TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

ALEKSI KUJALA

HW/SW TESTING FRAMEWORK

Master of Science thesis

Examiner: Professor Timo D. Hämäläinen
Examiner and topic approved on
30th November 2017

ABSTRACT

Aleksi Kujala: HW/SW testing framework
Tampere University of Technology
Master of Science Thesis, 46 pages, 0 Appendix pages
February 2018
Master's Degree Programme in Electrical Engineering
Major: Embedded systems
Examiner: Timo D. Hämäläinen
Supervisor: Hannu Nieminen
Keywords: hardware-dependent software, continuous integration, regression testing

u-blox GNSS receivers are tested at several levels: system level testing is done by the separate testing team, unit test environment is for hardware-independent software modules and continuous integration is used to discover unwanted issues in software development. However, testing of the hardware-dependent software components and hardware drivers is not well organized. So far testing has been irregular and it has been done only manually. If something goes broken, it is not detected immediately.

The purpose of this work was to implement an automated testing environment for hardware-dependent software components in u-blox receiver firmware. Different testing environments and tools were used in the implementation: HW/SW testing framework in the receiver firmware handles the test execution and reporting whereas system testing environment and Jenkins continuous integration tool were used in the test automation.

The work began with exploration of the firmware part of the testing. When testing framework became more familiar, testing process could be developed step by step. When functional testing process was implemented, more test cases and adaptations were done in the framework. In addition, test automation was also implemented for the simulation environment in order to find failures in the IC design.

17 test cases were implemented to the embedded test framework in approximately 3 months. In this time test development and automated regression testing detected several issues such as hardware bugs, broken image or questionable changes in the receiver firmware. So all in all, the new testing environment implemented in this work found out functional and useful in hardware-dependent software development.

TIIVISTELMÄ

Aleksi Kujala: HW/SW testausympäristö
Tampereen teknillinen yliopisto
Diplomityö, 46 sivua, 0 liitesivua
Helmikuu 2018
Sähkötekniikan tutkinto-ohjelma
Pääaine: Sulautetut järjestelmät
Tarkastaja: Timo D. Hämäläinen
Valvoja: Hannu Nieminen

Avainsanat: laitteistoriippuvuus, jatkuva integraatio, regressiotestaus

u-bloxin GNSS vastaanottimia testataan usealla eri tasolla: järjestelmätestauksesta on vastuussa erillinen testaustiimi, laitteistosta riippumattomalle ohjelmistolle on erillinen yksikkötestausympäristö ja jatkuvaa integraatiota käytetään löytämään ei-toivotut ohjelmistomuutokset. Laitteistosta riippuvan ohjelmiston ja laiteajureiden testaus ei kuitenkaan ole organisoitua. Tähän saakka testaus on ollut epäsäännöllistä ja manuaalista.

Tämän työn tarkoituksena oli toteuttaa testausympäristö laitteistosta riippuvalle ohjelmistolle u-bloxin vastaanottimen firmwaressa. Toteutuksessa käytettiin jo olemassa olevia testaustyökaluja: sulautettu testausympäristö firmwaressa toteuttaa testien ajamisen ja tulosten raportoinnin kun taas järjestelmätestausympäristöä ja jatkuvan integraation Jenkins työkalua käytetään testiautomaatiossa.

Työ alkoi firmwareen tutustumisella. Kun sulautetun testausympäristön käyttö tuli tutummaksi, testausta voitiin laajentaa vaihe vaiheelta kohti automatisoitua testausprosessia. Testejä voitiin lisätä firmwareen kun toimiva testausprosessi oltiin saatu aikaiseksi. Testiautomaatio laajennettiin kattamaan myös simulaatioympäristöä, jotta havaittaisiin IC suunnittelussa tapahtuneita ongelmia.

3 kuukauden aikana 17 testiä lisättiin testiympäristöön. Tänä aikana testien kehitys ja automatisoitu regressiotestaus havaitsi virheitä vastaanottimen hardwaressa ja firmwaressa, joten uusi testausympäristö voidaan todeta toimivaksi ja käytännölliseksi ohjelmistokehityksessä.

PREFACE

First, I would like to mention how great it has been in u-blox as a Master's thesis worker. Atmosphere in the Tampere office is always pleasant and co-operative.

I would like to thank hardware bring up team for the opportunity to work with this interesting topic. Since the work included tasks from different fields, there are many people who could be thanked here, but the firmware related support from my supervisor Hannu Nieminen and Jenkins related support from Marko Kaapu and Continuous Integration team has been valuable. In addition, guidelines for writing given by Timo Hämäläinen have been really helpful.

I hope you will have a pleasant reading!

Tampere, 23.2.2018

Aleksi Kujala

CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

HW	Hardware
SW	Software
SoC	System on a Chip
IP block	Intellectual Property block
GNSS	Global Navigation Satellite System
IC	Integrated Circuit
HAL	Hardware Abstraction Layer
RF	Radio Frequency
PMU	Power Management Unit
CPU	Central Processing Unit
HdS	Hardware-dependent Software
RTOS	Real Time Operating System
API	Application Programming Interface
UART	Universal Asynchronous Receiver Transmitter
Jenkins	Continuous Integration tool
RTL	Register-Transfer Level
CI	Continuous Integration
ReleaseTestGUT	Receiver system testing environment
Codename	Receiver firmware repository
HDL	Hardware Description Language
FPGA	Field-Programmable Gate Array
SSH	Secure Shell
DUT	Device Under Test
CMB	Current Measurement Board
PC	Personal Computer
TCP/IP	Transmission Control Protocol / Internet Protocol
GUI	Graphical User Interface
VCAD	Virtual Integrated Computer-Assisted Design
RPi	Raspberry Pi
GPIO	General Purpose Input/Output
ADC	Analog to Digital Converter
RAM	Random Access Memory

1. INTRODUCTION

Hardware-dependent Software (HdS) development is an essential part of the System on a Chip (SoC) design cycle. Each Intellectual Property (IP) hardware block needs a specific driver to be accessed and if changes in the hardware occurs or new hardware design begins, corresponding drivers also needs to be modified. Early detection of the hardware or low-level software bugs shortens SoC design cycle and thereby also costs.

u-blox is a global company providing wireless communication and positioning solutions for the automotive, industrial and consumer markets. Positioning product center is responsible for research, development, marketing and maintenance of the positioning products. Those are primarly Global Navigation Satellite System (GNSS) chips, but also other technologies such as dead reckoning.

u-blox reciver firmware is tested at several levels:

- Testing team is responsible for system testing. System testing is used to verify that the receiver operation corresponds to the specification.
- Higher level, hardware-independent software module unit tests are used in manual and automated regression testing.
- Proper hardware operation is verified with specific hardware block level testing environment.

However, testing of the hardware-dependent software components and hardware drivers is not well organized. So far testing has been irregular and it has been done only manually. If something goes broken, it is not detected immediately.

The purpose of this work is to implement fully automated testing environment for hardware related software components. The idea is that test development is done in parallel with the software development so only a few test cases will be implemented as an example for future test development.

Following chapters describes how the new testing environment is implemented, how it is used and which tools are used to achive desired functionality. Chapter 2 describes briefly the system architecture of the receiver. Hardware architecture of the baseband integrated circuit (IC) is introduced followed by the introduction of the main software modules and their role. Hardware Abstraction Layer (HAL) and the way how the hardware peripherals on the chip can be accessed from the software are focused more closely.

Chapter 3 describes different levels of testing in u-blox and objectives for this work while Chapter 4 gives an introduction to existing test environment and tools. This chapter tells also how the existing testing environments are used in this work. Chapter 5 deals with the actual testing framework. Steps implemented in this work are introduced in this chapter and testing workflow is described at low level. Chapter 6 evaluates results of this work. Some statistic of the testing and found bugs are introduced in this chapter as well. Future work possibilities are also considered.

2. SYSTEM ARCHITECTURE

This chapter briefly describes the hardware and software architecture of the u-blox GNSS receiver. The layers of the software stack are introduced briefly, but focus in this chapter is in issues that are relevant for this work.

2.1 Hardware architecture

Figure 1 shows NEO-M8P high precision GNSS module block diagram. The u-blox baseband chip integrates Radio Frequency block (RF), Power Management Unit (PMU), Digital Block and interfaces for communication. Central Processing Unit (CPU) is connected to the memories and a wide selection of peripherals and engines with bus. Special hardware engines required for signal acquisition and tracking are integrated to the chip. They are capable of search and track satellites defined by control software. Depending on the receiver model, optional external oscillators may be included to the module.

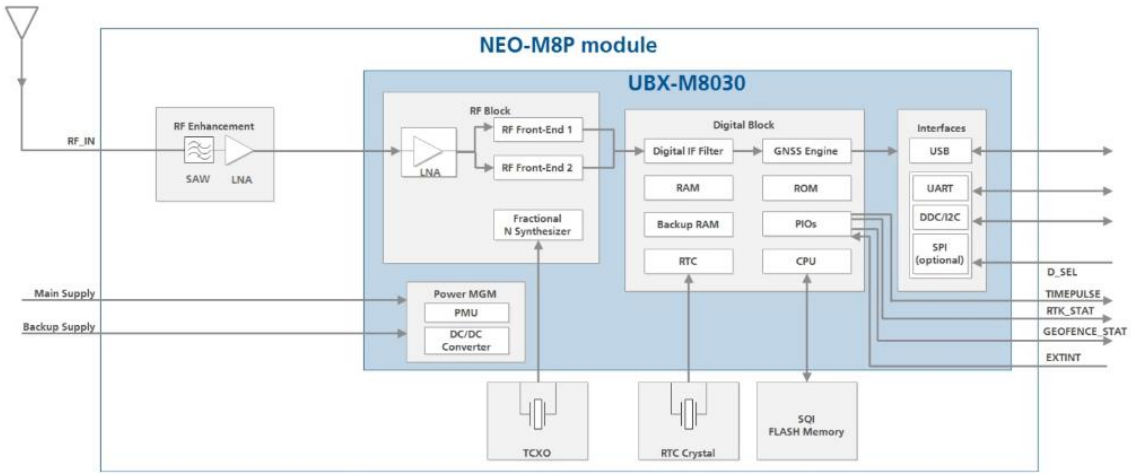


Figure 1. *Hardware architecture. [1]*

2.2 Software architecture

A common software stack is presented in Figure 2. Software can usually be divided into two layers: application and hardware-dependent software. Software in application layer is independent from the hardware and it implements the functionality of the system. Application can be multi-tasking or a single task function. Software in HdS-layer is

dependent on the hardware. It includes for example operating system, hardware drivers and boot code. Operating system in HdS-layer handles the resource management, and schedules the tasks based on the scheduling strategy. Another component in Hds-layer is for communication. It is responsible for communication between tasks in the same processing unit or external subsystems. Hardware Abstraction Layer (HAL) is the lowest level component of the HdS-layer. HAL is directly dependent on the hardware, which means that it has to be changed if underlying hardware architecture changes. HAL includes both processor specific code and device drivers, and it offers access to hardware resources for operating system and communication libraries. HAL is also used to initialize system and all the required harware componenents before application starts execution. [2]

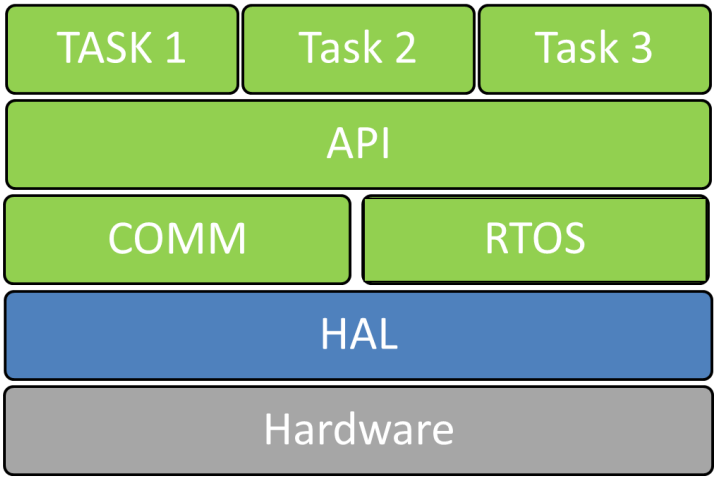


Figure 2. *Generic embedded software stack. [2]*

The role of HdS-layer in the u-blox receiver is more complicated: HAL provides features mentioned above such as system initialization and device drivers which are under investigation in this work. However, HdS-layer also includes receiver manager and tracking control modules which are also dependent on existsing hardware at some level. Scope of this work is not to focus on them, but high level functionality and the role of the software modules is following: receiver manager (RXM) manages receiver acquisition and tracking strategies. It decides which satellites will be searched and informs tracking control module (TRK) which satellites will be tracked. Receiver manager and tracking control modules are accessing GNSS hardware engines through device drivers defined in HAL. Figure 3 shows the u-blox receiver software stack including the main software modules.

u-blox receiver Real Time Operating System (RTOS) implements a priority-based, pre-emtive multitasking scheduler. Task switches are forced by timer counter and each task has its own stack where state of the task is stored in switches. To find a next task for

execution, scheduler iterates through the task list starting from the interrupted task and continues next highest priority ready state task.

