OULUN AMMATTIKORKEAKOULU

Aapo Rotonen

Qt Safe Renderer automated testing in the target hardware

Qt Safe Renderer automated testing in the target hardware

Aapo Rotonen
Thesis
Autumn 2022
Bachelor of Engineering, Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Bachelor of Engineering, Information Technology

Author(s): Aapo Rotonen
Title of the thesis: Qt Safe Renderer Automated Testing in the target hardware
Thesis examiner(s): Teemu Holappa The Qt Company, Kari Jyrkkä OAMK
Term and year of thesis completion: Autumn 2022
Pages: 36

The thesis purpose is to develop automated hardware testing in a project called Qt Safe Renderer. Qt Safe Renderer is a safety-critical software that provides a solution on how to detect and inform possible malfunctions or errors in a system. When dealing with such important topics, it is essential to have wide and versatile testing. Creating this kind of development in the tests means that other tests could be developed and expanded with a similar approach in the future for wider test coverage.

The primary goal of the thesis is to examine how the rendered output could be verified from an external monitor in target hardware by using Squish GUI Automation Tool automatically.

The study consists of introduction section, a brief analysis of different options that could be used in the implementation, the chosen option used in the implementation itself, and finally conclusions.

The results of the thesis work provided an alternative way how to verify the rendered output from hardware and gave the possibility for further development and expansion in test coverage in the Qt Safe Render project.

Keywords: The Qt Company, Qt, RTA, Test Automation, QSR, Squish

ACKNOWLEDGEMENT

First and foremost, I would like to thank Teemu Holappa for being my supervisor from The Qt Company. Teemu offered endless help every time it was required and guided me towards correct directions if I got stuck at some point. Then I would like to thank my supervisor Kari Jyrkkä from OAMK. Kari did actively read my progress and give good feedback, tips, and perspective about the work.

Additionally, I would like also to thank rest of the QSR team for the support, wisdom, and help along the way.

And lastly but not least, I would like to also thank Evgenii Kucheruk for helping me to get started with Squish at the beginning and setting everything up.

CONTENTS

1 INTRODUCTION 7
2 QT SAFE RENDERER 9
3 TEST AUTOMATION........ 12
3.1 Squish GUI Test Automation Tool 12
3.2 Test Automation in QSR........ 14
4 VERIFYING THE RENDERING OF THE OUTPUT OPTIONS 17
4.1 Screen capture and OCR feature. 17
4.2 Monitor example and a hook. 18
4.3 Test Harness. 20
4.4 Choosing the option. 22
5 IMPLEMENTATION 24
5.1 Installing Qt Creator 24
5.2 Building platform adaptation 26
5.3 Deploying Indicators to the hardware 29
5.4 Verifying the rendered output 30
6 CONCLUSION........ 34
REFERENCES........ 35

GLOSSARY

QSR Qt Safe Renderer

UI User Interface

RTA Release Test Automation

IPC Inter-Process Communication

GUI Graphical User Interface

IDE Integrated Development Environment

AUT Application Under Test

OCR Optical Character Recognition

CRC Cyclic Redundancy Check

1. INTRODUCTION

In this thesis, it is studied how automated testing on target hardware could be part of Qt Safe Renderer (QSR) and how this could be implemented. QSR is a safety-critical software that provides a solution on how to detect and inform possible malfunctions or errors in a system. This is an ongoing project at The Qt Company [1].

QSR is already part of automation testing and is tested in many different operating systems and platforms with multiple combinations, but all these tests are run in a desktop environment by using virtual machines created by the process. This thesis could be useful in the future if automated testing could be expanded more to hardware. This is because QSR supports hardware adaptation with different options that include many different graphical backends. Expanding the automated testing in the hardware expands testing coverage in this field and more data can be collected which then could be processed and used as an advantage to develop the project further.

The Qt Company is a global company that provides different kinds of software development platforms and frameworks, as well as expert consulting services. The most used product is Qt Creator, which is a multi-platform GUI framework and is written in C++ [18]. Qt Creator is software that can be used for creating GUIs as well as different kinds of cross-platform applications. It supports many variations of different software and hardware platforms. It also supports all major desktop platforms and most of mobile or embedded platforms [2].

The goals here are to take a closer look at several possible options that could be used at verifying the rendered output in the device and choosing the best one. Then include it in the implementation that is a fully automated environment where tests could be run in the target hardware (Figure 1).

Figure 1. Qualcomm Snapdragon 8155P

In the automated environment current QSR's Indicators example (Figure 2) will be used as a demonstrator and as a first test case that will be deployed to the hardware [19]. As if these tests are developed further later on, then the Qt Cluster example (Figure 3) might be also added as part of the whole hardware testing. The main goal is to build a fully automated environment that will do the required things to verify the rendered output in the hardware (Qualcomm Snapdragon 8155p) along with the QNX real-time operating system with one of the studied options.

By accomplishing this, it would provide suggestions on how to expand the rest of the current automated tests to the target hardware which is essential in this project in general. Currently, these tests are run on different platforms and operating systems but only in a desktop environment.

2. QT SAFE RENDERER

QSR has a solution for rendering safety-critical information in functional safety applications with Qt Creator. This will be demonstrated by showing different examples made with QSR. QSR provides a module that offers a user interface rendering component, that can be used for safety-critical based information processing. This helps to avoid any risks of injuries or damage to health [3]. It is also a commercial feature that comes with Qt for the Device Creation Enterprise license.

The QSR component used in runtime is designed to be integrated into a system that uses different and separate processes for safety-critical and non-safety functionalities [3]. There are many situations and use cases that deal with safety-critical information. This kind of solution is essential when dealing with such topics [3]. For example, one case could be any malfunction in a medical device or a brake failure in a car. Based on these examples, we can assume that QSR applies to many vertical fields such as embedded car systems, health-related applications, and automation [3].

In the case of safety-critical information, the information must be processed correctly and safely. If there were any malfunctions or occurring errors, the safety-critical information would still be rendered in the UI to let the user know what takes place behind the process [3].

