OULU UNIVERSITY OF
APPLIED SCIENCES

Veli-Pekka Porrassalmi

CI TEST AUTOMATION SYSTEM FOR SW4STM32 IDE AND ARM MBED CLI EXPORTER TOOLS

CI TEST AUTOMATION SYSTEM FOR SW4STM32 IDE AND ARM MBED CLI EXPORTER TOOLS

Veli-Pekka Porrassalmi
Bachelor’s Thesis
Spring 2018
Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Information Technology, Option of Equipment and Product Design

Author: Veli-Pekka Porrassalmi
Title of Bachelor's Thesis: CI Test Automation System for SW4STM32 IDE and ARM Mbed CLI Exporter Tools
Supervisor: Timo Vainio
Term and year of completion: Spring 2018
Number of pages: 79 + 6 appendices

The aim of this thesis was to design a CI test automation system to automatically verify integrity between the SW4STM32 IDE and ARM's Mbed CLI exporter tools. The thesis was commissioned by Etteplan Oyj as a part of the SW4STM32 IDE maintenance project. Another aim of the thesis was to provide Etteplan with a Jenkins tests server to be utilized in other projects as well.

The integrity was verified every time ARM released a new version of their Mbed OS. To automate the verification process, a Python test script was written. The Python test script implemented the core features required by the verification process. A Jenkins server was hosted to provide remote access to the test system for Etteplan's employees. The Jenkins server offered simple user interface to initiate test runs and to access HTML test reports and log files from the test runs. The Jenkins server also sent email notifications to the defined employees after every test run.

Keywords: CI, Test automation, Jenkins, SW4STM32, Mbed OS

PREFACE

I would like to thank Etteplan for commissioning this Bachelor’s thesis. I also want to thank my colleagues for keeping up the humor and great company, and my dear wife for being supportive at home.

I also wish to thank Timo Vainio, Kaija Posio and Niina Kuokkanen at OAMK for their assistance during this Bachelor’s thesis.

Oulu, 2.5.2018
Veli-Pekka Porrassalmi

CONTENTS

VOCABULARY

OS	Operating System
IDE	Integrated Development Environment
SCM	Source Control Management
CI	Continuous Integration
DUT	Device Under Test
MCU	Microcontroller Unit
Eclipse	Free open source software development IDE
SMTP	Simple Mail Transfer Protocol
HTML	Hypertext Markup Language
.bin	Binary file format
.py	Python program file
.html	HTML file

1 INTRODUCTION

Nowadays, test automation is a crucial part of software development process. In today’s software projects, even hundreds of developers may be involved in developing the same software. These developers are usually working with some sort of source control management (SCM) system in order to obtain code integrity and version control. When a considerable number of changes in code are made by multiple developers at the same time, errors cannot be avoided. Thus, every single time a developer makes changes in code, the code must be tested in order to obtain working and high-quality software. [1]

When the software is large and there are hundreds of developers making changes to the code, it is impossible to carry out the testing manually. It would take an enormously large amount of time to test the changes made by each developer by hand. This is where test automation comes into the picture. As the designation itself implies, test automation is used to test those hundreds or even thousands of changes in code automatically. Usually test automation compiles and builds the software, runs the defined tests and generates some sort of test report. Test automation does not necessarily have to be triggered automatically every time a developer makes changes in the code, but it can also be triggered manually. In either case, test automation saves a lot of time and is a very efficient way to detect errors. Testing software by hand can take months while with test automation it takes only a portion of that, maybe even less than a day. The context described above is the essential definition of the concept called Continuous Integration (CI). [1]

ARM, a British company specialized in software and semiconductor design, offers a real time operating system (RTOS) for embedded devices called the Mbed OS. Mbed OS is a rather large open source software project that gets updated constantly. ARM releases a new major release of their Mbed OS four times a year, but minor updates are more frequent. ARM’s Mbed tools have support to export Mbed OS projects to various Integrated Development Environments (IDEs) for software developing and debugging. One supported IDE is the

STMicroelectronics’ own IDE called System Workbench for STM32 or SW4STM32 in short.

Etteplan commissioned this thesis to create a CI test automation system for the SW4STM32 IDE and STM32 Nucleo development boards. The main goal was to maintain integrity between the SW4STM32 IDE and the ARM’s Mbed exporter tool. The integrity was tested every time a new major Mbed OS release was published by exporting Mbed OS project to the SW4STM32 IDE and building binary files for the selected STM32 Nucleo development boards. What is problematic though, is since major releases are published every quarter of a year and the employee responsible for the testing changes almost every time, it takes a lot of time to familiarize oneself with the project, set up the testing environment and write the test reports. Thus, Etteplan wanted to create a CI test automation system to be used with said project.

The test automation system would download the latest Mbed OS version, export Mbed OS project to the SW4STM32 IDE, build binaries using the SW4STM32 IDE, verifying results by flashing 10 predefined Nucleo development boards and running the test applications on those, and finally generate a test report. To make this easily accessible, a Jenkins server would be configured on a host PC that could be accessed remotely from any employees’ computer. The Jenkins server would then be responsible of running said test automation and sending test reports to defined employees via email with a single press of a button. With the Jenkins server configured, it could also be used as a general test platform in other projects inside Etteplan as well.

This thesis work was requested by Etteplan Oyj in Oulu. Etteplan was established in 1983 and it has been vastly growing ever since. Etteplan has more than 2800 employees in Finland, Sweden, the Netherlands, Germany, Poland and China. Etteplan’s expertise are Engineering, Embedded systems and IoT, and Technical documentation. [2]

2 DEVELOPMENT TOOLS

This chapter introduces the tools used during development work and by the test system.

2.1 Python language

Python is an open source, high level interpreted programming language that supports modules, classes, exceptions, and dynamic data types. Python's efficiency is based on its exceptionally clear syntax and portability between different operating systems. [3]

Python programming language was used to implement the test script responsible in the core features of this project. Also, the HTML reports were automatically generated via Python.

2.2 C++ language

C++ is an object-oriented programming language based on the C programming language. The C++ programming language features both high- and low-level features which is why it is considered a middle-level programming language. C++ is a widely used programming language in system and application programming as well as in embedded firmware development. [4]

C++ was used to implement the *main.cpp* program for the STM32 Nucleo development boards.

2.3 Arm Mbed OS

Mbed OS offered by ARM is a free and open source operating system for embedded devices. Mbed OS offers easy access for required tools to develop a product based on ARM Cortex-M processor architecture such as connectivity features, a real time operating system and drivers for various sensors and I/O devices. [5]

2.4 Arm Mbed CLI

Arm Mbed CLI (aka. mbed-cli) is a Python based command-line tool. Arm Mbed CLI enables the use of Arm Mbed OS build system, export functions, support for Git-based version control and remotely hosted repositories such as GitHub and many other features. [6]

In this thesis, the following Arm Mbed CLI commands were used:

- mbed detect
    - Detect connected Mbed devices
- mbed new
    - Import latest Mbed OS release from GitHub
- mbed ls
    - Print current Mbed OS information
- mbed export
    - Export Mbed OS project to external IDE

2.5 System Workbench for STM32 (SW4STM32)

STMicroelectronics offers their own Eclipse based toolchain called the System Workbench or SW4STM32 in short. SW4STM32 is a free IDE available for Windows, Linux, and OS X operating systems with full support for STM32 microcontrollers and related boards. SW4STM32 toolchain features GCC C/C++ compiler, GBD debugger, Eclipse IDE with support for Eclipse plug-ins and ST-LINK support. [7]

SW4STM32 IDE was used to compile and build Mbed OS projects.

2.6 STM32 Nucleo development boards

STMicroelectronics offers their own series of development boards called the STM32 Nucleo. They offer an easy approach to new project ideas and prototypes with a wide selection of STM32 MCUs (microcontroller units) and great extensibility via Arduino Uno R3 connectors. [8]

The STM32 Nucleo development boards were used to verify binary files by running a verification test between host PC and Nucleo development boards.

2.7 Jenkins

Jenkins is an open source automation server. Jenkins can be used in various tasks such as automated software building, testing, and deploying software. Jenkins supports a significant number of plugins that allow extensible Jenkin configurations. [9]

A Jenkins server was used to implement a user interface to initiate the test script and to browse test reports and logs.

2.8 Oracle VM VirtualBox

VirtualBox is a high performing virtualization tool for home users and enterprises. VirtualBox software offers usage of virtual machines to run numerous versions of operating systems regardless of the operating system running on the host computer. [10]

In this project, VirtualBox was used to run the Ubuntu Linux distribution during the early development phases before Ubuntu was installed on the final host PC.

3 SYSTEM DESIGN

In this chapter the general system design is introduced. More detailed implementation is introduced in later chapters.

3.1 Features

The core feature of the test setup was a Python test script. The test script was responsible for setting up the Mbed OS environment, building binary files, and verifying binary files by running tests on flashed Nucleo development boards. In addition, the test script generates HTML test reports and gathers detailed logs during test runs. The general test flow is represented in figure 6 in chapter 5.1.

The test script was initiated by a Jenkins server running locally on a host PC. The Jenkins server initiates a test run by calling the test script. After the test script has been executed, Jenkins collects the generated HTML reports and logs, and saves them with the test run. Jenkins server also sends an email notification along with the HTML test report to the defined employees' emails.

The Jenkins server can be accessed from a remote computer. This allows employees to run tests from their own computer.

3.2 System architecture

The system architecture consists of a host PC running a Jenkins server and the employees' computers connected to the office network. Static routing makes it possible to trigger the Jenkins server to run the test script from an employee's computer in the office network. This eliminates the need for physical access to the host PC. After the test run is completed the test reports are sent to the defined email addresses. The Nucleo development boards are connected via a USB hub. (Figure 1.)

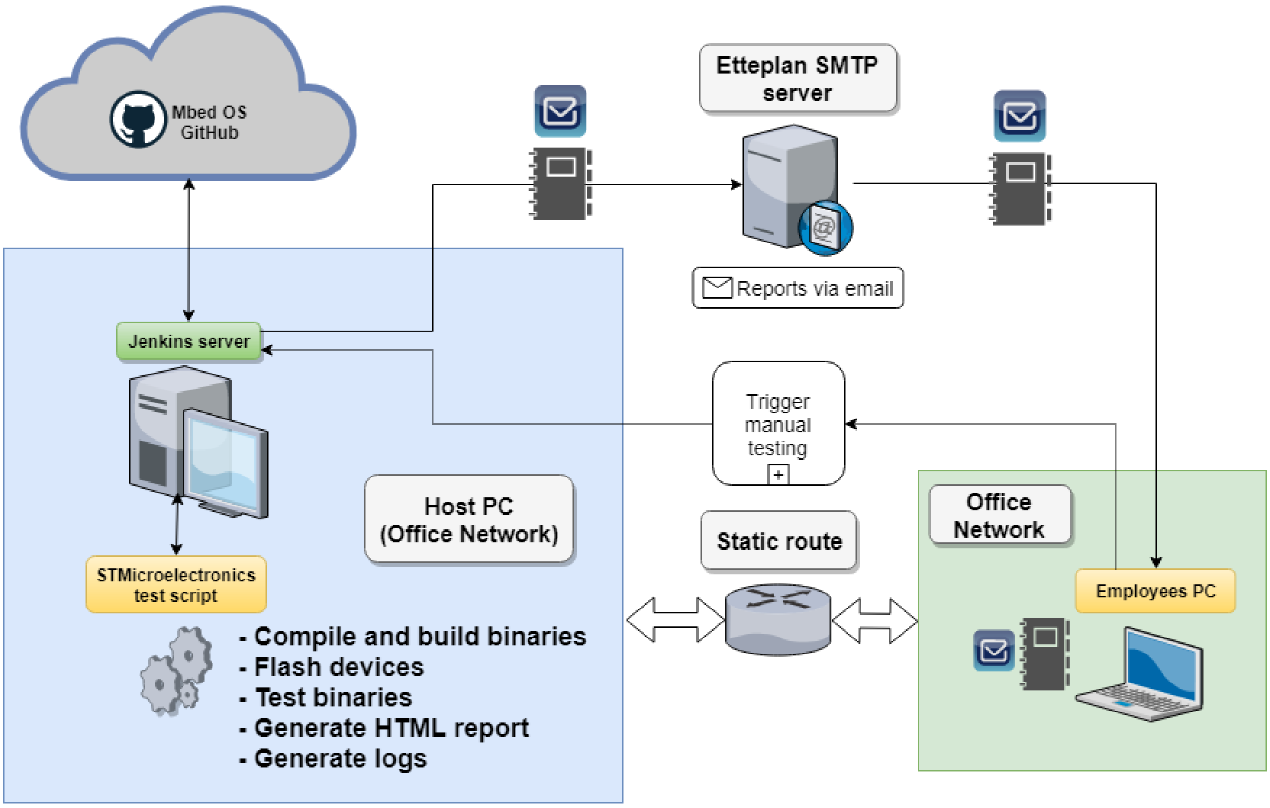


FIGURE 1. General system architecture

3.3 Directory tree structure

The overall directory tree structure or directory architecture is shown in figure 2 below. The directory tree root *STM32_CI* has three main subdirectories: *Scripts*, *Test_run*, and *temp_workspace* directories. The files in this chapter are further explained in later chapters.

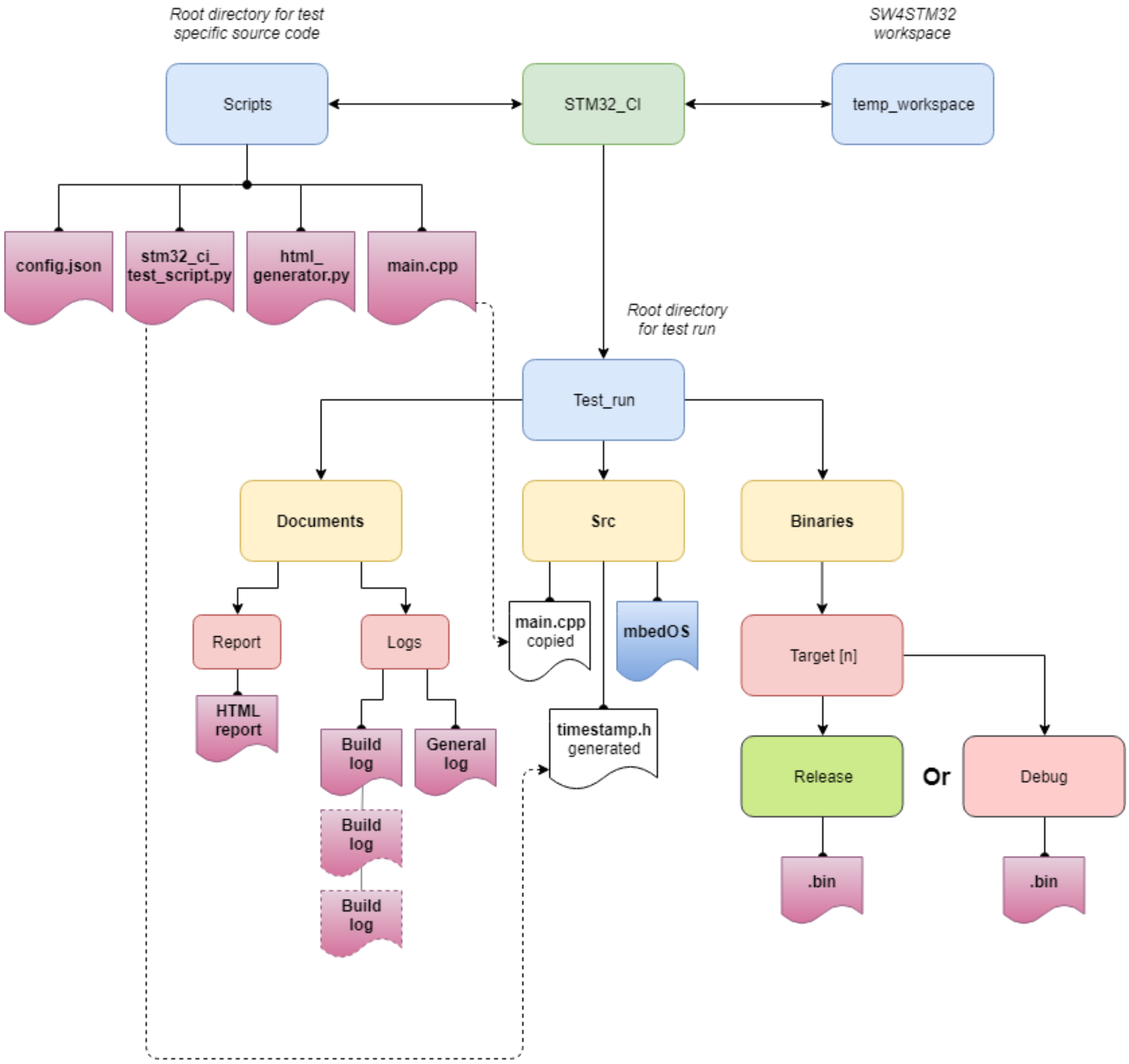


FIGURE 2. System directory tree

The *temp_workspace* directory is a temporary working directory required by the SW4STM32 IDE. It may or may not exist in the *STM32_CI* root directory depending on how it is defined in the *STM32_CI/Scripts/config.json* file. In this project, it was created in the *STM32_CI* root directory as shown in figure 2 above.

The *Scripts* directory contains the core files to implement the test automation system:

- Python test scripts
    - *stm_ci_test_script.py*

- *html_generator.py*
- Test configuration file
    - *config.json*
- C++ application for the Nucleo development boards
    - *main.cpp*

The *Test_run* directory contains all the test run specific directories and files. It is divided into three subdirectories

- Documents
- Src
- Binaries

The *Test_run/Documents* subdirectory contains the report and log files from a single test run. These files are stored in the *Report* and *Logs* subdirectories respectively. The second subdirectory, *Test_run/Src* contains the *timpesamp.h* header and *main.cpp* files. It is also the root directory for *Mbed OS*. The *Test_run/Src/mbedOS* directory contains the source code of *Mbed OS*. Finally, the *Test_run/Binaries* subdirectory contains the *.bin* binary files for the Nucleo development boards. The binary files are stored in the *Test_run/Binaries/BoardName/Release* directory. Depending on the build configuration, the last subdirectory name can be either *Release* or *Debug*. For this project the *Release* build configuration was used.

4 SYSTEM IMPLEMENTATION

In this chapter, the overall system implementation is introduced. It goes through the test setup and introduces the source files.

4.1 Development platform

The operating system chosen for the host PC was Ubuntu 16.04. In the early development phase, the Ubuntu OS was run on a virtual machine provided by Oracle VirtualBox. This way Ubuntu could be easily installed on an existing development computer already running Windows OS. Once the development work was mostly done, the Ubuntu OS was installed on the actual host PC.

Laptop computers were used as development and host PCs which do not usually have many USB ports. Thus, all ten Nucleo development boards were connected to the development PC and the host PC via a 10-port USB hub. (Figure 3.)

