be useful for users to be able to use Jenkins to build from source code branches other than the master branch to be able to test more experimental features. Even better, if the custom branch functionality could also be extended to the downstream dependencies such as the ExGUI. Users should be able to specify an identifiable name for their builds, but a sane default should be available too.

Some of the build toolchains have already been containerized, but a few of them are installed locally on the Jenkins server. This will also need to be considered when planning the Jenkins pipeline.

When dependencies' source code is changed, every relevant firmware and application need to be rebuilt to ensure that no module ends up being broken until someone in the future tries to compile its firmware. The CI pipeline acts as a type of smoke test in that it ensures that each and every module's source code remains buildable and that the modules can run any tests implemented for them.

5.5 Challenges

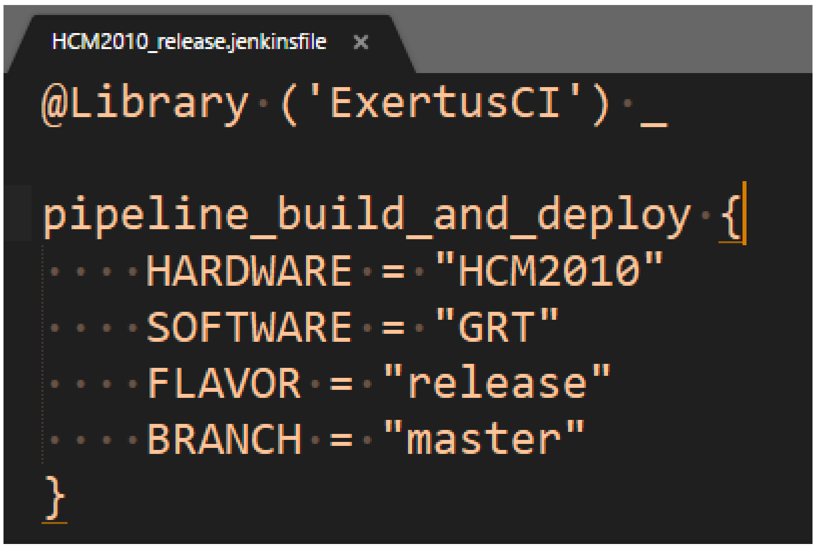
It is clear that a single declarative style pipeline per source code repository won't be enough to meet all these requirements, as its syntax and functionalities are quite restrictive by design (Jenkins.io, Documentation: Pipeline syntax. Accessed on 2nd April 2018). It won't allow for the conditionality that the requirements specified in the previous chapter set. The declarative pipeline can include blocks of scripted pipeline code, which are far more flexible, to the limits imposed only by the Groovy scripting language (Jenkins.io, Documentation: Pipeline syntax. Accessed on 2nd April 2018).

However, without a way to parameterize the pipeline through some sort of external input, the pipeline structure would have to be repeated in multiple Jenkinsfiles with different built in variable values for the different hardware. That would increase the administrative burden whenever a change must be made and opens up for introducing bugs and inconsistent behavior down the road. However, the declarative syntax is desirable, as it is the one recommended by Jenkins team and it also provides a better visualization of the CI pipeline in the Jenkins GUI. Jenkins has available a plug-in that provides parameterized builds, but the parameters for that plug-in would need to be inserted in a job specified inside the Jenkins GUI, which would further break away from the endeavor of putting parts of the infrastructure and processes into revision controlled files and add functionality duplication and administrative overhead.

5.6 Implementation

The common denominator for all the different requirement permutations is the stages and steps that are necessary to implement. What varies though is how each step is implemented for the various scenarios described above. Question then becomes how to both encapsulate the similarity of the overall Jenkins pipeline structure but facilitate different implementations details. Consult Appendix 1 for an overall structure of the pipeline and variations that need to be facilitated at each stage.

This was accomplished by reducing the actual Jenkinsfiles to maps containing parameters that describe the target of the build and include all the necessary information to facilitate runtime decisions on how to accomplish the pipeline steps.