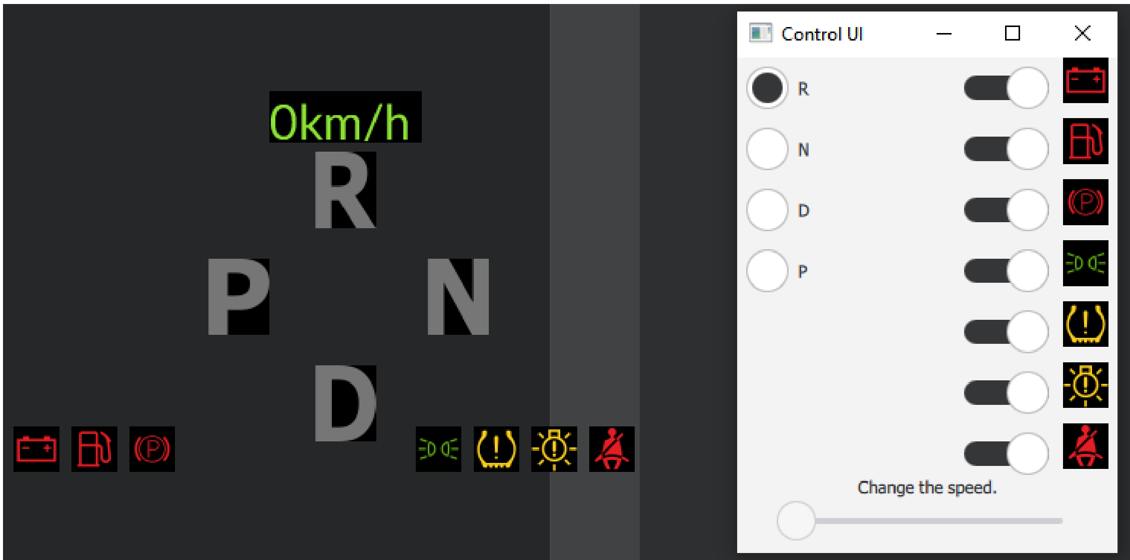

Figure 2. Indicators example.

Indicators example (Figure 2) provides safety-critical indicators and gear shifting that is used for demonstration purposes on how the safety-critical UI is rendered in a separate process by QSR [19].

Figure 3. Qt Cluster example.

The Qt Cluster example (Figure 3) demonstrates rendering on *QNX* and recovering from a sudden failure in the main UI. The Demo randomly changes different icons on and off, it also has switchable user interfaces called Hybrid car and Sports car [5].

QNX is a real-time operating system for embedded systems that are widely used in the car and vehicle industry for possible infotainment systems. QNX OS for

Safety enables certification for ASIL-D safety integrity level which is one of four identified standards and also has the highest integrity requirements on the product [4].

In the Qt Cluster example, three different processes are running in the background: The main instrument process to render the non-safe QML, the Monitoring process monitoring the main UI rendering, and QSR process that handles the telltales rendering. QSR process ensures that in case the main process is crashed or has an error, the telltales are still rendered and visible to the screen to inform the user [5].

Figure 4. Qt Cluster example with sports car user interface.

3. TEST AUTOMATION

This chapter discusses the meaning of test automation in general and what kind of automation testing QSR already has. Different types of testing can be used for example for testing some specific functionality of an application. Unit testing, integration testing, smoke testing, and regression testing have their purpose and benefits. The same purposes and benefits also apply to automated testing [6]. It executes test cases that could also be part of the functionality of an application. After tests are run, the actual test results could be compared with the expected results.

There are lots of benefits of automated testing instead of managing everything manually. Automated testing gives higher accuracy in finding errors and bugs and it is more effective. When someone checks a system manually, mistakes can be made more easily, especially when an application may contain hundreds to thousands of lines of code. Automation helps avoid these human errors [7].

Some test automation tools have reporting capabilities that log each test script to show users the status of every test. After the tests are run, the tester can then compare the results with the expectations to see how that specific part of the software works and looks compared to the requirements [7].

3.1 Squish GUI Test Automation Tool

Squish is used as an automation tool for functional testing, and it can be used for testing UI and applying test scripts for the regression testing and system testing for Qt Widgets, Qt Quick, and QML applications as well as automating interactions with embedded web content [8].

Qt Widgets are primary elements when creating user interfaces in Qt Creator. They can be used for displaying information and different kind of data and receive input from the user [9].

Qt Quick is a standard library module for making QML applications. It provides all types necessary for creating user interfaces with QML. It can be used for creating and animating visual components, getting input data from the user, and creating data models and views [10].

QML applications are made with declarative language that allows user interfaces to be written in terms of their visual components and how they relate and interact with each other [11]. Building animated UIs and connecting those to any back-end C++ libraries is made easy with the Qt Quick module.

The architecture behind Squish IDE is shown in the picture below (Figure 5). Every Squish tool works together building up a package that is not noticeable because all communication is done transparently behind the scenes. When executing Squish IDE, it automatically starts the squish server used for communication and stops whenever Squish IDE is shut down.

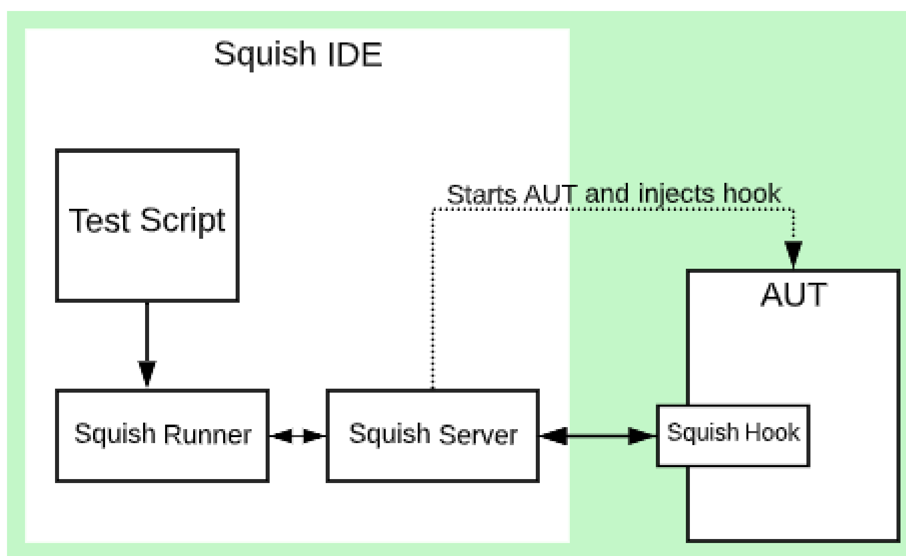

Figure 5. The architecture behind Squish IDE.

To make it possible for Squish to communicate and control an AUT (Application Under Test), it must have access to the internals of AUT. This is done by using bindings which are a combination of libraries allowing access to the objects required in the test case. This also includes the object properties and methods which are accessible after those bindings are set properly.