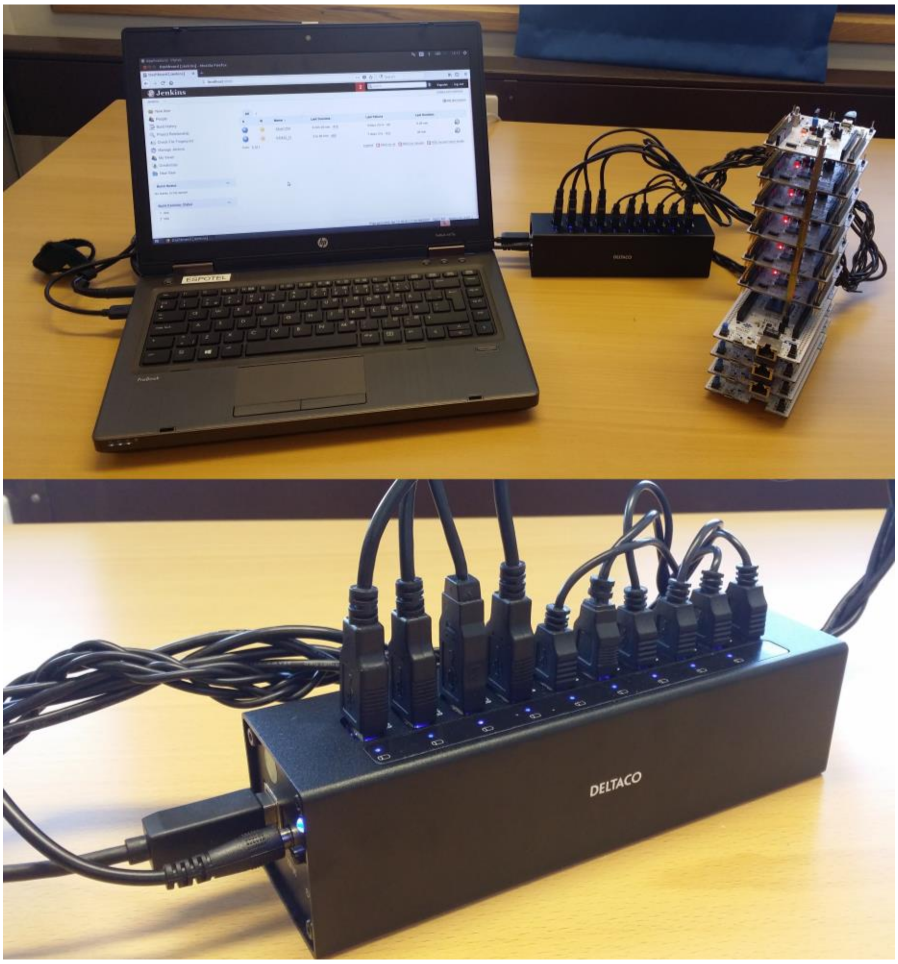


FIGURE 3. Test setup

4.2 Source files

The test system has four source code files (see figure 4). These files are the following:

- Configuration file *config.json*
- Main test script *stm32_ci_test_script.py*
- HTML generation script *html_generator.py*
- Test application for Nucleo development boards *main.cpp*

```
espotel@espotel-OU-DEMO01:~/STM32_CI/Scripts$ ls
config.json  html_generator.py  html_generator.pyc  main.cpp  stm32_ci_test_script.py
```

FIGURE 4. Source files

4.3 Test configuration file

The test configuration file config.json is used to configure test parameters. The configuration file is a JSON object that includes paths to working directories and supported boards. This chapter goes through that JSON object and explains the purpose of the JSON keys inside the config.json.

The *path* key contains all the important root directory paths for the test. The first two keys are related to the test directory. The *script-root-path* contains the path to the directory where all the test source files (scripts, configuration file, and *main.cpp* program) are located. The *test-root-path* contains the path to the test run root directory that is the directory where all the source-code, documents and binary files are created. (Figure 5.)

The next three keys are the SW4STM32 IDE related. The *systemworkbench-path* contains the path to the SW4STM32 IDE installation directory. The *tool-chain-path* contains the path to the GNU Embedded toolchain for ARM. This directory contains the GCC compiler. The last SW4STM32 IDE related key in the JSON object is the *workspace-path*. This contains the path to a SW4STM32 workspace which will be created while building binary files if it does not already exist. (Figure 5.)

The last key in the *path* is the *jenkins-job-workspace-path* key. This contains the path to the test's unique workspace in Jenkins. This directory is used to temporarily store HTML reports and logs before saving them in Jenkins. (Figure 5.)

```json
{
  "STM32": {
    "path": {
      "script-root-path": "/home/username/STM32_CI/Scripts",
      "test-root-path": "/home/username/STM32_CI/Test_run/STM32",
      "systemworkbench-path": "/home/username/Ac6/SystemWorkbench",
      "toolchain-path": "/home/username/Ac6/SystemWorkbench/plugins/fr.ac6.mcu.ext
      "workspace-path": "/home/username/STM32_CI/temp_workspace",
      "jenkins-job-workspace-path": "/var/lib/jenkins/workspace/STM32_CI"
    },
    "supported-boards": [
      "NUCLEO_F091RC",
      "NUCLEO_F103RB",
      "NUCLEO_F207ZG",
      "NUCLEO_F303ZE",
      "NUCLEO_F401RE",
      "NUCLEO_F429ZI",
      "NUCLEO_F767ZI",
      "NUCLEO_L073RZ",
      "NUCLEO_L152RE",
      "NUCLEO_L476RG"
    ]
  }
}
```

FIGURE 5. Configuration file (config.json)

5 TEST SCRIPT (*STM32_CI_TEST_SCRIPT.PY*)

The *stm32_ci_test_script.py* Python script acts as the main test script and is the heart of this project. From now on, the *stm32_ci_test_script.py* script is referred to as *the test script*.

During the following chapters the functions of the test script are introduced. On some occasions when functions with multiple parameters are introduced, the functions are referred to *function_name(params)*. Actual parameters are introduced in associated figures.

5.1 General test flow

The test script goes through four main phases (see figure 6):

- Detect devices
- Initialize test
- Build binary files
- Verify binary files

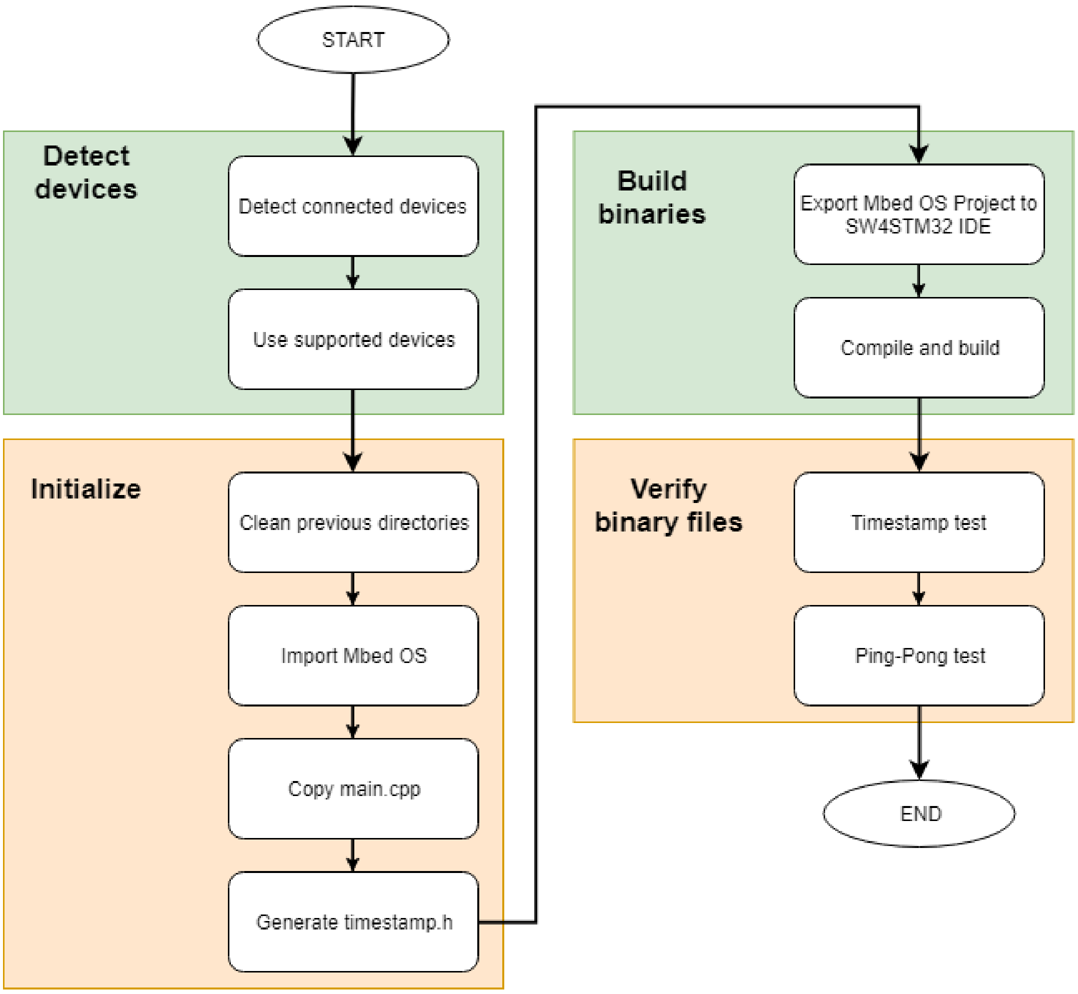


FIGURE 6. General test flow

The test script begins with saving the current timestamp (script starting time) to be used further in the test run and with some initialization phases which are explained in more detail later. All connected Nucleo development boards are also detected at the beginning of the test.

After the detection phase, the test initializes itself by removing directories related to old test runs. This gives the test a clean start every time the test is run. When old directories are removed, the test script recreates those directories.

Next, Mbed OS is imported from the GitHub repository. When Mbed OS has been downloaded, the *main.cpp* program is copied to the Mbed OS project. Also, a *timestamp.h* header file containing the starting time from the beginning of the test is created for the Mbed OS project.

At this point, the test contains all the required code for compiling and building the source code. Since the goal was to build binary files using the SW4STM32 IDE, the Mbed OS had to be exported to it. Exporting means that the Mbed OS project is exported to an external IDE and said IDE is used to compile and build the source code instead of the original Mbed CLI tools. After the Mbed OS project is exported, it is compiled and built using the SW4STM32 IDE, resulting in the targeted *.bin* binary files. (Figure 7.)

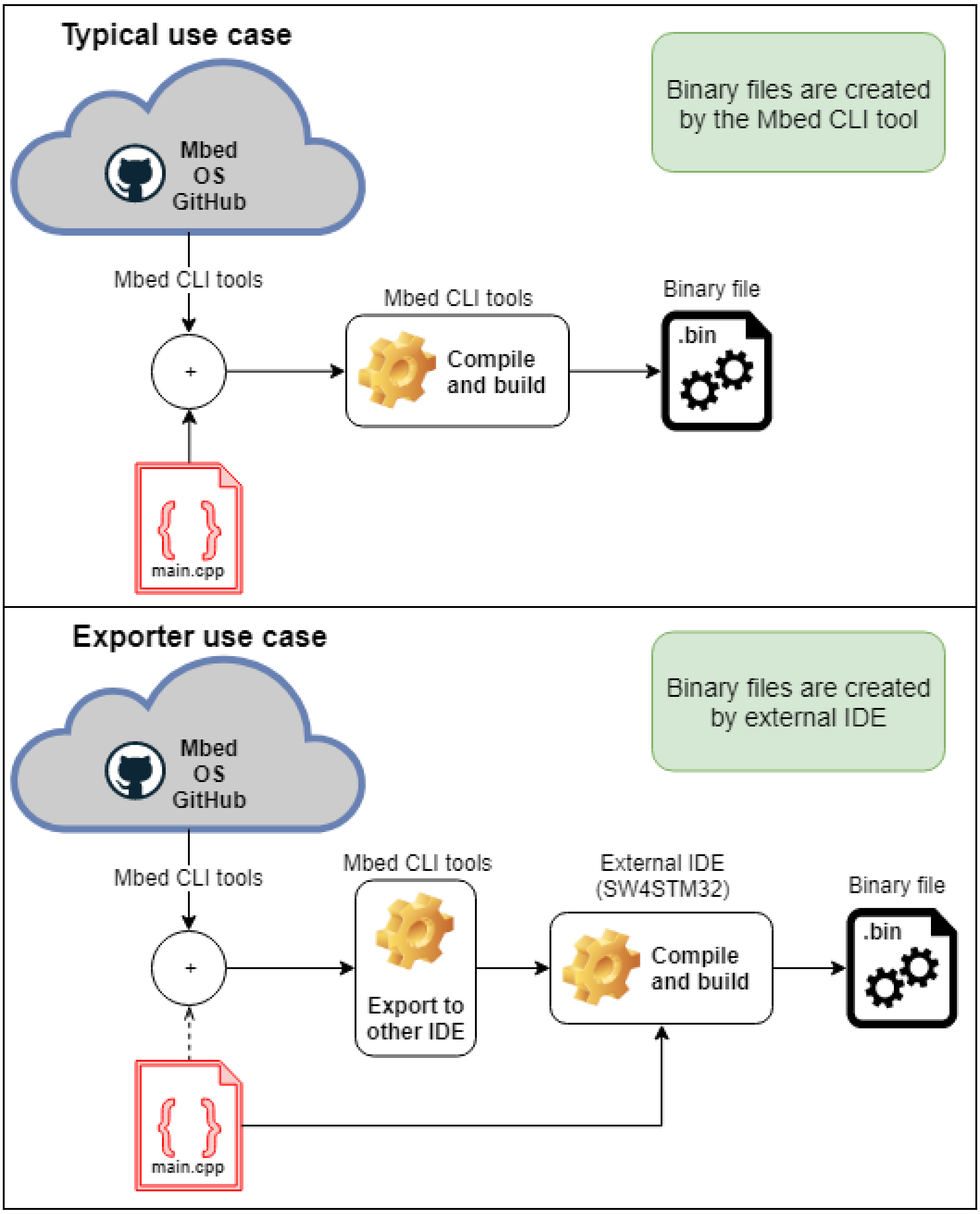


FIGURE 7. Exporter use case

To verify the integrity of the *.bin* binary files, every Nucleo development board is flashed with the binary files and tested further with a serial connection-based test sequence implemented in the test script and the *main.cpp* program. The *main.cpp* program running on the Nucleo development boards reads four characters on the serial bus and responds accordingly (see figure 62). The verification sequence is presented in figure 8 below. During the verification process, the host PC communicates with the Nucleo development boards via a USB connection by sending and receiving specific messages. The verification results depend on what messages are received on the host PC. If the received messages correspond to the expected values, the building of the *.bin* binary files has succeeded. (Figure 8.)

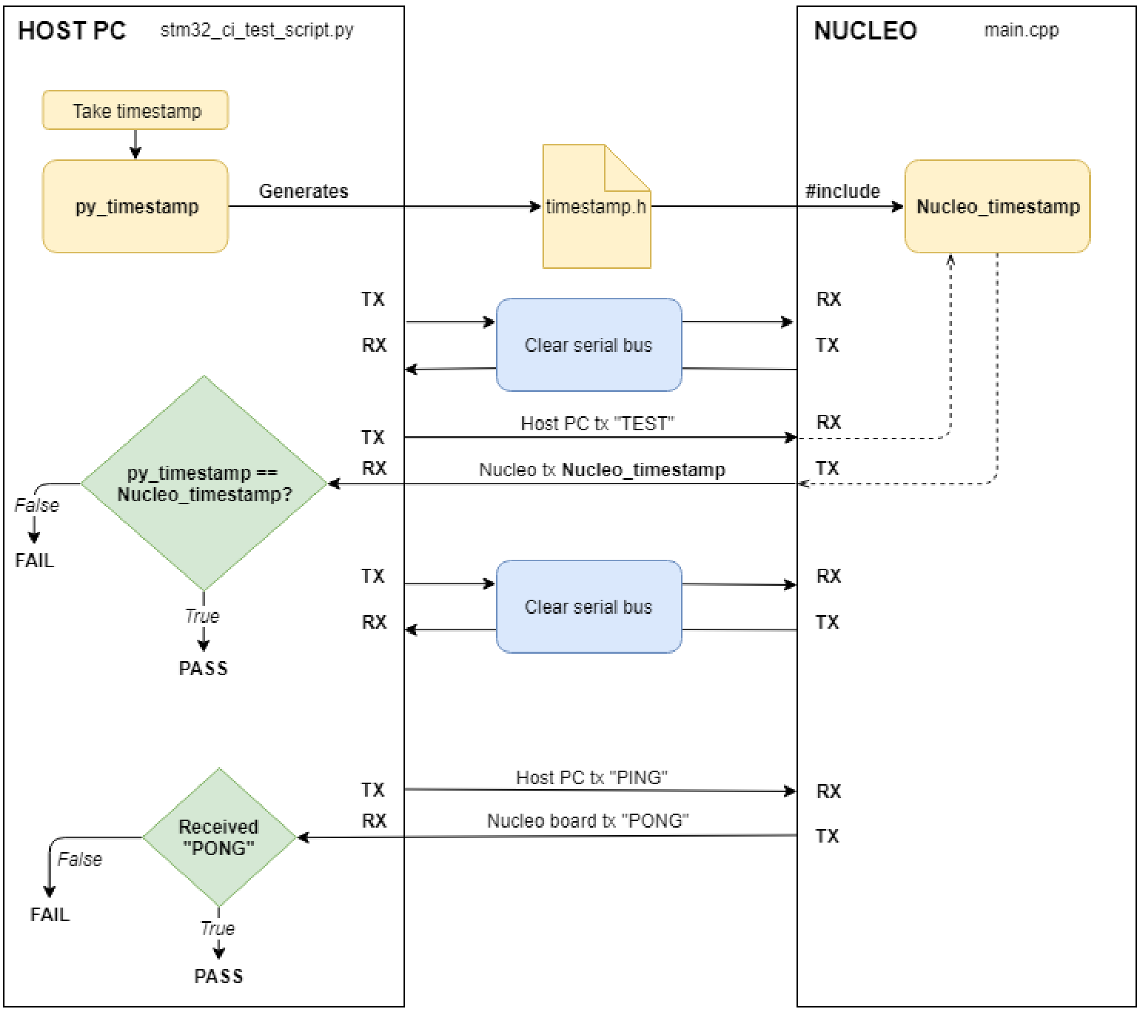


FIGURE 8. Binary file verification flow

The test script also has a logging method to collect runtime data from the test run as well as generate HTML reports. These logs and reports are discussed in detail in chapters 5.3.

5.2 Test script usage

The test script can be issued from command line by using *python stm32_ci_test_script.py*. By using the *--help* flag, all the additional arguments and their actions are shown in the screen. (Figure 9.)

