Bachelor's thesis

Information Technology / Embedded Software

NTIETS13S

2017

Mikael Laine

DESIGN AND IMPLEMENTATION OF A TEST ENVIRONMENT FOR RFIC FIRMWARE

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information Technology / Embedded Software

2017 | 31

Mikael Laine

DESIGN AND IMPLEMENTATION OF A TEST ENVIRONMENT FOR RFIC FIRMWARE

The goal of this thesis was to create a test environment for testing firmware on physical samples of an RFIC under development. The test environment was required to have support for automation and have possible uses with other test tools.

The test environment needed to communicate with the RFIC using a DigRF v4 interface in order to emulate the real environment where the RFIC will finally be used. In order to communicate in DigRF v4, the Keysight M9252A host adapter was used and controlled programmatically with its provided interface. The support for automation was implemented using Robot Framework, a test framework written in Python. An external library was created in order to support usage between Robot Framework and the M9252A.

The final result of this thesis was a working test environment with support for automation. The test environment successfully allows the user to test the firmware, and components of the test environment have been taken into use in other pre-existing testing tools.

KEYWORDS:

Embedded systems, RFIC, firmware, testing, programming

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tietotekniikka / Sulautetut ohjelmistot

2017 | 31

Mikael Laine

RFIC:N SULAUTETUN OHJELMISTON TESTAUSYMPÄRISTÖN SUUNNITTELU JA KEHITYS

Tämän opinnäytetyön tavoitteena oli kehittää testausympäristö RFIC:n sulautetulle ohjelmistolle. Testit piti suorittaa valmistetuilla RFIC piireillä. Testausympäristön täytyi tukea automatisointia sekä tarjota käyttömahdollisuuksia muiden testaustyökalujen kanssa.

Testausympäristön piti kommunikoida RFIC:n kanssa käyttäen DigRF v4 -rajapintaa. Tämä emuloi todellista tilannetta, jossa piiriä käytetään. Mahdollistaakseen DigRF v4 -kommunikoinnin käytettiin Keysight M9252A -isäntäadapteria, jota ohjattiin ohjelmoimalla käyttäen sen rajapintaa. Tuki automatisoinnille toteutettiin käyttäen Robot Framework -testausohjelmistoa. Jotta mahdollistettiin Robot Frameworkin sekä isäntäadapterin yhteensopivuus, kehitettiin ulkoinen ohjelmistokirjasto.

Opinnäytetyön lopullinen tulos oli toimiva testausympäristö, sisältäen tuen automatisoituihin testeihin. Testausympäristöä voitiin käyttää onnistuneesti sulautetun ohjelmiston testaukseen, ja ympäristön komponentteja otettiin myös käyttöön muissa testausympäristöissä.

ASIASANAT:

Sulautetut järjestelmät, RFIC, sulautettu ohjelmisto, testaus, ohjelmointi

CONTENTS

FIGURES

TABLES

LIST OF ABBREVIATIONS

API	Application programming interface
BBIC	Baseband integrated circuit
CI	Continuous integration
CLC	Control logical channel
CLI	Command line interface
COM	Component object model, an interface standard by Microsoft
DigRF	MIPI High speed interface connecting RFIC and BBIC
DLC	Data logical channel
FW	Firmware
GUI	Graphical user interface
HW	Hardware
IC	Integrated circuit
IQ data	Analytic signal
JTAG	Joint Test Action Group, a board boundary scan method
LTE	Long Term Evolution, a standard for high speed wireless communication
RAT	Radio access technology
RFIC	Radio frequency integrated circuit
RX	Radio receiver
TX	Radio transmitter
UML	Unified markup language

1 INTRODUCTION

The assignment in this thesis was to create an automated testing environment for Radio Frequency Integrated Circuit (RFIC) firmware in LG Electronics Finland Lab (LGEFL). LGEFL is a research and development office developing new RFIC solutions.

The tests were to be carried out on firmware running on actual samples of the RFIC. All firmware testing prior to this had been conducted in simulation. The IC testing was to be used in parallel with simulation testing. The IC tests give more reliable measurement data and verify the functionality of the firmware by giving demonstrable evidence.

There were two major use cases for the testing environment, development/debugging and automated regression testing. The test environment needed to enable the firmware developers to test and debug features during development and automate the execution of tests.

The test environment needed to be developed during two phases, a development phase and an automation phase. The purpose of these two phases was to have a rudimentary test environment quickly ready for usage with RFIC samples. This way the firmware developers could begin testing sooner and provide feedback for fixing existing test environment features as well as requirements for new features. The latter phase would focus on automating execution of well-established test cases.

1.1 Requirements

LGEFL required that the test environment would be built with Robot Framework. This was because it provides automation, plenty of community support and has a free license. Additionally, the following features were required:

- maintainable
- easy to use
- configurable
- modular

To make the test environment maintainable it needed to be well documented, have well-structured code and have an ordered filesystem. To make it easy to use, a simple

interface needed to be implemented for which the user could provide feedback that could be swiftly implemented. Configurability meant that the code needed to be able to run on multiple setups without switching between versions. In order to be compatible with other existing tools, the test environment needed to be built modularly so that components could be easily reused elsewhere.

Communication with the RFIC needed to be carried out with a pre-existing tool, the DigRF Host Adapter. Other implementations of the DigRF Host Adapter were available in LGEFL, but none of them had an extensive solution for firmware control and command communication.

1.2 Thesis scope

This thesis focuses mainly on the development of the RFIC firmware test environment. The automation of the tests were simplified to provide a proof of concept of how automation with Robot Framework would work. This thesis is framed in this way due to lack of time for the automation phase. There were several technical issues in the development phase, many of which were external and out of the scope of this thesis.

2 THEORY

This chapter explains some of the underlying theory and concepts regarding this thesis. It explains the goals and methods of software testing and it gives an overview of how an RFIC works when applied to a larger system of circuits. The final part of this chapter explains the tools and the software which were used during this thesis.

2.1 Software testing

The goal of software testing is to eliminate ideally all defects from it and to make sure that the software is actually doing what it was intended to do. Software defects occur due to a variety of reasons, but many of them can be accredited to human error, such as failure to analyze the specifications or logical errors in design and/or implementation. Errors like these can be difficult to notice simply by code review, especially if the system has a lot of complexity to it. (Homes 2013, 1 – 10)

There are many forms of software testing and each of them have different goals. Some tests focus on ensuring that the software meets the requirements of the specification, while other tests examine the compatibility of component's interfaces. This thesis focuses mainly on the method of unit testing and integration testing.

The goal of unit testing is to verify that small components of the software are working as intended. An example measure of a unit in code could be a coded function. The unit needs to pass a set of criteria as well as previous tests that have been moved into regression testing. The data acquired from these tests are then stored for analysis in conjunction with later tests.

Integration tests can have different meanings depending on the system under test. In this thesis, integration tests mean tests that examine the functionality and compatibility of components and use cases define which components are being tested. The specifications of the components and the requirements of the mentioned use cases define the criteria for when the tests are passed.