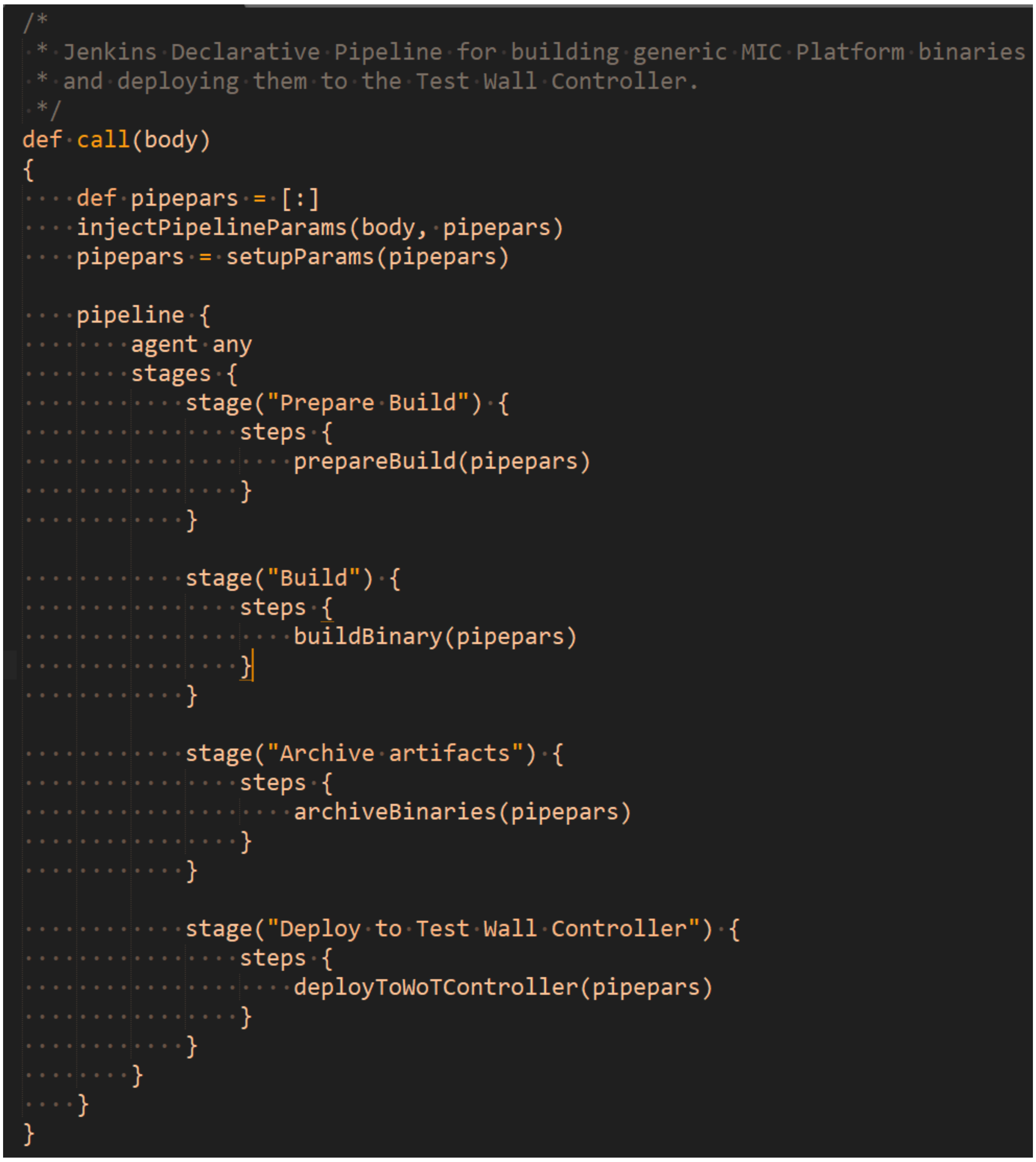
```
@Library ('ExertusCI') _

pipeline_build_and_deploy {
    HARDWARE = "HCM2010"
    SOFTWARE = "GRT"
    FLAVOR = "release"
    BRANCH = "master"
}
```

Figure 7. A Jenkinsfile containing a map of the different build parameters

The parameters are stored inside a custom Jenkins DSL block that retrieves the necessary pipeline structure from a Jenkins shared library file. The parameters that are brought in to the pipeline environment as input from the custom Jenkins DSL block are passed through a function that validates them and computes some additional parameters based on the ones defined in the Jenkinsfile.

After validation and computation, the extended set of parameters is injected into the pipeline's runtime environment. The pipeline can then use these parameters as input to Jenkins Shared Library functions.



```
/*
 * Jenkins Declarative Pipeline for building generic MIC Platform binaries
 * and deploying them to the Test Wall Controller.
 */
def call(body)
{
    def pipepars = [:]
    injectPipelineParams(body, pipepars)
    pipepars = setupParams(pipepars)

    pipeline {
        agent any
        stages {
            stage("Prepare Build") {
                steps {
                    prepareBuild(pipepars)
                }
            }

            stage("Build") {
                steps {
                    buildBinary(pipepars)
                }
            }

            stage("Archive artifacts") {
                steps {
                    archiveBinaries(pipepars)
                }
            }

            stage("Deploy to Test Wall Controller") {
                steps {
                    deployToWoTController(pipepars)
                }
            }
        }
    }
}
```