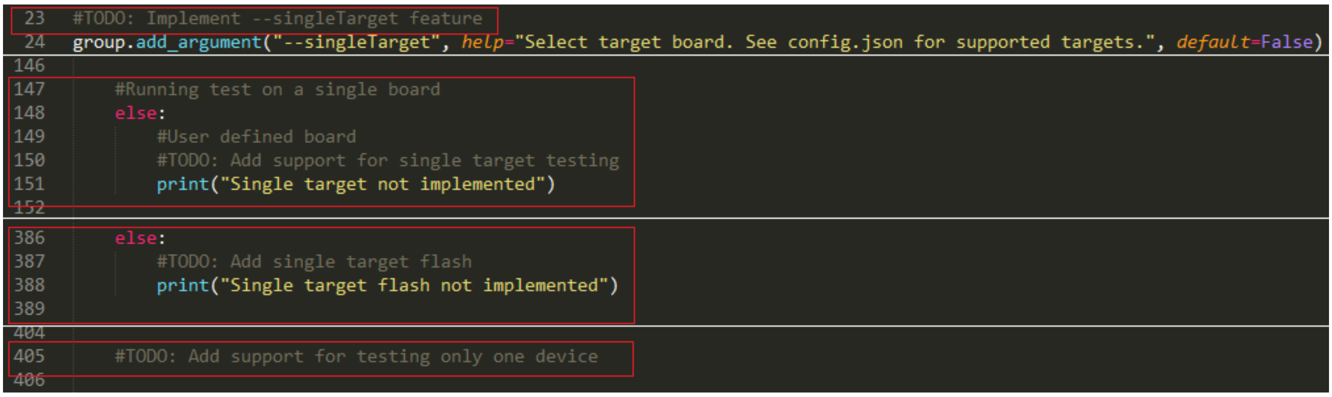
```
espotel@espotel-OU-DEMO01:~/STM32_CI/Scripts$ python stm32_ci_test_script.py --help
usage: stm32_ci_test_script.py [-h] [--branch BRANCH]
                               [--buildConfig {Release,Debug}]
                               [--singleTarget SINGLETARGET] [--skipBuild]
                               [--skipClean] [--skipFlash] [--skipTest]

optional arguments:
  -h, --help            show this help message and exit

  --branch BRANCH       Choose Mbed OS GitHub branch.
  --buildConfig {Release,Debug}
                        Choose build configuration
  --singleTarget SINGLETARGET
                        Select target board. See config.json for supported
                        targets.
  --skipBuild           Skips compile and build phases
  --skipClean           Don't remove Src directory and skip GitHub downloads
  --skipFlash           Skip flashing phase
  --skipTest            Skip testing phase
```

FIGURE 9. Usage of the test script (stm32_ci_test_script.py)

In the current version of the *stm32_ci_test_script.py,* only the *--help*, *--branch* and *--buildConfig* arguments are implemented. The *--skip** arguments are implemented but are only used for debugging and development work and thus should not be used in the final application. The *--singleTarget* argument was initially implemented for debugging and development use. Since it was a very early draft, it became obsolete and was removed during the later development phases. However, the placeholders were left in for further development purposes (see figure 10).



```
#TODO: Implement --singleTarget feature
group.add_argument("--singleTarget", help="Select target board. See config.json for supported targets.", default=False)

        #Running test on a single board
        else:
            #User defined board
            #TODO: Add support for single target testing
            print("Single target not implemented")

        else:
            #TODO: Add single target flash
            print("Single target flash not implemented")

        #TODO: Add support for testing only one device
```

FIGURE 10. Place holders for the “--singleTarget” argument

5.3 Logging

The test script generates two kinds of logs: general test log called *TEST_LOG* and Nucleo development board specific build logs called *BUILD_LOG*. The logging logic can be seen from figure 11 below.

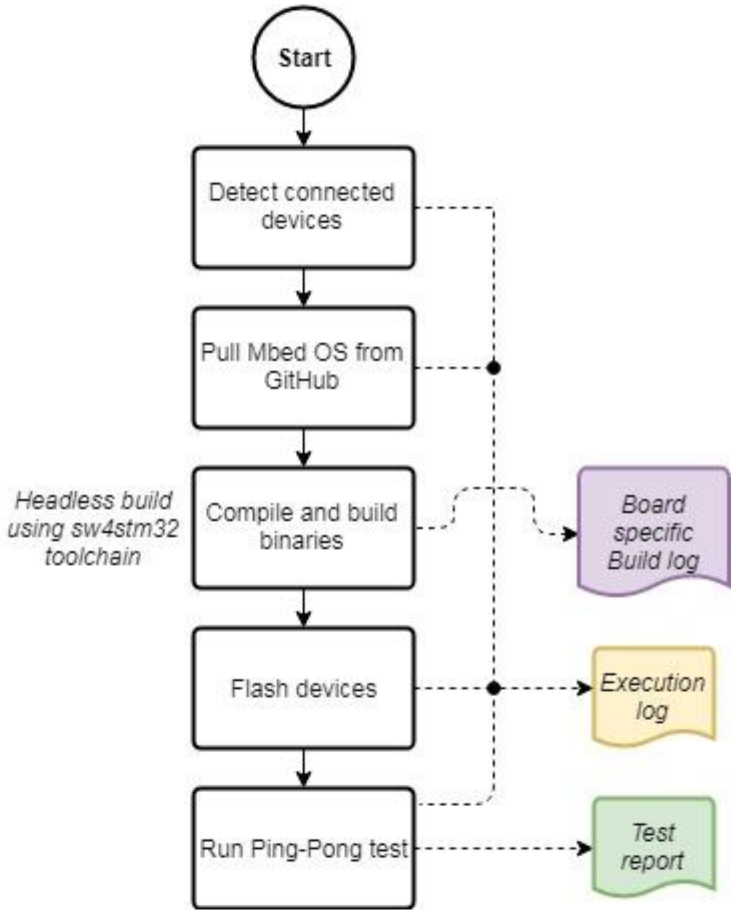


FIGURE 11. Log and report generation

Detailed information from the test execution is gathered in the *TEST_LOG* log file. Only one *TEST_LOG* file is generated during a single test run named *test_log.txt*. This *TEST_LOG* file is created at the beginning of the test script. If the *TEST_LOG* file already exists, it is overwritten. (Figure 12.)

```
#Open test log file for logging (overwrite previous log file)
TEST_LOG = open(test_log_file, 'w')
```

FIGURE 12. Create or overwrite the general log file

The methods used for logging are represented in the figure below. The *write()* function is used to add new entries such as strings and variables to the log file. Another method to add entries to the log files is the usage of *stdout* as on line 214 in figure 13. This method simply stores any output from the command issued by the *subprocess.call()* into a defined log file (in this case *TEST_LOG*). While writing to any log file, data is buffered before it is written to the file. When using the *write()* and *stdout* methods interchangeably, the log entries may not appear in chronological order due to the buffering feature. This is prevented with the use of the *flush()* function. The *flush()* function stores the buffer into the log file instantly. (Figure 13.)

```
212        TEST_LOG.write("\nFetching Mbed OS...\n")
213        TEST_LOG.flush()
214        subprocess.call('mbed new .', stdout=TEST_LOG, shell=True)
```

FIGURE 13. Logging methods

Detailed output from the compilation and build processes is gathered in the BUILD_LOG log files. Every Nucleo development board has its own *BUILD_LOG* log file named as *BoardName_build_log.txt*. The *BUILD_LOG* log files are used in the same way as the *TEST_LOG* files. (Figure 14.)

```
    #Store data from compile & build phases.
    build_log_file = "{}/{}_build_log.txt".format(log_path, board)
    BUILD_LOG = open(build_log_file, 'w')
    BUILD_LOG.write("Timestamp: {}\n".format(timestamp))
    BUILD_LOG.flush()

    #Export source tree to STM32 System Workbench IDE (SW4STM32)
    retcode = subprocess.call('mbed export -i sw4stm32 -m {}'.format(board), stdout=BUILD_LOG, shell=True)
    if retcode == 0:
        print("Export PASS")
        TEST_LOG.write("Export PASS!\n")
    else:
        print("Export FAIL")
        TEST_LOG.write("Export FAIL!\n")
    TEST_LOG.flush()

    #Compile and build via headless build
    #Refer to https://gnu-mcu-eclipse.github.io/advanced/headless-builds/, searched 11.4.2018
    retcode = subprocess.call('{}/eclipse \
        --launcher.suppressErrors \
        -nosplash \
        -application org.eclipse.cdt.managedbuilder.core.headlessbuild \
        -data {} \
        -import {} \
        -cleanBuild "Src"/{}'\
        .format(systemworkbench_path, workspace_path, src_root_path, build_configuration), stdout=BUILD_LOG, shell=True)
```

FIGURE 14. Build log generation and usage

Figure 15 below represents the generated log files. The *NUCLEO_ID_build_log.txt* files are build logs and the *test_log.txt* file is the general test log.

```
espotel@espotel-OU-DEMO01:/home/espotel/STM32_CI/Test_run/STM32/Documents/Logs
 Binaries     Logs                          11   NUCLEO_F091RC_build_log.txt
 Documents    Reports                        1   NUCLEO_F103RB_build_log.txt
 Src                                             NUCLEO_F207ZG_build_log.txt
                                                 NUCLEO_F303ZE_build_log.txt
                                                 NUCLEO_F401RE_build_log.txt
                                                 NUCLEO_F429ZI_build_log.txt
                                                 NUCLEO_F767ZI_build_log.txt
                                                 NUCLEO_L073RZ_build_log.txt
                                                 NUCLEO_L152RE_build_log.txt
                                                 NUCLEO_L476RG_build_log.txt
                                                 test_log.txt
```

FIGURE 15. Log files

5.4 Python modules

All the Python modules used are shown in figure 16 below. Chapters 5.4.1 through 5.4.10 explain what these modules are used for in the test script.

```python
import subprocess
import os
import sys
import shutil
import argparse
import datetime
import json
import time
import serial
import html_generator as HTML
```

FIGURE 16. Python modules used

5.4.1 subprocess

The *subprocess* module allows the user to spawn new processes and control input/output data pipes [11]. The *subprocess* module was used to issue Linux command line tools and commands inside the test script while saving the output data in the log files.

An example of using the *subprocess* module is given in figure 17 below. In this example, the *subprocess* module calls the Mbed-CLI tool (*mbed new .*) and stores the output data into the log file (*stdout=TEST_LOG*). (Figure 17.)

```python
    subprocess.call('mbed new .', stdout=TEST_LOG, shell=True)
```

FIGURE 17. Usage of the subprocess module

5.4.2 os

The *os* module allows the use of operating system dependent functionality [12]. In the test script the *os* module is used to create environment variables and directories, change directories, and check the existence of directories and files. In figure 18 below an example is given of checking if a directory exists and of directory creation. If the directory does not exist (*os.path.exists(documents_root_path)*), it will be created (*os.mkdir(documents_root_path)*). (Figure 18.)

```
#Create documents root directory
if os.path.exists(documents_root_path) is not True:
    os.mkdir(documents_root_path)
```

FIGURE 18. Usage of the os module

5.4.3 sys

The *sys* module provides access to the Python's interpreter variables [13]. It was used in the main script to abort the test run in case a crucial test phase fails. An example of *sys* module usage is given in figure 19 below. In this example, the python script exits (aborts the test run) if no Nucleo development boards are detected.

```
#Check if at least one device is connected. Otherwise abort test.
if nb_devices == 0:
    TEST_LOG.write("No devices found. Exiting program...\n")
    sys.exit("No devices found. Exiting program...")
```

FIGURE 19. Usage of the sys module

5.4.4 shutil

The *shutil* module offers multiple high-level file operations such as copying and removing files [14]. In the test script, the *shutil* module was used to remove directories including files with the *shutil.remtree(path)* function (see figure 20).

```
print("Remove " + binaries_root_path + "...")
TEST_LOG.write("Remove " + binaries_root_path + "...\n")
shutil.rmtree(binaries_root_path)
```

FIGURE 20. Usage of the shutil module

5.4.5 argparse

The *argparse* module makes it possible to easily write command line interfaces by allowing users to define command line arguments. Also, the *argparse* module automatically generates the *--help* argument to be used (see figure 9). [15]

The use of the *argparse* module is represented in figure 21 below.

```
18 #Commandline arguments
19 parser = argparse.ArgumentParser()
20 group = parser.add_argument_group()
21 group.add_argument("--branch", help="Choose Mbed OS GitHub branch.", default="latest")
22 group.add_argument("--buildConfig", help="Choose build configuration", choices=['Release', 'Debug'], default="Release")
23 #TODO: Implement --singleTarget feature
24 group.add_argument("--singleTarget", help="Select target board. See config.json for supported targets.", default=False)
25
26 #Used for debugging
27 group.add_argument("--skipBuild", help="Skips compile and build phases", action='store_true', default=False)
28 group.add_argument("--skipClean", help="Don't remove Src directory and skip GitHub downloads", action='store_true', default=False)
29 group.add_argument("--skipFlash", help="Skip flashing phase", action='store_true', default=False)
30 group.add_argument("--skipTest", help="Skip testing phase", action='store_true', default=False)
31 args = parser.parse_args()
```

FIGURE 21. Usage of the argparse module

In figure 21 on line 20, an argument group is created. After this, arguments are created by using the *add_argument(definitions)* function. The first parameter defines how the argument is called from the command line. The *help* parameter is the argument description. The *default* parameter defines a default value that is used if the argument is not overwritten by the user from the command line. The *choices* parameter defines the possible argument values and no other argument values are accepted. The *action* parameter defines what to do if the argument in question is called from the command line. This kind of argument does not require any additional argument when called from the command line. Finally, all the arguments are parsed on the line 31 (*args = parser.parse_args()*). From now on, the argument values can be accessed via *args.ARGUMENT* as shown in figure 22 below.

```
#Define build configuration
build_config = args.buildConfig
```

FIGURE 22. Accessing an argument value

5.4.6 datetime

The *datetime* module offers time and date manipulation tools [16]. In the test script it was used to define a timestamp by getting the starting time in the beginning of the test and storing it to the *start_time* variable. The *datetime* module was also used to further modify the timestamp into desired format. (Figure 23.)

```
#Get start time
start_time = datetime.datetime.now()
start_time = start_time.strftime("%Y-%m-%d_T%H:%M:%S")
```

FIGURE 23. Usage of the datetime module

5.4.7 json

The *json* module allows interaction between JSON objects and Python. As described, the *json* module was used to read the *config.json* configuration file [17]. The usage of *json* module is shown in figure 24 below.

```
42 '''
43     PARSE config.json FILE
44 '''
45 #Open config.json file
46 config = json.load(open('config.json'))
47 #Parse and set path variables from config.json
48 config = config["STM32"]
49 systemworkbench_path = config["path"]["systemworkbench-path"]
50 toolchain_path = config["path"]["toolchain-path"]
51 workspace_path = config["path"]["workspace-path"]
52 script_root_path = config["path"]["script-root-path"]
53 test_root_path = config["path"]["test-root-path"]
54 jenkins_job_workspace_path = config["path"]["jenkins-job-workspace-path"]
```

FIGURE 24. Usage of the json module

First, the *config.json* object is opened and loaded into the *config* variable. From now on the JSON object can be accessed via indexes like shown on the line 49 in figure 24.

5.4.8 time

The *time* module offers time related functions similar to the *datetime* module [19]. In the test script only the *time.sleep(sec)* function was used to suspend the execution of the script.

5.4.9 serial

The *serial* or *pySerial* module provides access for serial ports via Python [21]. The *serial* module was used in the *test_device(serial_port)* function in the test script to host a serial connection between the host PC and the Nucleo development boards during the binary file verification process. The usage of the *serial* module is further described in chapter 5.9.

5.4.10 html_generator

The *html_generator* imports the *html_generator.py* Python script into the main test script. The *as HTML* defines the local name for the *html_generator* module (see figure 16). This way the functions in *html_generator.py* can be accessed via *HTML.function_name(params)* method. The *html_generator.py* script is further introduced in chapters 5.10 and 5.12.

5.5 Detect devices

Detecting devices is one of the first steps to do in the *main()* function. The test script has a function called *detect_boards()* that can be used to automatically detect connected development boards. The function takes no parameters and returns the number of connected devices *nb_devices*. (Figure 25.)

```
'''
    DETECT CONNECTED DEVICES
    - Detect connected and supported boards
    - Return number of connected devices
'''
def detect_boards():
```

FIGURE 25. Declaration of the “detect_boards()” function

The *detect_boards()* function calls the command *mbed detect* of the Mbed CLI tool to list all the Mbed OS supported devices which are connected to the host PC (see figure 27). The raw output data from *mbed detect* command gives the following output data: *Detected “TARGET” connected to “MEDIA DIRECTORY” and using com port “SERIAL PORT* (see figure 26).

```
espotel@espotel-OU-DEM001:~$ mbed detect
[mbed] WARNING: The mbed tools were not found in "/home/espotel".
[mbed] WARNING: Limited information will be shown about connected mbed targets/boards
---
[mbed] Detected "NUCLEO_F207ZG" connected to "/media/espotel/NODE_F207ZG" and using com port "/dev/ttyACM5"
[mbed] Detected "NUCLEO_F767ZI" connected to "/media/espotel/NODE_F767ZI" and using com port "/dev/ttyACM7"
[mbed] Detected "NUCLEO_F429ZI" connected to "/media/espotel/NODE_F429ZI" and using com port "/dev/ttyACM3"
[mbed] Detected "NUCLEO_L476RG" connected to "/media/espotel/NODE_L476RG" and using com port "/dev/ttyACM6"
[mbed] Detected "NUCLEO_L073RZ" connected to "/media/espotel/NODE_L073RZ" and using com port "/dev/ttyACM0"
[mbed] Detected "NUCLEO_F303ZE" connected to "/media/espotel/NODE_F303ZE" and using com port "/dev/ttyACM9"
[mbed] Detected "NUCLEO_F091RC" connected to "/media/espotel/NODE_F091RC" and using com port "/dev/ttyACM1"
[mbed] Detected "NUCLEO_L152RE" connected to "/media/espotel/NODE_L152RE" and using com port "/dev/ttyACM2"
[mbed] Detected "NUCLEO_F401RE" connected to "/media/espotel/NODE_F401RE" and using com port "/dev/ttyACM8"
[mbed] Detected "NUCLEO_F103RB" connected to "/media/espotel/NODE_F103RB" and using com port "/dev/ttyACM4"
```

FIGURE 26. Raw output from the "mbed detect" command

To find the required information (target name, media directory and com port) from the data, the raw output must be processed. This is done by using the *string.split(condition)* function. The raw output data is first split from the line brake (*\n*) characters and stored in the *devices* list (see figure 27). The warning messages are not included in the *devices* list (26).

```python
    #Detect connected mbed devices
    print("Detecting mbed devices...")
    TEST_LOG.write("Detect mbed devices...\n")
    proc = subprocess.Popen('mbed detect', stdout=subprocess.PIPE, shell=True)
    #Read the stdout list from 'mbed detect', split it from new lines (\n) and store split list to "devices" variable
    output = proc.stdout.read()
    devices = output.split('\n')
```

FIGURE 27. Detecting connected devices

Every cell in the *devices* list now contains the required information from each development board. However, these cells are still merely long strings that also contain excess characters such as quotation marks, commas, and white spaces which are not relevant.