Squish also has a tool called Coco [20]. It is a cross-platform and cross-compiler code coverage analysis tool for C, C++, SystemC, C#, Tcl, and Qml languages. Automatic source code instrumentation is used to measure test coverage. This testing metric determines the number of lines of code that are successfully validated under a test, what additional tests need to be written and how the test coverage has changed for example over time [12].

Squish GUI test automation tool was chosen to be used in this thesis because it is already used by RTA (Release Test Automation) teams in Qt and has very good compatibility with Qt.

3.2 Test Automation in QSR

QSR already has test automation where a lot of different platforms, operating systems, and plugins are tested in versatile ways but mostly in the host environment (Figure 6), so there might be some space to expand automation in hardware testing.

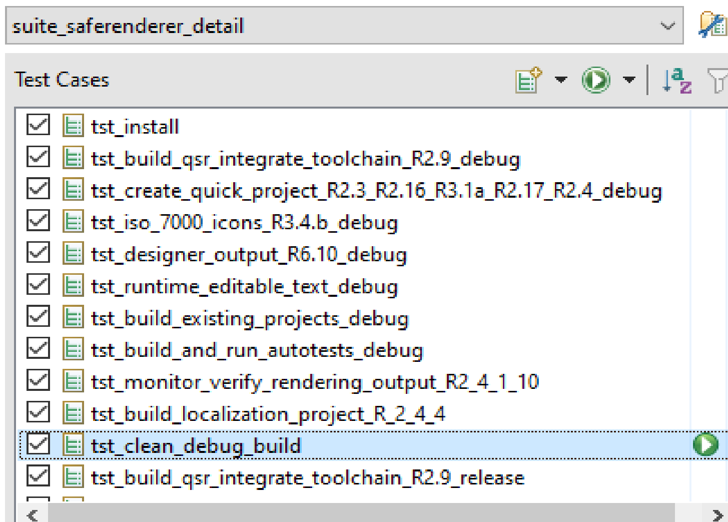

Figure 6. Some of the QSR test cases in a test suite.

Tests are mostly being run from Jenkins using Squish. Jenkins is an open-source automation server that can be used to build, deploy or automate any project. It allows setting up continuous integration (CI) or continuous delivery (CD) environments for almost any combination of languages and repositories, as well as automating other development tasks [13].

Squish is a convenient platform for test automation for Qt Creator as well as QSR because it has script bindings for all the GUI toolkits it supports, which also includes the ones Qt Creator has. It gives direct access to all functions, properties, and widgets that Qt Creator provides. Those bindings give the possibility to query and set object properties and call object methods like widgets.

What is also convenient is that Squish has a built-in object map which is designed to help a user to maintain test scripts when an application chooses to change its object hierarchy or object names. It helps when the set of test cases grows in the test suite for example in the test script code there could be multiple same places for GUI controls that they are referencing and pointing to [14].

In this thesis, all required and used Qt-related objects are saved into “names” import where they can easily be found and used. It can be seen in Figure 7.

```
6⊖def main():
7     startApplication("qtcreator")
8     activateItem(waitForObjectItem(names.qt_Creator_QtCreator_MenuBar_QMenuBar, "Edit"))
9     activateItem(waitForObjectItem(names.qt_Creator_QtCreator_Menu_Edit_QMenu, "Preferences..."))
10    mouseClick(waitForObjectItem(names.preferences_QListView, "Devices"), 70, 14, Qt.NoModifier, Qt.LeftButton)
11    clickTab(waitForObject(names.preferences_Devices_QTabWidget), "QNX")
12
```

Figure 7. Code that opens the devices tab from options in Qt Creator.

4. VERIFYING THE RENDERING OF THE OUTPUT OPTIONS

This chapter introduces a brief look into three different options that could be used as part of the implementation to verify the rendering output from the device. These options are *Screen capture and OCR feature*, *Monitor example and hook*, and *Test harness*. Lastly choosing which of the options will be used to achieve the wanted result and why.

4.1 Screen capture and OCR feature

One of the options could be using screen capture and OCR feature. OCR System combines hardware and software to convert physical documents for example into machine-readable text.

OCR stands for Optical Character Recognition and it is used to recognize computer-printed characters, read text, and different kinds of fonts. Some advanced OCR systems could also recognize even hand printing [15].

In this case, it could be used in analyzing a screen capture taken off an external monitor where our application runs for example. The way the screen capture works is that screen capture could be taken of a specific view and since the screen capture is a bitmapped image of pixels, OCR could then analyze the pixels to recognize letters and digits and compare them to some specific data to verify the output [15].

Squish GUI also has a similar kind of feature that is helpful and could be used here to verify the output. Different verification points could be manually placed anywhere as part of the script and one of these verification points is screenshot verification (Figure 8).

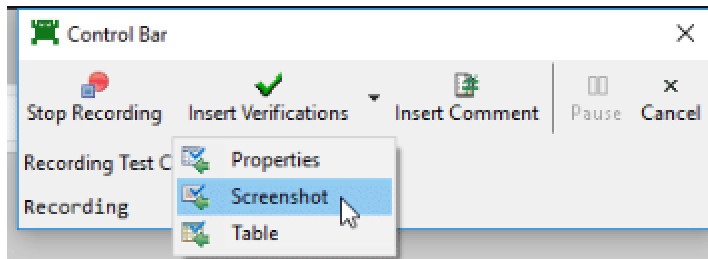

Figure 8. Squish screenshot.

You can insert a screenshot verification point to any part of the code. When the option is selected from the Control Bar (Figure 8) it gives you the possibility to select an object from the "Application Objects" list where all automatically identified objects are gathered or you can manually select a specific object from the screen with the "Pick" tool. Now Squish takes a screenshot of that object and inserts it into the verification point that was earlier set in the code. When the test case is running it will automatically compare the current view with the taken screenshot when it arrives at that verification point. If the current view is the same as the compared screenshot, the test case continues moving forward. And if there are some differences between them, then the test case will fail.

4.2 Monitor example and a hook

The second option could be using the current QSRs monitor example (Figure 9) and creating a hook between Squish and the monitor. Monitor example reads and compares that reference values match with the actual content in the used display. CRC values are 32-bit numbers generated by an algorithm called CRC based on the contents of a specific file so CRC check has a big role here. The way how the CRC check works is that when safety-critical content, like gear selector or other safety-critical indicators, is rendered to the screen, QSR reads the CRC checksum from the displayed interface (Figure 10). After that, the checksum is provided to the Monitor which runs in a separate process to conduct an independent check for the verified and correct rendered output [16].