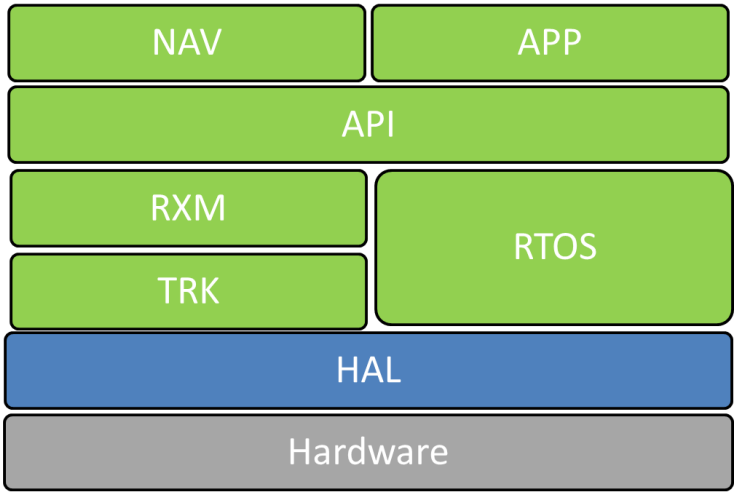


Figure 3. *u-blox receiver software stack.*

Fully hardware independent software modules are Navigation and Application modules. Receiver manager provides the data measured by the GNSS engines to the Navigation module which announces calculated positioning solution for the Application module. Application module is responsible for the system output. u-blox receivers use a u-blox proprietary UBX communication protocol. UBX messages are used to transmit GNSS data to the host, but moreover they are used to configure the receiver the desired way, and monitor the receiver. Navigation and Application modules can be tested offline in Linux or Windows environment. Ubx-log is given to the offline tool which produces new positioning solution with the Navigation module. Figure 4 shows offline tool usage.

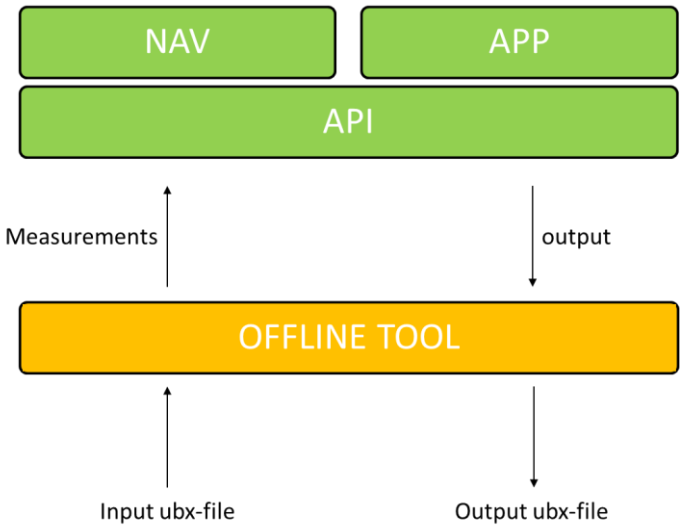


Figure 4. *Offline tool.*

2.3 Hardware Abstraction Layer

Now if we separate the two lowest layers from the stack to Figure 5, we have only Hardware Abstraction Layer and Hardware under investigation. This work is focused to implement a test environment for device drivers in Hardware Abstraction Layer and other hardware-dependent software components. Other testing tools in u-blox and their targets are introduced in Chapter 4.

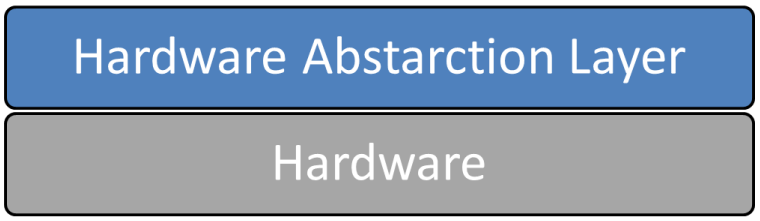


Figure 5. *In this work we are interested in hardware-dependent software components in the hardware abstraction layer.*

In order to understand how the receiver firmware is accessing the underlying hardware, those two layers in Figure 5 should be divided into smaller pieces. As can be seen in Figure 6, P1 – Pn are hardware IP blocks on the IC and each of them have own control registers Reg 1 – Reg n. Registers can be accessed through the defined base address of the hardware block.

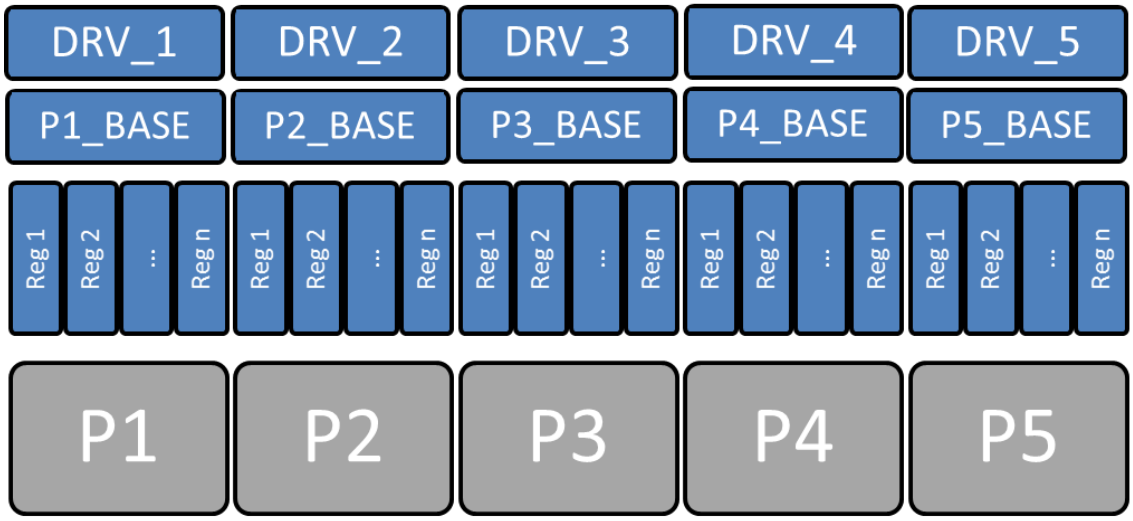


Figure 6. *Hardware peripherals, registers and drivers.*

In order to keep higher level software independent from hardware. Drivers (DRV) are used to access hardware peripherals. Of course each controle register has its own driver, but for the clearness they are represented as a one block DRV_1 – DRV_5.

The ARM CPU contains a bus matrix that allows access to the external and internal peripherals and memories. System address map contains fixed memory region for external peripherals. [3] This memory region is divided into fixed memory regions for the peripherals. Fixed memory mapping for peripheral base addresses is done in the macro file:

```
#define P1_BASE          0x4000A000
```

Peripheral control regisers are defined in the driver header file:

```
//Define registers map for hardware peripheral
typedef struct  DRV_P1_REG_s
{
volatile uint32_t        Reg_1;
volatile uint32_t        Reg_2;
volatile uint32_t        Reg_3;
volatile uint32_t        Reg_n;
} DRV_P1_REG_t;
```

Pointer is created to the defined control registers which is used in the hardware peripheral pointer:

```
//Create a pointer to the registers
typedef DRV_P1_REG_t *DRV_P1_REG_pt;

//Create a pointer to the hardware.
#define pP1 ((DRV_P1_REG_pt) P1_BASE)
```

Driver header file includes also register configurations which can be used in the source file where driver functions are implemented:

```
//Register value (in header file).
#define DO_SOMETHING        0x00000002

//Driver function implementation (in source file).
void some_function()
{
    //Changing the peripheral register value.
    pP1->Reg_1 |= DO_SOMETHING;

}
```

Hardware architecture block diagram seen in the Figure 1 does not reveal all the hardware peripherals available, but some of the communication interfaces can be used as an example. Changed hardware register can be e.g. UART transmission buffer which is used to send desired byte.

Previous code sample for the driver was a simple example. Drivers tested in this work may also be dependent on other drivers or the testable module may have only a minor hardware dependency which could also be replaced with some constant solution.

3. SOFTWARE VERIFICATION

3.1 Testing process

The main goal of the testing is to verify correct operation of the tested feature, or to find a reason why the feature is not working like it should. So at high level, it can be summarised that we are always comparing some requirements to the test observations. Testing after system design process can be modelled with Figure 7 diagram:

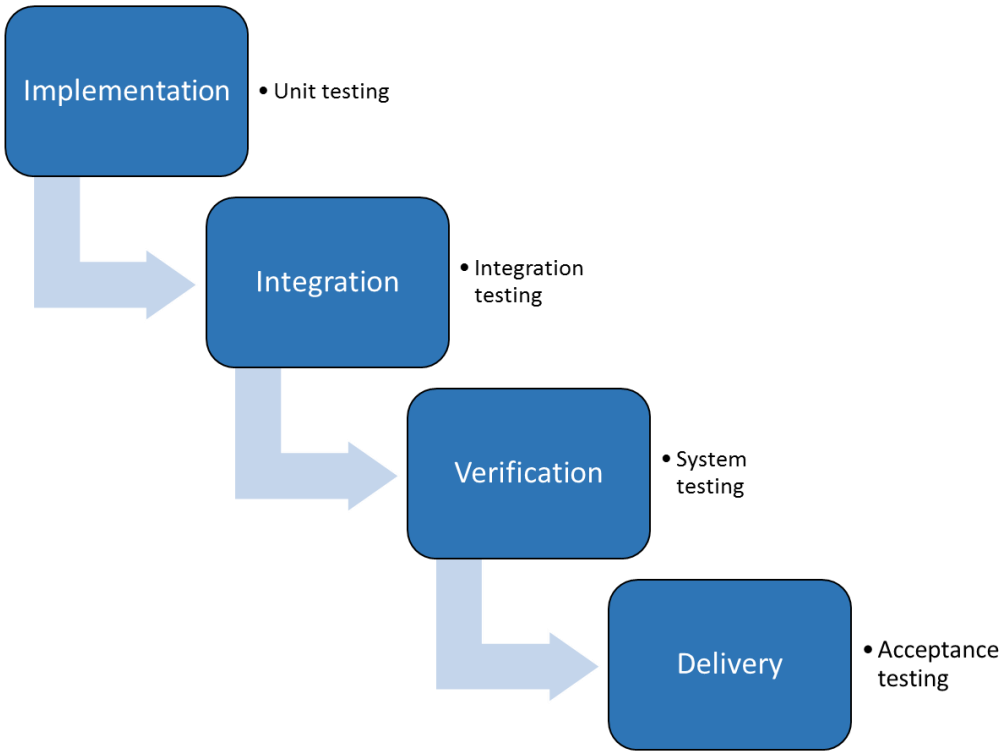


Figure 7. *Testing workflow. [4]*

Unit testing is a phase where each unit is tested independently from the other system components. [4] Unit testing is done in parallel with the software development and a unit can be for example a module, class or process.

Integration testing is the level of testing where various components are integrated to the overall system. [4] As can be seen in Figure 8, a software module may consist of several module layers so integration tests can also be done at several levels.

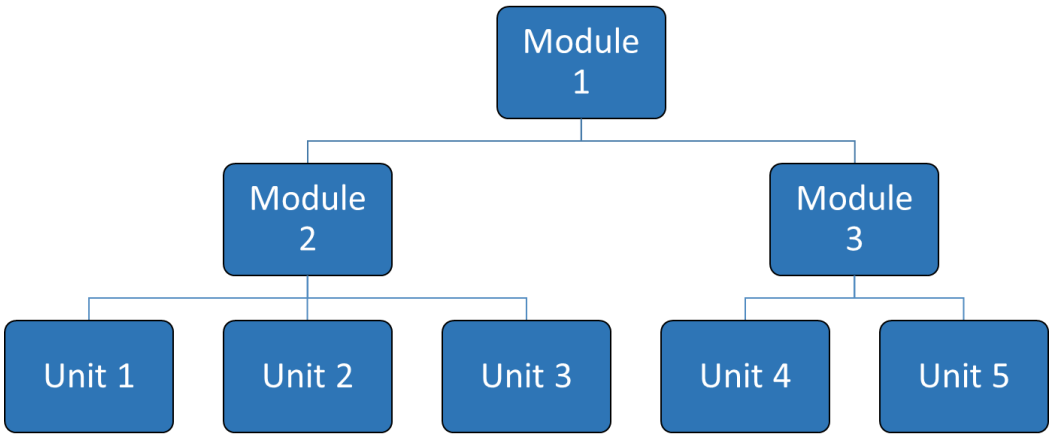


Figure 8. *In this case, integration tests can be done for Modules 1-3.*

System testing is usually done by the independent testing team. It is focused on the overall functionality of the system and the main goal for system testing is to verify that software is behaving according to its specifications. [4]

Acceptance testing is often done by the customers or their representatives in the realistic environment. Focus in the acceptance testing is also in system verification. But if system testing is more focused on finding errors, acceptance testing is trying to proof their absence. [4]

3.1.1 Test development and execution

Test environment, including e.g. test cases, test data and hardware is acquired in test development step. The following step is used for test execution and result observation. Now if these various level of testing and test steps are compared to the testing in u-blox, each of them have own teams and tools which are responsible for testing in concern.

Unit testing is mostly done by the developers. When some new feature is under development, *unit tests* are performed in local environment before integrating changes. However, *unit tests* are also part of the continuous integration (CI) so there is only a slight difference between *unit testing* and *integration testing*. *Integration testing* in u-blox is mainly continuous building of the firmware where unit tests are also performed as a *regression*. *Regression testing* is a type of software testing which is used to detect errors in already existing features. Errors may be a result of the recent modifications to the software. Testing can be done at all layers and it can be manual or automated. [5]

System testing is done by the separate testing team. Test environment including testdata, test cases and hardware are controlled from ReleaseTestGUT which is introduced in Chapter 4.3. Testing team is divided into three subteams: one team is responsible for module production testing, another team is responsible for test software development

and third team test execution and analysis. *System testing* has also become a part of the continuous integration, which can be imagined as *integration testing* at highest level. This means that a certain set of release tests are performed regurarly with different receivers.

As a part of the quality assurance, test sprints are performed for the Codename firmware on a regular basis. Test sprints are used to assert that the Codename stability and performance are at adequate level for future work and firmware product releases. Each sprint has its own topics where testing is focused. Test plan defines which receiver modules, firmwares and GNSS constellations are used in the sprint.

It is easy to define which is correct and which is not when performing *unit* and *integration tests*. We have an assumption how some software module should work or what kind of values it should return. However, environmental factors have a major affect on GNSS receiver functionality, so only a small part of the system tests have ON/OFF result. Because of this, extensive statistical testing and test data in different environments are needed.

3.2 Objectives

Main objective for this work is to offer a testing environment for hardware-dependent software components. So far they have been verified manually without organized test development, which means that new feature is tested when it is implemented and irregularly after that. So if something goes broken it is not noticed immediately. New testing environment offers a test implementation and execution environment for both manual and automated regression testing.