(Homes 2013, 58 – 66)

Using unit tests in conjunction with integration tests allows the tester to verify that the software is working as intended and that the outcome of a use case test is valid. There are more facets to software testing, but the test environment in this thesis focuses on facilitating unit and integration testing. This is because the development of the tested product is in a stage where basic functionality is being implemented and these functions serve simple use cases.

2.2 Wireless communication

Mobile devices are becoming increasingly more commonplace in today's world, and an essential part of what makes a device mobile is its ability to communicate wirelessly. Several devices are also becoming more dependent on being part of an interconnected system, which only increases the need for wireless communication.

2.2.1 Modem circuitry

Wireless communication is conducted in cellular networks over the medium of radio frequencies. The circuitry involved in achieving such communication is a matter of complex electrical engineering. It involves digital/analog conversion, (de)modulation, amplification and different kinds of signal filtering. However, this thesis simplifies these processes into the systems they create, such as the Baseband Integrated Circuit (BBIC) and the RFIC. Even though circuit design can have an effect on firmware functionality, electrical engineering is out of scope of this thesis.

The modem is a system of circuits that encodes and decodes data into signals that can be transmitted over radio frequencies. Figure 1 illustrates where and how the modem is used in a wireless device.

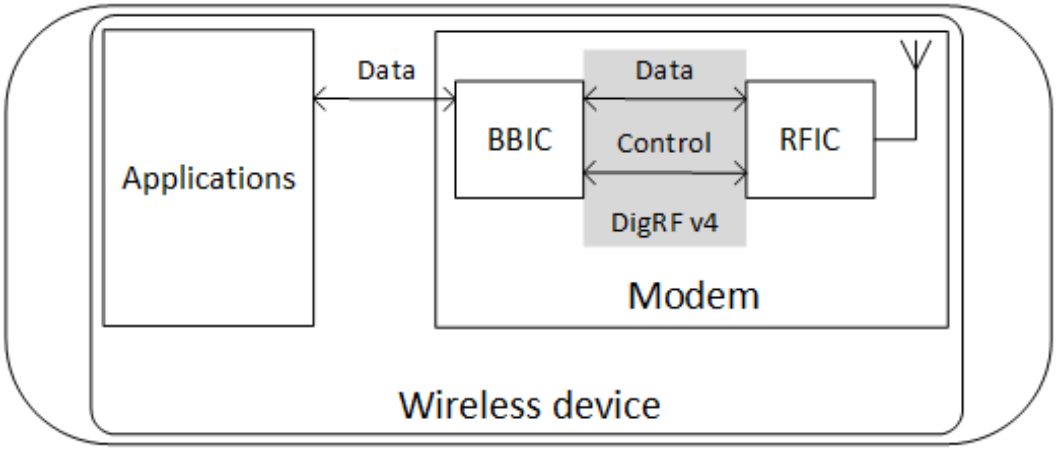

Figure 1. Modem components in a wireless device.

The applications of a wireless device uses the modem to communicate wirelessly. The process of using the modem is complex and has many layers to it, but it is simplified in this thesis to the end goal of sending and receiving data.

The BBIC encodes data into communication frames. These frames are then sent to the RFIC, which further processes them and finally transmits them wirelessly to the cellular network. This communication is bidirectional, i.e. signals that are received are sent to the BBIC and decoded back into data to be used by the applications running on the device.

The BBIC is also responsible for commanding the RFIC to configure itself for different use cases. These use cases are the different types of wireless radio frequency communication that are defined by technical standards, such as LTE. The BBIC also controls the RFIC with operational commands, such as booting up or image uploading.

(Mishra et al. 2015, 254 – 255)

2.2.2 MIPI DigRF Interface

DigRF is a MIPI Alliance working group which writes specifications for an interface between the BBIC and the RFIC. The DigRF working group was formed in 2007 (Wikipedia 2015). The current version – DigRF v4 – is designed to provide a convenient way for developers to implement new radio access technologies, such as LTE, while providing support for older existing ones. To ensure compatibility between different IC vendors, MIPI states the following: “MIPI DigRF focuses on the protocol and

programming used to interface the components, enabling vendors to differentiate their overlying system designs while providing interoperability at the interface level between compliant ICs.” (MIPI Alliance 2017)

Figure 1 illustrates where the DigRF interface is implemented in relation to the wireless device. Figure 2 illustrates the physical layout of a single lane DigRF configuration.

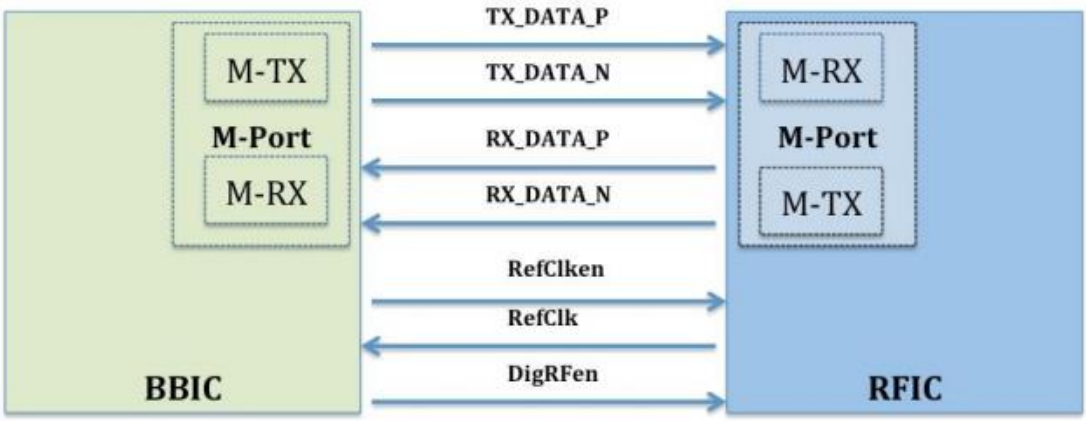

Figure 2. Single lane DigRF layout (Prodigy Technovations 2013).

DigRF uses the M-PHY port, a physical interfaces also created by MIPI Alliance, to transmit data. The interface can optionally utilize up to 3 more pairs of lanes for RX and TX each. The interface has also lanes for clock synchronization.

The DigRF interface uses a protocol for sending frames over the TX and RX lanes. These frames are sent either over the Control Logical Channel (CLC) or the Data Logical Channel (DLC). The CLC frames are used for control data, such as sending configuration commands to the RFIC, and the DLC frames are used for IQ transmission.

(Prodigy Technovations 2013)

2.3 Used technologies

This section gives an overview of the tools and the software that were used when implementing the resulting test environment.

2.3.1 Keysight M9252A DigRF Host Adapter

The hardware that was used to facilitate DigRF communication was Keysight's M9252A DigRF Host Adapter. It is meant for testing RFIC's by emulating a BBIC. It is suitable for development tests, validation and device integration. Table 1 lists the product features.