If the CRC values match, it means that the system works and the correct CRC value is shown in the command line. If the icon does not show the CRC value is

then “0” because it calculates the data of the icon and this way it is possible to know when the icon is OFF and when ON. If there are errors in the QSR process it gives an error message or mismatching fingerprint to inform what has happened. With the Monitor component, QSR can be used for creating safety-critical user interfaces to different kinds of processors and makes it also possible for more versatile systems to be part of [16].

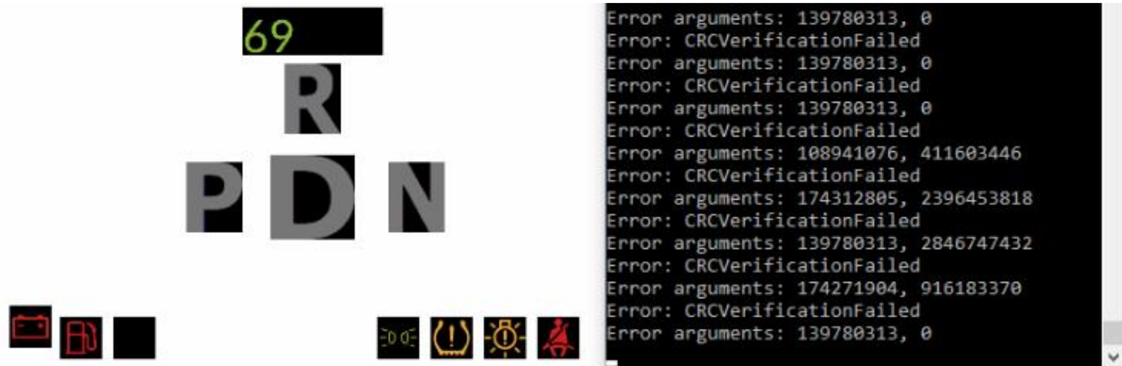

Figure 9. Monitor example with Indicators.

The Monitor component verifies that rendered safety critical items are displayed at the right times and if not, it will inform the user about it. For each safety-critical QML type in Safe Renderer’s example application, the Monitor reads the expected CRC value from the layout file and compares it with the actual CRC value that comes from the display [16].

QSR’s tooling calculates during build time of the correct checksums when using the Monitor. This unique checksum is created for each of the safety-critical items, with the needed identifiers. [16] The Monitor is also run in a separate process from the rendering which allows independent operation from QSR.

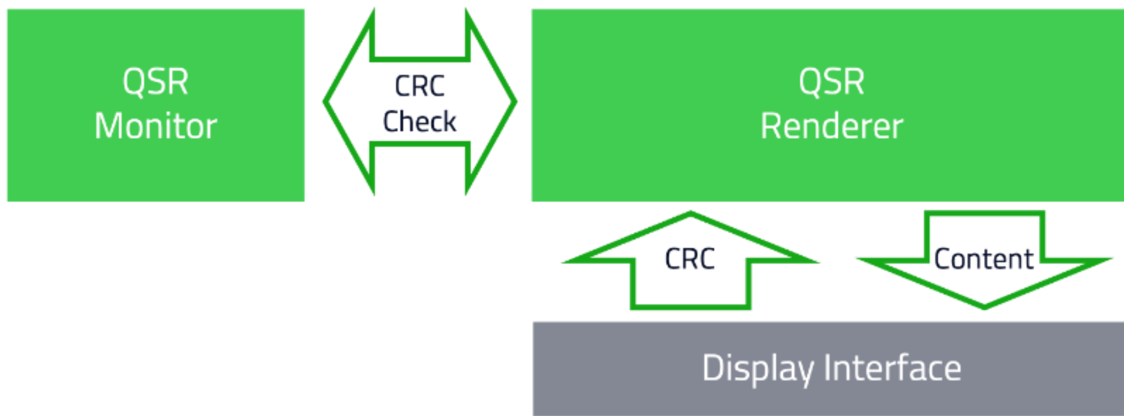

Figure 10. How QSR works with the Monitor component.

4.3 Test Harness

Test Harness (Figure 12) is a Python script that creates a GUI to make manual testing easier for QSR. It uses network sockets for communication between applications. Different kinds of events can be sent to the applications with Test Harness (Figure 12) once the settings are set correctly pointing to the environment where the application is running from the script (Figure 11). For example, if the Indicators example (Figure 2) runs in the host environment then IP and PORT have to be set pointing to that, so in that case, IP would be “localhost” and PORT “32123”.

```
# network settings
self.runtime1_IP = 'localhost' # 127.0.0.1
self.runtime1_Port = "32123"
self.runtime2_IP = "10.204.0.127"
self.runtime2_Port = "22"
self.runtime1_connectionstate = False
self.runtime2_connectionstate = False
self.mySocket1 = socket.socket()
self.mySocket2 = socket.socket()
self.errMessage = "Error: EventSender is n
```

Figure 11. Network settings in the Test harness script.

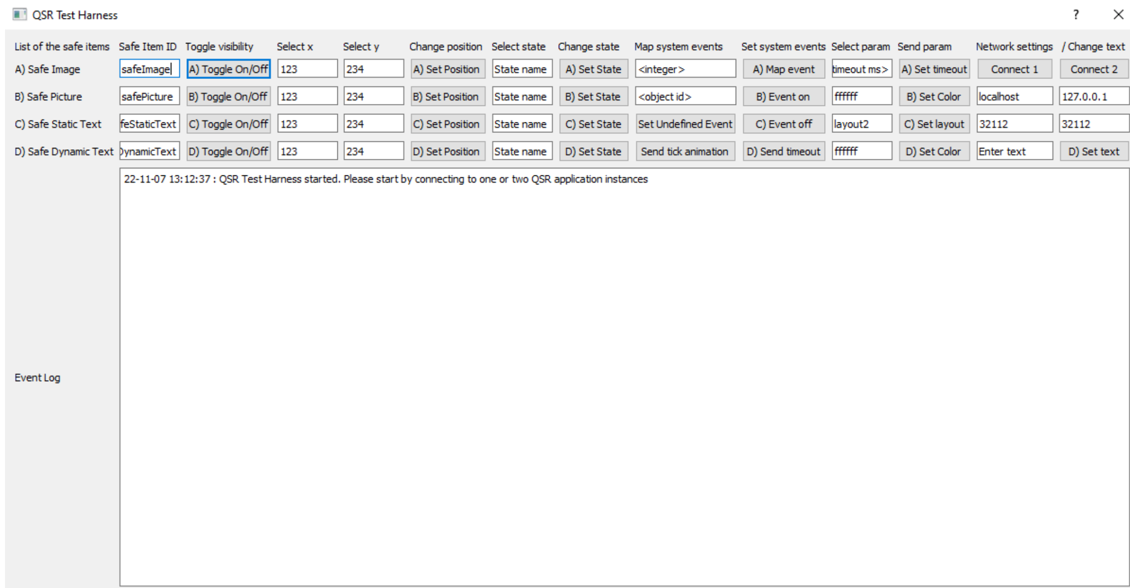

Figure 12. Test harness GUI.