To get rid of the excess characters, the *string.split()* function is used again. This time, the data is split from quotation marks (“) and stored in the *target_data* list. The *target_data* list is a two-dimensional array containing all the required information and excess characters in separate cells per board. The *target_data* list is iterated and only target name, media directory and com port information are stored in the final two-dimensional list *target_list*. The *detect_boards()* function also checks if the connected board is supported by the test – only the supported boards are accepted in the final *target_list* list. The supported boards are defined in the *config.json* file (see figure 5). (Figure 28.)

```python
#Select all connected boards
if args.singleTarget == False:
#All connected boards
    for i in range(0, int(nb_devices), 1):
        #Split from "
        target_data = devices[i].split('"')
        #Check if connected board is supported by the conf.json
        for board in supported_boards:
            if target_data[1] == board:
                #Structure of target_list[1, 3, 5] => [TARGET_NAME, MEDIA_DIR, COM_PORT]
                target_list.append([target_data[1], target_data[3], target_data[5]])
                break
```

FIGURE 28. Parse device data

The return variable *nb_devices* is calculated from the original *devices* list. Since the *devices* list contains the information of one board per cell, it is enough to only count the number of cells. If no Nucleo development boards are detected, the test run is aborted to prevent futile test runs. (Figure 29.)

```python
#Number of connected devices
nb_devices = int(len(devices)) - 1
print("Number of devices: " + str(nb_devices))
TEST_LOG.write("Number of devices: " + str(nb_devices) + "\n")

#Check if at least one device is connected. Otherwise abort test.
if nb_devices == 0:
    TEST_LOG.write("No devices found. Exiting program...\n")
    sys.exit("No devices found. Exiting program...")
```

FIGURE 29. Calculate number of connected devices

5.6 Initialize test

Every time the test script is run it must be initialized. This means that the previous Mbed OS project must be deleted and replaced with a new one along with *main.cpp* and *timestamp.h* files. The *initialize_test()* function is used to implement these features (see figure 30). This function takes no parameters and returns two variables: *mbed_version* containing the version of current Mbed OS and *mbed_sha* containing the SHA-1 hash from GitHub. The SHA-1 hash value can be used to track down the exact version of Mbed OS to reproduce test cases if required.

```
'''
    CLEAN AND INITIALIZE TEST
    - Clear existing Mbed OS directories, create new ones and download Mbed OS
    - Copy main.cpp to Mbed OS directory
    - Generate timestamp.h file
    - Return Mbed OS version and GitHub hash
'''
def initialize_test():
```

FIGURE 30. *Declaration of the "initialize_test()" function*

5.6.1 Creating the environment variable for SW4STM32

At the beginning of the initialization function, an environment variable for the SW4STM32 IDE and ARM toolchain is created. This allows the use of SW4STM32 IDE from the command line. The environment variable is modified by appending the SW4STM32 IDE and ARM toolchain paths to the current path environment variable. (Figure 31.)

```
    #Create enviroment variable
    path_list = []
    print(os.environ.get('PATH'))
    TEST_LOG.write("\n" + str(os.environ.get('PATH')) + "\n")
    path_list.append(systemworkbench_path)
    path_list.append(toolchain_path)
    os.environ["PATH"] += os.pathsep + os.pathsep.join(path_list)
    print(os.environ.get('PATH'))
    TEST_LOG.write(str(os.environ.get('PATH')) + "\n")
    TEST_LOG.flush()
```

FIGURE 31. *Create environment variable for the SW4STM32 IDE*

5.6.2 Clean directories

To achieve a clean test run every time, the Src and Binaries directories are removed and then recreated. Also, if they do not already exist, the Documents and its subdirectories Logs and Report are created as well. (Figure 32.)

```python
    #Remove previous files and folders if exists
    if os.path.exists(src_root_path):
        print("\nRemove " + src_root_path + "...")
        TEST_LOG.write("\nRemove " + src_root_path + "...\n")
        shutil.rmtree(src_root_path)

    if os.path.exists(binaries_root_path):
        print("Remove " + binaries_root_path + "...")
        TEST_LOG.write("Remove " + binaries_root_path + "...\n")
        shutil.rmtree(binaries_root_path)

    #Create required directories. Required if --skipClean flag is used
    #Create source code root path
    if os.path.exists(src_root_path) is not True:
        os.mkdir(src_root_path)

    #Create binaries root path
    if os.path.exists(binaries_root_path) is not True:
        os.mkdir(binaries_root_path)

    #Create documents root directory
    if os.path.exists(documents_root_path) is not True:
        os.mkdir(documents_root_path)

    #Create test log directory
    if os.path.exists(log_path) is not True:
        os.mkdir(log_path)

    #Create test report directory
    if os.path.exists(report_path) is not True:
        os.mkdir(report_path)
```

FIGURE 32. Remove old directories and recreate new ones

5.6.3 Prepare Mbed OS project

The latest Mbed OS version is imported from GitHub using the *mbed new* command provided by the Mbed CLI tool. (Figure 33.)

```python
    subprocess.call('mbed new .', stdout=TEST_LOG, shell=True)
```

FIGURE 33. Usage of the "mbed new" command via subprocess module

After the Mbed OS is imported from GitHub, the current Mbed OS version and SHA-1 hash value are stored in the *mbed_version* and *mbed_sha* return variables respectively. The current Mbed OS version is resolved using the *mbed ls* command that lists all the imported libraries and their information including the Mbed OS version (Figure 34.).

```
velipp@velipp-VirtualBox:~/Thesis_work_vp/CI_STM32/STM32/Src/mbed-os$ mbed ls
mbed-os (#16bac101a6b7, tags: latest, mbed-os-5.7.7)
```

FIGURE 34. Raw output from the "mbed ls" command

The version number is parsed from this output using the same *split()* function as when detecting devices and further stored in the *mbed_version* variable. The SHA-1 hash value can be shown with *git rev-parse HEAD* command. This output is stored in the *mbed_sha* variable. (Figure 35.)

```python
    os.chdir("{}/mbed-os".format(src_root_path))

    #Parse current mbed-os version and git hash
    proc = subprocess.Popen('mbed ls', stdout=subprocess.PIPE, shell=True)
    output = proc.stdout.read()
    temp = output.split(" ")
    temp = temp[4].split(")")
    mbed_version = temp[0]

    TEST_LOG.write("Mbed OS Version: {}\n".format(mbed_version))
    TEST_LOG.flush()

    #Parse Mbed OS GitHub SHA hash
    proc = subprocess.Popen('git rev-parse HEAD', stdout=subprocess.PIPE, shell=True)
    mbed_sha = proc.stdout.read()
    TEST_LOG.write("Mbed OS GitHub SHA: {}\n".format(mbed_sha))
```

FIGURE 35. Parse the Mbed OS version and SHA-1 hash value

The final steps in the initialization are to copy the main.cpp file and to generate the *timestamp.h* header file into the Mbed OS project directory. Finally, the existence of *timestamp.h* header file is checked. If the file cannot be found, the test run is aborted to prevent a futile test run. (Figure 36.)

```
    #Copy test application (main.cpp) and to the Mbed OS directory root
    print("Copying main.cpp to Mbed OS...")
    TEST_LOG.write("\nCopy main.cpp to Mbed OS...\n")
    TEST_LOG.flush()
    subprocess.call('cp -v {}/{} {}'.format(script_root_path, "main.cpp", src_root_path), stdout=TEST_LOG, shell=True)

    print("Create timestamp.h file")
    TEST_LOG.write("\nCreate timestamp.h file\n")
    os.chdir(src_root_path)
    #Create timestamp.h header file
    with open("timestamp.h", "w") as header_file:
        header_file.write("#define TIMESTAMP \"{}\"".format(start_time))

    #Check if timestamp.h header file was created
    if os.path.exists(src_root_path + "/timestamp.h"):
        print("timestamp.h created")
        TEST_LOG.write("timestamp.h created\n")
    else:
        TEST_LOG.write("Could not create timestamp.h -> Aborting test...\n")
        sys.exit("Could not create timestamp.h -> Aborting test...")

    return mbed_version, mbed_sha
```

FIGURE 36. Prepare main.cpp and timestamp.h files

5.7 Build binaries

The *build_binary(board, build_configuration, timestamp)* function is used to build *.bin* binary files for Nucleo development boards. The function takes three parameters and returns *build_result* describing whether the build process was *PASS* or *FAIL*. (Figure 37.)

```
'''
    BUILD BINARY FILE
    - Export Mbed OS to SW4STM32 IDE
    - Compile and build .bin file using headless build mode
    - Return build result (PASS/FAIL)
'''
def build_binary(board, build_configuration, timestamp):
```

FIGURE 37. Declaration of the “build_binary(params)” function

The *board* parameter defines the Nucleo development board for which the binary file is built. The *board* parameter is mainly required by Mbed CLI tool to export the Mbed OS project into the SW4STM32 IDE, but it is used for logging and creating directories as well. The *build_configuration* parameter defines the build configuration, i.e., if the binary files are either built in *release* or *debug* version. The default build configuration is *release*. The last parameter *timestamp* is used to write the start time of the test run to build logs.

The *build_binary()* function begins with a removal of old binary directories from previous builds (see figure 38).

```
    #Change to the mbed-os root
    os.chdir("{}/mbed-os".format(src_root_path))

    #Remove Debug and Release directories from the project (mbed-os root)
    if os.path.exists("Debug"):
        shutil.rmtree("Debug")
    if os.path.exists("Release"):
        shutil.rmtree("Release")

    #Change back to source root
    os.chdir(src_root_path)
```

FIGURE 38. Remove old binary directories

Next, the Mbed OS project is exported to the SW4STM32 IDE using Mbed CLI's *mbed export -i -m* command. The *mbed export* takes two flags: *-i* defining the IDE to be used and *-m* defining the target board. (Figure 39.)

```
    #Export source tree to STM32 System Workbench IDE (SW4STM32)
    retcode = subprocess.call('mbed export -i sw4stm32 -m {}'.format(board), stdout=BUILD_LOG, shell=True)
```

FIGURE 39. Export Mbed OS project to the SW4STM32 IDE

After exporting the Mbed OS project to the SW4STM32 IDE, a headless build process is initiated to compile and build *.bin* binary files (see figure 40). The headless build allows developers to compile and build software without any GUI's (graphical user interface) using a build script instead. Since the SW4STM32 IDE is based on Eclipse IDE, it is possible to use the headless build feature supported by the Eclipse IDE itself. Usage of the headless build is represented in figure 40 below. Further reading about the Eclipse headless build can be found at https://gnu-mcu-eclipse.github.io/advanced/headless-builds/ (date of retrieval: 19.4.2018).

```
    #Compile and build via headless build
    #Refer to https://gnu-mcu-eclipse.github.io/advanced/headless-builds/, searched 11.4.2018
    retcode = subprocess.call('{}/eclipse \
        --launcher.suppressErrors \
        -nosplash \
        -application org.eclipse.cdt.managedbuilder.core.headlessbuild \
        -data {} \
        -import {} \
        -cleanBuild "Src"/{}'\
        .format(systemworkbench_path, workspace_path, src_root_path, build_configuration), stdout=BUILD_LOG, shell=True)

```

FIGURE 40. Headless build

Finally, the *.bin* binary file is copied to corresponding directory (test directory). The test script also verifies that *.bin* file exists. If the corresponding file is found, the return variable *build_result* is set to *PASS*. If the file was not found the *build_result* is set to *FAIL*. (Figure 41.)

```
    TEST_LOG.write("Copy .bin file to test directory\n")
    TEST_LOG.flush()
    #Copy .bin file to board specific directory
    subprocess.call('cp -v ./{}/Src.bin {}/{}/{}'.format(build_config, binaries_root_path, board, build_config), \
        stdout=TEST_LOG, shell=True)

    #Check if .bin files are generated
    if os.path.exists("{}/{}/{}/Src.bin".format(binaries_root_path, board, build_config)):
        build_result = "PASS"
        print("Success! .bin file generated for board " + board + "\n")
        TEST_LOG.write("Success! .bin file generated for board " + board + "\n")
    else:
        build_result = "FAIL"
        print("Fail! .bin file not found for board " + board + "\n")
        TEST_LOG.write("Fail! .bin file not found for board " + board + "\n")

    TEST_LOG.flush()

    return build_result
```

FIGURE 41. Checking if the binary file exists

5.8 Flash device

To test the integrity of the *.bin* binary files they must be flashed to the corresponding development boards. This is done in the *flash_device(board, board_dir)* function. The *flash_device(board, board_dir)* function takes two parameters and returns the *flash_result* variable containing the *PASS/FAIL* result from the flashing process. The *board* parameter describes which board's binary file is going to be used and the *board_dir* describes the media directory where the Nucleo development board is mounted in the Linux system. (Figure 42.)

```
'''
    FLASH DEVICE
    - Copy .bin file to device
    - Return flash result (PASS/FAIL)
'''
def flash_device(board, board_dir):
```

FIGURE 42. Declaration of the ”flash_device(params)” function

Flashing a Nucleo development board is done by simply copying the *.bin* binary file to the media directory of selected Nucleo development board. The copying process is retried five times at most in case the process fails. The flashing process is verified as *PASS* or *FAIL* depending on the return value (*retcode*) from the function calling the copying method. Finally, the *flash_result* variable is returned. (Figure 43.)

```
        #Flash the device by copying binary file to target boards media directory
        for i in range(0, 5, 1):
            TEST_LOG.write("Copy .bin file to {}\n".format(board_dir))
            TEST_LOG.flush()
            retcode = subprocess.call('cp -v {}/{}/{}/Src.bin {}'.format(binaries_root_path, board, build_config, board_dir), \
                stdout=TEST_LOG, shell=True)

            if retcode == 0:
                flash_result = "PASS"
                print("Flash PASS")
                TEST_LOG.write("Flash PASS!\n")
                break
            else:
                flash_result = "FAIL"
                print("Flash FAIL")
                TEST_LOG.write("Flash FAIL! Retries {}/5\n".format(i))

    else:
        #TODO: Add single target flash
        print("Single target flash not implemented")

    return flash_result
```

FIGURE 43. Flash device by copying the .bin file to the Nucleo development board

5.9 Verify integrity of binary files

In this chapter the Nucleo development boards are expressed as device under test (DUT). The verification process is represented in figure 8 in chapter 5.2.

The integrity of *.bin* binary files are verified in the *test_device(serial_port)* function. This function takes in one parameter *serial_port* and returns the *timestamp_result* and *ping_result* variables containing the results of this verification process. The *serial_port* parameter is used to host a USB serial connection between the host PC and the Nucleo development boards. The *test_device(serial_port)* function is mainly based on the pySerial module. (Figure 44.)

```python
'''
    TEST DEVICE
    - Run Timestamp test
    - Run Ping-Pong test
    - Return timestamp and ping-pong test results (PASS/FAIL)
'''
def test_device(serial_port):
```

FIGURE 44. Declaration of the "test_device(serial_port)" function

The verification process begins by hosting a serial connection between the host PC and the DUT. The *serial_port* parameter defines which serial port is used to create the connection. The serial connection is created on the *ser* (short for serial) object that can be further used to interact with the DUT. After creating the *ser* object, the serial connection is verified to be open. If the serial connection is closed, the script retries to open it at maximum of three times. (Figure 45.)

```python
    #Test boards over serial connection
    TEST_LOG.write("\nBegin test sequence:\n")
    try:
        #Create serial connection (COM PORT, BAUD, TIMEOUT)
        with serial.Serial(serial_port, 9600, timeout=2) as ser:
            ser.flush()
            print("\nBegin test at {}".format(serial_port))
            TEST_LOG.write("Begin test at {}\n".format(serial_port))

            #Check if serial connecion has been established. Try opening COM port again if closed
            for i in range(0, 3, 1):
                if ser.is_open != True:
                    print("COM closed")
                    TEST_LOG.write("COM closed\n")
                    ser.open()
                    time.sleep(1)
                else:
                    print("COM open")
                    TEST_LOG.write("COM open\n")
                    break
```

FIGURE 45. Open serial connection between host PC and DUT

After the serial connection is established, the serial bus is cleared from any excess data. The DUT reads four characters from the serial bus and responds according to the received buffer. How the *main.cpp* program works is discussed in more detail later. The *stm32_ci_test_script.py* script writes the *0* character to the serial bus and waits for the respond from the DUT. If the respond is not received, the previous sequence is repeated until four characters containing *0000* are received from the DUT. The clearing sequence is repeated up to the maximum of ten times to prevent infinite loop in case the DUT does not respond at all. (Figure 46.)

```
            #Clear any extra garbage from serial bus. Send character "0" until "0000" buffer is received
            print("Clear serial bus:")
            TEST_LOG.write("Clear serial bus:\n")
            for i in range(0,10,1):
                try:
                    #Write "0" to serial bus
                    ser.write("0")
                    #Read 4 characters from serial bus
                    foo = ser.read(4)
                except:
                    print("Device returned no data (device disconnected or multiple acces on port?)")
                    TEST_LOG.write("Device returned no data (device disconnected or multiple acces on port?)\n")
                    break
                #Received four characters containing "0000"
                if len(foo) == 4 and foo == "0000":
                    print("Serial bus cleared!")
                    TEST_LOG.write("Serial bus cleared!\n")
                    serial_ready = 1
                    break
```

FIGURE 46. Clearing serial bus

If the serial bus is cleared successfully, the script writes the *TEST* string to the serial bus and waits for an answer from the DUT by reading twenty characters from the serial bus. The DUT responds with the timestamp that was stored at the beginning of the test and given for the DUT in the *timestamp.h* header file. If the received buffer from DUT matches the timestamp, the test has been successful and the *timestamp_result* is set to *PASS*. If the received buffer does not match with the timestamp, the test fails and the *timestamp_result* is set to *FAIL*. (Figure 47.)

```
                if serial_ready == 1:
                    #Ask for timestamp from DUT
                    TEST_LOG.write("Send \"TEST\"\n")
                    try:
                        ser.write("TEST")
                        timestamp = ser.read(20)
                        print(timestamp)
                        TEST_LOG.write("Received: {}\n".format(timestamp))

                        if timestamp == start_time:
                            #PASS
                            timestamp_result = "PASS"
                        else:
                            #FAIL
                            timestamp_result = "FAIL"
```

FIGURE 47. The timestamp test

After verifying the timestamp, the serial bus is cleared again. Next, a Ping-Pong test is initiated. This phase works with the same principle as the timestamp test: The script writes *PING* to the serial bus and waits for an answer from DUT but reading only four characters this time. The DUT should answer with *PONG*. If the received buffer from the DUT matches with *PONG,* the test is successful and the *ping_result* is set to *PASS*. If the received buffer does not match with *PONG,* the *ping_result* is set to *FAIL*. This Ping-Pong test is repeated three times to ensure the serial connection is still intact after the timestamp test. Finally, the *timestamp_result* and *ping_result* variables are returned. (Figure 48.)

```
                    #Run PING-PONG test three times
                    for i in range(0, 3, 1):
                        TEST_LOG.write("Send \"PING\"\n")
                        ser.write("PING")
                        answ = ser.read(4)
                        print(answ)
                        TEST_LOG.write("Received: {}\n".format(answ))
                        if answ == "PONG":
                            #PASS
                            ping_result = "PASS"
                        else:
                            #FAIL
                            ping_result = "FAIL"
                            break
                except:
                    TEST_LOG.write("Error! Could not write or read serial!\n")
    except:
        #Serial connection could not be established
        TEST_LOG.write("Error! Could not establish serial connection!\n")

    print("Test done!\nResults:\nTimestamp: {}\nPING-PONG: {}\n".format(timestamp_result, ping_result))
    TEST_LOG.write("Test done!\nResults:\nTimestamp: {}\nPING-PONG: {}\n\n".format(timestamp_result, ping_result))
    TEST_LOG.flush()
    return timestamp_result, ping_result
```

FIGURE 48. Running the Ping-Pong test

5.10 HTML test report generation

From every test run, an HTML test report is generated. The purpose of this test report is to present the test results and general information from test run. An example of the HTML test report is represented in figure 49 below.