Table 1. M9252A features (Keysight Technologies 2014).

Feature	Benefit
For RFIC test, simulates a BB-IC	Control and configure the RFIC to match the test setup
Gear 2 (HS2x) support	Test high throughput to the latest MIPI specifications
Multi-link	Support for 1Rx/1Tx at low power or 4Rx/2Tx at Gear 2 high speed
Automation API	Complete programmatic control for complete production testing requirements
Seamless integration with Keysight 89600 VSA software	Immediate access to the industry's broadest, most advanced standards-based demodulation and signal analysis

The M9252A comes with software that allows generation and customization of DigRF traffic. It provides a graphical user interface and a COM API for Windows 7 (32/64-bit) and Windows XP (32-bit). The API can be used to programmatically command the host adapter.

(Keysight Technologies 2014)

2.3.2 Python

Python is an interpreted programming language first released in 1991. An interpreted language means that the program is executed directly rather than compiled in advance into an executable binary. Python is designed to promote the readability of the code by eliminating special characters from its syntax and replacing them with whitespaces. It is

a very popular language and has been in the top ten popularity list maintained by TIOBE since 2003.

Python is known for its extensive catalogue of libraries. At the time of writing there are 106 819 packages in the Python package index (Python Software Foundation 2017a). These packages together with its built-in libraries offer a wide variety of support for web interfaces, testing, networking, documentation etc.

(Wikipedia 2017)

2.3.3 Robot Framework

Robot Framework is a test automation framework written in Python. It uses a keyword-driven tabular syntax for writing test cases. However, it also supports data-driven tests. It automatically produces reports and logs from tests that have been run. Similar to Python, Robot Framework has a large selection of libraries that provide a wide range of features. It also has support for integration into different Continuous Integration (CI) systems.

Robot Framework is application and platform independent. Figure 3 demonstrates the modularity of its architecture.

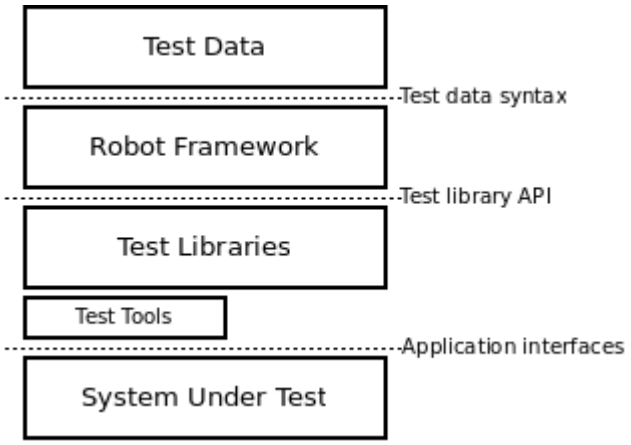

Figure 3. Robot Framework architecture (Robot Framework Foundation 2016).

The test libraries equip Robot Framework with interfaces to the system under test. It does so by programmatically interacting with it, either directly or through other tools. They provide the test writer with keywords that execute certain functions. These keywords are

designed to make the tests easy to understand as it gives the tests a very imperative structure.

Figure 4 illustrates the structure we get when applying Figure 3 to this thesis.

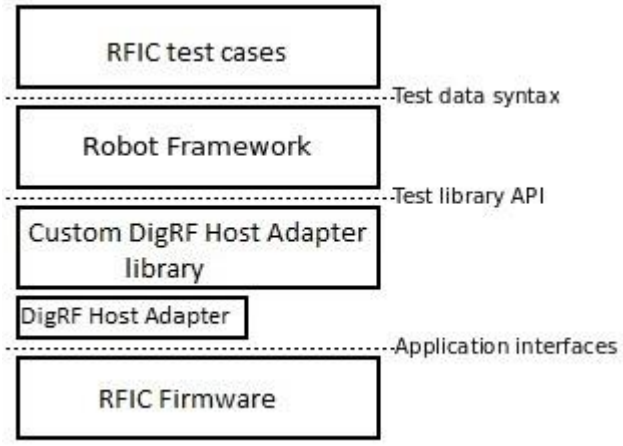

Figure 4. Implementation of Robot Framework architecture.

The DigRF Host Adapter does not have an existing test library for Robot Framework. Robot Framework does however have an API and supports the use and creation of custom external test libraries. An external library can be created for the DigRF Host Adapter by using its own API as mentioned in Section 2.3.1.

(Robot Framework Foundation 2016)

Tests are written in Robot Framework into test cases which form test suites. Test cases should be as stable as possible in order to keep them maintainable. This is because the product requirements are well established and it is merely the implementation that is fluid. A use case should be explainable in a test in a very simple and somewhat abstract manner.

As an example, let us assume we want to test that the system under test should be able to execute task A and B in succession. We would create the following test case:

```
# Test name          Action              Argument
A and B              Execute task        A
                     Execute task        B
```

It is quite rarely that specifications change so drastically that such a use case would no longer be valid. The implementation of how the task is executed might change and thus also change what the keyword `Execute task` would do, but the actual test case stays intact as it is. Figure 5 demonstrates how often a resource in a test system is subject to change.

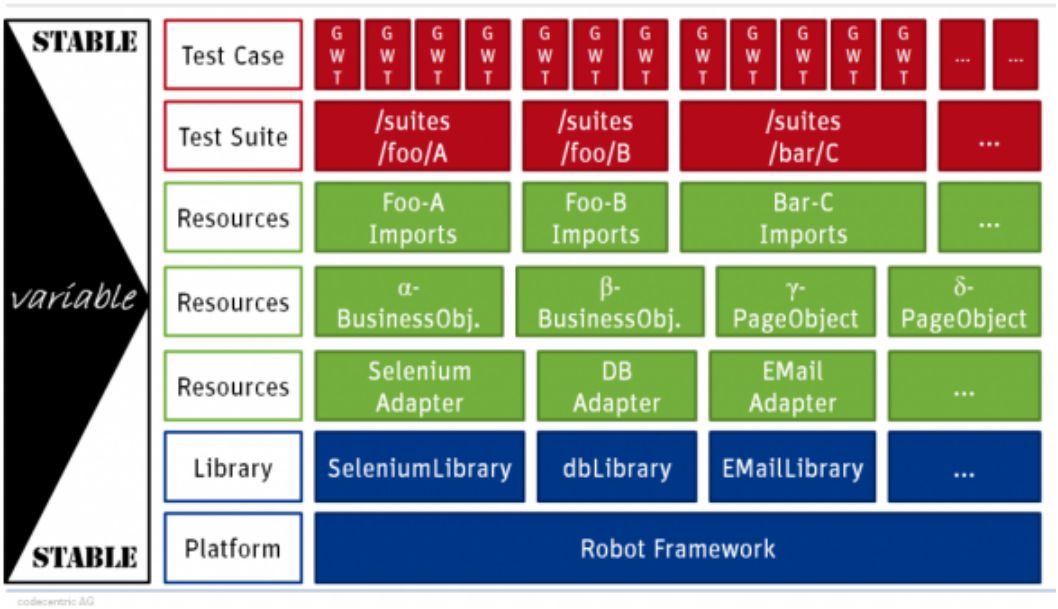

Figure 5. Scalable and maintainable test suite (Ebbert-Karroum 2010).