A socket is one endpoint of a two-way communication link between two different programs running on the network. The structure and properties of a socket are defined by an API for the networking architecture. Sockets are created during the time of the process of an application running in the node and it is bound to the network protocol to be used for transmissions, a network address of the host, and a port number [17].

This could be chosen as part of the implementation and as the option how to verify the output in the device. The way it could work is that the Test harness would be started as an AUT from Squish inside of a test case and there could be a hook between QSR and this AUT. This way the wanted actions could be also recorded with the record option in Squish. For example, a simple action could be that the item ID would be written in the text field under “Safe Item ID” for Safe Picture, and then the mouse click would also be recorded for the “Toggle On/Off” button (Figure 12). That would send a change visibility event to QSR and it would turn the desired item on or off if the network connections were first set.

4.4 Choosing the option

The conclusion here was to use a few similarities between *Test Harness* (Figure 12) and the *Monitor example with a hook* (Figure 9) to build the implementation. The general idea was to create a test case with Squish that includes creating a connection through a socket and sending safe events to the QSR process just like *Test Harness* would do (Figure 13) but now straight from Squish. Then return the desired data to Squish to verify the rendered output from the hardware by comparing the expected and actual CRC values like the *Monitor example* would do but only with the command line.

The main reason that led to this decision was that building a working implementation like this could also mean that it could be easily maintained and developed in the future if other tests were expanded with a similar approach to the hardware without any dependencies on hooks or other applications.

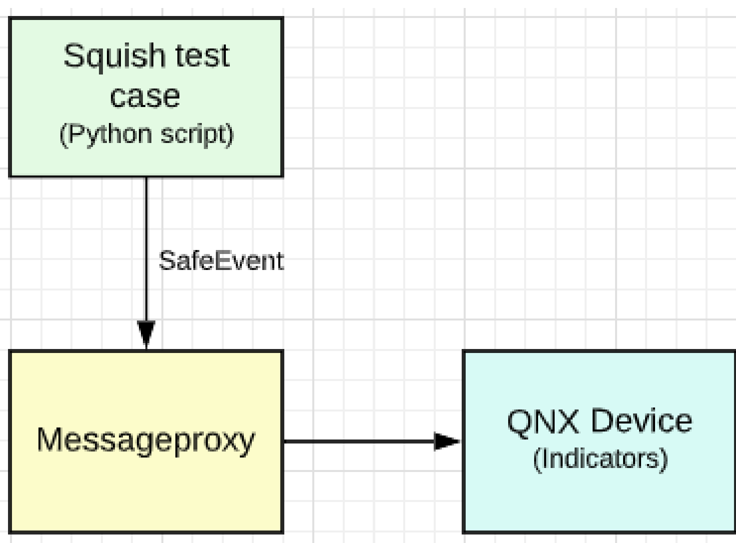

Figure 13. The process of sending events from Squish to the Device

Messageproxy (Figure 13) is an application made for testing the messaging interface in QSR. It includes network settings which must be set for a proper connection between applications. Messageproxy application listens to events coming from a network socket and then passes them to the QSR process via IPC (Inter-

Process Communication) so in this case, it has to be also deployed and running in the device.

5. IMPLEMENTATION

The implementation should contain everything from the beginning where Qt Creator is installed to the end where the Indicators example is deployed to the target hardware and the rendered output is verified with the chosen option. This means that there is a test suite that includes many different test cases that makes together a fully automated environment that performs all this.

5.1 Installing Qt Creator

The implementation process starts with the Qt Creator installation test case to choose the correct configuration including addons, plugins, and components which will be installed depending on what the user has declared in the file named "env_variables.py" (Figure 14) and in the code itself. The file (Figure 14) contains variable initializations that may need to be hardcoded when tests are run on a local computer.

When the installation begins and the Qt maintenance tool will open, the main script will check the variables given in the file so it can choose the correct product, components, plugins, etc that will be included in the installed package.

get_job_name() returns the name of the tested configuration which includes compiler version, information that is our target machine physical or virtual, and to which module is installed. *get_version()* returns the correct product version required. In this case, Safe Renderer 2.0 is used, so 2.0.0 will be the declared version. *get_qt_version()* returns the correct Qt-version that will be used and those test cases will be run against. *get_license()* declares what kind of license is used. In this case, it is enterprise. *get_installer()* declares which package type will be used.

```python
def get_job_name():
    """
    Function returns Jenkins job name
    """
    #return "RTA_QtSafeRenderer_win/cfg=win-MinGW8.1.0-Windows10-x64"
    return "RTA_smoke_installer_windows/cfg=win-MinGW8.1.0-Windows10-x64_QNX700-armv8"
def get_version():
    """
    Function returns the Product version
    """
    return "2.0.0"
    #return os.environ.get(envVariable_product_version, "ERROR_not_defined")
def get_qt_version():
    """
    Function returns the Product version
    """
    return "5.15.10"
    #return os.environ.get(envVariable_qt_version, "ERROR_not_defined")
def get_license():
    """
    Function returns used license type, enterprise/opensource. Defaults to enterprise
    """
    return os.environ.get(envVariable_license, "enterprise")
def get_installer():
    """
    Function returns used package type, online/offline/monolite. Defaults to online
    """
    return os.environ.get(envVariable_contentsource, "online")
```

Figure 14. Env_variables.py file.

The test suites usually start fresh, meaning that they do not have any dependencies on other test suites or cases. So at the beginning of the installation of the test case, it checks that folder with the same name does not exist, and if does, it will be removed and created again. After this, the installation will begin and all those declared versions of compilers, add-ons, and the main product will be installed, and the folder will be created which in this case is called “RTA” (Figure 15).