Back to STM32_CI | general_report

STM32 test report

Test info:

Start time: *2018-04-13_T09:27:35*

No. included boards: *10*

Mbed OS version: *mbed-os-5.8.2*

Git branch: *latest*

Git hash: *f9ee4e849f8cbd64f1ec5fdd4ad256585a208360*

Test results:

Target	Overall result	Create .bin file	Flash device	Timestamp test	Ping-Pong test
NUCLEO_F207ZG	**PASS**	PASS	PASS	PASS	PASS
NUCLEO_F767ZI	**PASS**	PASS	PASS	PASS	PASS
NUCLEO_F429ZI	**PASS**	PASS	PASS	PASS	PASS
NUCLEO_L476RG	**PASS**	PASS	PASS	PASS	PASS
NUCLEO_L073RZ	**PASS**	PASS	PASS	PASS	PASS
NUCLEO_F303ZE	**PASS**	PASS	PASS	PASS	PASS
NUCLEO_F091RC	**PASS**	PASS	PASS	PASS	PASS
NUCLEO_L152RE	**PASS**	PASS	PASS	PASS	PASS
NUCLEO_F401RE	**PASS**	PASS	PASS	PASS	PASS
NUCLEO_F103RB	**PASS**	PASS	PASS	PASS	PASS

FIGURE 49. HTML test report

The HTML test report is generated via *html_generator.py* Python script. The script dynamically creates an HTML document during test run by simply feeding

HTML syntax into a *.html* file. HTML documents are created with the same procedure as the log files. The desired HTML syntax is first defined as a string variable which is stored to the *.html* file.

The generation of the HTML test report begins with the *start_html(html_file_path)* function. This function creates and initializes the HTML test report with default HTML syntax. If the HTML test report already exists, it is overwritten. The *html_file_path* parameter defines the directory path and the name of the *.html* file. (Figure 50.)

```
'''
    INITIATE HTML REPORT
'''
def start_html(html_file_path):
    #HTML syntax
    html_syntax = """
    <!DOCTYPE html>
    <html>
    <head>
    <title>
    </title>
    <meta name=\"viewport\" content=\"width=device-width, initial-scale=1\">
    <style>
        body {
            background-color: #ffffff;
            background-repeat: no-repeat;
            background-position: top left;
            background-attachment: fixed;
        }
        h1 {
            font-family: Arial, sans-serif;
            color: #000000;
            background-color: #ffffff;
        }
        p {
            font-family: Georgia, serif;
            font-size: 18px;
            font-style: normal;
            font-weight: normal;
            color: #000000;
            background-color: #ffffff;
        }
        table,th,td {
            border: 1px solid black;
        }
    </style>
    </head>
    <body>"""

    #Write HTML syntax to HTML file
    with open(html_file_path, 'w') as report:
        report.write(html_syntax)
```

FIGURE 50. Definition of the "start_html(html_file_path)" function

Filling information to the HTML test report is done by using two specific fun-tioncs: *fill_general_info(params)* and *fill_board_info(params)*.

The *fill_general_info(params)* function creates and fills in the *Test info* section (Figure 49.). The defined parameters are embedded into the *html_syntax* string variable. In the *fill_general_info(params)* function the *html_syntax* also creates the heading columns for the test result table (see figure 49). Finally, the *html_syntax* string variable is appended into the *.html* file. (Figure 51.)

```
'''
    FILL IN GENERAL TEST INFO
'''
def fill_general_info(html_file_path, start_time, nb_boards, mbed_os_version, git_branch, git_hash):
    html_syntax = """
    <h1>STM32 test report</h1>
    <h2>Test info:</h2>
    <p>Start time: <i>{}</i></p>
    <p>No. included boards: <i>{}</i></p>
    <p>Mbed OS version: <i>{}</i></p>
    <p>Git branch: <i>{}</i></p>
    <p>Git hash: <i>{}</i></p>
    <h2>Test results:</h2>
    <table>
        <tr>
            <th>Target</th>
            <th>Overall result</th>
            <th>Create .bin file</th>
            <th>Flash device</th>
            <th>Timestamp test</th>
            <th>Ping-Pong test</th>
        </tr>
    """.format(start_time, nb_boards, mbed_os_version, git_branch, git_hash)

    with open(html_file_path, 'a') as report:
        report.write(html_syntax)
```

FIGURE 51. Declaration of the “fill_general_info(params)” function

The *fill_board_info(params)* creates a new entry for a single board and fills in the test results. As in the previous function, the parameters are embedded in the *html_syntax* string variable and appended into the *.html* file. In the *fill_board_info(params)* function, a new table row including the board name and test results is created. Also, the test results are colored in either red, green, or black depending on what the test result was. (Figure 52.)

```
'''
    FILL BOARD INFO AND TEST RESULTS
'''
def fill_board_info(html_file_path, board, overall_result, binary_result, flash_result, timestamp_result, pinpong_result):
    overall_result_color = define_result_color(overall_result)
    binary_result_color = define_result_color(binary_result)
    flash_result_color = define_result_color(flash_result)
    timestamp_result_color = define_result_color(timestamp_result)
    pinpong_result_color = define_result_color(pinpong_result)

    html_syntax = """
        <tr>
        <td>{}</td>
            <td><span style="color:{}; font-weight:bold">{}</span></td>
            <td><span style="color:{}">{}</span></td>
            <td><span style="color:{}">{}</span></td>
            <td><span style="color:{}">{}</span></td>
            <td><span style="color:{}">{}</span></td>
        </tr>
    """.format(board, overall_result_color, overall_result,\
     binary_result_color, binary_result, flash_result_color, flash_result,\
     timestamp_result_color, timestamp_result, pinpong_result_color,  pinpong_result)

    with open(html_file_path, 'a') as report:
        report.write(html_syntax)
```

FIGURE 52. Declaration of the “Fill_general_info(params)” function

The *define_result_color(result)* function is used to define the color of the *PASS/FAIL* result in the HTML test report. The function takes in the test result as the *result* paramter and returns a color as a string according to the value of the *result* parameter. (Figure 53.)

```
'''
    DEFINE TEXT COLOR
'''
def define_result_color(result):
    if result == "PASS":
        return "green"
    elif result == "FAIL":
        return "red"
    elif result == "N/A":
        return "black"
```

FIGURE 53. Declaration of the "define_result_color(result)" function

Finally, the *.html* file is terminated with the *end_html(html_file_path)* function. This function simply feeds the closing tags to the *.html* file. (Figure 54.)

```python
'''
    END HTML REPORT
'''
def end_html(html_file_path):
    html_syntax = "</table></body></html>"

    with open(html_file_path, 'a') as report:
        report.write(html_syntax)
```

FIGURE 54. Declaration of the "end_htlm(html_file_path)" function

5.11 Initializing the Jenkins workspace

When a test run comes to its end and all the test run specific documents are generated, the documents are copied into the Jenkins workspace. Since build logs require a considerably large amount of storage space, all the log files are packed and compressed into a single *Logs.tar.gz* archive file. This process is implemented in the *prepare_jenkins_workspace()* function (see figure 55).

```python
'''
    PREPARE JENKINS WORKSPACE
    - Compress and copy log files to Jenkins workspace
    - Copy HTML test report to Jenkins workspace
'''
def prepare_jenkins_workspace():
```

FIGURE 55. Declaration of the "prepare_jenkins_workspace()" function

The log files are first packed using the *tar* tool. This tool is used to archive and compress files and directories. The use of the *tar* tool is shown in figure 56 below. The *-czvf* flag defines the actions the *tar* tool performs: *-cf* flags create a new archived file and *-z* flag compresses the archive. The *-v* flag stands for verbose which outputs detailed information about the *tar* process. After the required flags are defined, the path to the archive file is given, followed by the path to the source. (Figure 56)

```python
    retcode = subprocess.call(('sudo tar -czvf {}/Logs.tar.gz {}').format(jenkins_job_workspace_path, log_path),
```

FIGURE 56. Archiving and compressing the log files to the Jenkins workspace

The HTML test report document is copied to the Jenkins workspace as well (see figure 57).

```
    #Copy the HTML report to Jenkins STM32 job workspace
    retcode = subprocess.call(('cp {} {}').format(report_file, jenkins_job_workspace_path), stdout=TEST_LOG, shell=True)
    if retcode == 0:
        print("HTML report copied succesfully")
        TEST_LOG.write("HTML report copied succesfully\n")
    else:
        print("HTML report copy failed!")
        TEST_LOG.write("HTML report copy failed!\n")
```

FIGURE 57. Copy the HTML test report to Jenkins workspace

5.12 Main function

The *main()* function defines the test logic and uses the *html_generator.py* script to generate HTML reports (see figure 6). The *main()* function takes no parameter nor returns anything.

The *main()* function begins with saving the starting time to the test log. Next a dictionary variable called *test_results* is declared. In Python, dictionaries are similar to lists but instead of indexing the cells with numbers, the cells are indexed with keys. For example, in the *test_results* dictionary *Binary* is the key and *N/A* is the value of the key. The *test_result* dictionary contains the following keys:

- *Overall*
    - Contains overall test result
- *Binary*
    - Contains *.bin* binary file build result
- *Flash*
    - Contains flashing process result
- *Timestamp*
    - Contains the timestamp test result
- *PingPong*
    - Contains the Ping-Pong test result

In addition, two string variables are declared: *mbed_version* and *mbed_sha*. (Figure 58.)

The variables described above are initially set to *N/A (*not available). This is in case something goes wrong and some steps are not completed. For example, if serial the serial connection cannot be hosted the, *Timestamp* and *PingPong* tests will not be run. In this case, the test results for those cases would be *N/A* instead of *FAIL*. This indicates that the test cases were not run at all instead of falsely indicating test failures. In addition, *N/A* is more informative than returning an empty result.

```
'''
    MAIN LOOP
'''
def main():
    print("Test started at {}".format(start_time))
    TEST_LOG.write("Test started at {}\n".format(start_time))

    #Dictionary to store test results during every run
    test_results = {"Overall": "", "Binary": "N/A", "Flash": "N/A", "Timestamp": "N/A", "PingPong": "N/A"}
    mbed_version = "N/A"
    mbed_sha = "N/A"
```

FIGURE 58. Declaration of the "main()" function

The first step is to detect connected devices by calling the *detect_boards()* function. As discussed earlier in chapter *5.5*, the *detect_boards()*, the function returns the number of connected devices. This number is stored to the *nb_devices* variable. (Figure 59.)

Next, the test environment is initialized using the *initialize_test()* function. This function returns the version of Mbed OS and the SHA-1 hash value. These values are stored in the *mbed_version* and *mbed_sha* variables respectively. (Figure 59.)

After the devices are detected and the test environment is initialized, the *main()* function prepares the HTML test report document. This is done with the *HTML.start_html(report_file)* function call where the *report_file* is the path to the defined *.html* file. Another part of preparing the HTML test report document is to fill in the general test information (see figure 6). This information is filled in using the *HTML.fill_general_info(params)*. (Figure 59.)

```
#Detect boards
nb_devices = detect_boards()

#Initialize test
if not args.skipClean:
    temp = initialize_test()
    mbed_version = temp[0]
    mbed_sha = temp[1]

#Initialize HTML report
HTML.start_html(report_file)

#Fill in general test information
HTML.fill_general_info(report_file, start_time, nb_devices, mbed_version, args.branch, mbed_sha)
```

FIGURE 59. Test run initialization

Next, the *main()* function proceeds with a *build – flash – verify* sequence for each Nucleo development board at a time. The results from these phases are stored in the *test_results* dictionary in corresponding keys. Each Nucleo development board is iterated in a *for* loop. The number of times the *for* loop runs the said sequence is defined by the *nb_devices* variable. This way every detected Nucleo development board is taken into account. (Figure 60.)

The *.bin* binary file is built using the *build_binary(params)* function. This function returns the *PASS/FAIL* result as discussed in chapter *5.7*. The result is stored in the *Binary* key. (Figure 60.)

Next, the integrity of the *.bin* binary file is verified. This begins by flashing the DUT using the *flash_device(params)* function. As described in chapter *5.8*, the function returns the *PASS/FAIL* result from the flashing process. This result is stored to the *Flash* key. After the DUT is flashed, the verification process can be initiated. This is done by calling the *test_device(com_port)* function. The *test_decive(com_port)* function runs the *timestamp* and *Ping-Pong* tests and returns the *PASS/FAIL* results from them. These results are stored in the *Timestamp* and *PingPong* keys respectively. (Figure 60.)

Once the *build – flash – verify* sequence is completed, an overall test result is defined. This is done by examining if any of the phases failed, i.e., whether a

FAIL or *N/A* value exists in the *test_results* dictionary. If one or both of the values are found, the *Overall* key is set to *FAIL*. Otherwise, the *Overall* key is set to *PASS,* indicating that the test run was successfully completed for current DUT. (Figure 60.)

Finally, the results in the *test_results* dictionary are saved in the HTML test report. This is done by calling the *HTML.fill_board_info(params)* function. After the results are saved in the HTML test report, the *test_results* dictionary is reinitialized with the *N/A* values for the next *build – flash – verify* sequence run. Yet again, the sequence described above is repeated until every detected Nucleo development board is tested. (Figure 60.)

```python
    #Begin the test sequence
    for i in range(0, int(nb_devices), 1):
        target = target_list[i][0]
        media_dir = target_list[i][1]
        com_port = target_list[i][2]

        #Build .bin file
        if not args.skipBuild:
            test_results["Binary"] = build_binary(target, build_config, start_time)

        #Flash device
        if not args.skipFlash:
            test_results["Flash"] = flash_device(target, media_dir)

        #Test device
        if not args.skipTest:
            temp = test_device(com_port)
            test_results["Timestamp"] = temp[0]
            test_results["PingPong"] = temp[1]

        #Check overall result. If any test case fails, the overall result is "FAIL". Otherwise "PASS".
        #If N/A was found Overall is set to "N/A"
        if "FAIL" in test_results.values():
            test_results["Overall"] = "FAIL"
        elif "N/A" in test_results.values():
            test_results["Overall"] = "FAIL"
        else:
            test_results["Overall"] = "PASS"

        #Fill test results into HTML report
        HTML.fill_board_info(report_file, target, test_results["Overall"], test_results["Binary"],\
            test_results["Flash"], test_results["Timestamp"], test_results["PingPong"])

        #Reset test results for next run
        test_results = {"Overall": "", "Binary": "N/A", "Flash": "N/A", "Timestamp": "N/A", "PingPong": "N/A"}

        TEST_LOG.write("Progress: {}/{} completed\n\n\n".format((i+1),nb_devices))
        TEST_LOG.flush()
```

FIGURE 60. Build and verify .bin files

After the test sequence is completed for every detected Nucleo board, the HTML test report is terminated by using the *HTML.end_html(report_file)*. The next step is to archive the log files and copy the HTML test report document to the Jenkins workspace. This is simply done by calling the *prepare_jenkins_workspace()* function. Finally, the ending information is stored to the log file and the log file is closed. (Figure 61.)

```

    #End HTML report
    HTML.end_html(report_file)

    print("Compress \"Logs\" directory")
    TEST_LOG.write("Compress \"Logs\" directory\n")
    TEST_LOG.flush()

    #Copy HTML report and Log files to Jenkins workspace
    prepare_jenkins_workspace()

    print("Process completed!")
    TEST_LOG.write("Process completed!\n")

    #Get end time
    end_time = datetime.datetime.now()
    end_time = end_time.strftime("%Y-%m-%d_T%H:%M:%S")

    print("Test ended at {}".format(end_time))
    TEST_LOG.write("Test ended at {}".format(end_time))
    TEST_LOG.close()

```

FIGURE 61. Test run finishing

6 C++ APPLICATION FOR NUCLEO DEVELOPMENT BOARDS

The *main.cpp* program, written in the C++ programming language, implements the test application for Nucleo development boards. The *main.cpp* program reads the serial bus of a Nucleo development board and answers to the received message buffer accordingly. The general work flow is represented in figure 62 and an explanation is given below.

The *main.cpp* program includes the *timestamp.h* header file generated during the initialization phase in the *stm32_ci_test_script.py* script. The *timestamp.h* header file contains the starting time (timestamp) of the test run.

The *main.cpp* program reads four characters from the serial bus. If the received buffer was *TEST*, the timestamp from the *timestamp.h* header file is written to the serial bus. Alternatively, if the received buffer was *PING*, then *PONG* is written to the serial bus. However, if the received buffer does not match neither of the above, the received buffer itself is written to the serial bus. The latter feature is typically used to clear the serial bus.

The source code for the *main.cpp* can be found in appendix 1 at the end of this document.

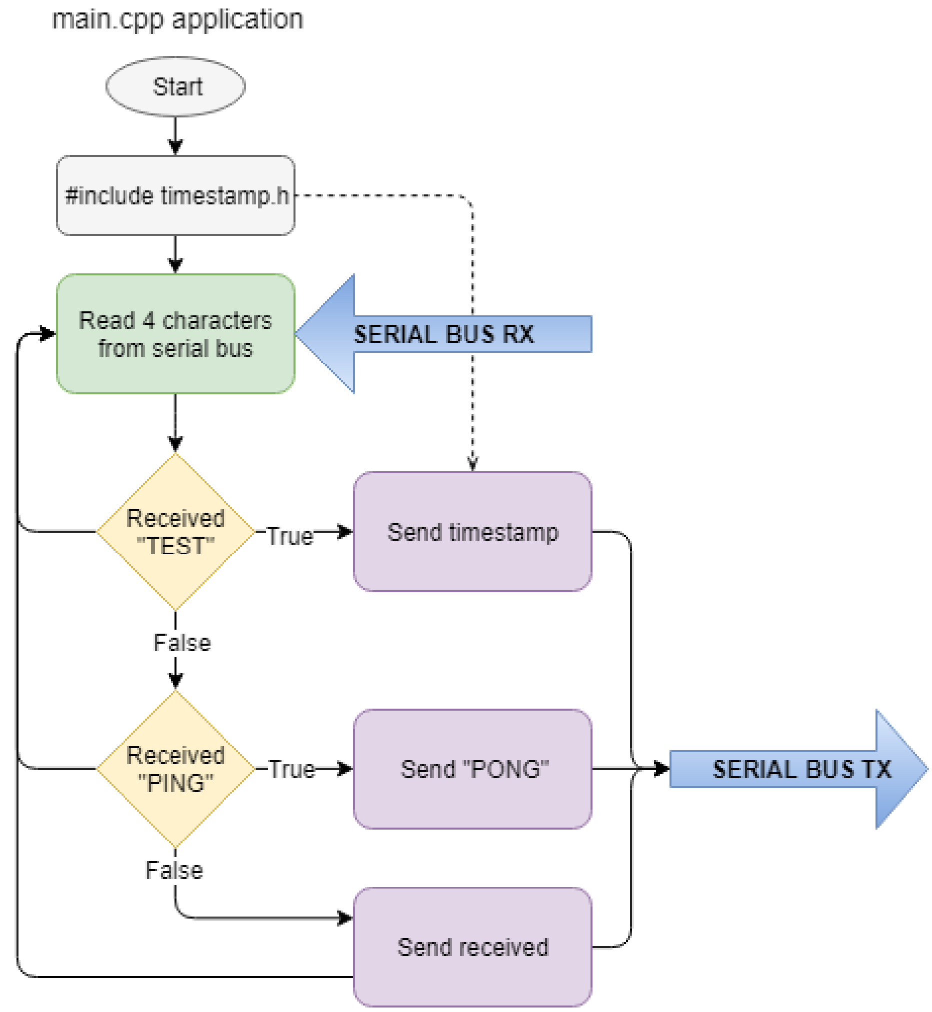


FIGURE 62. Flow chart of the "main.cpp" application

7 JENKINS SERVER

A Jenkins server was hosted to offer a simple user interface for the test system. The Jenkins server made it easy to run the test script and access test reports and logs from a remote computer. The Jenkins server was also created to offer Etteplan a general-purpose test server for other projects. The Jenkins server was installed on the host PC.