Figure 5 suggests that the further away a component is from the system under test, the more stable it should be. In the case of this thesis, the DigRF Host Adapter library would be situated in the adapter and library layer of the figure. This means that well established functionality with the DigRF Host Adapter and the DigRF interface should remain stable, while functionality for different test cases should implemented or updated as the need arises.

(Ebbert-Karroum 2010)

3 ARCHITECTURE

This section explains the structure of the final product, i.e. the test environment. A brief overview of the complete system is given and each component is explained in more detail in its own section.

The typical RFIC physical test setup is a computer with external hardware interfaces connected to the IC that is being tested. The test environment is run on the computer by the user. The user can be either a person, such as an RFIC firmware developer, or an automated program as part of a CI system. Figure 6 illustrates where the developed test environment is situated with regards to the physical test setup.

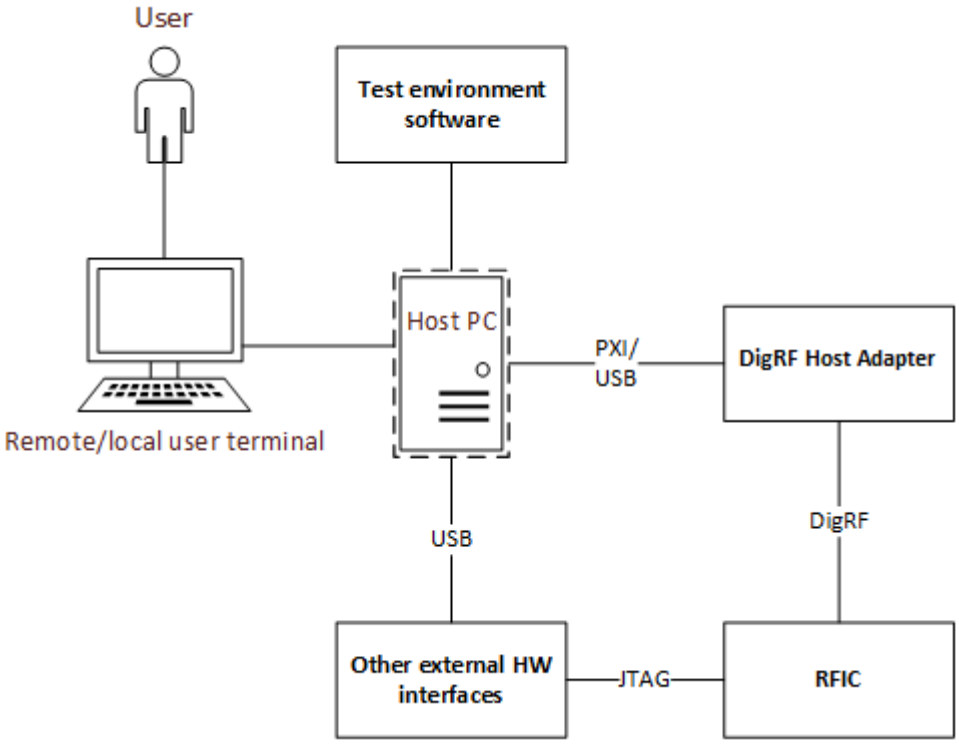

Figure 6. Physical test setup.

The DigRF Host Adapter facilitates DigRF communication on a hardware level. The RFIC is operated by emulating a BBIC on the host through the means of sending commands and IQ data to the RFIC. This is done with DigRF communication by transmitting and receiving CLC frames for commands and DLC frames for IQ data. See Section 2.2.2 for further details.

Other external hardware (HW) interfaces refer to measurement and instrumentation equipment that are connected to the RFIC, other than the DigRF Host Adapter. These interfaces can be implemented into the test environment similarly to the DigRF Host Adapter interface.

3.1 Test environment overview

The test environment is comprised of multiple components. These components together make it possible to use the DigRF Host Adapter with Python and in Robot Framework. The most central component is the DigRFExerciserLibrary as it is the bridging component between the DigRF Host Adapter and the rest of the test environment. Figure 7 illustrates a component diagram of the test environment.