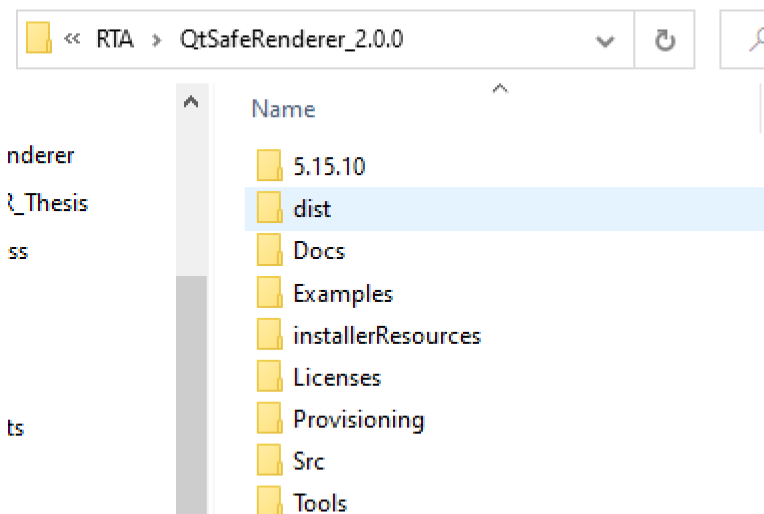

Figure 15. RTA folder including the product and files that are installed.

5.2 Building platform adaptation

For downloading, required files from URLs with python, some kind of library is needed so in this case “urllib-library” will be used. It is a module that is used for fetching URLs using different protocols. Also used paths and functions in the code are declared in a different file called “qsr_imports” from where they are imported to the main code for usage.

The first thing in building the platform adaptation is that *openWFD* files (Figure 16) and *QNX* files (Figure 17) should be downloaded and extracted to a folder called “libopenwfd” and “qnx700” (Figure 18) and they are located in a target directory which is called “QnxFiles” that is also created in the process (Figure 18).

```
def download_wfd_libs():
    wfdPath = qsr_imports.wfdPath_dl
    #wfdLib = os.chdir(wfdPath + "\lib")
    #wfdInclude = os.chdir(wfdPath + "\include")
    os.chdir(targetDir)
    try:
        # Download files
        file.retrieve(wfdPath, "libopenwfd.zip")
        # Extract files
        os.system("tar -xvf libopenwfd.zip")
        test.log("Download successful! files can be found at:" + " " + targetDir)
    except:
        test.log("Something went wrong with wfd libs.")
```

Figure 16. Downloading and extracting WFD files to the target directory.

```
def download_qnx_files():
    # Path to QNX files
    qnx700_snap = qsr_imports.qnx700
    qnxSrcs = qsr_imports.qnxFiles
    # Change directory
    os.chdir(targetDir)
    try:
        # Download files
        file.retrieve(qnx700_snap, "qt5.15.10-armv8-qnx700-snap.tar.gz")
        file.retrieve(qnxSrcs, "qnx700-20210323-windows.7z")
        # Extract files
        os.system("tar -xvf qt5.15.10-armv8-qnx700-snap.tar.gz")
        os.system("7zr x qnx700-20210323-windows.7z")
        test.log("Download successful! files can be found at:" + " " + targetDir)
    except:
        test.log("Something went wrong with qnx files.")
```

Figure 17. Downloading and extracting QNX files to the target directory.

These files are required for proper implementation. Once this is completed, the next objective is to run simple tests from Squish to check that all the required files and folders are found before proceeding to the next step.

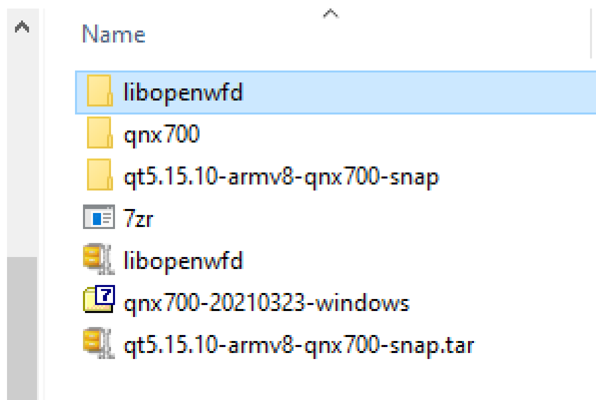

Figure 18. Required files for building the testing environment in a folder called “QnxFiles”.

The “qnx700” folder includes a batch file called “qnxdsp_env.bat” which needs to be run from the command line so environment variables will be set automatically pointing to the correct paths and directories. Then setting up the compiler path needs to be done so that the rest of the files can be configured and installed properly and this can be done with a simple command (Figure 19). Since the compiler path is pointing to the correct directory, the test case can proceed to the next step.

```
path=Qt/Tools/mingw810_64/bin;%PATH%
```

Figure 19. Setting up compiler path from the command line.

Configuring Qt for *QNX* needs to be done if Qt Creator needs to be built again. Giving the following long command from the command prompt is required for correct configuration in this case (Figure 20). This is just an example of a similar command.

```
./configure -confirm-license -commercial -release -xplatform \
qnx-aarch64le-qcc -opengl es2 -nomake tools -skip qtactiveqt \
-skip qtandroidextras -skip qtconnectivity -skip qtdoc \
-skip qtlocation -skip qtmacextras -skip qtsensors -skip qtserialport \
-skip qtwinextras -skip qtx11extras -no-openssl -nomake examples -nomake tests \
-force-debug-info -separate-debug-info -skip qtwebengine -no-fontconfig \
-prefix /opt/qt5.15.10/install/qnx7_armv8
```

Figure 20. Example of configuration command.

Then running commands “make” and “make install” needs to be done to build and install the whole environment properly.

Use **System Environment** and
Set **WFD_INCLUDE** to **C:\Users\aarotone\QnxFiles\libopenwfd\include**
Set **WFD_LIB** to **C:\Users\aarotone\QnxFiles\libopenwfd\lib**

Variable	Value
QNX_HOST	C:/Users/aarotone/QnxFiles/qnx700//host/win64/:
QNX_TARGET	C:/Users/aarotone/QnxFiles/qnx700//target/qnx7
qtcreator	C:\Users\aarotone\RTA\QtSafeRenderer_2.0.0\Tool
QTDIR	C:\Users\aarotone\RTA\QtSafeRenderer_2.0.0\5.15.
SESSIONNAME	Console
SQUISHCOCO	C:\Program Files\squishcoco
SystemDrive	C:
SystemRoot	C:\WINDOWS
TEMP	C:\Users\aarotone\AppData\Local\Temp
TMP	C:\Users\aarotone\AppData\Local\Temp
UATDATA	C:\WINDOWS\CCM\UATData\D9F8C395-CAB8-49
USERDNSDOMAIN	INTRA.QT.IO
USERDOMAIN	INTRA
USERDOMAIN_ROAMINGPROFILE	INTRA
USERNAME	aarotone
USERPROFILE	C:\Users\aarotone
WFD_INCLUDE	C:\Users\aarotone\QnxFiles\libopenwfd\include
WFD_LIB	C:\Users\aarotone\QnxFiles\libopenwfd\lib
windir	C:\WINDOWS
ZES_ENABLE_SYSMAN	1

Figure 21. Setting OpenWFD paths in Qt Creator.