This thesis will not go into detail regarding the Jenkins since Jenkins is a rather complex system itself. Furthermore details about the Jenkins configuration were not introduced in this document for security reasons.

7.1 Jenkins installation

This chapter introduces the very basics on how to install Jenkins server. Since the Ubuntu Linux distribution was used on the host PC, this introduction mainly applies to Linux operating systems.

The Jenkins server was installed following the instructions on the Jenkins web page. The Jenkins server was installed following these commands:

```
wget -q -O - https://pkg.jenkins.io/debian/jenkins.io.key | sudo apt-key add
-
sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ >
/etc/apt/sources.list.d/jenkins.list'
sudo apt-get update
sudo apt-get install Jenkins
```

By using these commands, Jenkins was downloaded, installed and pre-configured to get the user started. After this, the Jenkins server could be accessed via web browser at the address *http://localhost:8080*. When the Jenkins server was accessed for the first time, the user was guided through a *Post-installation setup wizard*. [22]

Further reading about the installation process can be found at https://jenkins.io/doc/book/installing/ (date of retrieval: 25.4.2018).

7.2 Jenkins configuration

The Jenkins server was configured according to the requirements set by the features designed. Due to security reasons, only general description is given in this chapter.

To gain remote access to the Jenkins server, the host PC running Jenkins was connected to a separate router. The router was configured to have a static IP address to provide a static access point for the host PC. By default, the Jenkins server listens to the port 8080. The port 8080 was forwarded in the router to allow employees to connect to the Jenkins server inside Etteplan's network. Finally, the static IP address was also mapped in the Jenkins server.

To allow the Jenkins server to send emails, SMTP settings had to be configured. These settings were configured by filling in Etteplan's SMTP server information and by defining the *System Admin e-mail address* in the Jenkins system configuration.

7.3 Jenkins architecture design

Jenkins' general architecture is represented in figure 63 below. In Jenkins, users can create test cases called *jobs*. When these *jobs* are executed, they create instances from the jobs called *builds*. These *builds* use the w*orkspace* as their actual working area while the test case is running. For example, the *workspace* can be used to temporarily store log files.

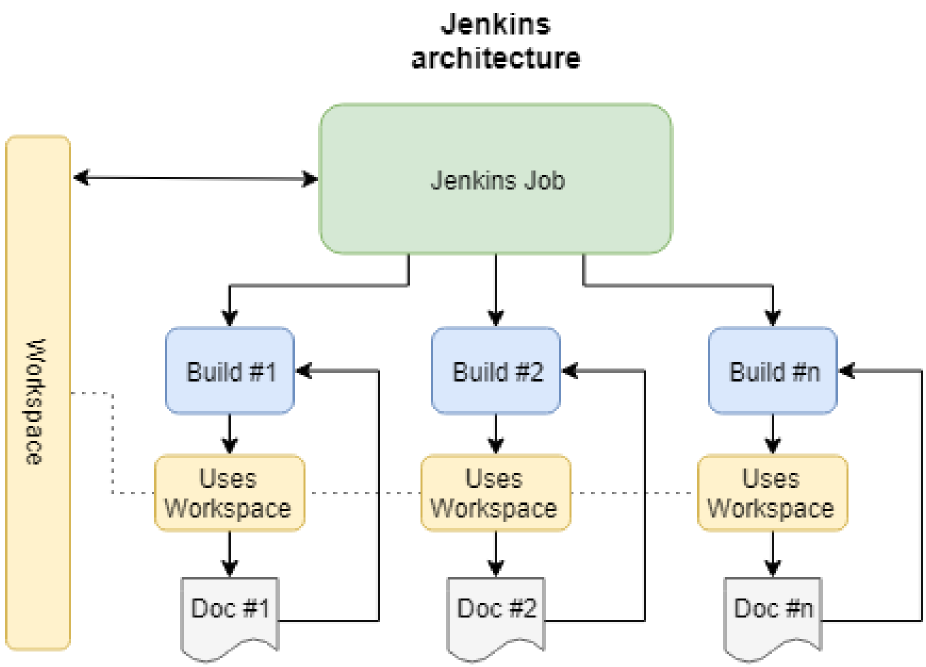


FIGURE 63 Jenkins' general architecture

7.3.1 Jobs

As already mentioned, Jenkins allows the user to create *jobs*. These *jobs* define the implementation and behavior of test cases. For example, during this thesis a Jenkins *job* was created to run the Python test script, store the test documents and send an email notification to defined employees.

7.3.2 Builds

When a Jenkins *job* is initiated, a new instance from the *job* is created. These instances are called *builds*. *Builds* are test run specific and they hold information from each initiated test run (*build)*. Jenkins saves a finite amount of *builds* that can be examined afterwards. For example, a developer can compare a build that was done a week ago with the latest *build*. Previous builds can be examined from the jobs *Build History* panel (see figure 64).

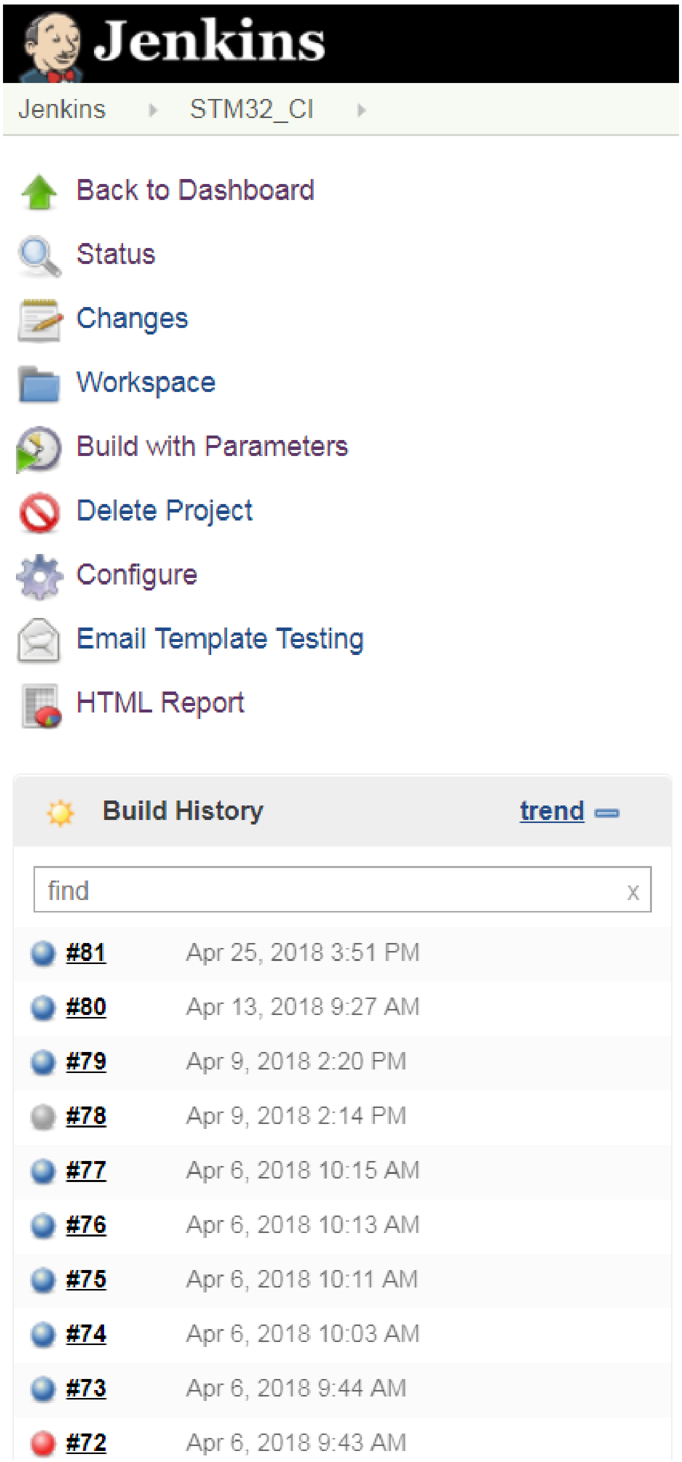


FIGURE 64. STM32_CI job's Build History panel

7.3.3 Workspaces

Worskpaces are unique, temporary working directories for Jenkins *jobs*. For example, *workspaces* can be used to temporarily store log files. Jenkins can be further configured to archive any files or directories from the *workspace* for more permanent storing.

7.4 The STM32_CI Jenkins job

A Jenkins job called *STM32_CI* was created for the test automation system. This job was used to automatically run the Python test script, store the test documents, and send email notifications to the defined employees.

The Python test script is executed from the *STM32_CI* job by using the *Execute shell* build step option. This allows the usage of command line commands in Jenkins. First, current working directory is changed to the *Scripts* root directory (*cd /home/espotel/STM32_CI/Scripts*) where the *stm32_ci_test_script.py* Python script is located. After this, the test script can be executed with the command *sudo python stm32_ci_test_script.py*. (Figure 65.)

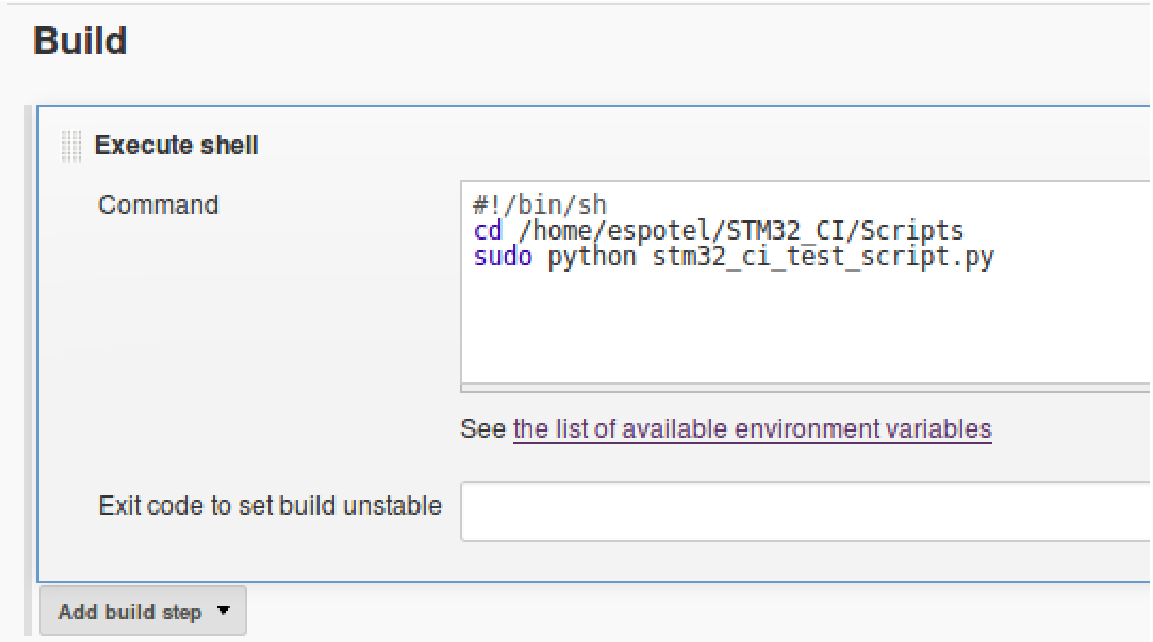


FIGURE 65. STM32_CI Build configuration

7.4.1 Workspace usage

In the *STM32_CI* Jenkins job, the workspace was used to temporarily store the HTML test report and the log files during every build. The *stm32_ci_test_script.py* Python script copies the generated HTML test report document and compressed log archives to the *STM32_CI* workspace. The *STM32_CI* job was configured to store these files. This was done in the Jenkins job's *Post-build Actions* configuration menu. (Figure 66.)

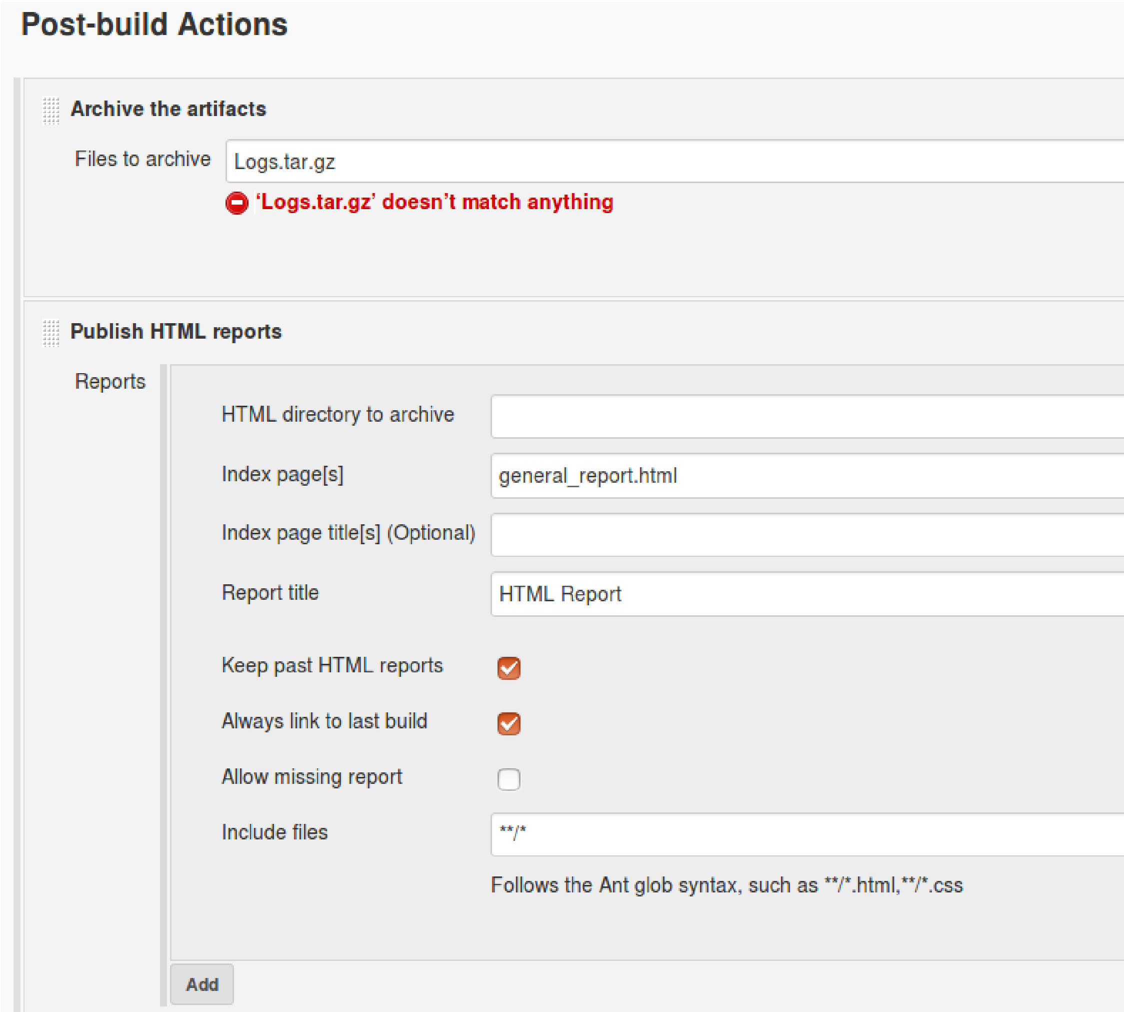


FIGURE 66. Archive HTML report and log files from STM32_CI workspace

Another *Post-build Action* configured was the *Delete workspace when build is done*. This cleans the workspace from the defined files. The *STM32_CI* job was configured to delete the HTML test report and the log archives. (Figure 67.)

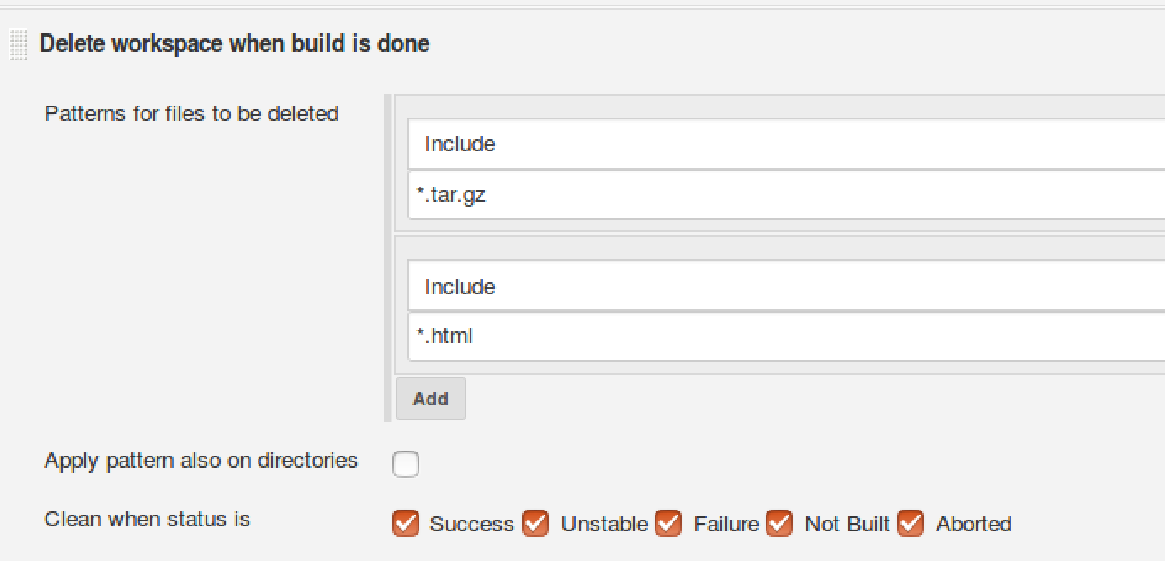


FIGURE 67. Cleaning the STM32_CI workspace

7.4.2 Email notifications

The *STM32_CI* job was configured to send the HTML test report via email after every build. This feature was configured in the *Post-build Actions – Editable Email Notification* section. The *Project Recipient List* defines to whom the email is sent. In this case, email notifications are sent to pre-defined recipients as well as additional recipients that can be defined at the beginning of every build. (Figure 68.)