3.2.1 Manual tests

Some of the tests are not reasonable to include in the automated regression testing, because they may need external equipment such as oscilloscope, multimeter or adjustable power source during testing. In this case the test itself does not necessarely give any information of the testable feature, but it may be controlled by the test and observing is done with the external meters.

Manual tests are intended to be executed in local environment e.g. own desktop by the developer. Test case development is done concurrently with the other software development. Test case is then used in manual regression testing each time changes may have an affect on module functionality.

3.2.2 Automated tests

Firmware part of the whole HW/SW testing process is introduced more closely in Chapter 4 and 5, but the tests which are not defined as manual tests are executed sequentially by sending one command to the receiver. These tests are part of the automated regression testing which is done every time something is changed in the source code version control. Test automation is handled with Jenkins continuous integration servers.

Tests are executed with the real receiver and in the simulation environment where receiver hardware is modeled with register-transfer level (RTL) description. Both targets have own objectives: purpose of the testing with the real chip is to detect failures in the receiver firmware. That's why it is triggered by the changes in the source code management. Purpose of the simulations is to detect failures in the IC design and the testing is done less frequently.

4. CURRENT TEST ENVIRONMENT

This chapter describes already existing testing environment and how different parts of it can be used in this work. In order to understand different test environments Figure 9 shows different software layers with the corresponding test environment. Each testing environment is introduced in own chapter.

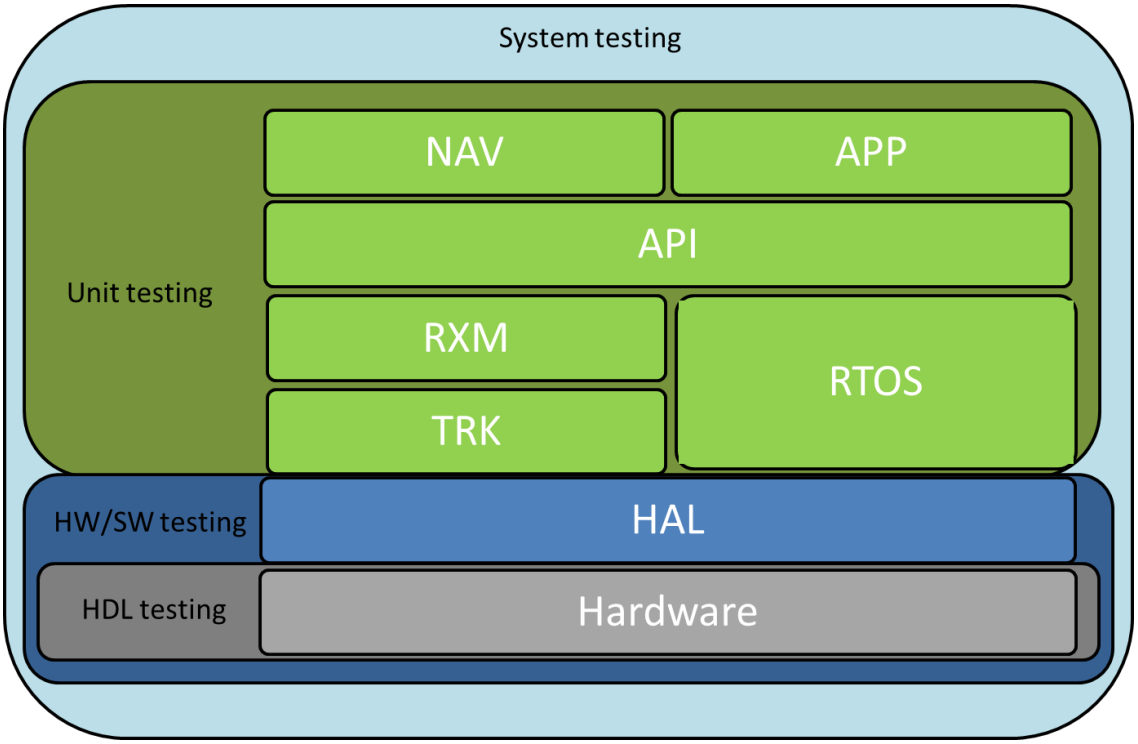


Figure 9. *Software stack with corresponding test environments.*

4.1 HDL test environment

HDL test environment is an embedded block level testing environment in the receiver firmware. It is used to test basic register access and to verify correct behavior of the logic before it is deployed and it can be runned on a real chip, FPGA or simulator. This HDL test environment has a special image for the mentioned targets where only essential files are compiled to get minimum system running. Program image is the executable binary which is loaded to the flash memory.

Name of the environment may be a bit confusing because it refers to the Hardware Description Language. However, test environment itself does not have anything to do with the language even if the tests are also done in the simulations where hardware description language is used to model receiver hardware. HDL test software architecture can be seen in Figure 10, only HAL layer with the embedded HDL test framework is compiled in the image. Test is a single task function so operating system can also be removed. More precise generic example of a HDL test case can be seen in Figure 11.

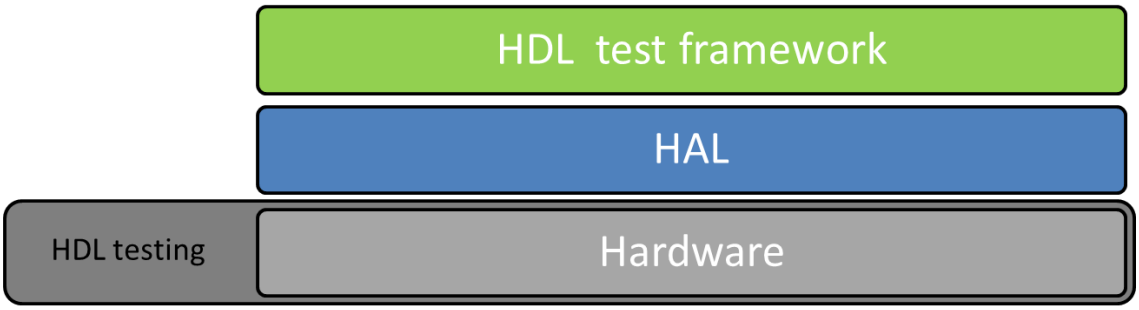


Figure 10. *HDL test software architecture.*

HDL tests are performed by using only a minimal number of other peripheral blocks seen in Figure 11. Tests are usually writing something to the hardware registers and after a while register value is read and it is compared to the written value. It is also possible that register write may launch some hardware block and in that case its operation is checked. If register value is not what it should be or hardware block is not operating correctly, error count is increased.

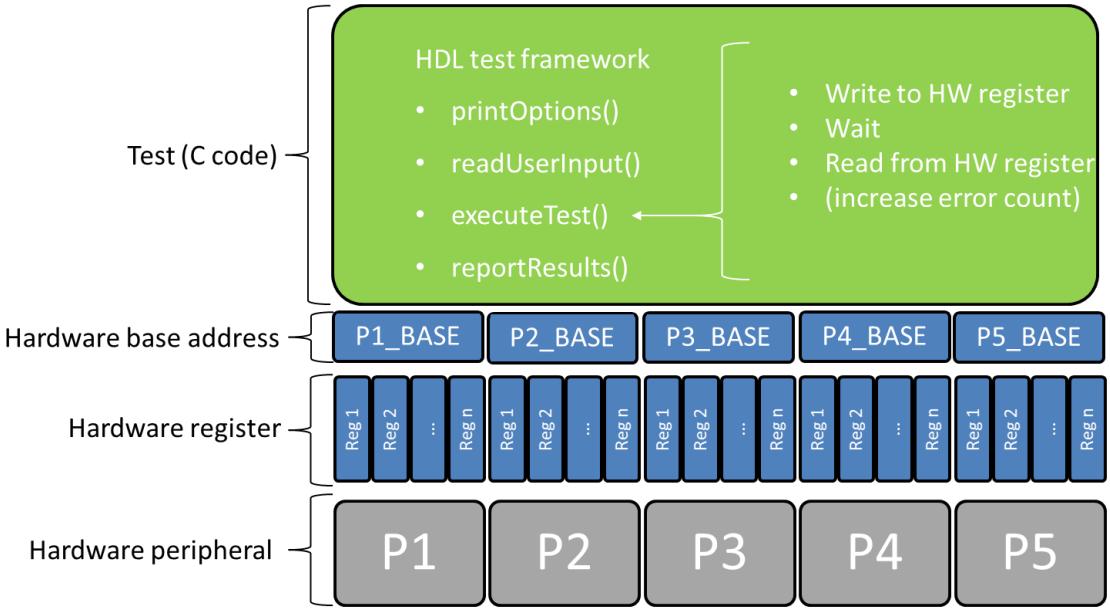


Figure 11. *Generic HDL test case.*

In practise a HDL test with the real chip is done by taking a serial connection to the receiver, for example with Putty SSH client and by choosing which test will be performed. Hardware setup for the test can be seen in Figure 12. In the figure DUT is connected to the current measurement board (CMB), but test could also be executed on FPGA platform or in simulation environment. Because we are verifiying correct behavior of the logic, external meters, signal source or other RF components are not needed. Receiver firmware is testing itself so basically Personal Computer (PC) is needed only for sending commands and observing the output.

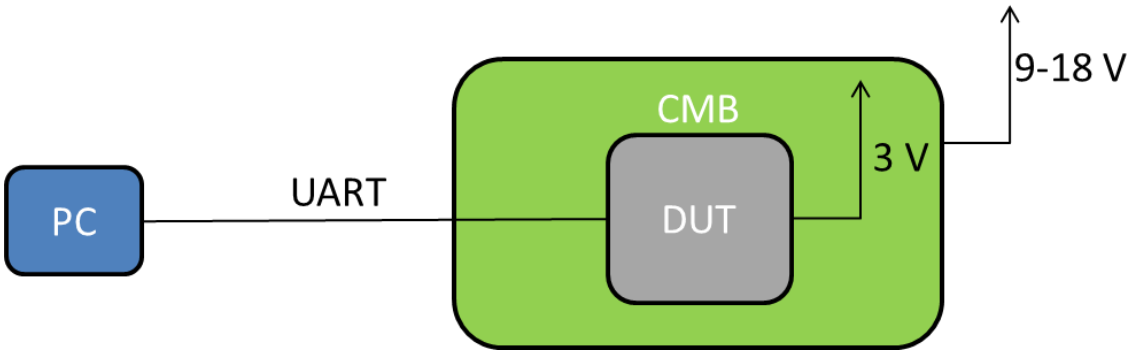


Figure 12. *Hardware setup for HDL test.*

Figure 11 shows a system architecture and a generic test case in HDL test environment. After the receiver is booted and C execution is moved to the HDL test main function, the test framework prints all the tests available and a set of options to the terminal wherefrom most essential commands for this work are listed below:

- s Start HDL test
- r Software reset
- b Safeboot (exit now!)
- [1-..]<enter> Run only that test

By sending some of the options to the receiver, appropriate actions are performed. Command *s* executes all the tests available sequentially, *r* will perform software reset and *b* will set the receiver to the safeboot mode. User can also run a certain test by choosing the number of the test. Results and possible errors are printed to the terminal after the tests

HDL test framework is also used as a platform for HW/SW tests. It offers an access to the hardware resources, test handler and user interface so embedding the new HW/SW test framework into already existing HDL test framework was the best solution. The difference between new HW/SW and HDL test framework is that HW/SW is meant for hardware dependent software testing whereas HDL is meant for hardware block testing.

Of course hardware is also tested at the same time when HW/SW test is performed. Both HDL and HW/SW images include HAL layer as can be seen in Figure 13, but focus of the testing is different.

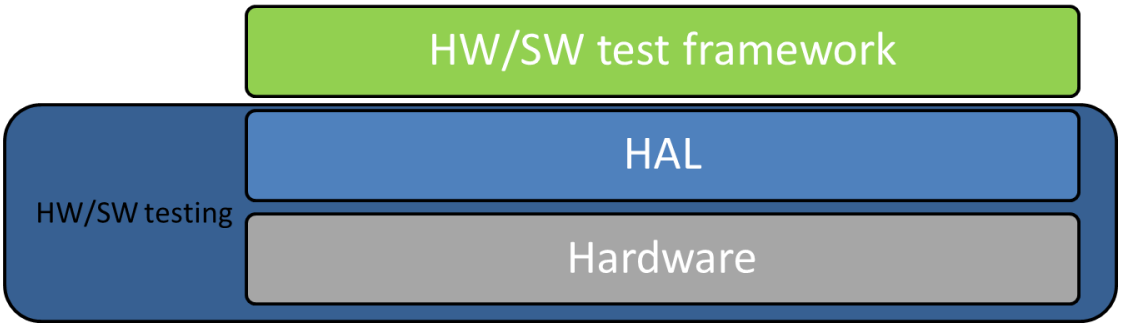


Figure 13. *HW/SW test software architecture.*

4.2 Unit test environment

Part of the Codename firmware repository are firmware unit tests. Unit tests are used to verify correct operation of all the software modules excluding hardware-dependent components. Unit tests are performed offline in Linux or Windows environment and tracing of the test coverage can be included into tests also. Because this unit test environment can be performed only in Linux or Windows environment, lower level software components are not included in the tests. Tested software modules can be seen in Figure 14.