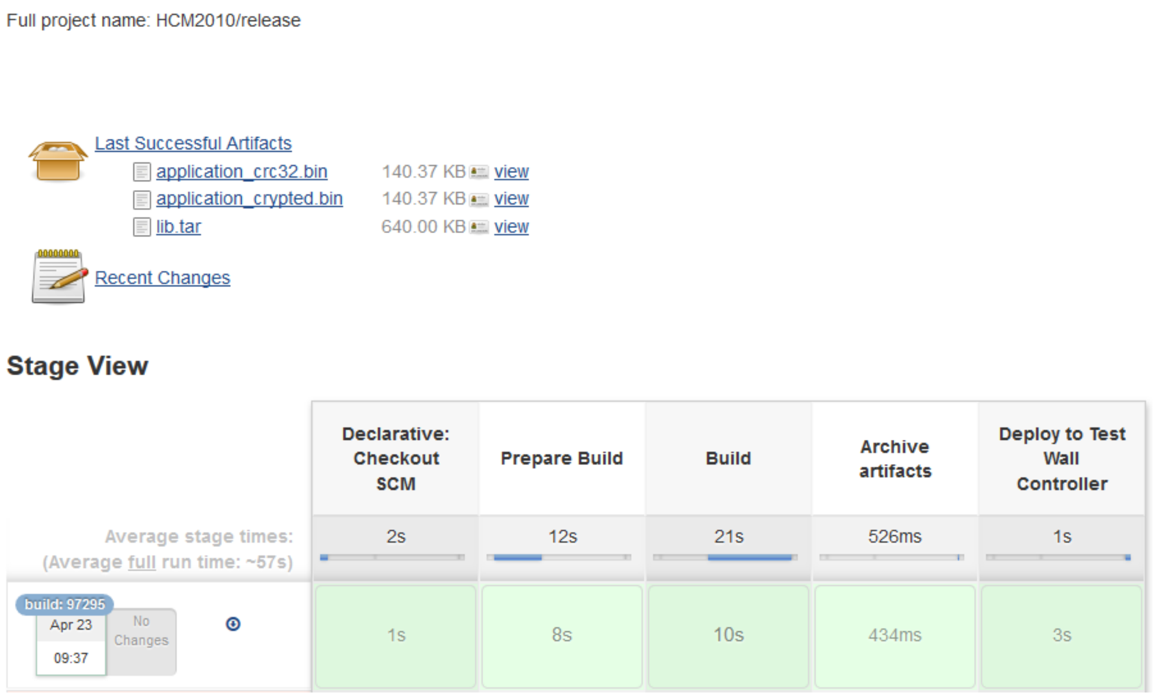
Figure 8. A Jenkins pipeline that uses 'pipepars', a map of pipeline parameters to control the different steps

The function can do its workload and branch the execution based on these parameters as necessary.

Shell scripts were created to include new build targets to reduce the need for the developers to remember different sets of build flags and to provide a simple interface for Jenkins to build different binaries. Now the Jenkins functions can just call these shell scripts to run the build process. The scripts are also available for anyone who checks out the source code repository.

Where Docker is used, Windows bat files and Linux shell scripts were added to the repositories to include suitable Docker run commands. These can be used by both Jenkins as well as anyone who checks out the source code and has Docker installed locally. The script files, whether for Windows or Linux, utilize Make and the Makefiles inside a Docker container containing the suitable build toolchain.

The resultant build artifacts are stored and made available in a predefined format for everyone through the Jenkins GUI. After this, they are deployed to WoT Controller, which acting as the bus NMT master will pick up the new firmware files and update the CAN bus nodes. After this, a suitable XContinuousIntegration call is issued and the test automation can take over.


Full project name: HCM2010/release

Last Successful Artifacts

application_crc32.bin	140.37 KB	view
application_crypted.bin	140.37 KB	view
lib.tar	640.00 KB	view

Recent Changes

Stage View

	Declarative: Checkout SCM	Prepare Build	Build	Archive artifacts	Deploy to Test Wall Controller
Average stage times: (Average full run time: ~57s)	2s	12s	21s	526ms	1s
build: 97295 Apr 23 09:37 No Changes	1s	8s	10s	434ms	3s

Figure 9. HCM2010 latest build artifacts in the Jenkins GUI

6 IMPLEMENTING THE TEST AUTOMATION

It was decided that the first type of testing that would be implemented as a proof of concept would be testing of Guitu scripting block functionality. Testing the scripting blocks already tests a lot more than the blocks themselves. It also executes a lot of the underlying GRT code to verify its functionality. Testing will need to happen every night on the latest versions of each firmware. Both Jenkins and the team of testers will need to be able to trigger any single test or a set of tests using XStudio.

6.1 Challenges

In the beginning, most of the challenges stemmed from the fact that the testing needs to be implemented in Guitu. There seemed to be a lot of logic that needs to be similar between the different SUTs and possibly different Guitu test configurations in the future. All of this foretold about a lot of code duplication resulting in a heavy administrative burden, especially if some of the duplicated code units needed to be changed. Another challenge was to figure out how to provide suitable test inputs and get the test result back from the test configuration during runtime and be able to compare that to what was expected from the test. In the end, both questions had a very straightforward answer, but figuring them out took a few iterations. The intermediate iterations will be discussed in the Retrospective chapter. Integrating Guitu scripts and GRT with XStudio also poses some questions that need answering.

6.2 Implementing the Wall of Test

The WoT starts logically inside XStudio. There, a test campaign session is run. This campaign session includes one or more tests which in turn include one or more test cases. XStudio schedules this test campaign session with an XAgent that is running on a Linux laptop colloquially known as The WoT Master.

The WoT Master is connected to a CAN network through a USB-CAN adapter and the SocketCAN software interface. SocketCAN uses Berkeley socket API, the Linux network stack and implements the CAN device drivers as standard Linux network interfaces (The Linux Kernel Archives. [Accessed on 22nd April 2018]). The laptop is also setup with python-canopen, a FOSS Python library that implements the CAN-Open protocol.

The XAgent on The WoT Master picks up this campaign session and runs XStudio Python launcher passing it a set of parameters that are used to control the actual testing. For this, the launcher was customized a bit to better suit Exertus' needs. The launcher expects to find a Python script located in the following folder:

```
<test_root>/<test_name>/<test_name>.py
```

This is a challenge, since the current WoT implementation only uses one generic Python script instead of many specific ones. The challenge was overcome by modifying the Java code of the Python launcher so that when it is run, it creates this directory structure, but with a symbolic link pointing at the actual script.