Every email notification includes the HTML test report. This was made possible by the defining attachments in the *Attachments* section: Every file in the workspace ending with *html* is included in emails. Also, the link to the build in question is included in the email notifications. (Figure 68.)

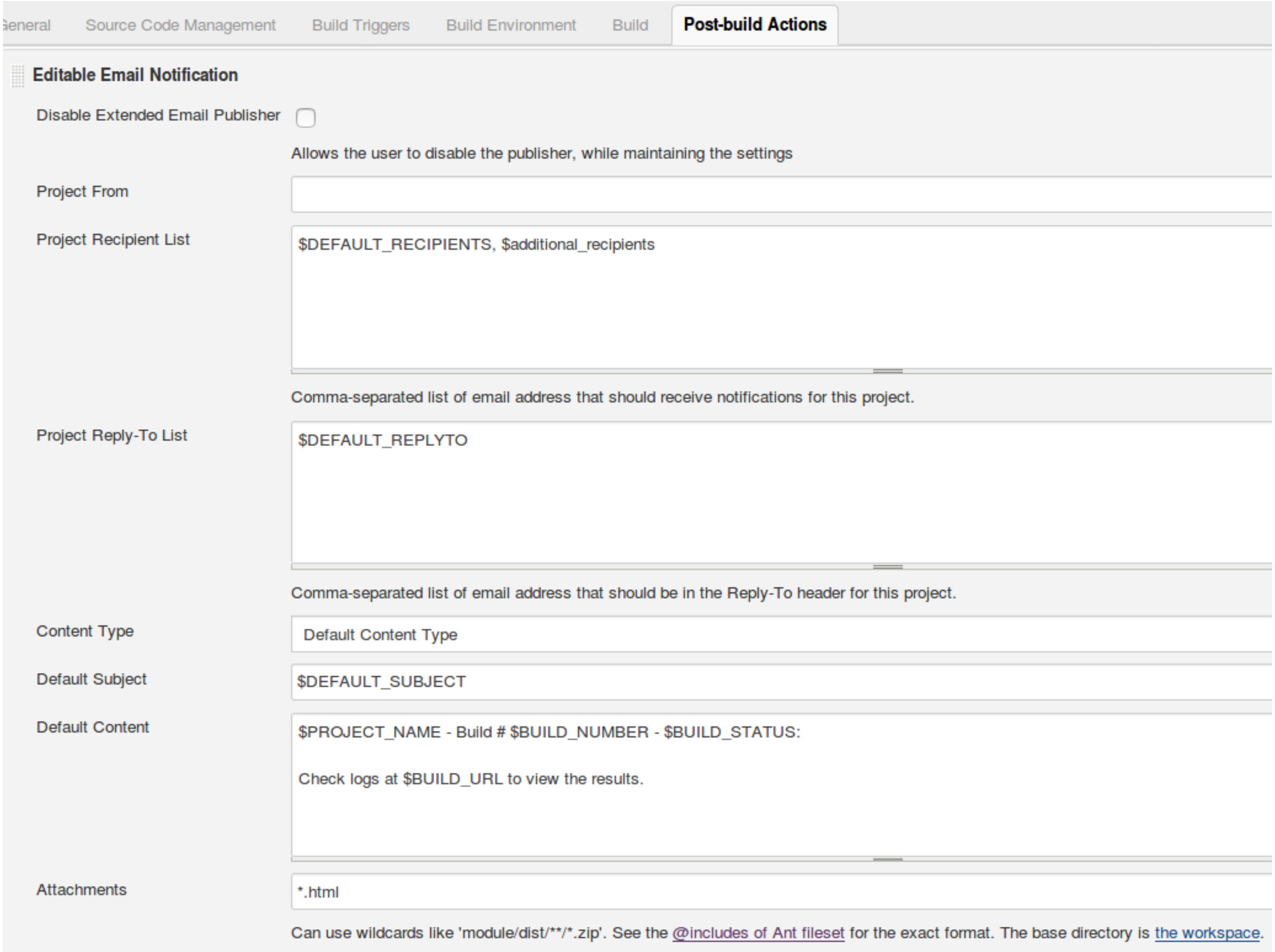


FIGURE 68. STM32_CI email notification configuration

Figure 69 below an example is given of an email notification. The email notification includes the HTML test report as an attachment (*general_report.html*). Also, a link to the job is given. (Figure 69.)

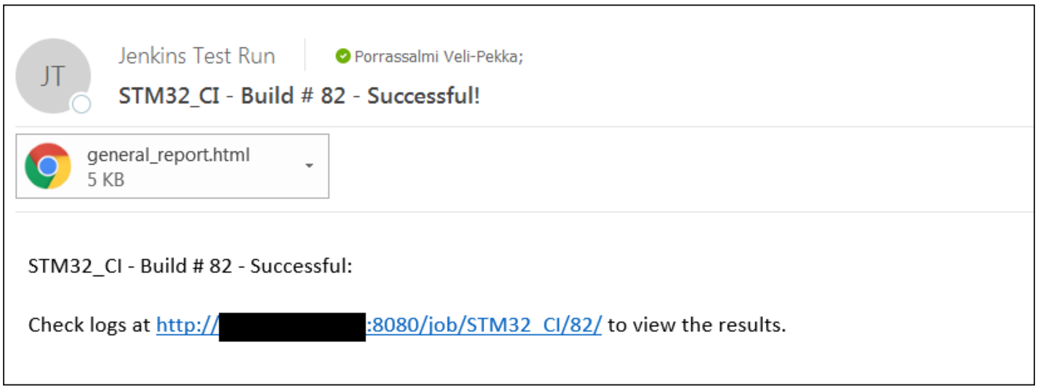


FIGURE 69. Email notification

7.4.3 Usage (Initiating build)

The *STM32_CI* job can be initiated from the Jenkins' home page by clicking the *Build* button on the right-hand side. This creates a new build from the *STM32_CI* job. (Figure 70.)

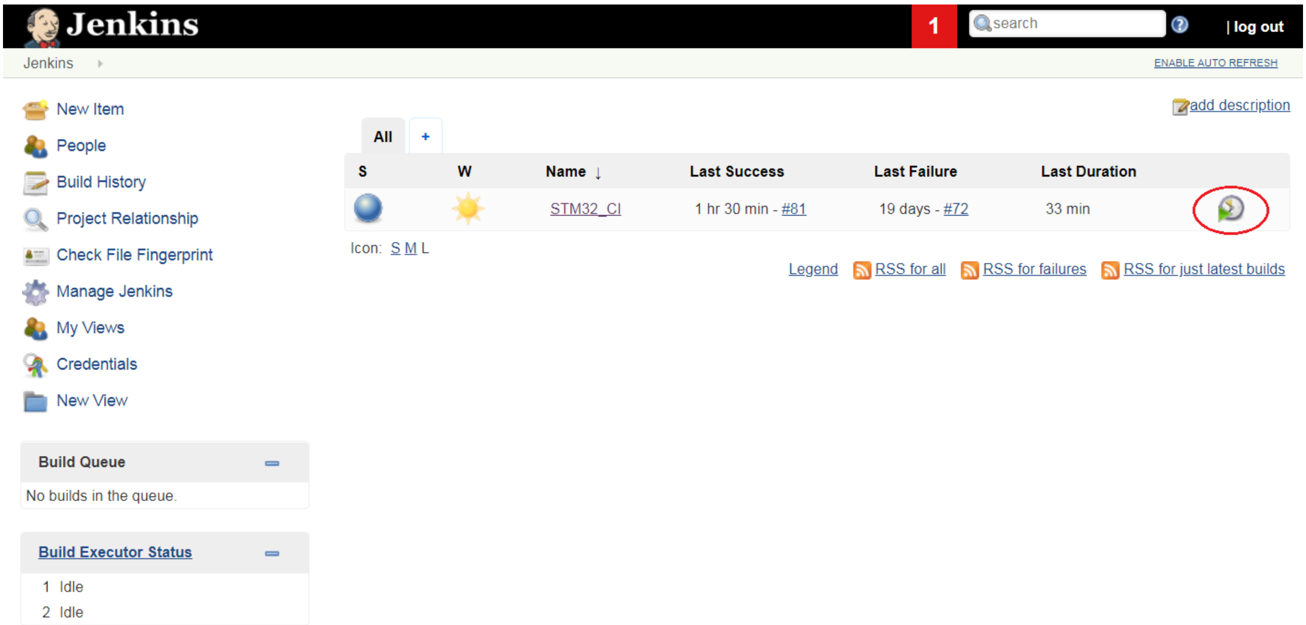


FIGURE 70. Jenkins front page. The "build" button circled.

After initiating a new build from the STM32_CI job, the user is given an option to add additional email notification recipients in the *additional_recipients* field. If multiple recipients are to be defined, the email addresses must be separated by commas. Also, if no additional recipients are to be defined, the *additional recipients* field is left empty. (Figure 71.)

FIGURE 71. Option for adding additional email notification recipients

After additional email notification recipients are defined, Jenkins begins executing the *STM32_CI* job (see figure 72). Thus, the Python test script will be run, HTML test report and log files will be generated, and the test results are sent via email to the defined employees with a single press of a button.

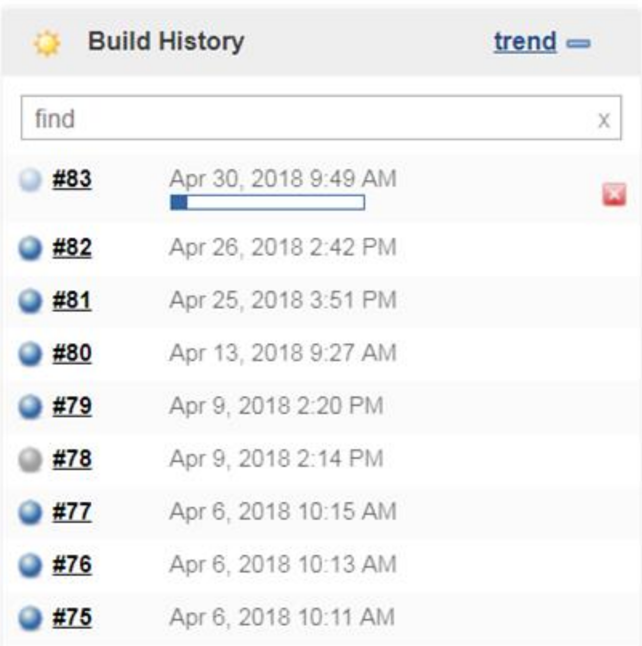


FIGURE 72. STM32_CI build #83 in progress

After a build is completed, it can be examined in Jenkins. The build results are accessible through the build page. The test run specific logs can be downloaded from the *Build Artifacts* link and the HTML test report can be examined from the *HTML Report* link. (Figure 73.)

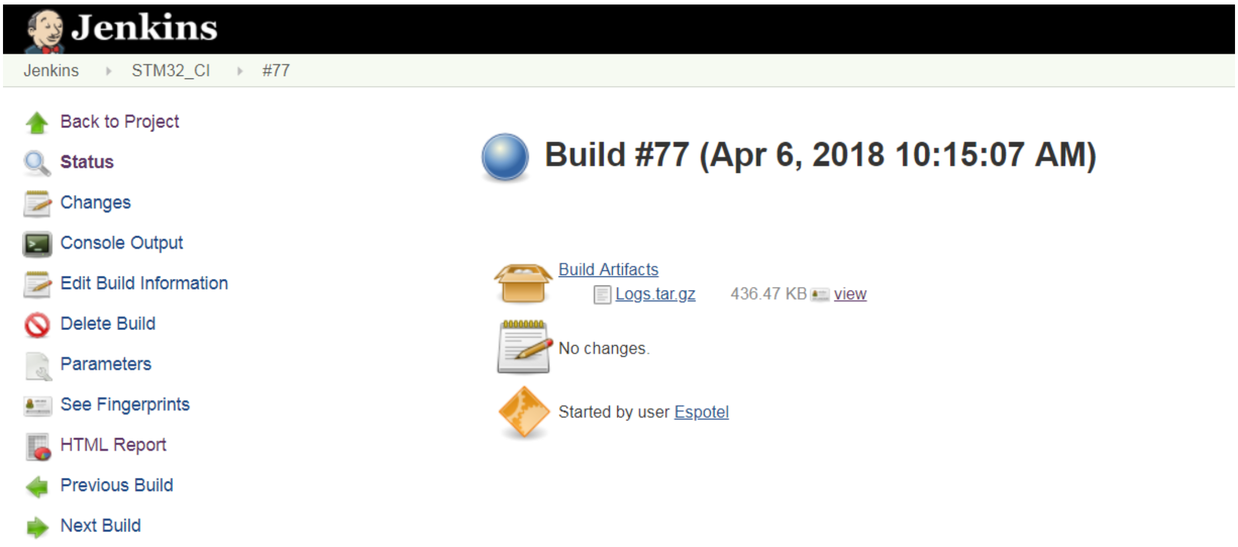


FIGURE 73. Build #77 from the STM32_CI job

8 RESULTS

This chapter goes through the results. In addition, any problems encountered are analyzed and further development ideas are introduced, such as bug fixes.

8.1 Results achieved

The objectives of this thesis were achieved successfully. The Python test script that was created, automatically imports the latest Mbed OS version, exports the Mbed OS project to the SW4STM32 IDE, build binaries using the SW4STM32 IDE, and verifies the integrity of the created binary files using the test between the host pc and Nucleo development boards. The Python script also generates the HTML test reports and logs from every test run.

A Jenkins test automation server was hosted in Etteplan's network with remote access feature. A Jenkins job was created to run the Python test script, store the HTML test report and test logs, and send email notifications to the defined employees (see figures 70, 49 and 69). In addition, the Jenkins server can be further utilized in Etteplan's other projects as well.

In case any of the previous test cases must be reproduced, it is possible with the SHA-1 hash value seen in the HMTL test reports. With this hash value, the exactly the same version of Mbed OS can be imported. (Figure 49.)

8.2 Problems encountered

This chapter analyzes some of the encountered problems. Also, if the problem was solved, solutions are given.

8.2.1 Unsynchronized serial bus

The verification process of the binary files was not acting as specified during the early development phases. When the incoming serial traffic from the Nucleo boards was observed, some peculiar messages were received. For example, if *PING* was written to the serial bus, the Nucleo development board might have answered with *ONGP* or *STPO*. This was due to the excess data in the serial bus from previous test runs. This problem was solved with the *Clear serial bus* phase that synchronized the serial bus between the host PC and the Nucleo development boards. Clearing the serial bus process is introduced in chapter 5.9.

8.2.2 Unique verification process

If the flashing process of a Nucleo development board fails, the previous program will stay in the memory of the device. This feature set requirements for the Python test script and the *main.cpp* program for the Nucleo development boards: The *main.cpp* program had to be unique for every test run and the Python test script had to be aware of that.

This problem was solved by implementing the *timestamp* test. Every time a test run was initiated, the current timestamp was stored in the Python test script. This timestamp was embedded in the *timestamp.h* header file that is included in the *main.cpp* program. This way the *timestamp.h* header file was unique during every test run and both the Python test script and the *main.cpp* program were aware of the timestamp. If the flashing process failed, the previous program sent wrong timestamp to the host PC during the verification process. The verification process of the binary files is introduced in chapter 5.9.

8.2.3 USB device routing in a virtual machine

The Ubuntu Linux distribution was initially installed on a virtual machine running on the Windows 7 OS. The software used to host the virtual machine was VirtualBox. Two major problems were encountered with this setup. The first was the number of USB devices the Windows 7 OS could recognize. All ten Nucleo development boards were connected to the host PC via a 10-port USB hub. However, the Windows 7 OS only recognized seven out of ten USB devices (the Nucleo development boards). This problem was not resolved since the final host PC would run Ubuntu natively. With the final host PC running Ubuntu, this problem was not encountered.

The second problem was with routing the USB devices, i.e., the Nucleo development boards, in the virtual machine. This means that a USB device connected to the computer running the Windows 7 OS has to be routed to the operating system in the virtual machine. This feature worked fine but occasionally the USB devices were suddenly not detected anymore even though the VirtualBox software implied that the USB devices were routed. (Figure 74.)

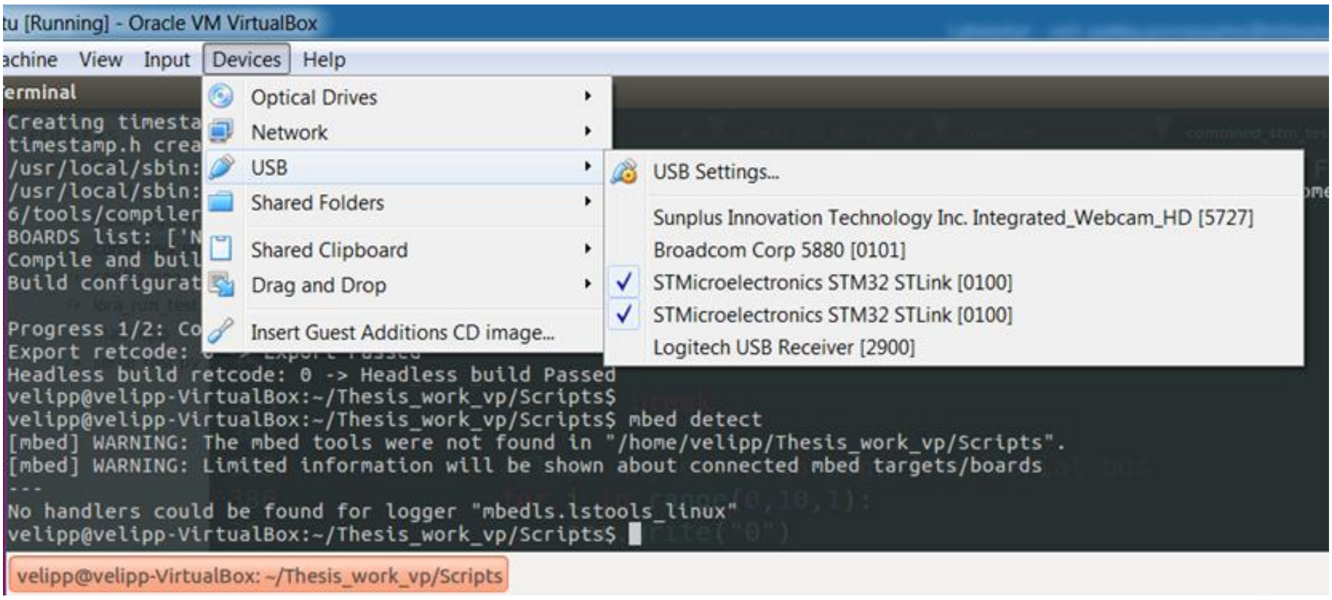


FIGURE 74. Ubuntu not detecting Nucleo devices

8.2.4 Serial communication problem

On some rare occasions the Nucleo development boards could not communicate via the serial bus for some reason. In these cases, the test script verified the serial connection to be open. This problem is relatively new, and no permanent solution is yet devised. Physically re-plugging the Nucleo development boards temporarily solved this problem. There has been some discussion on wether a software reset could be implemented for the Nucleo development boards to permanently solve this problem.