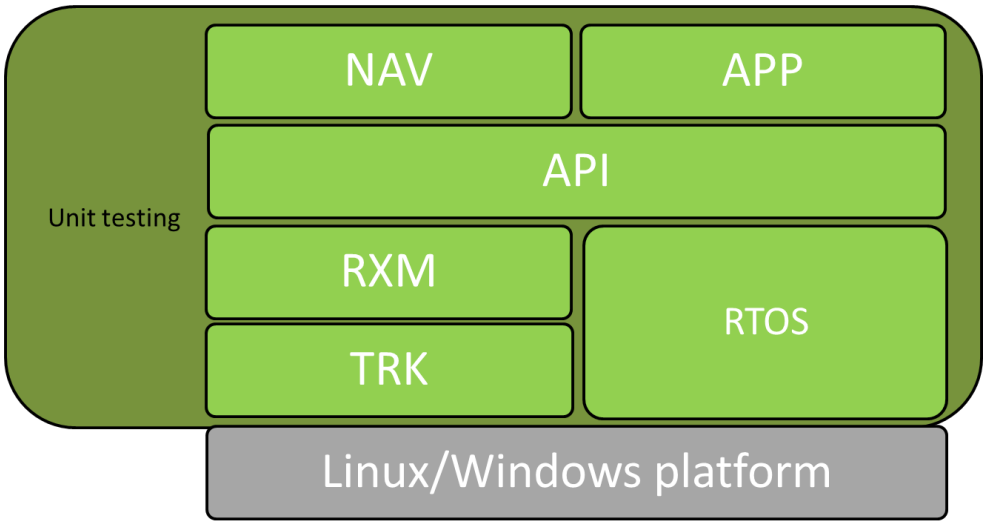


Figure 14. *Unit tests are for hardware-independent components.*

In order to test low level software, current unit test environment should be upgraded with transaction accurate hardware platform model implemented in SystemC. Hardware model would contain all the resources required for low level software execution and verification. [2] However, hardware modeling would be fairly slow option compared to real hardware and if we want to test every new commit in the repository, the time interval between commits might be too short. Modeling of the certain hardware resources would also be challenging. Because of these reasons, this option is left out of consideration.

4.3 ReleaseTestGUT

ReleaseTestGUT is a Perl written test framework for receiver system testing. System testing is focused on overall functionality of the system as can be seen in Figure 15. It verifies that system reflects to the system specification. In our case interesting values are e.g. accuracy of the receiver and its power consumption.

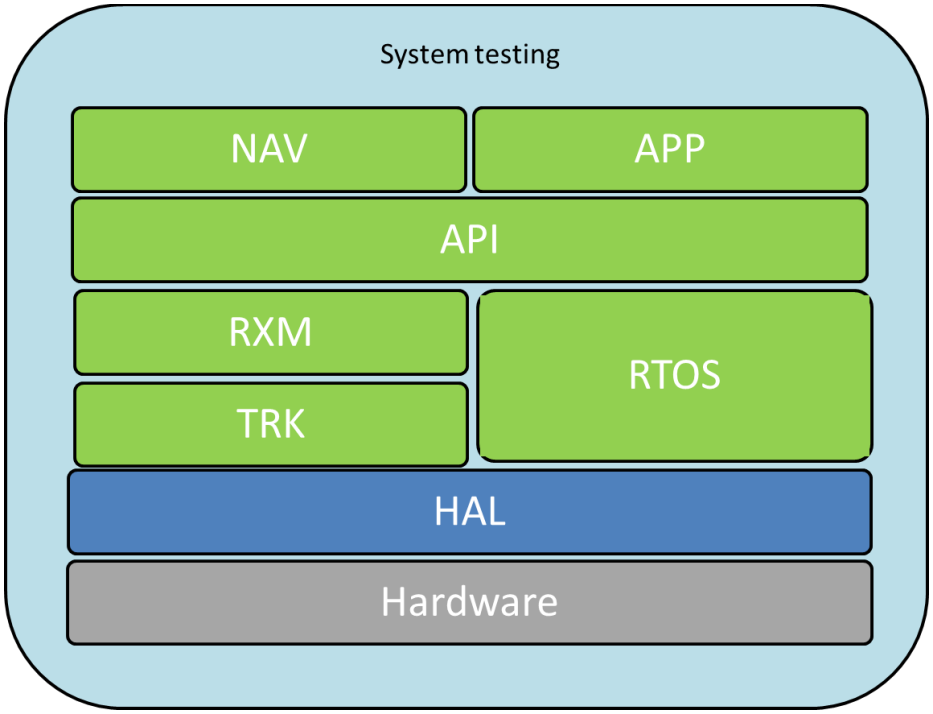


Figure 15. *System testing covers the whole system.*

ReleaseTestGUT includes e.g. test cases, testdata used in the tests, test hardware drivers and results. Release tests are performed on a specified test sites. Typical components for a test site are listed below and the block diagram can be seen in Figure 16:

- Ethernet switch
- Serial device server

- Power supply
- Signal source
- Amplifiers, attenuators and RF splitters
- Current measurement board with one or several receivers
- Hardware for additional measurements

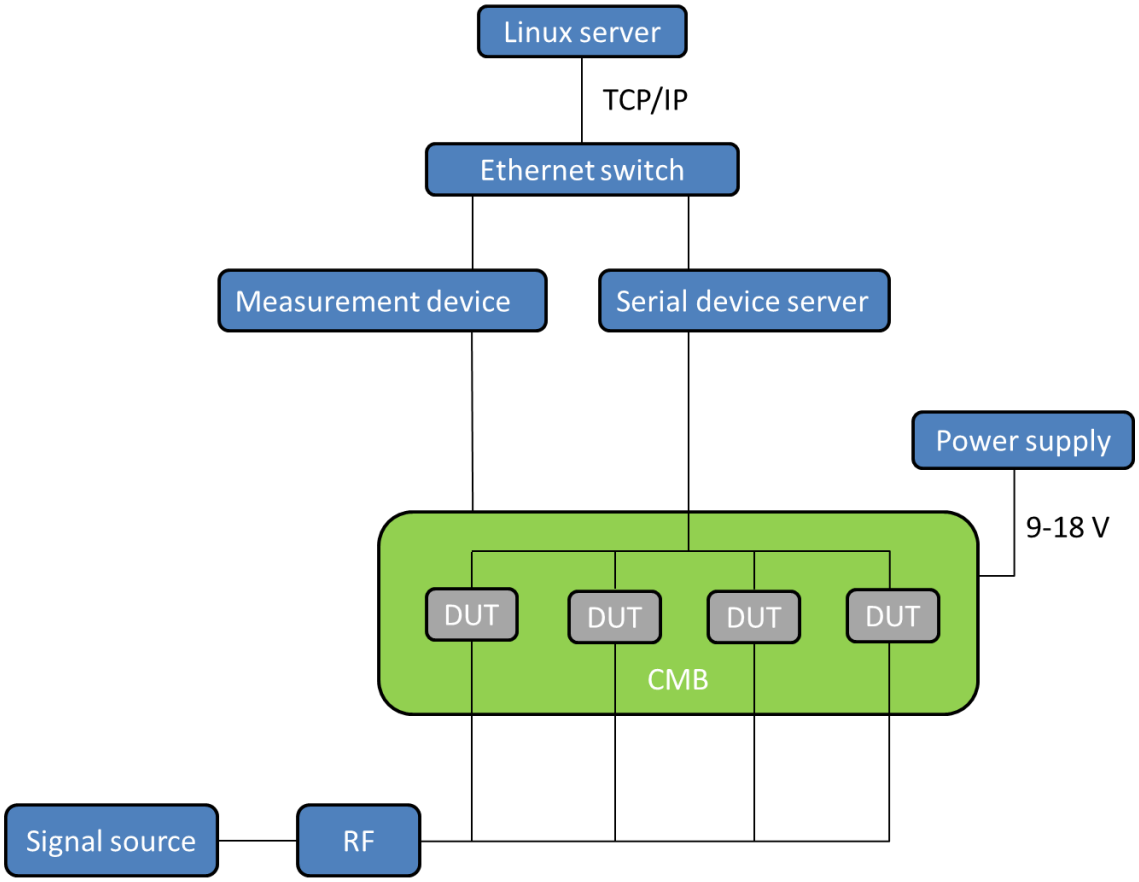


Figure 16. *Test site block diagram.*

Signal source can be a signal generator, replay device or a static antenna. Signal generator can be used to generate a signal which can not be recorded or got from the antenna, e.g. test site is in Europe and the receiver is configured to use satellite system from Asia. Replay source is used if we want to replay some recorded scenario. Recorded scenario can be e.g. from near environment. In some test cases static roof antenna is enough for a signal source. External LabJack devices are used to measure for example power consumption of the receiver.

Basically a release test can only be a set of phases where the scenario is repeated for the receiver with the desired firmware. Then the produced positioning solution is compared to the truth solution. Truth positioning solution may be generated for example with the Applanix positioning system where integrated inertial technology is utilized in order to

get a reliable solution. With produced positioning solution and recorded truth solution, measurements such as positioning error in 2D and 3D dimensions can be generated.

Release tests can be performed from either command line interface or graphical user interface (GUI). Usage of ReleaseTestGUT GUI is not relevant in this work so it is not handled more closely, but GUI is more or less using same scripts to perform test compared to command line way of testing. Three different perl scripts are needed to perform release test from command line interface:

- testconfig.pl generates a test configuration file. Test configuration file includes e.g. choosed test case and possible test parameters.
- rxconfig.pl generates a receiver configuration file. It includes information used to configure the receiver for the test. For example used firmware revision and GNSS constellations are included.
- releasetest.pl script is used for test execution. It needs generated configuration files generated with the scripts above as a parameter.

From this work point of view, ReleaseTestGUT offered a simple receiver interface to use. Communication would have been possible with the receiver even without ReleaseTestGUT, but then the interface should have been done in the Jenkins. Perl modules in ReleaseTestGUT offered ready-made functions for receiver connectivity and communication e.g. functions that can be used to send and store incloming data from the receiver.

4.4 Jenkins

Jenkins is an open source continuous integration tool which is used to automate building, testing and deployment of the software. [6] Several in-house servers has a Jenkins installed and they are maintained by the continuous integration team. In u-blox, Jenkins is used for several intention: continuous building in the release branches, and unit tests are normal use, but Jenkins is also used to run release tests with the receiver and analyze static receiver logs. Continuously performed release tests with the receivers are corresponding with this work, only the objectives of the testing are different. Release tests are focused on system level verification whereas we are interested of individual software units.

Figure 17 shows the Continuous Integration process: developers checks in their changes to the release branch in the source code management which triggers a set of actions where new changes are checked out to the CI server workspace and build is started. Observations of the build is then notified to the concerning developers and possibly to the project management. In fault situation responsible continuous integration team is also notified.

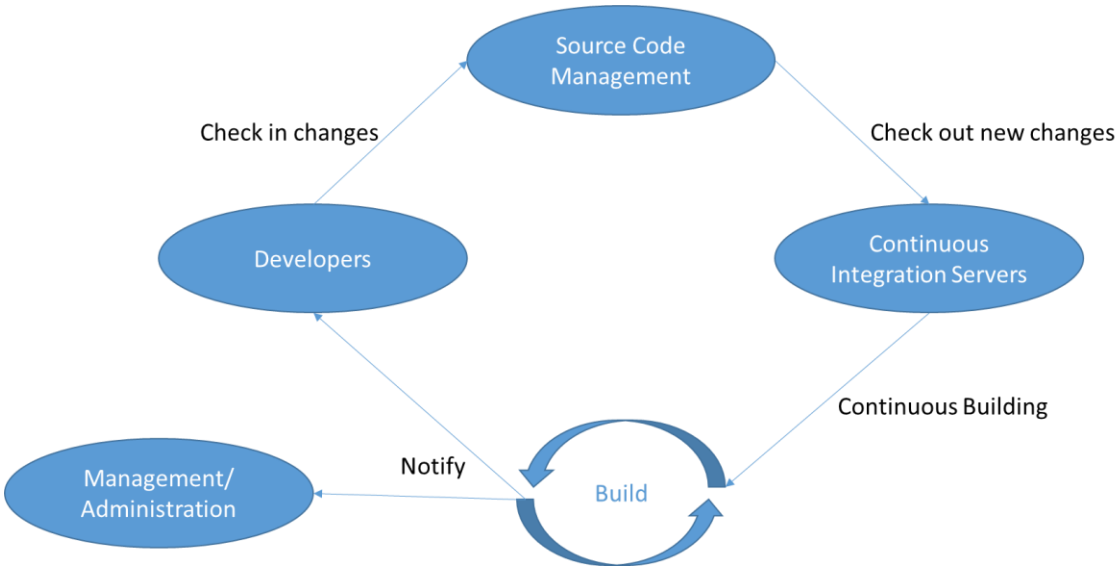


Figure 17. *Continuous Integration process.*

4.5 VCAD

Cadence provides the expertise and tools for the entire electronics design chain. [7] Cadence runs a hosted Virtual Integrated Computer-Assisted Design (VCAD) environment for u-blox. Environment offers revision control for IC development and tools for digital verification and synthesis. Verification is done with the NCSim simulation engine and SimVision (Figure 18). SimVision offers waveform viewer to debug digital, analog, or mixed-signal designs written with several different hardware description languages. [8]

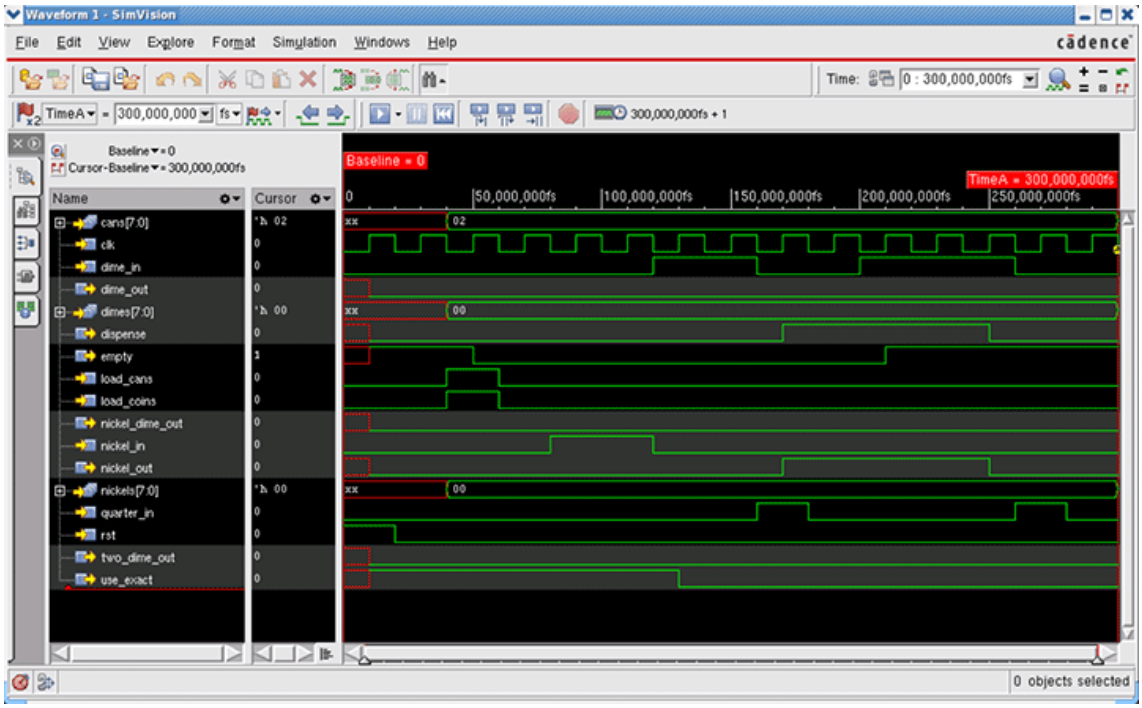


Figure 18. *SimVision waveform window. [8]*

In this work we are running HW/SW tests with RTL simulations automatically without user interaction. Thereby we are only interested in NCSim console output and not signal level events. However, signal level verification can be considered in the future work.