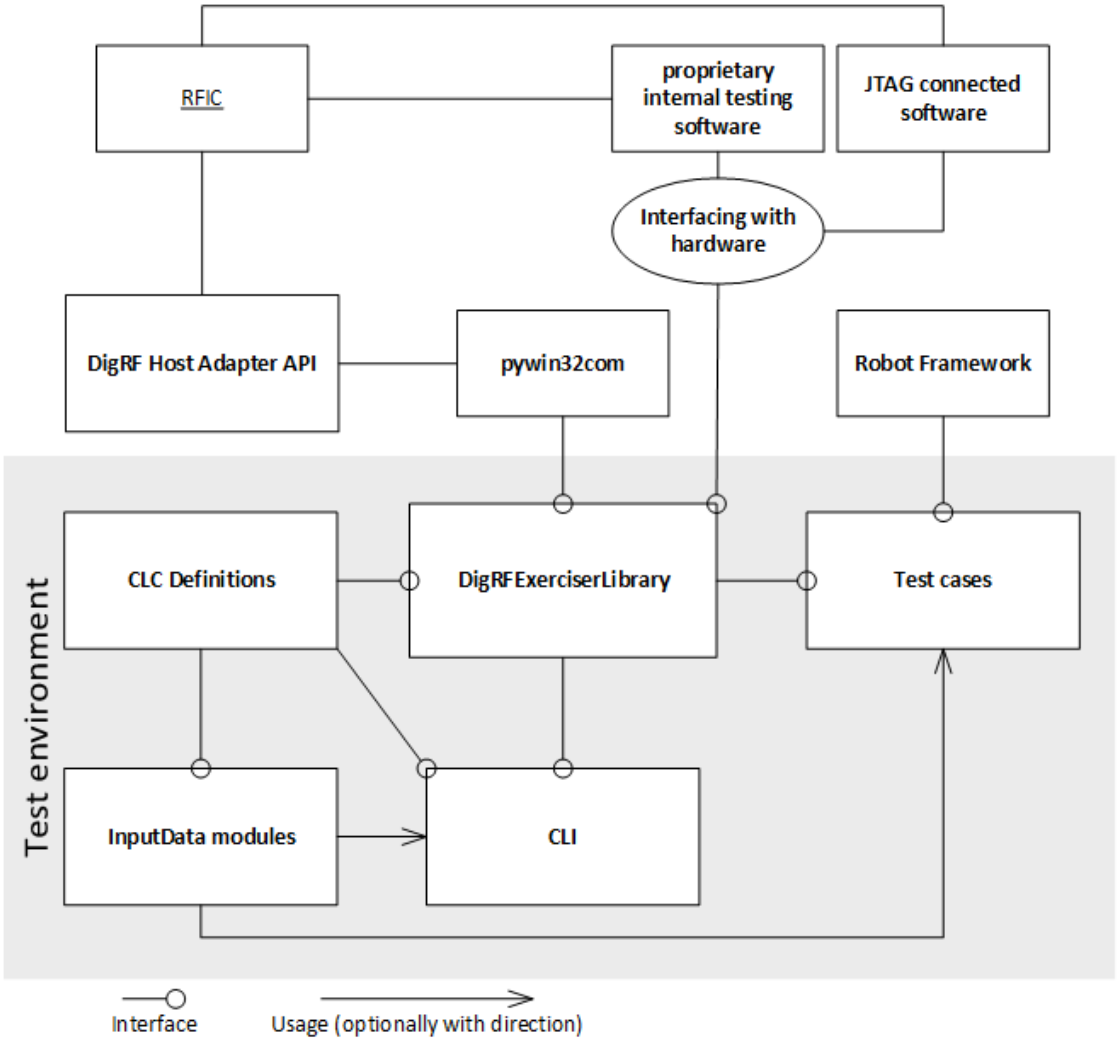

Figure 7. Test environment component diagram.

The components of the test environment developed in this thesis are highlighted with a darker background in Figure 7. The test environment is connected to the RFIC through the DigRF Host Adapter using its API. The API is used with Python by utilizing the *pywin32com* module. The test environment is also connected to the RFIC through other instrumentation devices.

The Command Line Interface (CLI) enables real time usage and self-testing of the test environment. The CLC Definitions and InputData modules are components that allow programmatic representation of CLC frames and its data. The test cases are tests written in Robot Framework compatible syntax.

3.2 DigRFExerciserLibrary

The DigRFExerciserLibrary (later referred as library) is a component that interfaces with the DigRF Host Adapter's API. The purpose of this library was to facilitate DigRF communication with the RFIC and to provide an external library for Robot Framework usage. Communicating with the RFIC with the DigRF protocol simulates BBIC-RFIC communication, as opposed to directly writing values into the RFIC's processor registers through an external interfacing method, e.g. JTAG. It provides a more accurate testing environment as this is the way the RFIC communicates with the BBIC in a modem.

Robot Framework is a test automation tool that supports external libraries written in the Python, Java and C programming languages. The library was written in Python so that it could be used with Robot Framework without any intermediary components, such as Jython, if it were written in Java. The DigRF Host Adapter API is provided as a COM object. To interface with it, the library uses the *pywin32com* module which provides support for using COM objects.

3.2.1 Configuration system

The commissioning organization has multiple HW testing setups at multiple locations that can have many different use cases. These varying environments sometimes have incompatible configurations. To deal with this issue, a configuration system was implemented into the library.

The configuration system is a function that reads an INI-file and configures itself and the DigRF Host Adapter accordingly. The INI-file is written and given by the user. It is a text-based file, which means that it can easily be included into version control tools.

The configuration file contains properties for configuring what type of session is created (e.g. online or simulation) and different link properties (e.g. clock frequency). The correct values for these properties are referenced from the DigRF Host Adapter API documentation. Additionally, miscellaneous configurations are also given in the INI-file, such as paths to software which controls other external instrumentation devices.

3.2.2 Usage of the DigRF Host Adapter API

The library has functions for programming CLC frames into the DigRF Host Adapter's memory, controlling the transmission and reception engines as well as parsing CLC frames from the DigRF Host Adapter's memory. These functions compile the correct workflow so that the DigRF Host Adapter is easier to use from the CLI and Robot Framework. It also stores the sent and received CLC frames into memory so that they can be used later on for verifying information. As an example, the workflow for sending CLC frames is described in Figure 8.

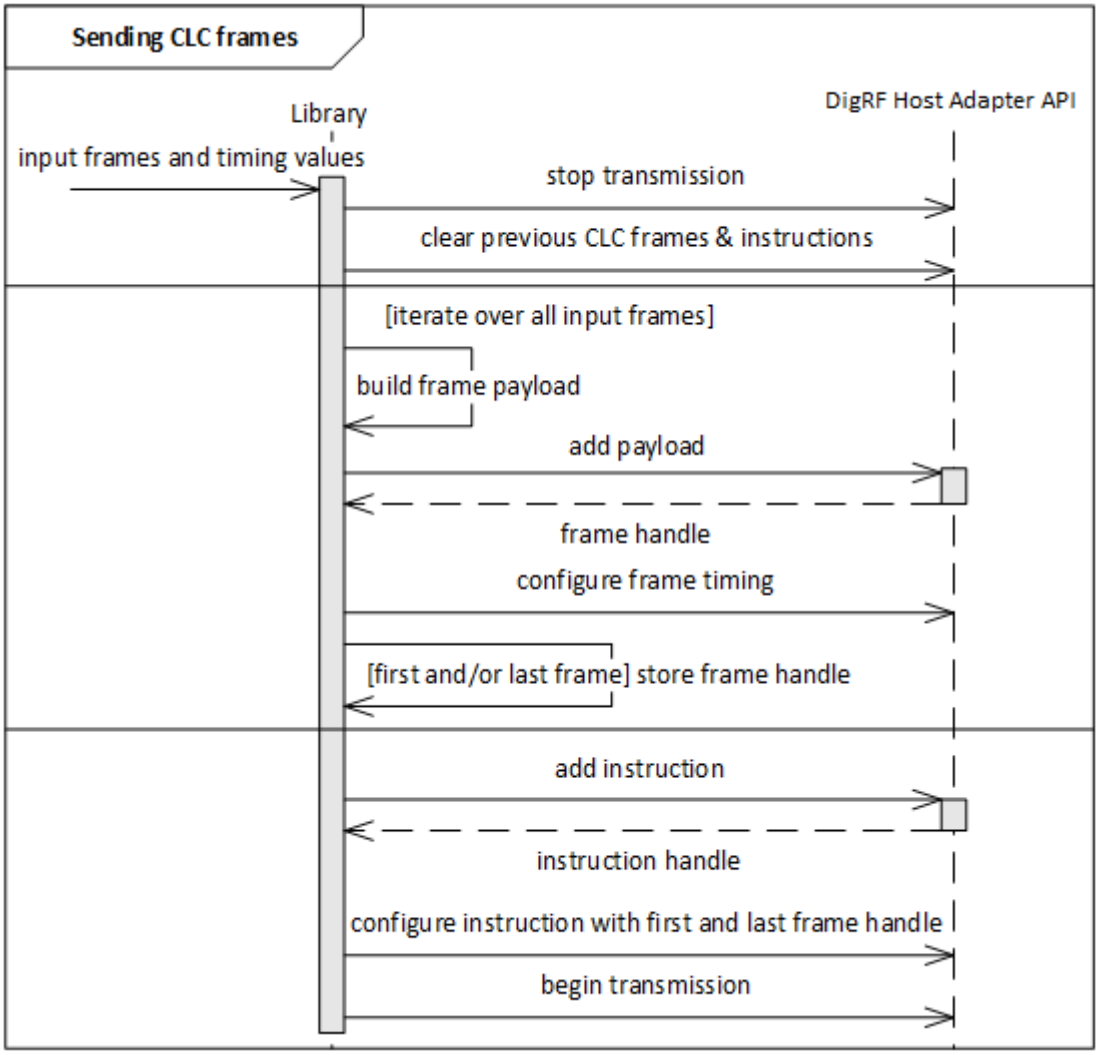

Figure 8. Workflow for sending CLC frames.