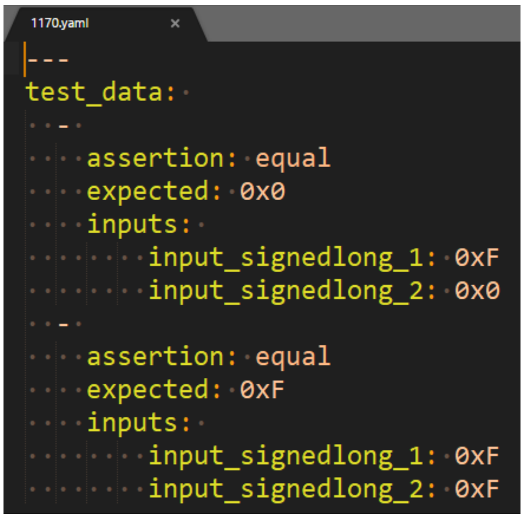
```
<test_root>/<test_name>/<test_name>.py -> <test_root>/WoT.py
```

It would have been simpler just to hard code the actual launcher script, but because of the way XStudio expects to be notified of finished tests, this would've blocked any chance of running tests in parallel in the future.

The XStudio Python launcher is modified so that it launches WoT.py with the following set of parameters:

- --test_id: ID number of the test in XStudio
- --sut_name: Name of the SUT in XStudio
- --testcaseindex: Index of the test case in XStudio
- --log: Name of the test result log file to be created in format <test name>_<test case index>.txt

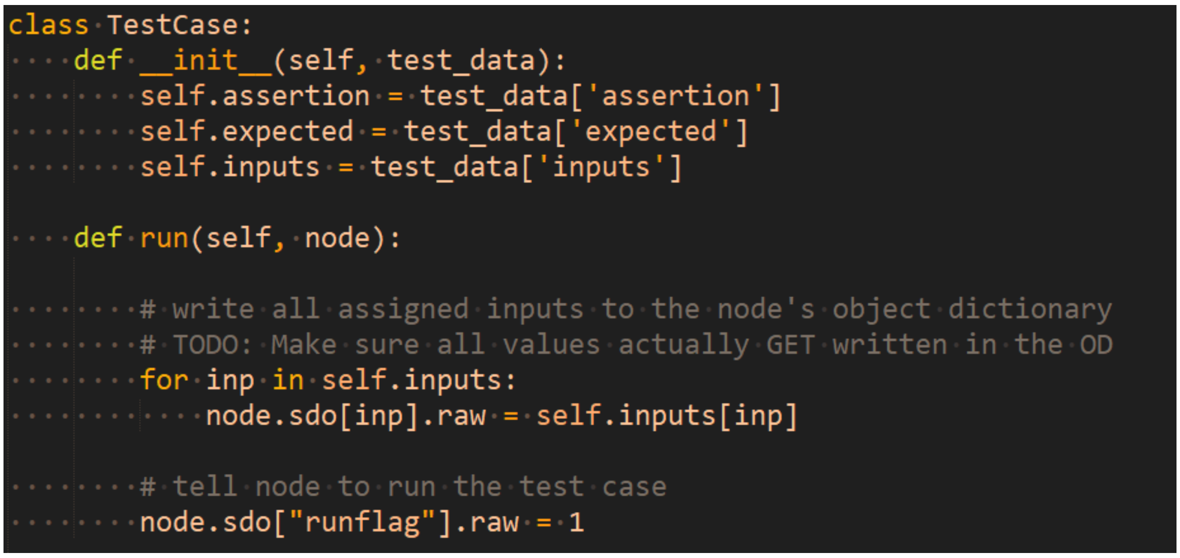
WoT.py then parses these command line arguments to control the rest of the testing. It reads a YAML file whose name corresponds with --test_id. The file contains a test configuration, which for now is a map of test inputs that should be used for the test case indicated by --testcaseindex together with an expected output of the test case with the given inputs.


1170.yaml

```yaml
---
test_data:
  -
    assertion: equal
    expected: 0x0
    inputs:
        input_signedlong_1: 0xF
        input_signedlong_2: 0x0
  -
    assertion: equal
    expected: 0xF
    inputs:
        input_signedlong_1: 0xF
        input_signedlong_2: 0xF
```

Figure 10. A YAML file containing two inputs for test id 1170

--test_id and the test inputs read from the YAML test configuration are then written into correct SUT's object dictionary through the SDO protocol. When all the necessary information is written in to the OD, one last entry called runflag is set to a certain value to let the SUT know that it can now start running a test.



```python
class TestCase:
    def __init__(self, test_data):
        self.assertion = test_data['assertion']
        self.expected = test_data['expected']
        self.inputs = test_data['inputs']

    def run(self, node):

        # write all assigned inputs to the node's object dictionary
        # TODO: Make sure all values actually GET written in the OD
        for inp in self.inputs:
            node.sdo[inp].raw = self.inputs[inp]

        # tell node to run the test case
        node.sdo["runflag"].raw = 1
```

Figure 11. Parts of class TestCase that show writing the CANOpen object dictionary and running the test case

A Guitu script that is run periodically on the SUT then picks up this runflag signal, inspects the test-id and runs a system-global test selector script that points it to the script that implements the actual test. This test case is then run with the values written in to the OD used as inputs to the Guitu script that implements the test case. When the test is finished, the result is written in to the OD and runflag is set to a value that signals the controlling Python script that the test was finished successfully. The Python script polls the runflag value. When it detects a finished test, it reads the test result from the OD of the current SUT and outputs it into a log file in a format expected by XStudio. It then signals XStudio that the test is completed by writing an empty file with the name test_completed.txt.