5. HW/SW TESTING ENVIRONMENT

This chapter describes implemented work including testing framework in the receiver firmware and test automation. Test automation includes both tests with the real chip and in simulation environment. Figure 19 shows used testing architecture. We have two Git repositories which are checked out to the workspace. Codename is the firmware repository where specific image is built whereas ReleaseTestGUT is the system testing environment used as an interface to communicate with the receiver.

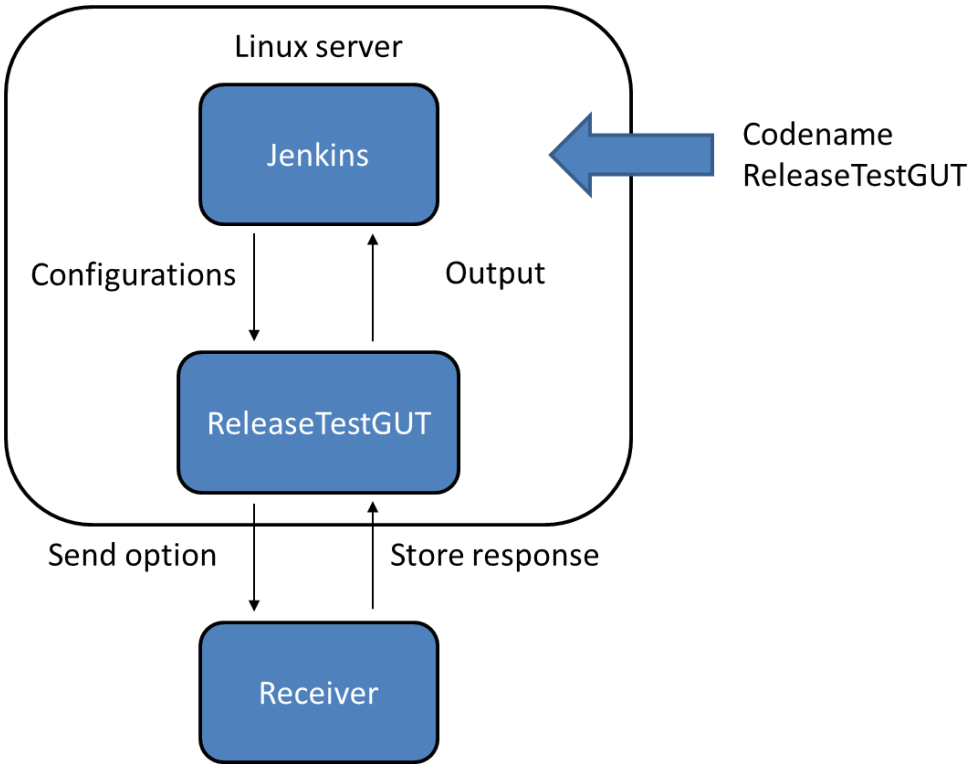


Figure 19. *Test architecture with the real chip.*

5.1 Test automation

The whole test automation is based on Jenkins. Jenkins pipeline script is configured to poll changes from the master branch of the Codename repository, which means that the job is built each time someone merges a development branch to the master branch as can be seen in Figure 20.

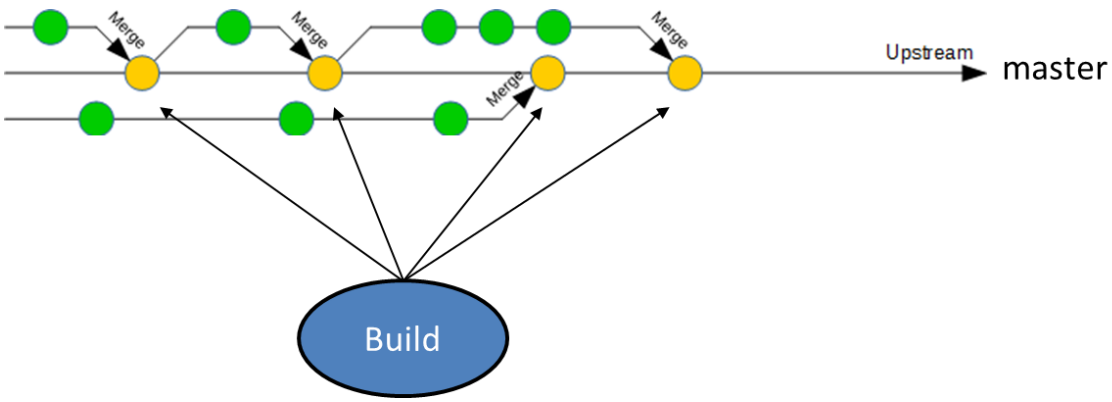


Figure 20. *HW/SW tests are executed each time changes in the master branch occurs.*

During build, both Codename and ReleaseTestGUT with its submodules are checked out to the HWSW_test subdirectory of the specified workspace. Structure of the workspace can be seen in Figure 21.

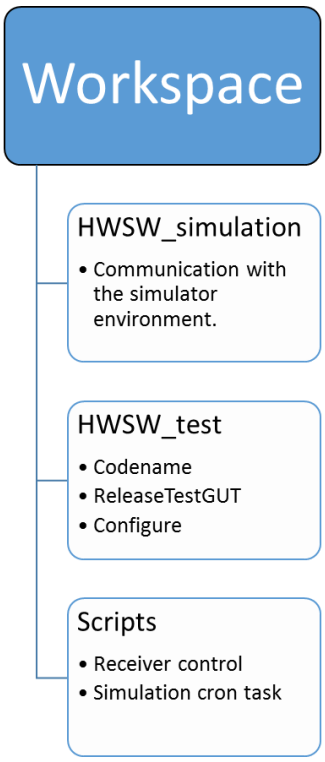


Figure 21. *Workspace structure.*

Jenkins pipeline script compiles the HW/SW image with latest changes in the *Codename* subdirectory and uses perl scripts located in *ReleaseTestGUT* subdirectory to create the configuration files described in Chapter 4.3 for the test. Release test is then

executed with these files. Both configuration files and release test output is located in the *Configure* subdirectory. Release test establishes a connection to the receiver and test case passes an option *s* to the receiver through UART. After receiving the option, embedded test framework executes all the defined tests for testable software components while release test is storing incoming results to the ubx log file.

When release test is ready, Jenkins script unzips the ubx-log located in the *Configure* subdirectory and parses the log for errors. Existence of the errors prescribes whether Jenkins build status is success or failure. Information of the status is then sent to the defined users. In the beginning information is only sent to the test framework developers, but in the future information will be sent only to the developer who has made the changes. The whole test process can be seen in Figure 22.

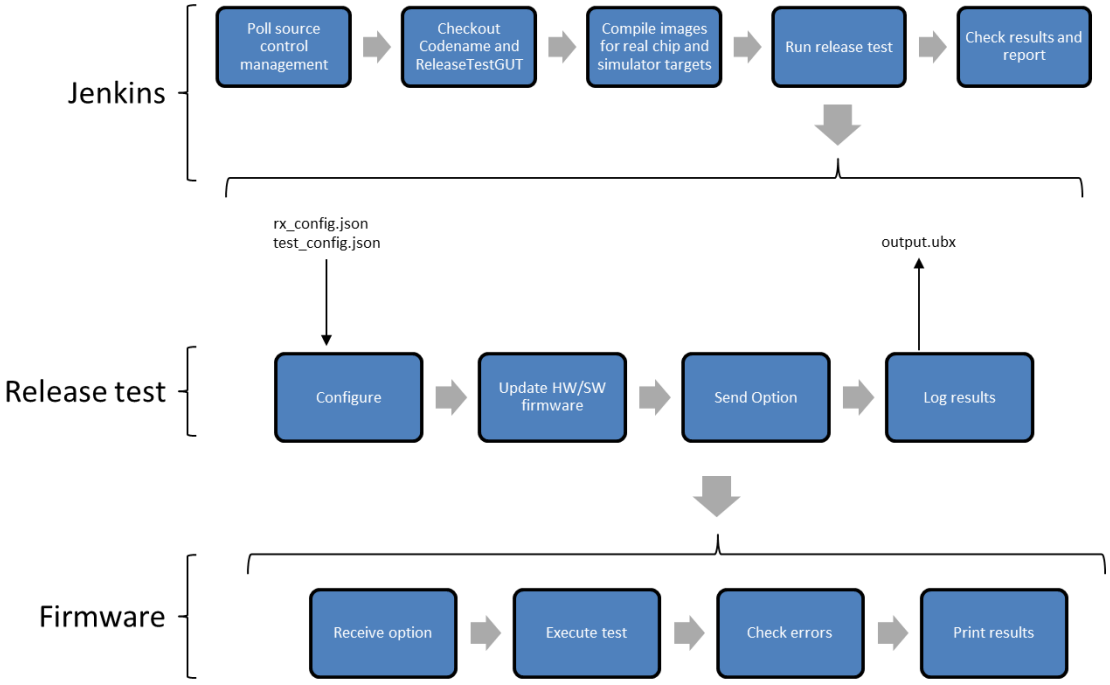


Figure 22. *Testing process.*

Defined users will receive the status and a link to the Jenkins build by email. User can then explore the console output of the build where output ubx-log is printed. If build fails, console output can be explored to see which part of the test failed and fixes can be done immediately. The files which are changed in the build are displayed and searching of the bug can be limited to the corresponding files. HW/SW project page in Jenkins is shown in Figure 23. All five stages of the build and their durations can be seen on the front page of the project and if build fails for some other reason, it can be seen there. Build history can also be seen on the left and results of the previous builds can be compared together.

Program 1 shows the stage structure of the Jenkins build. A node is an executor where the job is executed and desired server can be chosen by filtering node with server name. For example in this case with *ch-thl&&devpos&&linux*, executor is *devpos Linux* server in *Thalwil*. *run_test()* function is executed twice in case of exceptions. Second time, receiver is reseted with the Raspberry Pi before firmware update and test execution.

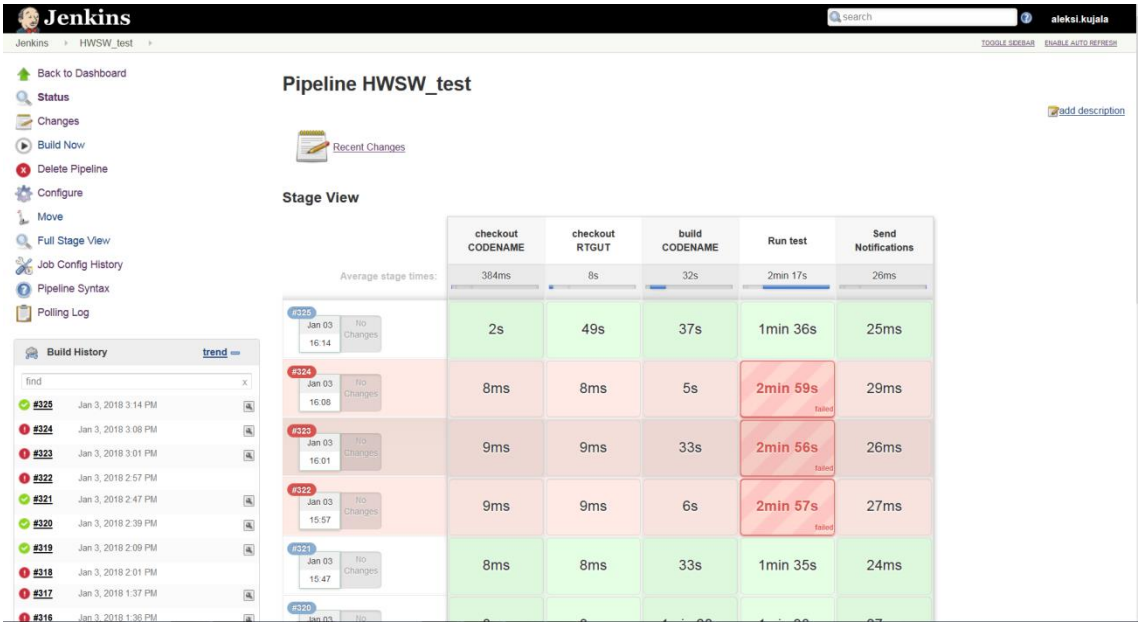


Figure 23. *Jenkins project page.*

```
node('ch-thl&&devpos&&linux') {

    try {
        def Firmwares = [:]

        currentBuild.result = 'SUCCESS'

        stage('checkout CODENAME') {
            checkout_CODENAME()
        }
        stage('checkout RTGUT') {
            checkout_RTGUT()
        }
        stage('build CODENAME') {
            build_CODENAME(Firmwares)
        }
        stage('Run test') {
            retry(2) {
                run_test(Firmwares)
            }
        }
    } catch (e) {
        currentBuild.result = 'FAILURE'
        throw e
    } finally {
```

```
        stage('Send Notifications') {
            send_Email()
        }
    }
}
```

Program 1. *Jenkins project stages.*

Release test configuration files are generated in *Run test* stage. Test and result parsing are also done is this stage. Program 2 is a code snippet from Perl written release test implemented in this work. In the beginning we are sending option *s* to the receiver and we start to read incoming data. If expected string from the receiver does not appear, timeout will interrupt logging and Jenkins build will end up in failure.

```
$rxh->send("$unitTest");

my $t0 = time();
my $timeout = 90;

#Read messages until tests are finished or stop logging when timeout.
DEBUG1("------------Waiting for results------------");
while(time() - $t0 < $timeout) {
      my $msg =$rxh->getMessage();
      if(defined($msg->{_string}))
      {
            if(index($msg->{_string},"more to be implemented") != -1)
            {
                  last;
            }
       }
}
```

Program 2. *Release test code snippet.*

In this work, a specific test site in Tampere office laboratory was implemented for HW/SW testing. If common test site was used, knowledge of the existing hardware would have been necessary. Like mentioned in Chapter 2.2, UBX-messages can be used to configure and monitor the receiver. However, UBX communication is not included into HW/SW image so any information of the existing hardware is not available. It could be possibe to run HW/SW tests in the common test sites by updating the standard firmware to the receiver after the test. However, there still would be problems with the queuing of the tests.