When sending CLC frames with the DigRF Host Adapter API, the frames are first programmed into its memory. This sequence of frames are then programmed into a queue, which is called an instruction. Each frame that is programmed into the instruction needs to be configured with a time offset, relative to the previous frame, in order to avoid overlapping frames in a sequence.

In order to send CLC frames with the test environment library, a list of frames and their timing values are given to a function. This function serves as a keyword for Robot Framework. The function iterates over all the given frames and uses the CLC Definitions interface to build a compatible payload (see Section 3.4 for further details on CLC Definitions). This payload is then added to the DigRF Host Adapter, which returns a memory handle for the frame. The library stores the handle of the first and last frame of

the sequence. After all frames have been added and their timing offsets have been configured, an instruction is added to the DigRF Host Adapter. The instruction is then programmed with the first and last frame handles. This will instruct the DigRF Host Adapter to send all the frames found between those handles.

For the capture and transmission of IQ data the library needs to configure the data frames in a similar fashion. These values are static and are based on the type of IQ data which is being sent, such as Radio Access Technology (RAT) and bandwidth.

Besides having functions to interface with the DigRF Host Adapter, the library provides interfacing functions outwards. This feature makes the library modular, in the way that it can be used with other tools. It is easy to implement more functionality, simply by writing more functions to the library. An example of this is explained in more detail in Section 3.7.

3.3 Command line interface

The Command Line Interface (CLI) is used to test the library and to conduct RFIC FW tests manually in real time. It was created to facilitate development of the test environment in a feedback loop as explained in the agile software development methodology (Lui & Chan 2008, 14). The CLI provides a tool for quickly testing the functionality of the library and as such can also be used for prototyping test cases. The CLI enabled the FW developers to begin testing the software on the RFIC samples in parallel with the development of the test environment.

3.3.1 Main structure

The CLI implements the Python *argparse* module. This module simplifies the implementation of parsing command line arguments (Python Software Foundation 2017b). In addition to the in-built help flag, the CLI has arguments for a debug flag, initialization file and which version of the DigRF Host Adapter API to use.

The CLI runs in an infinite loop which parses user input. Figure 9 illustrates the logic of the CLI main loop.

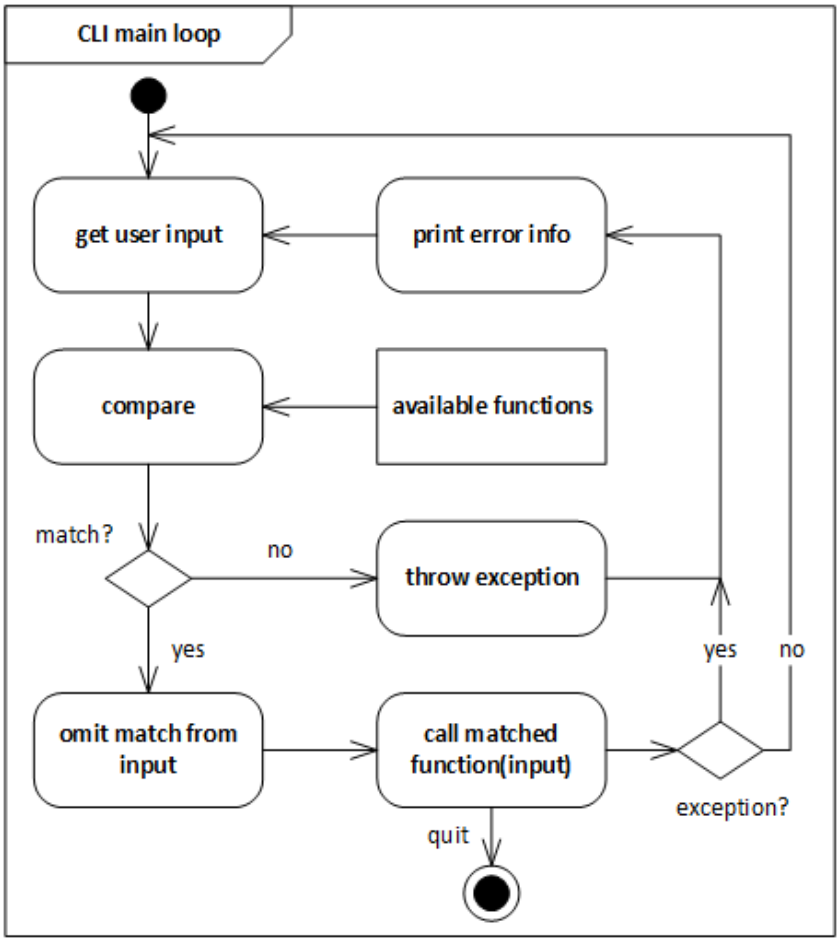

Figure 9. CLI main loop activity diagram.

The input is compared to a list of available functions which is then executed if a match is found. Any additional input is passed on to the function as arguments. The main loop is set to catch all errors so that any tests aren't interrupted. Each component in the test environment has its own exception objects so that it is easier to pinpoint the origin of the error. The available functions are either well established workflows, basic operational commands (e.g. exiting and configuration) or functions for prototyping new functionality.

3.3.2 Available functions

The CLI provides the user the ability to initialize and configure the library, transmit and receive IQ data and CLC frames. The IQ data that is sent are taken from files that are defined by the user with external tools. When the user wishes to send CLC frames, she

specifies a sequence of frames in files. These files make up the InputData modules (see Section 3.5), which implement the CLC Definitions interface (see Section 3.4).

The CLI provides a help function which prints all the available functions. When given a function name as an argument, the inline code documentation, the docstring, is displayed (Python Software Foundation 2001).

Other available functions are used for controlling the transmission and reception engines, clearing memory and displaying statistics and statuses of the DigRF Host Adapter. Additionally, the available functions include basic operations and configuration.

3.4 CLC Definitions

CLC Definitions is a component that implements the structure of the control frames that are defined in the DigRF v4 protocol and in the specifications by the commissioning organization. It provides the test environment the ability to pack and unpack data, and to represent that data in a more human-readable form.

The data that is sent in the payload of the CLC frames need to be provided as raw bytes to the DigRF Host Adapter API. Calculation of the bytes can be quite a laborious task for a human, due to the way data is represented in the payload. Figure 10 shows an example of such a representation.