XStudio launcher picks up the creation of this file, parses the log file and writes the test result in to the XStudio database. If there are remaining test cases in the test, they are run similarly. When all the test cases have been completed, the test is ready and the next one can be run until all the tests in the campaign session have been finished.

A test operator can just build a test campaign in XStudio, create the test case configuration file with YAML and implement the test scripts in Guitu. After this these can be run on demand from the XStudio or by Jenkins via XContinuousIntegration. A minimal amount of utility scripting needs to be done for each SUT in Guitu.

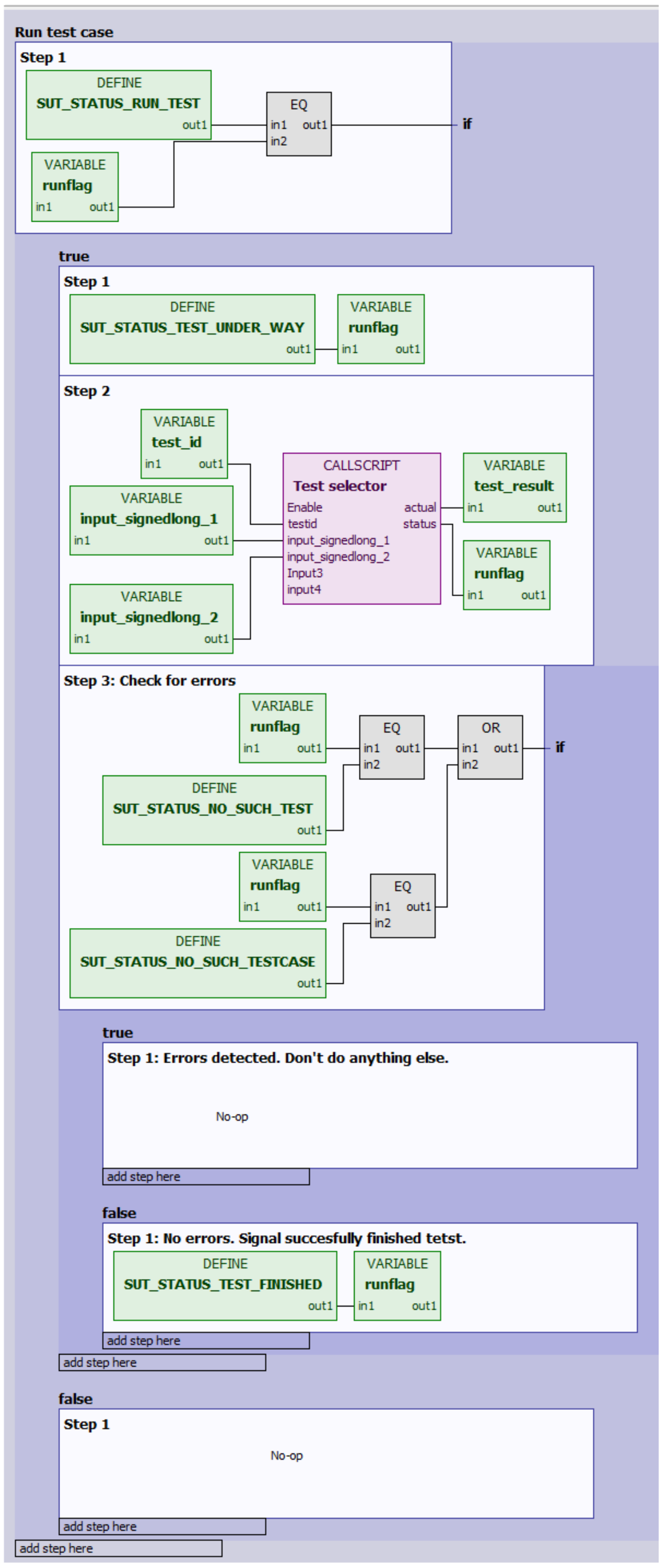


Figure 12. The full utility script that is required for each SUT in Guitu.

7 CONCLUSIONS

The main goal of this thesis was accomplished. Exertus made a major stride towards CI by implementing nightly build and building facilities for automated building and testing. Further developments are now a matter of implementing more and different kinds of tests and bringing in more hardware modules under CI and testing. Exertus can now start collecting experience and feedback to guide future development of the system and make informed decisions on future development based on this experience. With the power of Python, CAN bus, the CANOpen protocol and all the functionality provided by the GRT the system is very robust and extensible. The testers can implement the test scripts in the familiar Guitu environment and specify all the test inputs for the test scripts in a user-friendly YAML configuration file that can be version-controlled. The author strongly believes that it is a great compliment to Exertus' pre-existing testing regimen given some time to mature.