In order to be sure of the existing hardware it was reasonable to implement an own test site where the desired receiver module can be set permanently. When the test site is allocated only for the one task, tests can be executed immediately when changes in the receiver firmware repository occurs. Test site architecture used in the HW/SW tests can be seen in Figure 24.

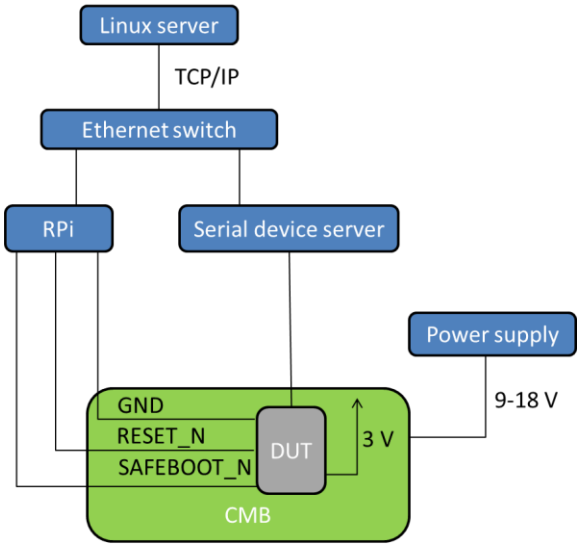


Figure 24. *HW/SW test site architecture.*

Typical test site with its components was defined in the previous chapter. Signal source and other RF components are not needed in low level software testing so they can be removed from the test site. Basically only current measurement board with the receiver is enough for a test site. Current measurement board is connected to the serial device server, which transfers serial data to the local network.

Once in a while some of the tests may get stuck. In this situation receiver has to be booted manually before continuing tests with new firmware revision. In order to boot the receiver remotely, a Raspberry Pi (RPi) is connected to the current measurement board. Ground levels of the CMB and RPi are connected while RPi General Purpose Input/Ouput (GPIO) pins are controlling the reset and safeboot pins of the receiver. Raspberry Pi and receiver connected to the current measurement board can be seen in Figure 25.

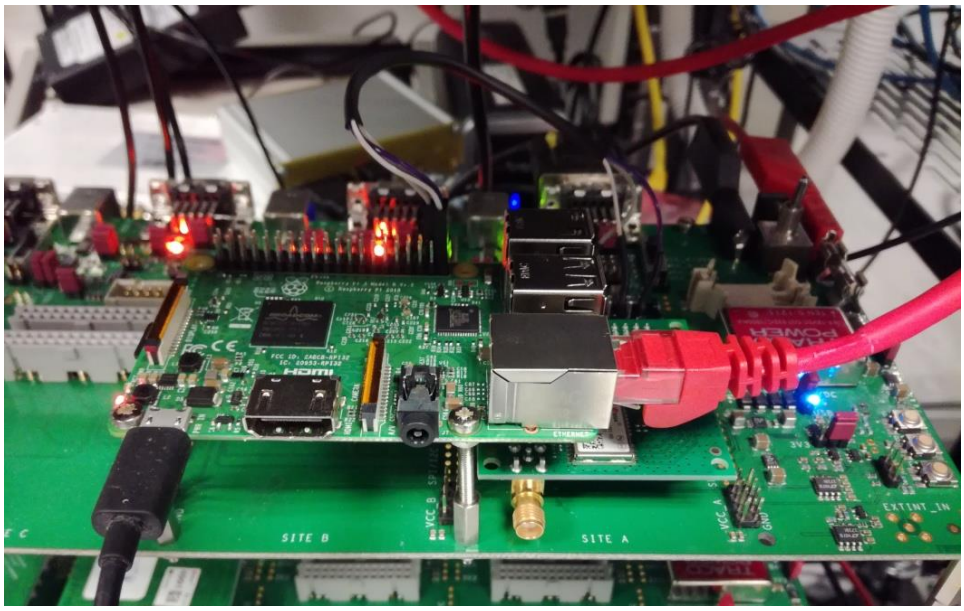


Figure 25. *Raspberry Pi is connected to the receiver to enable remote boot.*

RPi GPIOs are controlling *SAFEBOOT_N* and *RESET_N* pins which can be seen in Figure 26 NEO-M8 pin assignment.

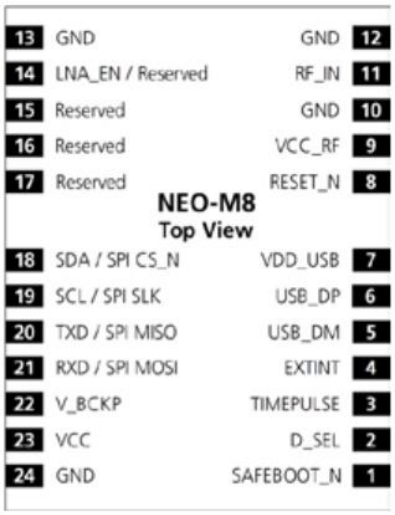


Figure 26. *u-blox receiver pin assignment. [1]*

Figure 27 shows pin control timing diagram. In reset, RESET_N pin is held down for a 0.5 second and then released. In safeboot, SAFEBOOT_N pin has to be held down when RESET_N pin is released. RPi GPIOS are set to high impedance mode when they are not used.

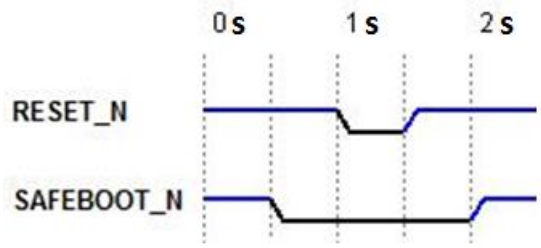


Figure 27. *Safeboot timing.*

RPi is connected to the local network so basically everyone can reset the receiver by executing Python written script located in the RPi. Test case in the ReleaseTestGUT also detect if the receiver does not respond, in this situation SSH Perl modules are used in Jenkins build to reset the receiver with the RPi. Python script can be seen in Program 3 code snippet.

```
GPIO.setmode(GPIO.BOARD)
GPIO.setwarnings(False)

if action == "-help":
        print(help)
elif action == "-reset":
        GPIO.setup(40, GPIO.OUT, initial=GPIO.LOW)
        sleep(0.5)
elif action == "-safeboot":
        GPIO.setup(38, GPIO.OUT, initial=GPIO.LOW)
        sleep(0.5)
        GPIO.setup(40, GPIO.OUT, initial=GPIO.LOW)
        sleep(0.5)
        GPIO.cleanup(40)
        sleep(0.5)
else:
        print("Unknown command!")
        print(help)

GPIO.cleanup()
```

Program 3. *Raspberry Pi Python code.*

5.2 Tool interaction

The second part of the test automation was simulations. Same framework can be used for simulations and real hardware. The image is different and defined target is used as a compile-time flag to tell what is included in the image. HDL tests are already done in Register-Transfer Level (RTL) simulations so the same flagging can be used for HW/SW image. Basic workflow with simulations is corresponding with the hardware tests. Another Jenkins job is implemented to run the tests and parse the results. Thereby many things can be reused from the previous job for this purpose. Access from the local

servers to simulation service provider servers and workspace turned out to be the most complicated part of the task.

There are two supported mechanisms for running simulations: makefile based flow and regression based flow. The makefile based flow is intended for interactive work, for example RTL development, and regression based flow is for automated testing of the blocks. So basically in order to run HW/SW tests in RTL simulation, simulation needs to be started as a regression. In order to get latest changes from the RTL sources, workspace login with data synchronization is needed before simulation. When simulation is ready, output log is available in the simulator workspace. The output is same as in hardware tests so the same functions can be used to parse the result in Jenkins build.

Jenkins jobs are built as a generic Jenkins user. This means that a private SSH key has to be given to the user in order to access VCAD account without login prompt. Basically everyone can login as a generic user so the idea of giving an access to personal VCAD account was not reasonable. Another solution was to use personal user account from local servers to handle communication between Jenkins and VCAD. This was not optimal solution either, because tieing a certain user into automated process is not reasonable in long-term. However, this was the least bad option and simulations were done using this way before a better one was found.

Different testing cycle is used for the simulations and tests with the real chip. Simulations consumes a lot of computing resources and it can take several hours to execute all the tests compared to a few minutes duration with the real chip. Because of this, simulations can not be executed on each commit. One simulation in a week was found for a appropriate testing cycle, which means that simulation is scheduled to run once a week in the weekend when it does not cause too much load for the resources. However, in order to get results from the simulation as soon as possible, time to time it has to be checked if the results are ready. Block diagram in Figure 28 shows the simulation workflow.

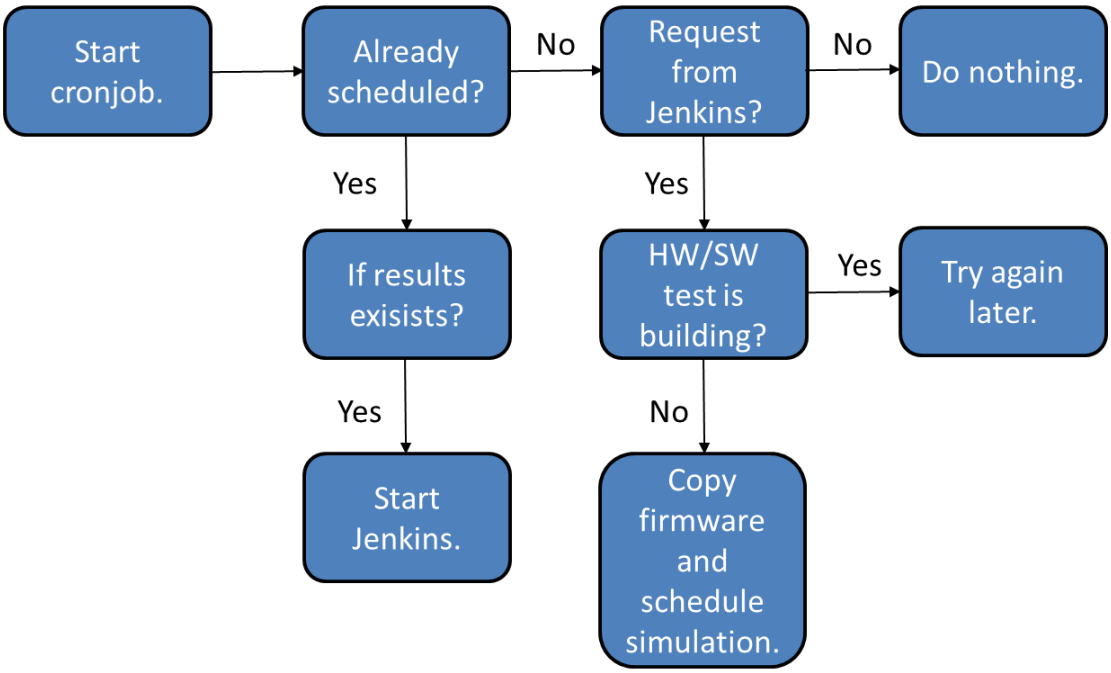


Figure 28. *Simulation workflow.*

Linux cronjob is configured to poll the simulation requests and results. Each Friday new request from Jenkins is sent and new simulation is scheduled. User can also manually start the Jenkins build and send a simulation request. Cronjob schedules a simulation over SSH connection by running a specific script in the remote server. Basically script loads the workspace, synchorinizes the RTL files and starts the simulation. When simulation is running, cronjob in the local server tries to copy the result log from the remote back to local server. If copying results exit code 0 and result file is not empty, simulation is ready. Then result log can be removed from the remote sever and Jenkins build can be triggered for parsing. If exit code is 1, simulation is not ready yet. Jenkins is also running on different server so the build command has to be given over SSH connection.

The purpose of the simulations is not to verify latest changes in the firmware, because it is done with the real chip every time someone makes changes to the master branch, but to verify latests changes in RTL sources to detect possible bugs in hardware design. Some of the tests are dependent on analog parts of the chip and that library is not included in the simulations so these tests are skipped in the simulator.

Because we are running simulations only once a week, it does not matter exactly after which commit the firmware image is built. This means that the simulated image can be built in the same workspace by using same *HWSW_test* Jenkins build used for the chip image. Because cron job is copying simulator image from the *HWSW_test* workspace, concurrent builds are prevented. This means that when the *HWSW_test* Jenkins build is

compiling the image, a lock file is generated to the workspace so that the cron script can not access to the same workspace.

5.3 HW/SW test framework

HW/SW testing framework is an embedded test framework for hardware-dependent software components in u-blox receiver firmware. Its basic intention is to test hardware drivers and hardware dependent software with a real receiver and report the test results. Basically the example test case in Figure 29 uses device drivers or other hardware-dependent component and checks its result or triggered action.