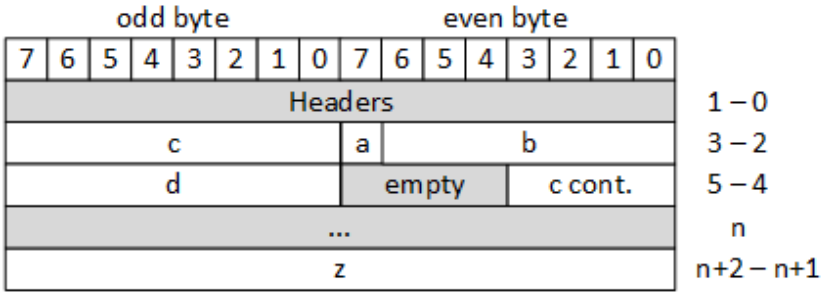

Figure 10. CLC frame data positions.

Each type of control frame has its own header and own set of parameters which are packed in the payload to minimize its size in order to reduce the load on the bus between the RFIC and BBIC. The placements of parameters in the payload are defined in CLC Definitions and are used to map the data into the payload.

As an example, let us consider parameter *a* = 1 and parameter *b* = 28. This would result in the byte at index 2 to be 156 (0x9C). Instead of manually calculating each byte and compiling a payload out of them, the CLC Definitions component allows the user to represent data in each parameter individually. See below for example code.

```
frame.data["a"] = 1
frame.data["b"] = 28
frame.data["c"] = 3000
  ...
frame.data["z"] = 123456
```

A hierarchical class structure was implemented for the different types of CLC frames, in order to represent them clearly and to keep the code maintainable. This structure also automates the process of applying default parameter values to the payload. Figure 11 illustrates how the frames are defined in UML.

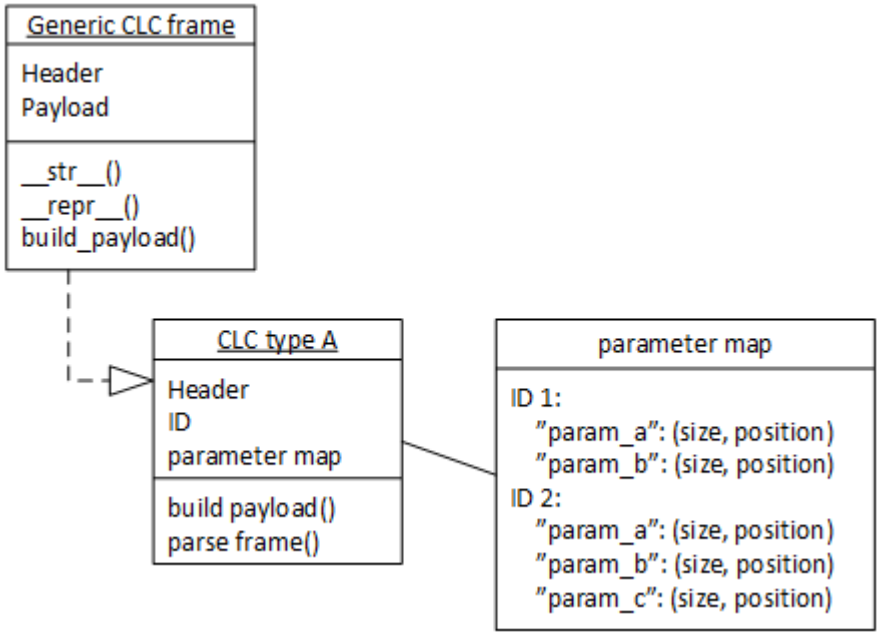

Figure 11. CLC frame class diagram.

The more general parts of a CLC frame are defined in parent classes, such as default variables and functions, while more type specific properties, such as sub headers or the positions of the parameters in the payload, are defined in child classes. The payload building function assigns all the necessary values to the payload and fills in zeroes into

positions where no values were given by the user. CLC Definitions also parses received payloads into human-readable data in a similar fashion, only vice versa.

Some parameters are expressed in the specifications in Q format, i.e., a fixed point format. This is a case by case format in which the developer can assign any amount of bits as the fractional part and as the integer part (ARM Ltd. 2001). The structure of the format can vary between parameters in this test system. A separate Q format type was implemented to provide support for it. Parameters are defined in CLC Definitions with Q format type which can then be assigned with integer and fractional values separately. As an example, let *q_param* be an 8-bit sized parameter with 5 integer bits and 3 fractional, i.e. Q5.3. We would assign the value 13.5 in the following way:

```
frame.data["q_param"] = Qformat(13, 5)
```

Alternatively, the parameter could still be assigned values normally with Python syntax:

```
frame.data["q_param"] = 0b01101101 # Binary
frame.data["q_param"] = 0x6D # Hexadecimal
frame.data["q_param"] = 109 # Decimal
```

3.5 InputData modules

The test modules, which are also called InputData modules in the system overview figure (Figure 7), are Python scripts that contain CLC frames. Since the firmware is controlled by CLC frames, its activity is described by what commands are sent to it, e.g. start receiving data with configuration x, wait 100 ms, change parameter y, wait 100 ms, stop receiving. Typical use cases like this can be defined by writing a specific set of commands in a Python script through the use of CLC Definitions (see Section 3.4).

The files written by the user are called modules. Python defines a module as "a file containing Python definitions and statements" (Python Software Foundation 2017c). These modules serve as a way for the user to store a command or a set of commands as an entity that can be called on to activate certain behavior in the RFIC. The modules could be as simple as sending one ping request, or something with more functionality, for instance turning on transmission with a specific set of configuration parameters.

These modules are arranged in a filesystem to provide an organized structure for easier management of the data. Since the modules define a certain behavior, they can be arranged into different categories. Figure 12 depicts an example of such a structure.