Setting up the *OpenWFD* paths to point to the correct directories in Qt Creator environment variables needs to be done before the adaptation can be implemented correctly. This allows Qt Creator to find the earlier downloaded files and adapt to them. For the device to recognize and be able to use the Qt libraries, they have to be deployed into the device also from Qt Creator (Figure 22).

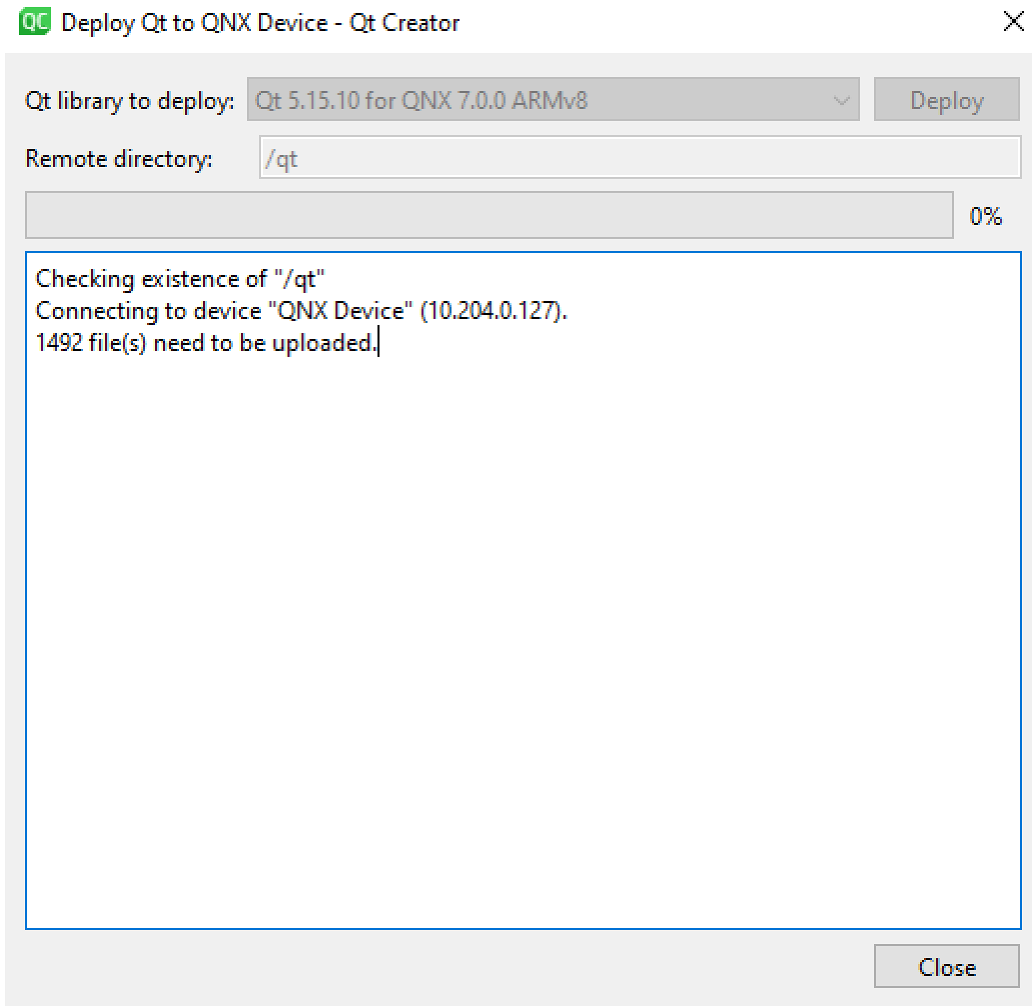

Figure 22. Deploying libraries into the device.

5.3 Deploying Indicators to the hardware

Deploying Indicators example into the hardware, SSH connection needs to be set up to the hardware. The test case starts the command line and takes an SSH connection to the hardware IP address and port and lets the command line window stay active as long as the test case is running. An application named Messageproxy (Figure 13) needs to be also started and run in the background.

The test case creates a connection through a socket and sends these events that are declared in the code to Messageproxy which then passes the events to the Indicators example that is running in the hardware. Once the Indicators example

is deployed and shown in an external monitor (Figure 23), it can start receiving events from other processes.

Figure 23. Indicators deployed to the device (Displayed in an external monitor).

5.4 Verifying the rendered output

The output needs to be verified to be sure that the Indicators example is rendered in the target device and that the QSR process that runs in the background and listens to different events and acts according to those. The following test case includes functions that send events to the device to turn the indicators on and off by changing their state (Figure 24).

```python
if (Host_Connection.ON or HW_Connection.ON):
    for i in gears:
        events = []
        toggleEvent = createStateChangeEvent("root", i)
        sendMessage(toggleEvent)
        test.log(i + " " + "gear state changed")
        time.sleep(0.4)
```

Figure 24. Sending state change event for an icon.

A few different functions had to be made in Python to send and receive data to and from QSR properly. These functions are called *createOutputVerifyItemEvent* (Figure 25), *createOutputRequestEvent* (Figure 26), and *createOutputReplyEvent* (Figure 27).

These events are already implemented in C++ in QSR because they are used in the Monitor example (Figure 10), but because they need to be used from Squish, these new functions that properly transfer the desired data had to be made.

```python
def createOutputVerifyItemEvent(target):
    bytes = EventId.OUTPUTVERIFICATIONSTATUSVERIFYITEM.to_bytes(4, byteorder='big', signed=False)
    targetId = calcHash(target)
    bytes += targetId.to_bytes(4, byteorder='big', signed=False)
    return bytes.ljust(128, b"\x00")
```

Figure 25. A function that updates the output CRC value of a single item.

```python
def createOutputRequestEvent():
    bytes = EventId.OUTPUTVERIFICATIONSTATUSREQUEST.to_bytes(4, byteorder='big', signed=False)
    return bytes.ljust(128, b"\x00")
```

Figure 26. A function that requests data from QSR.

```python
def createOutputReplyEvent(bytes):
    reply = EventOutputVerificationStatusReply()
    index = 0

    reply.eventId = int.from_bytes(bytes[index:index+4], byteorder='big', signed=False)
    index += 4
    reply.count = int.from_bytes(bytes[index:index+4], byteorder='big', signed=False)
    index += 4
    reply.variantId = int.from_bytes(bytes[index:index+4], byteorder='big', signed=False)
    index += 4
    reply.datapadding = int.from_bytes(bytes[index:index+4], byteorder='big', signed=False)
    index += 4

    count = reply.count

    while count > 0:
        elements = QsrVerifyPayload()
        elements.itemId = int.from_bytes(bytes[index:index+4], byteorder='big', signed=False)
        index += 4
        elements.displayCrc = int.from_bytes(bytes[index:index+4], byteorder='big', signed=False)
        index += 4
        elements.stateId = int.from_bytes(bytes[index:index+4], byteorder='big', signed=False)
        index += 4
        elements.inTransition = int.from_bytes(bytes[index:index+4], byteorder='big', signed=False)
        index += 4
        reply.payload.append(elements)
        count -= 1

    return reply
```

Figure 27. A function that contains the received data from QSR.