The final implementation of the test automation system is surprisingly simple, but the path to that solution went through a couple of less successful iterations, that were quickly deemed sub-optimal in varying aspects. The first version of the system was based entirely on Guitu configurations, which relied on text file based inputs to parameterize the testing. Similarly, all the log output was done via Guitu scripting. This does not scale very well and introduces a lot of copy and paste scripts that build an administrative burden for the future, especially if something needs to be changed. Some of this logic would also have to be copied to other Guitu configurations to implement different kinds of testing. Another downside to this approach was the fact that one extra MIC module would have to be dedicated as a controller module for the SUTs, introducing redundant hardware to the Wall of Test.

The next implementation relied on Linux shell scripts, Exertus CLI CAN client Execan and the CANOpen PDO protocol. The goal was to put more logic into version controllable, reusable code and less inside the Guitu configuration. All the code was still run on a MIC module. This was a step into the right direction, but due to some of the properties of Execan this was not ideal either. This could have probably been overcome by modifying Execan and to some extent by using SDO instead of PDO.

The implementation after this relied on Python and the CAN bus, but was still run on a MIC module. This wouldn't have been a problem otherwise, but an embedded hardware module such as the MIC is quite inflexible when it comes to installing new software, making it more difficult to leverage for example Python's external libraries. This implementation used some custom code to do basic SDO input and output through a Linux character device node representing the CAN bus. However, it was left unfinished when the final epiphany of moving all the Python code to the WoT Master struck resulting in more flexible development environment and the final implementation with python-can and python-canopen together with YAML test configuration files. This results in a minimal amount of logic stored inside the Guitu configuration.

Most of these sub- optimal implementations were result of the authors' unfamiliarity with the CAN bus and the ways that CANOpen allows interacting with the modules. The initial requirements also gave the author an impression that as much as possible of the testing should be done in Guitu, which resulted in the first implementation but also in the realization that it would not scale well to larger amounts and several types of tests. From that point onward, after having cleared that it would be acceptable to implement some logic in written source code instead of Guitu configurations, the progress towards the final implementation was relatively quick.

Implementing the build automation in Jenkins was not entirely straightforward either, but with the help of Jenkins Shared Library it was possible to create a single pipeline script that had enough intelligence built into it so that it can run the different variations of the build, deploy and test process stages. Everything is built and tested nightly to provide an overview of all the different hardware products and their software and their status after the past day's development work. Executables and programming libraries were made available through an easy to use GUI in Jenkins and employees can retrieve them without having to rely on the developers to get what they need, freeing up time and decreasing the amount of distractions. All the build procedures have now also been codified and the toolchains required to build the software are available for Jenkins reducing reliance on individual computers and persons.

7.1 Future

When the number and variety of implemented tests grow, it might be necessary to look into running some of the tests in parallel instead of the current sequential logic. Some corner stones to enable this were already laid in the design process. Tying together Phabricator and Jenkins to create more automation into the continuous integration environment, where every introduced change is automatically built and tested in real time is also a possible path worth considering.

BIBLIOGRAPHY

Burnstein, Ilene 2003. Practical software testing: a process-oriented approach. New York: Springer Verl

CAN in Automation (CiA). No date. CAN data link layers in some detail [web page]. [Accessed on 1st April 2018] Available at: https://www.can-cia.org/can-knowledge

CAN in Automation (CiA). CAN high-speed transmission [web page]. [Accessed on 1st April 2018]. Available at: https://www.can-cia.org/can-knowledge/can/high-speed-transmission/

CAN in Automation (CiA). No date. CAN lower- and higher-layer protocols [web page]. [Accessed on 1st April 2018]. Available at: https://www.can-cia.org/can-knowledge/can/can-data-link-layers/

CAN in Automation (CiA). No date. CANopen: CANopen internal device architecture [web page]. [Accessed on 1st April 2018]. Available at: https://www.can-cia.org/can-knowledge/canopen/device-architecture/

CAN in Automation (CiA). No date. CANopen: The standardized embedded network [web page]. [Accessed on 1st April 2018]. Available at: https://www.can-cia.org/canopen/

CAN in Automation (CiA). No date. Network management (NMT) [web page]. [Accessed on 1st April 2018]. Available at: https://www.can-cia.org/can-knowledge/canopen/network-management

CAN in Automation (CiA). No date. Process data object (PDO) [web page]. [Accessed on 1st April 2018]. Available at: https://www.can-cia.org/can-knowledge/canopen/pdo-protocol/

CAN in Automation (CiA). No date. Service data object (SDO) [web page]. [Accessed on 1st April 2018]. Available at: https://www.can-cia.org/can-knowledge/canopen/sdo-protocol/

Docker, Inc. No date. Get Started, Part 2: Containers [web page]: Available at: https://docs.docker.com/get-started/part2

Docker, Inc. No date. Get Started, Part 1: Orientation and setup [web page]: Available at: https://docs.docker.com/get-started/

Exertus Oy. No date. Yritys [web page]. [accessed on 23rd Jan 2018]. exertus.fi > Yritys. Available at: http://exertus.fi/?lang=fi&page=Yritys

Exertus Ltd. No date. About Us [web page]. [accessed on 23rd Jan 2018]. http://www.exertus.fi/?lang=en > About us. Available at: http://exertus.fi/?lang=en&page=About%20us.