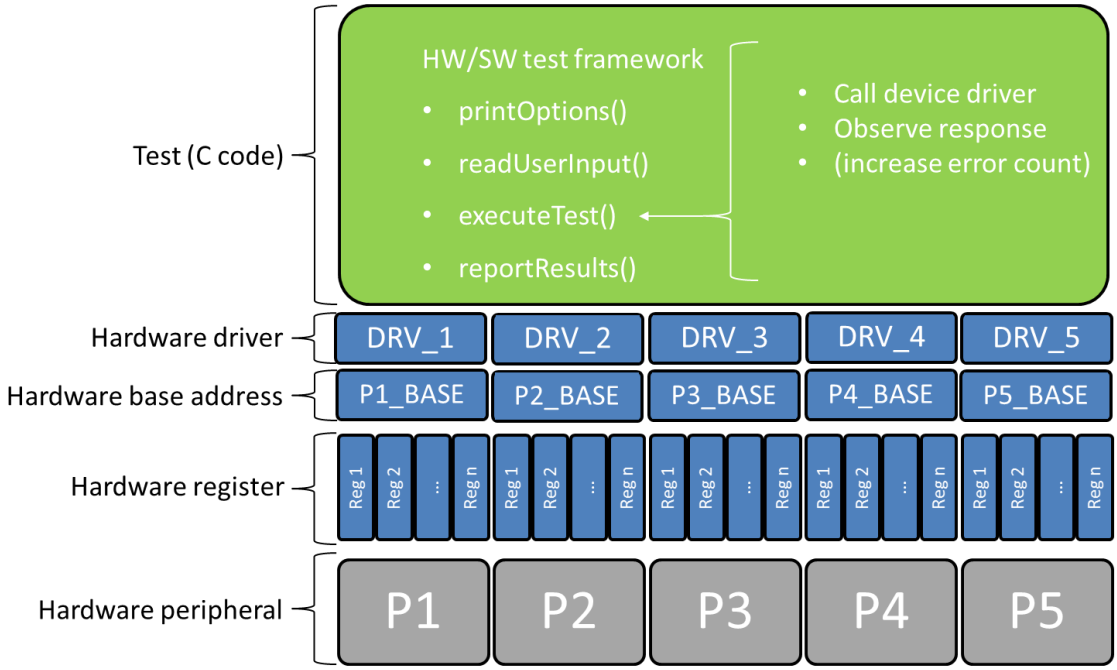


Figure 29. *Generic HW/SW test.*

5.3.1 Test cases

In this work, only a few tests were implemented in the framework as an example. The idea is that developers would implement a test case at the same time they are developing some new feature or performing some verification task. Test cases are listed in the framework in an data structure array. Data structure used in the test framework is defined in Program 4. Some of the fields are used only in the HDL tests, but it is reasonable not to remove them in case they are needed in the HW/SW tests later. These fields are *ioBuff* and *debuggerSim*. *ioBuff* is set to true if we want to buffer UART communication instead of straight printing while *debuggerSim* is true when we want to simulate random memory access by the debugger.

```
typedef struct TST_TEST_DESCRIPTOR_s
{
    const CH* const              kpkTestName;
    TST_TEST_FUNC_pt const       kpTestFunc;
    const TST_TEST_FUNC_PARAM_t kTestParam;
    U4                           errorCnt;
    U4                           perRstMask;
    U4                           perClkMask;
    L2                           skip;
    L1                           ioBuff;
    L1                           debuggerSim;
#ifdef HWSW_TEST
    L1                           manualTest;
#endif //HWSW_TEST
} TST_TEST_DESCRIPTOR_t;
```

Program 4. *Test structure.*

Meaning of the structure is following:

- *kpkTestName:* Name of the test case.
- *kpTestFunc:* Pointer to the test function.
- *kTestParam:* Test parameter.
- *errorCnt:* Test errors.
- *perRstMask:* Mask indicating the peripherals to be reseted before testing.
- *perClkMask:* Mask indicating the clocks to be enabled for the test.
- *skip* Skip this test.
- *ioBuff* : Buffered IO for UART.
- *debuggerSim:* Simulate debugger access.
- *manualTest:* Test which need external hardware to be performed.

kpTestFunc is the test function which performs the test for the software component. *errorCnt* is 0 by default and it is increased in the test function in case of errors in the test. *manualTest* field is added to the structure in this work. Meaning of this field is to separate manual tests from the test which are executed sequentially in the test automation. These kind of tests need some external equipment, such as oscilloscope or additional power source when performing the test.

Like mentioned before, the meaning of the test cases is to check functionality of the device drivers and hardware-dependent components by calling their functions and checking the response. One example of a tested driver is for analog to digital converter (ADC). ADC can be used to measure several different analog inputs and many of them have some documented predefined value. Program 5 pseudocode shows the basic structure of the ADC test case.

```
unsigned int test_function(kTestParam)
{
    //kTestParam is defined in the test description. Unused in most of
    the tests.
    UNUSED(kTestParam);

    initialize_ADC();

    for(value_to_be_measured < number_of_values)
    {
        result = start_Measurement(value_to_be_measured);

        if(result < minValue || result > maxValue)
        {
            print_value();
            errors++;
        }
        else
        {
            //Measurement passed!
        }
    }

    reportErrors(errors);

    return errors;
}
```

Program 5. *Simplified structure of the ADC test case.*

In the ADC test all the possible analog values available are measured and they are compared to the threshold values. If measured value is under or over the threshold value, error count is increased. Program 5 shows only a simple example how the test is done. In the real case, measured values are also used in the calculations afterwards. Calculated results are then compared to the guideline values.

Sometimes testable software component in HW/SW test may have unwanted dependencies to the software layers above. To avoid unnecessary files in the compilation, software stubs are introduced for the testable modules as can be seen in Figure 30. Their intention is to overwrite the functions which module is dependent on and they won't cause any actions. Usually depencies are related to operating system functions.

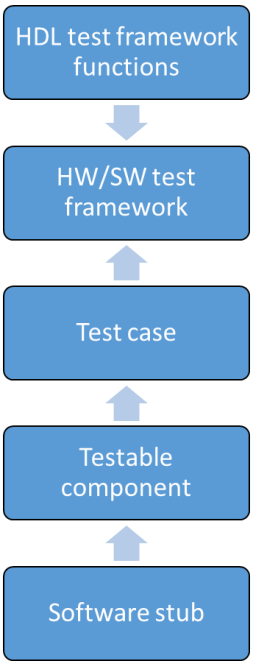


Figure 30. *HW/SW test dependencies.*

5.3.2 Main function

C execution starts at the image entry point. Boot process sets up all the hardware blocks to get the minimal system running, this is followed by the power on self test where all the memories are checked with other functional checks. Before execution is moved to the C main function, interrupt controller and A/D converter are initialized.

Program 6 shows the basic functionality of the hdl main function. Main file is relatively large, mainly because it includes test descriptions for almost 200 Hdl tests and for HW/SW tests implemented so far. However, the following code snippet gives an idea how the tests are performed. Functionality is described briefly in pseudocode so a lot of details and variables are dropped off.

```
int main()
{
    if(functionPointer != NULL && MagicWord)
    {
        for (testNumber < numberOfTests)
        {
            if (functionPointer == testNumber.kpTestFunc)
            {
                found = true;
                break;
            }
        }
        if (found)
        {
```

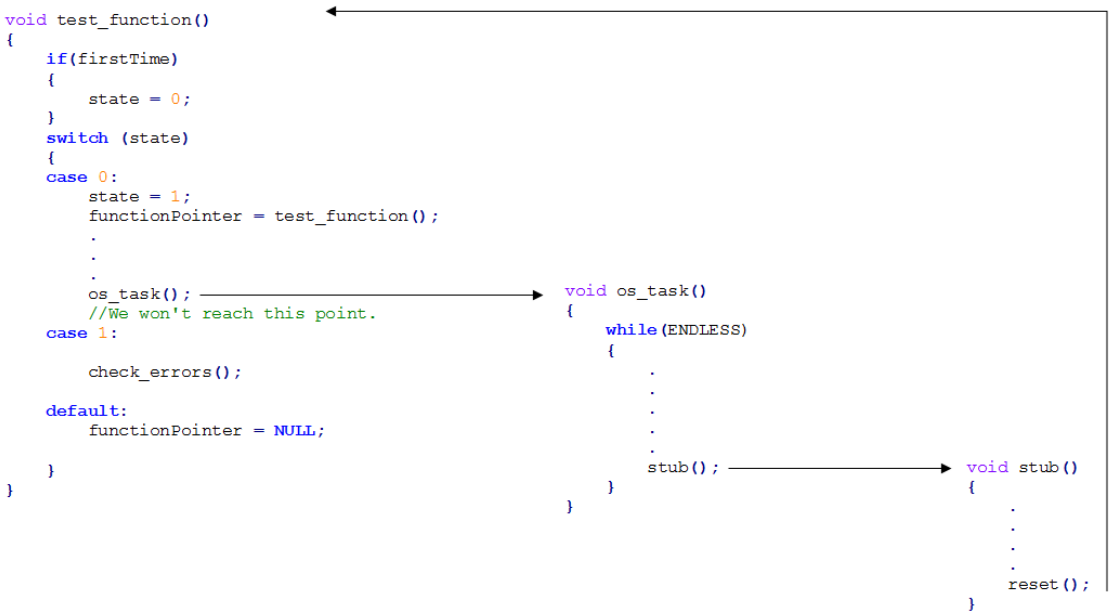
```
            //Reset was done in the test so continue execution and in-
            crease testnumber.
             executeTest(testNumber.kpTestFunc);
             testAfterReset =  testNumber + 1;
        }
    }
    while (1)
    {
        //Display options. User can choose all tests or a certain test
        to be executed.
        display_options();
        //Read user input.
        test = io_get_character();

        for(testNumber < numberOfTests)
        {
            if (testAfterReset != 0 && runAllTests)
            {
                //If some test case had a reset, we don't want to con-
                tinue from the beginning.
                testNumber = testAfterReset;
                testAfterReset = 0;
            }
            if (testNumber != test && !runAllTests)
            {
                //We are looking for a certain test case.
                continue;
            }
            if (testNumber.manualTest && runAllTests)
            {
                //We are running all test cases, but we want to skip
                manual tests
                continue;
            }
            if (testNumber.skip)
            {
                //Skip this test.
                continue;
            }
            else if (testNumber.kpTestFunc)
            {
                //Execute test.
                executeTest(testNumber.kpTestFunc);
            }
            else
            {
                //No such test case.
            }
        }
        //Print test results.
        printTestSummary();
    }
}
```

Program 6. *Main functionality pseudocode.*

Some of the tests may include resets, for example when testing wake up from the sleep mode. In this situation C execution starts from the beginning and the test procedure will not reach the end. To avoid this situation, executed function is stored into pointer and

with a certain *MagicWord* function execution can be continued from the point where the reset occurred. Function pointer and the *MagicWord* are stored into hard coded memory regions of the battery backed random acccess memory (RAM).

Sometimes a reset is a wanted action. For example if we are testing a operating system task which is an endless loop, we need to have some way how to get back to the test function. This situation can be seen in Program 7.



Program 7. *Reset in a test case.*

Resets also have an effect on testresults. Like mentioned above, *errorCnt* is increased in case of test errors. However, if some test includes a reset, results of the previous tests are reset to zero. For this situation, a memory region from the battery backed RAM is reserved for a result mask. Result mask is 8 byte long integer and each of its bits means one test case. If error occurs in the test, corresponding bit is set to 1 from the mask. This does not tell if there was more than one error in the test, but it gives an infromation to the user that something failed. Program 8 shows how the results are stored to the bitmask and how they are read from the mask. Storing is performed after each test and reading before test summary is printed.

```
//Store
if(testDescList[testNum].errorCnt != 0)
{
      *results |= (1 << testNum);
}
```

```
//Read
for(i = 0; i < NUMOF(testDescList); i++)
{
      if(*results & (1 << i))
      {
            testDescList[i].errorCnt = 1;
      }
}
```

Program 8. *Store and read the results from the mask.*

As can be seen from the Program 6, existence of the function pointer is checked in the beginning of the main function. If pointer is not null, test execution is continued in the test before reset and information of the next test in the test description array is stored. Otherwise the options are printed out. The most important options are described in the Chapter 4.1. A lot of details are dropped off from the pseudocode, but basically the test description list is iterated through: if user chooses a certain test to be performed, rest of the tests are ignored. When user chooses all test to be executed, only manual tests are ignored. When list iteration is completed, test summary is printed out.

5.4 Work stages

Different parts of the test framework are now introduced in the previous chapters. Now we can look closer in which order these parts are implemented. A new image for HW/SW test was already implemented before this work. So the work began with exploration of the firmware part of the testing. When HDL test framework became more familiar, adaptation of the ReleaseTestGUT could be started. Basically the first goal was to be able to run an empty HW/SW test from the Linux command line interface. When this was ready all the same things could be done by automated Jenkins script.

The second goal was to get fully automated test process where Jenkins script is polling the firmware master branch in version control and launching the tests. In this step the HWSW_test Jenkins project was created and a appropriate project workspace for Git repositories was allocated from Thalwil servers. After this step was done, we had a Jenkins task interacting with the receiver by using the ReleaseTestGUT. The test process was working, but any test cases was not embedded in the firmware. The next phase was to add a few test cases and adapt the embedded test framework in a needed way. Now also other developers started to include new test cases in the testing.

Basically testing with the real chip was now working as expected. The last changes in the firmware part was separation of the manual and automated tests. Now also Raspberry Pi was connected to the test site when it was noticed that the receiver may get stuck, because of inoperative test case.

After testing process with the real chip was functional, simulation automation could be started. In the beginning of this phase, an user account was created to VCAD environment and the latest revision of the chip RTL design was checked out to the user workspace. The first goal was to be able to start the simulations without user interaction. Then the same could be done from the local servers over SSH connection. Another Jenkins task and communication with the remote server was then implemented.