Now the rendered output can be verified by using these functions. First of all, a state change event (Figure 24) is sent for a battery icon (Figure 28) to change state to “OFF” (Figure 29), then *createOutputVerifyItemEvent* (Figure 25) is sent to read the bitmap of the icon for the CRC value.

Figure 28. Indicator icons are turned on (Battery icon first from the left).

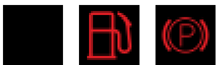

Figure 29. The battery icon is turned off.

After this, the CRC value is updated and stored in the queue which then can be read and requested with the *createOutputRequestEvent* (Figure 26). Then QSR answers with a reply which contains information about that single safe item “icon-Battery” in this case. Then the information can be returned with *createOutputReplyEvent* (Figure 27) and then compared with the expected CRC value which is parsed from an XML file that contains CRC values and other information about different items and assets (Figure 30).

```
tree = ET.parse("monitorconfigdata.xml")
root = tree.getroot()
# Parse target IDs and asset IDs from the
for id in root.iter('id'):
    ID.targetId.append(id.text)
    continue
for crc in root.iter('drawcrc'):
    ID.crcValue.append(crc.text)
    continue
```

Figure 30. Parsing XML file.

It will then inform the user about the state of the icon the moment it was verified (Figure 31) and at this point, it is in an "OFF" state (Figure 29), so it returns a value of 0. And in the same way, when the icon is turned back to the "ON" state (Figure 28) it returns that calculated CRC value which indicates that the icon is "ON" in the device and that the value is also found in the XML file which then confirms that it works (Figure 32).

Log	Item ID: 139780313
Log	Actual CRC: 0
Log	Item ID: 174249564
Log	Actual CRC: 0

Figure 31. Output when the indicator icons are off.

Log	Item ID: 139780313
Log	Actual CRC: 3975076789
Log	Found same CRC values from XML and QSR: 3975076789!
Log	Item ID: 174249564
Log	Actual CRC: 402835355

Figure 32. Output when the indicator icons are on.

6. CONCLUSION

As a conclusion, the automated environment and implementation of the thesis succeeded to meet the main goals set at the beginning. The rendered output in the target hardware running in an external monitor could be verified with the studied and chosen option. Required information about the safety-critical items showing on the screen can be gathered to Squish to confirm that the functionality works when sending different events to the QSR process.

There is also an intention to include Jenkins and Code Coverage integration and this will be part of further development with possible other expansions such as other tests that could be run in the hardware. Now as there is working main functionality done for this kind of implementation, it is a lot easier to expand to other tests to be run in the same hardware for more comprehensive testing in this kind of field.

Even though only the “main” functionality of the chosen option on how to verify the rendered output was shown properly with pictures of code. The work included many different test cases that were mentioned, and the actions and steps were shown with pictures but they included also a lot of code that built up the environment together.

As the implementor of the work, the thesis was educational personally because most of what it included was new so there was a lot of learning along the way. Squish was a new thing for me and it took some time in the beginning to understand and learn how it works. Also understanding the compatibility between different applications, processes, dependencies, and languages on how they can work together is something that also taught me a lot. After this thesis, I feel like I developed myself a lot in many different areas that the work included.

REFERENCES

1. The Qt Company. Date of retrieval 26.7.2022
   https://www.qt.io/company
2. Qt. Date of retrieval 26.7.2022
   https://en.wikipedia.org/wiki/Qt_(software)
3. Qt Safe Renderer Overview. Date of retrieval 18.7.2022
   https://doc.qt.io/QtSafeRenderer/qtsr-overview.html
4. QNX. Date of retrieval 23.11.2022
   https://blackberry.qnx.com/content/dam/qnx/products/certified_os/os-for-safety-auto-product-brief.pdf
5. Cluster Example. Date of retrieval 9.8.2022
   https://doc.qt.io/QtSafeRenderer/qtsaferenderer-saferenderer-qtcluster-example.html
6. Testing types. Date of retrieval 11.8.2022
   https://www.softwaretestinghelp.com/types-of-software-testing/
7. Automated Testing. Date of retrieval 18.7.2022
   https://www.techtarget.com/searchsoftwarequality/definition/automated-software-testing
8. GUI Test Automation. Date of retrieval 26.7.2022
   https://www.qt.io/product/quality-assurance/squish
9. Qt Widgets. Date of retrieval 24.8.2022
   https://doc.qt.io/qt-6/qtwidgets-index.html
10. Qt Quick. Date of retrieval 24.8.2022
    https://doc.qt.io/qt-6/qtquick-index.html
11. QML. Date of retrieval 24.8.2022
    https://doc.qt.io/qt-6/qmlapplications.html
12. What is CoCo. Date of retrieval 15.7.2022
    https://www.todaysoftmag.com/article/2287/how-to-use-squish-coco-to-determine-code-coverage-for-automated-tests

13. What is Jenkins. Date of retrieval 9.8.2022
https://www.jenkins.io/
14. Object map. Date of retrieval 11.8.2022
https://doc.froglogic.com/squish/latest/rg-objectmap.html
15. What is OCR. Date of retrieval 26.7.2022
https://www.pcmag.com/encyclopedia/term/ocr
16. QSR Monitor explained. Date of retrieval 30.8.2022
https://www.qt.io/blog/qt-safe-renderer-monitor
17. About network socket. Date of retrieval 25.7.2022
https://www.avast.com/c-what-is-tcp-ip
18. Qt Creator. Date of retrieval 23.11.2022
https://doc.qt.io/qtcreator/
19. Indicators example. Date of retrieval 23.11.2022
https://doc.qt.io/QtSafeRenderer/qtsaferenderer-saferenderer-indicators-example.html
20. Squish Coco. Date of retrieval 23.11.2022
https://doc.froglogic.com/squish-coco/latest/