Exertus Ltd. No date. Canto2 Monitoring and Commissioning Tool [PDF Document]. [accessed on 25th Feb]. Available at: http://www.exertus.fi/files/Tiedostot/Canto2_Commissioning_Tool.pdf

Exertus Ltd. No date. Guitu Programming Environment [PDF Document]. [accessed on 25th Feb]. Available at: http://www.exertus.fi/files/Tiedostot/GUITU_Programming_Environment.pdf

Exertus Ltd. No date. HCM2010S Tech Doc [PDF Document]. [accessed on 25th Feb]. Available at: http://www.exertus.fi/files/Tiedostot/TechDoc_HCM2010S.pdf

Exertus Ltd. No date. MIC2000S Tech Doc [PDF Document]. [accessed on 25th Feb]. Available at: http://www.exertus.fi/files/Tiedostot/TechDoc_MIC2000S_revD.pdf

Exertus Ltd. No date. Products [web page]. [accessed on 23rd Jan 2018]. http://www.exertus.fi/?lang=en > Products. Available at: http://exertus.fi/?lang=en&page=Products

flow.ci / Medium.com, 2016. Introduction to Containers: Concept, Pros and Cons, Orchestration, Docker, and Other Alternatives [web page]. Accessed on 2nd April 2018. USA: A Medium Corporation. Available at: https://medium.com/flow-ci/introduction-to-containers-concept-pros-and-cons-orchestration-docker-and-other-alternatives-9a2f1b61132c

Free Software Foundation. No date. GNU make manual [web page]. Available at: https://www.gnu.org/software/make/manual/make.html

Jenkins.io. No date. Documentation, Blue Ocean [web page]. [accessed on 3rd March]. Available at: https://jenkins.io/doc/book/blueocean/

Jenkins.io. No date. Documentation, Extending with Shared Libraries [web page]. [accessed on 3rd March]. Available at: https://jenkins.io/doc/book/pipeline/shared-libraries/

Jenkins.io. No date. Documentation, Installing Jenkins [web page]. Available at: https://jenkins.io/doc/book/installing/#installing-jenkins

Jenkins.io. No date. Documentation, Pipeline: Declarative versus Scripted Pipeline syntax [web page]. [accessed on 3rd March]. Available at: https://jenkins.io/doc/book/pipeline/#declarative-versus-scripted-pipeline-syntax/

Jenkins.io. No date. Documentation, Pipeline Syntax [web page. Available at: https://jenkins.io/doc/book/pipeline/syntax/

Jenkins.io. No date. Documentation, Pipeline: What is Jenkins Pipeline? [web page]. [accessed on 3rd March 2018]. Available at: https://jenkins.io/doc/book/pipeline/#overview

jenkinsci / jenkins. No date. GitHub source code repository: jenkins/LICENSE.txt [text document]. [accessed on 1st Feb 2018]. Available at: https://github.com/jenkinsci/jenkins/blob/master/LICENSE.txt

Kawaguchi, Kohsuke 2014. Hudson [PDF file]. Archived by Wayback Machine Internet archive. Original URI: https://www.java.net//blog/kohsuke/archive/20070514/Hudson J1.pdf. Available at: https://web.archive.org/web/20140701020639/https://www.java.net//blog/kohsuke/archive/20070514/Hudson%20J1.pdf.

Kautto, Tuomas 1996. Ohjelmistotestaus ja siinä käytettävät työkalut: ohjelmistotekniikan seminaariesitelmä [web site]. [accessed on 30th March 2018]. Available at: http://www.mit.jyu.fi/opiskelu/seminaarit/ohjelmistotekniikka/testaus/#RTFToC1. University of Jyväskylä, Jyväskylä, Finland

Limoncelli, Thomas A., Hogan, Christina J. and Chalup, Strata R. 2017. The Practice of System and Network Administration: DevOps and Other Best Practices for Enterprise IT Volume 1, Third Edition. USA: Addison-Wesley / Pearson Education.

The Linux Kernel Archives. No date. Readme file for the Controller Area Network Protocol Family (aka SocketCAN) [text document]. Available at: https://www.kernel.org/doc/Documentation/networking/can.txt

Phacility Inc. No date. Phacility [web page]. [accessed on 27th Jan 2018]. phacility.com > Phabricator. Available at: https://www.phacility.com/phabricator/

phacility / phabricator. No date. Github source code repository: phabricator/LICENSE [text document]. [accessed on 1st Feb 2018]. Available at https://github.com/phacility/phabricator/blob/master/LICENSE.txt

Ries Eric 2008. Startup Lessons Learned: Minimum Viable Product: a guide [blog post]. [accessed on 30th March 2018]. Available at http://www.startuplessonslearned.com/2009/08/minimum-viable-product-guide.html.

XQual. Contact us. 2018-03-13. Available at https://www.xqual.com/contact.html.

XQual. Continuous Integration. 2018-02-19. Available at https://www.xqual.com/documentation/continuous_integration/continuous_integration.html.