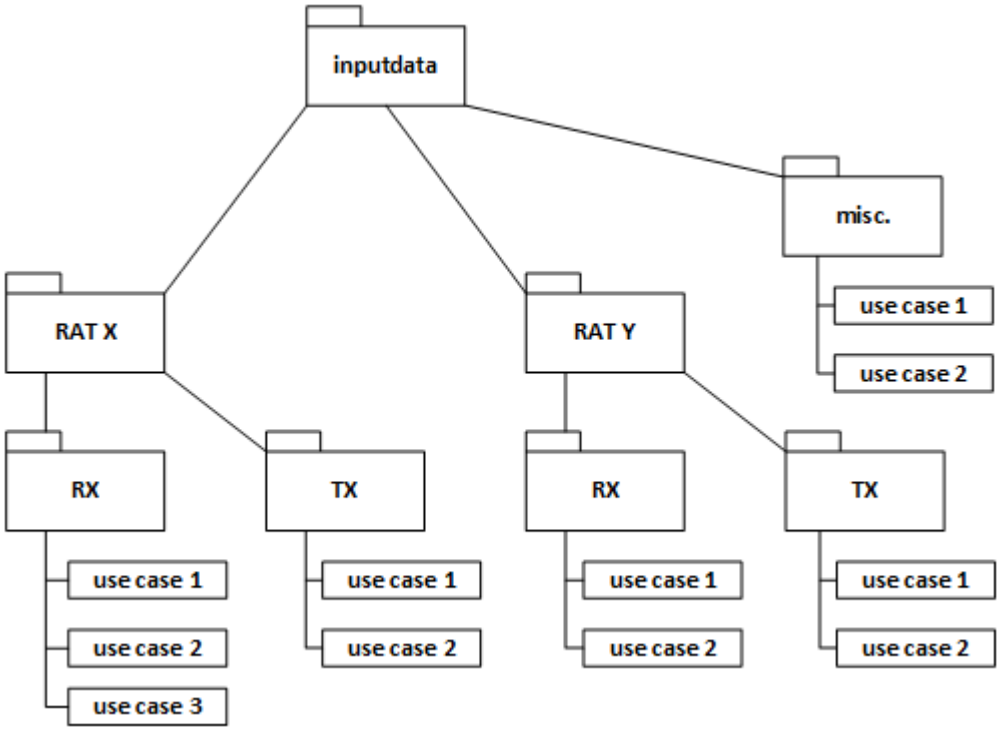

Figure 12. Test case filesystem.

The main folder contains subfolders which define a category. The pictured categories are modeled after pre-existing test cases from simulation testing. Each category can have an endless amount of subcategories, which contain InputData modules. These InputData modules define a sequence of control frames, which in turn define a certain behavior in the RFIC FW. The categories, however, may be subject to change when the testing of the firmware reaches deeper abstractions.

3.6 Test cases

The test cases component in the overview figure (Figure 7) refer to cases written in Robot Framework's syntax. These test cases are then executed as an automated process, such as a batch job or as part of a regression testing setup. When developing a test case, a basic workflow is established using the CLI. The user uses either previously defined InputData modules or writes new ones and goes through a specific process to verify

RFIC FW functionality. As an example, the user might want to do a test to verify that a command is executed successfully even though the previous failed. Below is an example of how such a test case might look like in Robot Framework's syntax.

```
# Test case name    Action              Argument
Failing command     [Documentation]
                    Send command        rat_y.rx.use_case_1
                    Verify response     False
                    Send command        rat_y.tx.use_case_2
                    Verify response     True
```

Test cases were developed as a proof of concept and as a template for further usage. These test cases demonstrate how to automate the following procedures in the test environment:

- send basic CLC commands
- automate the use of external tools in the library
- create task lists for different types of testing (regression, RAT specific etc.)
- self-test, i.e., verify connectivity to RFIC whenever it is reset
- structuring the filesystem into test suites and test cases

3.7 Interfacing with other external tools

The commissioning organization had a tool that used the DigRF Host Adapter but it was designed for sending DLC frames and had a very limited support for CLC frames. A method for using the CLC Definitions and the InputData modules components was added into the test environment.

The external tool makes use of standard streams to read the payload value in hexadecimal of the CLC frame. A script was developed to enable usage of InputData modules in the external tool in a similar fashion to the CLI. The script builds the payload of the given module's frames and outputs their value in hexadecimal. The external tool then reads the output stream and programs the payload into the DigRF Host Adapter's memory.

This part of the test environment serves as a demonstration of the usefulness and re-usability of the components. For future improvements, the external tool could use the parameter mapping values in the definitions to have it display all the parameter values in its GUI so that the user can modify the values on the fly.

4 CONCLUSION

This thesis explained the development of an RFIC firmware test environment by presenting the basic underlying theory and technology of the subject and by giving an overview of the final resulting system. The test environment was required to test firmware running on physical samples of the RFIC and support automation of those tests.

The development of the test environment was implemented with agile software development methods. A basic functionality was established before testing and was later improved and expanded when usage of the test environment began. Further analysis of technical requirements of the test environment was carried out during the usage of the test environment, which led to the development of further configurability and modularity of the components making up the system.

Automation was successfully established by providing a proof of concept. Complete automation of all developed tests was not implemented due to technical difficulties, some of which were external. Further automation is achievable with minor improvements on the test environment.

The initial requirements of the test environment were met. A working test environment with basic functionality was up and running during the start of sample testing, and required features were easily pinpointed and implemented as testing proceeded. The test environment was proven to be useful and compatible with other tools in use.

While working on this thesis the author learned much about the engineering involved in developing circuitry and its software. The author had not previously worked on designing and implementing a system of this scale, which turned out to be a very educational experience. It involved a lot of research into interfaces, how to use them and how to create them. The author was also exposed to the concepts of creating user friendly systems and user interfaces.

REFERENCES

ARM Ltd. 2001. ARM® Developer Suite AXD and armsd Debuggers Guide: Q-format. Referenced 20.4.2017 http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0066d/CHDFAAEI.html

Ebbert-Karroum, A. 2010. How to Structure a Scalable And Maintainable Acceptance Test Suite. Referenced 28.4.2017 https://blog.codecentric.de/en/2010/07/how-to-structure-a-scalable-and-maintainable-acceptance-test-suite/

Homes, B. 2013. Fundamentals of Software Testing, 1st edn. London: Wiley.

Keysight Technologies 2014. M9252A DigRF Host Adapter Data Sheet. Referenced 27.4.2017 http://literature.cdn.keysight.com/litweb/pdf/5991-2028EN.pdf?id=2313868

Lui, K.M. & Chan, K.C.C. 2008. Software development rhythms: harmonizing agile practices for synergy, 1st edn. Hoboken, New Jersey: John Wiley & Sons Inc.

MIPI Alliance 2017. MIPI DigRF. Referenced 26.4.2017 https://www.mipi.org/specifications/digrfsm-specifications

Mishra, S.; Singh, N.K. & Rousseau, V. 2015. System on Chip Interfaces for Low Power Design, 1st edn. Morgan Kaufmann.

Prodigy Technovations 2013. PGY-DGRF DigRF v4 Application Notes Version 1.0. Referenced 15.5.2017 http://www.prodigytechno.com/resources/DIGRFV4/PGY-DGRF_DigRF_v4_Application_Notes_Version1.0.pdf

Python Software Foundation 2001. PEP 257 -- Docstring Conventions. Referenced 24.4.2017 https://www.python.org/dev/peps/pep-0257/

Python Software Foundation 2017a. PyPi - the Python Package Index. Referenced 27.4.2017 https://pypi.python.org/pypi

Python Software Foundation 2017b. argparse — Parser for command-line options, arguments and sub-commands. Referenced 18.5.2017 https://docs.python.org/3/library/argparse.html

Python Software Foundation 2017c. Modules — Python 3.6.1 Documentation. Referenced 24.4.2017 https://docs.python.org/3.6/tutorial/modules.html

Robot Framework Foundation 2016. Robot Framework User Guide. Referenced 25.1.2017 http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html

Wikipedia 2015. DigRF. Referenced 26.4.2017 https://en.wikipedia.org/wiki/DigRF

Wikipedia 2017. Python (programming language). Referenced 27.4.2017 https://en.wikipedia.org/wiki/Python_(programming_language)