Rest of the work was used for writing this documentation and maintaining the systems. Some small modifications was done in the framework if deficiencies was found.

6. EVALUATION

This chapter evaluates the implemented work. Some statistic for the testing and discovered bugs are also introduced. The workload this far and features which could be done in the future are also considered.

6.1 Statistics

The testing environment is new and in the beginning there is only a few tests included in the automated regression testing. Because of this, it is not reasonable to evaluate benefits of the system by counting found bugs. The testing environment was launched in the beginning of November 2017. At that time only three tests were included in the embedded test framework and the test part of the Jenkins build was only a couple of seconds. As can be seen in the Figure 31, building times in Jenkins were approximately 3 minutes in the beginning. Build time is dependent on several things: amount of changes in ReleaseTestGUT and Codename repositories, network traffic between Thalwil and Tampere sites, connection issues and of course number of tests. Build time varies a lot because of previous reasons, however linear approximation shows that build time has increased almost 2 minutes.

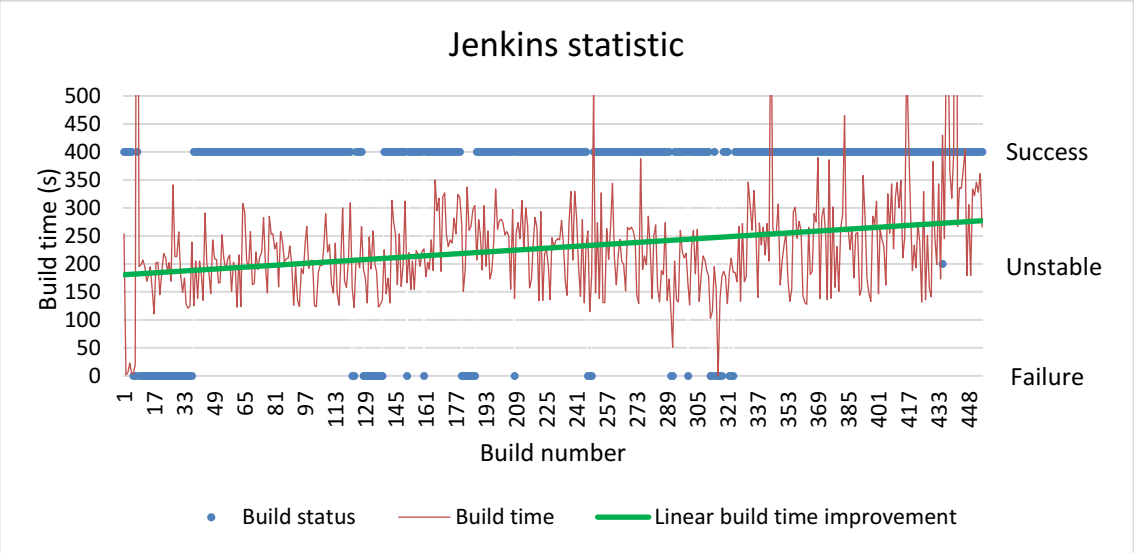


Figure 31. *Jenkins statistic.*

In the end of January 2018, total number of test cases was 17 wherefrom 13 were included in the automated regression testing. The last four test were for manual testing.

Even though the build time has increased almost 2 minutes, test execution part of the build has increased only approximately 15 seconds. So the build time has increased mainly because of grown image size. When tests are implemented for new components, new files are included in the compilation and therefore compilation of the firmware takes longer and so does the firmware update.

Blue markers in the Figure 31 are showing status for each Jenkins build. Like mentioned before, it is not reasonable to evaluate benefits of the system by counting the found bugs in the software, because amount of tests is still relatively small. This is the reason why almost all the builds are successfully finished. Most of the failures are caused by the development of the system or network connection issues between Thalwil and Tampere sites. Connection issue was handled in the Jenkins script by starting the release test again if the first attempt fails. In these cases, the reset is also triggered by the Raspberry Pi.

However, a few issues were detected by the automated regression testing. Sometimes only the image was broken and failure was detected already in the compilation, but some issues were also detected by the HW/SW tests. As can be seen in the Figure 31, several failures were detected by the system in the beginning. The same bug was causing all the Jenkins build failures and it was detected already in the test development phase. The failure was in the chip prototype and the fix was done to the next version of the chip. Meanwhile the hardware bug was taken into account in the HW/SW test so that failures are ignored.

From the beginning of November to the end of January, there was approximately 450 builds in the Jenkins. During this time, only a few issues were found in the actual testing environment so it is relatively maintenance free system. Unfortunately same amount of metadata was not available for simulations, because they are executed only once a week and simulation part of the work was not completed until end of December. But it can be mentioned even without testdata that automated regression testing in third party simulation environment is much more unstable than with the real chip. However, it still decreases amount of work used by the developer compared to old procedure where user had to build and copy image manually to the server where user logged in afterwards to start the simulation and observe results.

6.2 Future work

In this kind of work, it is difficult to say when it exactly is ready. More tests can be implemented to cover as much code as possible. Some new test may have an affect on the system and it may need some improvements. And of course, all the systems need maintenance every now and then.

Some features done in this work were not optimal. The reason was not in the actual implementation, but in the surrounding environment and tools. The simulation part of the work is an example which could have been done differently compared to this solution. This solution is tied into certain user and the developer does not really have a clue what is happening in the simulation environment. If the hosted VCAD environment will some day be moved from the service providers servers to the local in-house servers, it is easier to implement a system where Jenkins generic user is accessing straight to the simulation environment and no user privileges are needed.

In this work, a new test site was implemented in the Tampere office laboratory and its description was added into ReleaseTestGUT. Own test site was mandatory, because the implemented test case is dependent on existing hardware. It would have been reasonable to implement a universal test case so that it could have been used also in common test sites. Then the test would follow the process described in Figure 32.

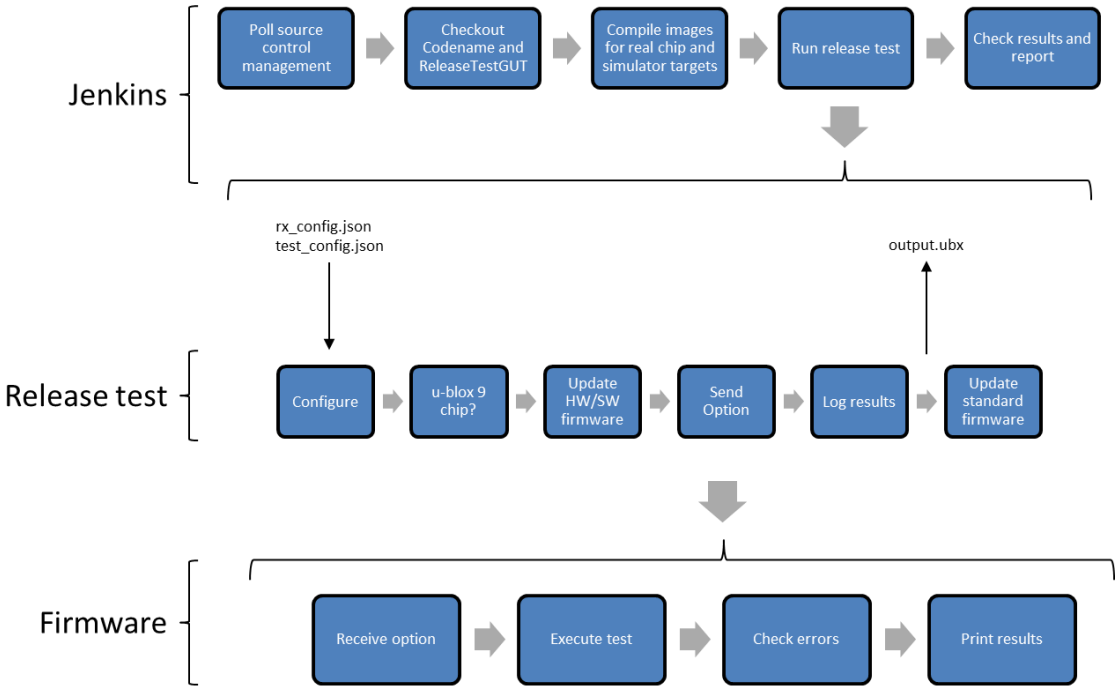


Figure 32. *Alternative test process.*

If HW/SW tests are excecuted in common test sites, we have to be convinced that the right receiver is attached to the test site. It is assured before firmware update. Like mentioned before, UBX communication is not included into HW/SW image. In order to use the same receiver for other release tests, the standard receiver firmware has to be updated in the end of the test so that the receiver is able to communicate with the next test.

Even if the test case was not dependent on the test site, HW/SW release test still cannot be executed in common test sites. Basically release tests are scheduled from the graphical user interface and the tests are executed sequentially based on the given priority. At the moment, there is not an easy way to set release tests to queue from the command line interface. If release test is performed from the command line interface, the test is performed immediately and if the test site is occupied, the test will fail.

If HW/SW release test is used in common test sites, some more changes is needed in the ReleaseTestGUT in order to queue jobs from command line interface, so this option was dropped out of consideration. At the moment the easiest way to run HW/SW release test in different location is to make an indentical copy of the original site description and to rename it and change the IP-address of the UART switch. Other things that can be considered in future work are:

- Part of the larger continuous integration system.
- FPGA-based test site.
- Interface testing.
- New tests.

If added to the larger continuous integration system, it has to be considered if only status of the HW/SW build is passed to the larger system or if the release test is performed straight from the larger system. FPGA based platform could be used to run tests on the hardware model and Raspberry Pi could be used in the interface testing.

One option which could be considered in the future is to include analog libraries into simulations. Some of the testable components are dependent on the analog parts of the chip and so far these tests are excluded from the simulations, because analog libraries are not included. Of course this would have an major affect on computing power usage so these tests would not be performed as a regression.

When this work was implemented, u-blox 9 generation GNSS chip was under development and HW/SW testing image was defined for that target. When the next generation chip development starts and driver implementation for new IP-blocks begins, this testing framework can be reused by defining a new target for the firmware.

6.3 Workload

Amount of work was not too big in this work when looking at the total number of codelines in the Table 1. Amount of lines does not include scripts implemented for support or additions in the already existed files. Even if the total number of codelines is relatively small, different programming languages are used in the different environments which increased the workload. Basic knowledge of the receiver firmware,

Jenkins, ReleaseTestGUT and simulation environment was essential before they could be combined.

If we are looking closer where the code is located, C code is used in the receiver firmware, Perl mainly in ReleaseTestGUT, Groovy in Jenkins pipeline scripts, Shell scripts in interaction between simulation service provider and in-house servers, and Python in Raspberry Pi application. So considering how many new environments were intruduced during this work, the workload was appropriate and a clear division into different fields made actually the whole work more interesting.

Table 1. *Amount of code in this work.*

Language	**Lines (appr.)**
C	1200
Perl	600
Groovy	350
Shell	150
Python	50
Total	2350

So all in all, most time consuming part of this work was to introduce to the different tools and testing environments. When basic understanding of each tool was at sufficient level, combining of them was more or less fluent.

7. CONCLUSIONS

The main objective of this work was to implement an automated test environment for hardware drivers and hardware-dependent software components in u-blox receiver firmware. In this work we became familiar with the testing in u-blox: different testing environments and tools were introduced and we looked closer which part of the system each environment is testing and how the they are used in this work.

More detailed objectives for this work were to adapt the embedded test framework in the new HW/SW firmware image for hardware related software testing and a few test cases. In addition a new test site, its configuration and test automation both with real hardware and in simulator were the objectives. All these were implemented and in addition the Raspberry Pi receiver boot controller was a bonus task. Each step of the test framework implementation and used hardware was introduced in this document.

The new firmware image for embedded test framework was already implemented in the beginning so this work began by exploring firmware part of the test architecture. After firmware and test framework usage became more familiar, a new test site and essential additions were done in the system test environment. When changes in the system test environment were ready, HW/SW test with the real chip could be triggered from Linux environment. This allowed the next step where tests could be done automatically each time new changes in the receiver firmware occurs. Test automation was done with the Jenkins continuous integration tool. In this phase we had a fully functional test process, but no tests to be executed. Next step was to add some test cases to the embedded test framework in the firmware and adapt the framework for the tests. Test cases were added by other developers, too.

When the test environment for the real chip was ready, test automation for RTL simulations was the next step. Communication between in-house and simulation servers was established, and a second Jenkins task to schedule simulations and report the results. Even if simulation automation was implemented succesfully, there is some future work and optimization what can be considered.

17 test cases were implemented to the embedded test framework in approximately 3 months. In this time test development and automated regression testing detected several issues such as hardware bugs, broken image or questionable changes in receiver firmware. So all in all, the new testing environment implemented in this work found out functional and useful in hardware-dependent software development. In the future it can be also used in next generation u-blox GNSS receiver chip development.

REFERENCES

[1] "u-blox Product resources," [Online]. Available: https://www.u-blox.com/sites/default/files/NEO-M8P_DataSheet_%28UBX-15016656%29.pdf. [Accessed 10 January 2018].

[2] K. Popovici and A. Jerraya, "Hardware Abstraction Layer—Introduction and Overview," in *Hardware-dependent Software*, Springer Netherlands, 2009, p. 299.

[3] "ARM Developer Documentation," [Online]. Available: https://developer.arm.com/docs. [Accessed 22 February 2018].

[4] A. Mili and F. Tchier, Software Testing: Concepts and Operations, John Wiley & Sons, Incorporated, 2015.

[5] S. Yoo and P. Runeson, "Guest editorial: special section on regression testing," *Software Quality Journal,* vol. 22, no. 4, pp. 699-699, 2014.

[6] "Jenkins User Documentation," [Online]. Available: https://jenkins.io/doc/. [Accessed 1 January 2018].

[7] "VCAD Services," Cadence, [Online]. Available: https://www.cadence.com/content/cadence-www/global/en_US/home/services/vcad-services.html. [Accessed 1 January 2018].

[8] "SimVision Debug," Cadence, [Online]. Available: https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/debug-analysis/simvision-debug.html. [Accessed 2 January 2018].