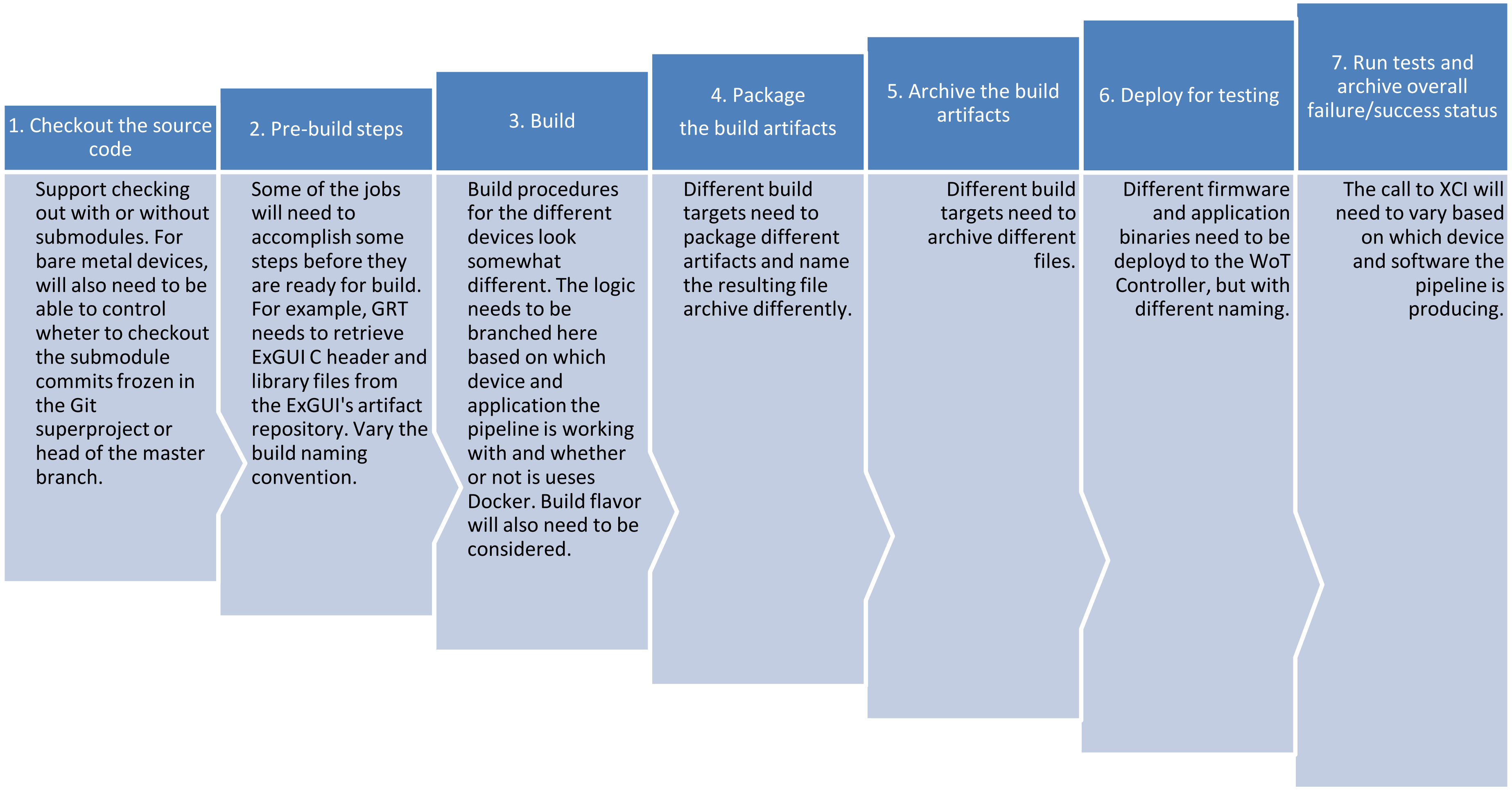
XQual. Open-Source launchers [web page]. 2018-02-18. Available at https://www.xqual.com/open_source/launchers.html.

XQual. XAgent [web page]. 2018-02-19. Available at https://www.xqual.com/documentation/xagent/xagent.html.

XQual. XStudio User Guide [web page]. 2018-03-13. Available at: https://www.xqual.com/documentation/user_guide.

APPENDICES

Appendix 1: Jenkins CI pipeline overall structure



1. Checkout the source code

Support checking out with or without submodules. For bare metal devices, will also need to be able to control wheter to checkout the submodule commits frozen in the Git superproject or head of the master branch.

2. Pre-build steps

Some of the jobs will need to accomplish some steps before they are ready for build. For example, GRT needs to retrieve ExGUI C header and library files from the ExGUI's artifact repository. Vary the build naming convention.

3. Build

Build procedures for the different devices look somewhat different. The logic needs to be branched here based on which device and application the pipeline is working with and whether or not is ueses Docker. Build flavor will also need to be considered.

4. Package the build artifacts

Different build targets need to package different artifacts and name the resulting file archive differently.

5. Archive the build artifacts

Different build targets need to archive different files.

6. Deploy for testing

Different firmware and application binaries need to be deployd to the WoT Controller, but with different naming.

7. Run tests and archive overall failure/success status

The call to XCI will need to vary based on which device and software the pipeline is producing.