8.2.5 No space left on device

This problem occurs on some occasions with the flashing process when a Nucleo development board has been flashed multiple times without a power reset. When the Nucleo development board is being flashed, the flashing fails with a following error: *cp: error writing 'path_to_binary_file: No space left on device*.

This problem is probably due to the Nucleo development board not refreshing its filesystem after the previous successful flash. Since the Nucleo development boards appear as media devices, attempts have been made to solve the problem, without success, by remounting the media directory of the problematic Nucleo development board. However, further investigation could be in order since this solution was implemented in a hurry. A software reset for the Nucleo development boards could be worth a try as well. For a temporary fix, the Nucleo development boards were simply re-plugged.

8.3 Further development

This chapter introduces some further development ideas. Some of the remaining bugs, with possible solutions are introduced as well.

8.3.1 Option to choose targeted Nucleo development boards

As already introduced, the *stm32_ci_test_script.py* Python test script has a placeholder for a single target testing feature (*--singleTarget* argument). The purpose of this command line argument was to define only one of the Nucleo development boards to be used in a test run.

In addition to implementing this feature, it could also be modified so that a user could define multiple Nucleo development boards to be used in a test run. The *--singleTarget* could be renamed for example as *--customTargets*. This argument would then take the target names of the desired Nucleo development boards as parameter. This feature could be further used in Jenkins. In the *STM32_CI* job, a drop-down menu could be created to choose the desired Nucleo development boards to be used. The list of the supported Nucleo development boards could be fetched from the already existing *config.json* configuration file.

8.3.2 Code refactoring

Refactoring code means improving the design of the code without changing its behavior or logic. Put simply, refactoring means cleaning the code to make it easier to read and comprehend.

Especially the *detect_devices()* function is somewhat complicated because it has multiple variables that could be replaced with just one. For example, the output from the *mbed detect* is stored to the *output* variable. Next, this *output* variable is split from the *line break* character (*n*) and the output is stored to a new variable called *devices* (see figure 27). This split variable could have been stored to the original *output* variable instead of creating a new one. Also, *output* is not a very informative name for this variable. It could have been named *devices* to begin with. The *detect_devices()* function is introduced in chapter *5.5.*

8.3.3 Automatic detection of new Mbed OS releases

The Jenkins server could automatically initiate a new build from the *STM32_CI* job every time a new Mbed OS version is released. The detection could be done in a separate Python script that polls the current Git SHA-1 hash value of Mbed OS. When a new hash value is detected, the script would ask Jenkins to initiate new build. The already existing *config.json* configuration file could hold the information about the previous hash value. The script would check the current Mbed OS hash value and compare it to the hash value in the *config.json* file. If the hash values differ, a new Mbed OS version is released and new build shall be initiated.

A proof of concept was already made regarding this feature, but it was not implemented in the scope of this thesis. This proof of concept was a Python script that managed to resolve the SHA-1 hash value of current Mbed OS version without first importing it from GitHub.

8.3.4 Existing bugs

In the *detect_boards()* function, the number of devices (*nb_devices*) is calculated. However, this is done right after the *mbed detect* command is issued. This list of devices also contains devices not supported by the test. This means that the list of devices, from where the *nb_devices* is calculated, is not yet filtered with the *supported-boards* list from the *config.json* configuration file. If any extra development boards are connected to the host PC, they are counted in the *nb_devices* variable giving false information about the number of devices.

These extra development boards are not really used in the actual test but the *nb_devices* variable is used in various cases during the Python test script such as in *for* loops, which could potentially cause a test breaking bug. To fix this bug, the *nb_devices* should be calculated after the device list is filtered with the *supporter-boards* list from the *config.json* configuration file. (Figure 29.)

There is a missing logging phase in the *test_device(serial_port)* function after the *clear serial bus* phase. During this phase, the *serial_ready* variable is used to determine if the serial bus was successfully cleared. However, there are no logging methods whatsoever if clearing the serial bus fails. The test sequences are initiated only if the *serial_ready* variable is equal to *1*, i.e., serial bus was successfully cleared. If the *serial_ready* is not equal to *0*, i.e., clearing the serial bus failed, every test sequence is skipped, and no log entries are made. To solve this, an *else* condition shall be added with logging features after the test sequence.

In the end of the *main()* function, the log files are archived and compressed to the Jenkins' workspace with the *prepare_jenkins_workspace()* function call. This function is called too early since information is still being appended to the log file after it. This could be fixed by simply moving the *prepare_jenkins_workspace()* function to the end of the *main()* function.

9 CONCLUSION

The aim of this thesis was to create a Jenkins based CI test automation system for Etteplan to automatically verify the integrity between STMicroelectronics' SW4STM32 IDE and ARM's Mbed CLI exporter tool. The original problem with the SW4STM32 IDE maintenance project was the large effort that had to be put in to accomplish the verification process. This required one to setup the test environment, create the binary files, flash the Nucleo development boards, and write the test reports by hand.

A Python script was created to execute the tasks mentioned above. The Python script was able to download the latest version of Mbed OS and export the project to the SW4STM32 IDE. In the script, binary files were created, and they were flashed to the corresponding Nucleo development boards. Finally, the binary files were verified via the verification sequence between the host PC and the Nucleo development boards. As a result, an HTML test report and test logs were generated from every test run.

A host PC was configured to run a Jenkins test server with the remote access feature. Jenkins was used to implement simple user interface to initiate the Python test script and to browse the test documents. The configured Jenkins server can be further utilized in Etteplan's other projects as well.

REFERENCES

1. Nikhil, Pathania. 2016. Learning Continuous Integration with Jenkins. Birmingham: Packt Publishing. Available as e-book: http://proquest.safaribooksonline.com.ezp.oamk.fi:2048/book/software-engineering-and-development/9781785284830/learning-continuous-integration-with-jenkins/pr01_html#X2ludGVybmFsX0h0bWxWaWV3P3htbGlkPTk3ODE3ODUyODQ4MzAlMkZjaDAxX2h0bWwmcXVlcnk9

2. Etteplan. About Etteplan. Date of retrieval 1.5.2018. Available: https://www.etteplan.com/about-us

3. Python. General Python FAQ: General Information. Date of retrieval: 1.5.2018. Available: https://docs.python.org/3/faq/general.html#what-is-python

4. cplusplus. C++ Language FAQ. Date of retrieval: 1.5.2018. Available: http://www.cplusplus.com/info/faq/

5. Arm Mbed. Mbed OS. Date of retrieval: 1.5.2018. Available: https://www.mbed.com/en/platform/mbed-os/

6. Arm Mbed. Docs: Tools: Arm Mbed CLI. Date of retrieval: 1.5.2018. Available: https://os.mbed.com/docs/v5.8/tools/arm-mbed-cli.html

7. STMicroelectronics. SW4STM32. Date of retrieval: 1.5.2018. Available: http://www.st.com/en/development-tools/sw4stm32.html

8. STMicroelectronics. STM32 MCU Nucleo. Date of retrieval: 1.5.2018. Available: http://www.st.com/en/evaluation-tools/stm32-mcu-nucleo.html?querycriteria=productId=LN1847

9. Jenkins. Documentation. Date of retrieval: 1.5.2018. Available: https://jenkins.io/doc/

10. VirtualBox. Date of retrieval: 1.5.2018. Available: https://www.virtualbox.org/

11. Python. subprocess. Date of retrieval: 1.5.2018. Available: https://docs.python.org/2/library/subprocess.html

12. Python. os. Date of retrieval: 1.5.2018. Available: https://docs.python.org/2/library/os.html

13. Python. sys. Date of retrieval: 1.5.2018. Available: https://docs.python.org/2/library/sys.html

14. Python. shutil. Date of retrieval: 1.5.2018. Available: https://docs.python.org/2/library/shutil.html

15. Python. argparse. Date of retrieval: 1.5.2018. Available: https://docs.python.org/3/library/argparse.html

16. Python. datetime. Date of retrieval: 1.5.2018. Available: https://docs.python.org/2/library/datetime.html

17. Python. json. Date of retrieval: 1.5.2018. Available: https://docs.python.org/2/library/json.html

18. Python. json. Date of retrieval: 1.5.2018. Available: https://docs.python.org/2/library/json.html

19. Python. time. Date of retrieval: 1.5.2018. Available: https://docs.python.org/2/library/time.html

20. Python. time. Date of retrieval: 1.5.2018. Available: https://docs.python.org/2/library/time.html

21. pySerial. Date of retrieval: 1.5.2018. Available: https://python-hosted.org/pyserial/index.html

22. Jenkins. Installin Jenkins: Linux. Date of retrieval: 1.5.2018. Available: https://jenkins.io/doc/book/installing/#linux

APPENDICES

Appendix 1 The main.cpp program for Nucleo development boards

Appendix 2 The *config.json* configuration file

Appendix 3 The *html_generator.py* Python script

Appendix 4 The *stm32_ci_test_script.py* Python script

Appendix 5 The HTML test report

Appendix 6 The general log file

```cpp
#include "mbed.h"
#include "timestamp.h"

//Create serial object
Serial serial(USBTX, USBRX);

int main()
{
    char buffer[5] = {0};
    char* timestamp = TIMESTAMP;

    while(1) {
    //Read 5 characters from serial to buffer
        serial.gets(buffer, 5);

        //Received "TEST"
        if(strcmp(buffer, "TEST") == 0) {
            wait(0.5);
            //Write timestamp to serial
            serial.printf("%s", timestamp);
        }

        //Received "PING"
        else if(strcmp(buffer, "PING") == 0) {
            wait(0.5);
            //Write "PONG" to serial
            serial.printf("PONG");
        }

        //Received anything else
        else {
            wait(0.5);
            //Write received buffer to serial
            serial.printf("%s", buffer);
        }
    }

    return 0;
}
```

```json
{
  "STM32": {
    "path": {
      "script-root-path": "/home/username/STM32_CI/Scripts",
      "test-root-path": "/home/username/STM32_CI/Test_run/STM32",
      "systemworkbench-path": "/home/username/Ac6/SystemWorkbench",
      "toolchainpath":"/home/username/Ac6/SystemWorkbench/plugins/
                    fr.ac6.mcu.externaltools.armnone.linux64_1.15.0.2017083
                    11556/tools/compiler/bin",
      "workspace-path": "/home/username/STM32_CI/temp_workspace",
      "jenkins-job-workspace-path": "/var/lib/jenkins/workspace/STM32_CI"
    },
    "supported-boards": [
      "NUCLEO_F091RC",
      "NUCLEO_F103RB",
      "NUCLEO_F207ZG",
      "NUCLEO_F303ZE",
      "NUCLEO_F401RE",
      "NUCLEO_F429ZI",
      "NUCLEO_F767ZI",
      "NUCLEO_L073RZ",
      "NUCLEO_L152RE",
      "NUCLEO_L476RG"
    ]
  }
}
```

```
#!/usr/bin/python

import subprocess
import os


'''
	INITIATE HTML REPORT
'''
def start_html(html_file_path):
	#HTML syntax
	html_syntax = """
	<!DOCTYPE html>
	<html>
	<head>
	<title>
	</title>
		<meta name=\"viewport\" content=\"width=device-width, initial-scale=1\">
	<style>
		body {
			background-color: #ffffff;
			background-repeat: no-repeat;
			background-position: top left;
			background-attachment: fixed;
		}
		h1 {
			font-family: Arial, sans-serif;
			color: #000000;
			background-color: #ffffff;
		}
		p {
			font-family: Georgia, serif;
			font-size: 18px;
			font-style: normal;
			font-weight: normal;
			color: #000000;
			background-color: #ffffff;
		}
		table,th,td {
		    border: 1px solid black;
		}
	</style>
	</head>
	<body>"""

	#Write HTML syntax to HTML file
	with open(html_file_path, 'w') as report:
		report.write(html_syntax)


'''
	FILL IN GENERAL TEST INFO
'''
def fill_general_info(html_file_path, start_time, nb_boards, mbed_os_version, git_branch, git_hash):
	html_syntax = """
	<h1>STM32 test report</h1>
	<h2>Test info:</h2>
	<p>Start time: <i>{}</i></p>
	<p>No. included boards: <i>{}</i></p>
	<p>Mbed OS version: <i>{}</i></p>
	<p>Git branch: <i>{}</i></p>
	<p>Git hash: <i>{}</i></p>
	<h2>Test results:</h2>
	<table>
		<tr>
			<th>Target</th>
			<th>Overall result</th>
			<th>Create .bin file</th>
			<th>Flash device</th>
			<th>Timestamp test</th>
			<th>Ping-Pong test</th>
		</tr>
	""".format(start_time, nb_boards, mbed_os_version, git_branch, git_hash)

	with open(html_file_path, 'a') as report:
		report.write(html_syntax)
```

```
'''
      DEFINE TEXT COLOR
'''
def define_result_color(result):
      if result == "PASS":
            return "green"
      elif result == "FAIL":
            return "red"
      elif result == "N/A":
            return "black"


'''
      FILL BOARD INFO AND TEST RESULTS
'''
def fill_board_info(html_file_path, board, overall_result, binary_result, flash_result, timestamp_result, pinpong_result):
      overall_result_color = define_result_color(overall_result)
      binary_result_color = define_result_color(binary_result)
      flash_result_color = define_result_color(flash_result)
      timestamp_result_color = define_result_color(timestamp_result)
      pinpong_result_color = define_result_color(pinpong_result)

      html_syntax = """
            <tr>
            <td>{}</td>
                  <td><span style="color:{}; font-weight:bold">{}</span></td>
                  <td><span style="color:{}">{}</span></td>
                  <td><span style="color:{}">{}</span></td>
                  <td><span style="color:{}">{}</span></td>
                  <td><span style="color:{}">{}</span></td>
            </tr>
      """.format(board, overall_result_color, overall_result,\
       binary_result_color, binary_result, flash_result_color, flash_result,\
       timestamp_result_color, timestamp_result, pinpong_result_color,  pinpong_result)

      with open(html_file_path, 'a') as report:
            report.write(html_syntax)


'''
      END HTML REPORT
'''
def end_html(html_file_path):
      html_syntax = "</table></body></html>"

      with open(html_file_path, 'a') as report:
            report.write(html_syntax)
```

```python
#!/usr/bin/python

import subprocess
import os
import sys
import shutil
import argparse
import datetime
import json
import time
import serial
import html_generator as HTML

#Get start time
start_time = datetime.datetime.now()
start_time = start_time.strftime("%Y-%m-%d_T%H:%M:%S")

#Commandline arguments
parser = argparse.ArgumentParser()
group = parser.add_argument_group()
group.add_argument("--branch", help="Choose Mbed OS GitHub branch.", default="latest")
group.add_argument("--buildConfig", help="Choose build configuration", choices=['Release', 'Debug'], default="Release")
#TODO: Implement --singleTarget feature
group.add_argument("--singleTarget", help="Select target board. See config.json for supported targets.", default=False)

#Used for debugging
group.add_argument("--skipBuild", help="Skips compile and build phases", action='store_true', default=False)
group.add_argument("--skipClean", help="Don't remove Src directory and skip GitHub downloads", action='store_true',
default=False)
group.add_argument("--skipFlash", help="Skip flashing phase", action='store_true', default=False)
group.add_argument("--skipTest", help="Skip testing phase", action='store_true', default=False)
args = parser.parse_args()


#Define build configuration
build_config = args.buildConfig

#Target list to store data about connected devices
target_list = []


'''
    PARSE config.json FILE
'''
#Open config.json file
config = json.load(open('config.json'))
#Parse and set path variables from config.json
config = config["STM32"]
systemworkbench_path = config["path"]["systemworkbench-path"]
toolchain_path = config["path"]["toolchain-path"]
workspace_path = config["path"]["workspace-path"]
script_root_path = config["path"]["script-root-path"]
test_root_path = config["path"]["test-root-path"]
jenkins_job_workspace_path = config["path"]["jenkins-job-workspace-path"]


'''
    DEFINE TEST DIRECTORIES
'''
#Define main directories
src_root_path = test_root_path + "/Src"
binaries_root_path = test_root_path + "/Binaries"
documents_root_path = test_root_path + "/Documents"

#Define log files and directories
log_path = documents_root_path + "/Logs"
report_path = documents_root_path + "/Reports"

#Test log file. Store test run data
test_log_file = "{}/test_log.txt".format(log_path)

#Test report file. General PASS / FAIL report from test
report_file = "{}/general_report.html".format(report_path)

#Create test root directory
if os.path.exists(test_root_path) is not True:
    os.makedirs(test_root_path)
```

```python
#Create documents root directory
if os.path.exists(documents_root_path) is not True:
    os.mkdir(documents_root_path)

#Create test log directory
if os.path.exists(log_path) is not True:
    os.mkdir(log_path)

#Open test log file for logging (overwrite previous log file)
TEST_LOG = open(test_log_file, 'w')


'''
    DETECT CONNECTED DEVICES
    - Detect connected and supported boards
    - Return number of connected devices
'''
def detect_boards():
    nb_devices = 0
    #Parse supported boards list from the config.json
    supported_boards = config["supported-boards"]

    #Detect connected mbed devices
    print("Detecting mbed devices...")
    TEST_LOG.write("Detect mbed devices...\n")
    proc = subprocess.Popen('mbed detect', stdout=subprocess.PIPE, shell=True)
    #Read the stdout list from 'mbed detect', split it from new lines (\n) and store split list to "devices" variable
    output = proc.stdout.read()
    devices = output.split('\n')

    TEST_LOG.write(output + "\n")

    #Number of connected devices
    nb_devices = int(len(devices)) - 1
    print("Number of devices: " + str(nb_devices))
    TEST_LOG.write("Number of devices: " + str(nb_devices) + "\n")

    #Check if at least one device is connected. Otherwise abort test.
    if nb_devices == 0:
        TEST_LOG.write("No devices found. Exiting program...\n")
        sys.exit("No devices found. Exiting program...")

        #Select all connected boards
    if args.singleTarget == False:
    #All connected boards
        for i in range(0, int(nb_devices), 1):
            #Split from "
            target_data = devices[i].split('"')
            #Check if connected board is supported by the conf.json
            for board in supported_boards:
                if target_data[1] == board:
                    #Structure of target_list[1, 3, 5] => [TARGET_NAME, MEDIA_DIR, COM_PORT]
                    target_list.append([target_data[1], target_data[3], target_data[5]])
                    break

        print("Detected devices:")
        TEST_LOG.write("Detected devices:\n")
        for i in range(0, nb_devices, 1):
            print(str(target_list[i]))
            TEST_LOG.write(str(target_list[i]) + "\n")

        #Check if every supported board are detected
        if nb_devices != len(supported_boards):
            print("WARNING: Every supported board not detected!")
            TEST_LOG.write("\nWARNING: Missing supported board! Some device(s) not detected...\n")

    #Running test on a single board
    else:
        #User defined board
        #TODO: Add support for single target testing
        print("Single target not implemented")

    TEST_LOG.flush()
    return nb_devices
```

```
'''
    CLEAN AND INITIALIZE TEST
    - Clear existing Mbed OS directories, create new ones and download Mbed OS
    - Copy main.cpp to Mbed OS directory
    - Generate timestamp.h file
    - Return Mbed OS version and GitHub hash
'''
def initialize_test():
    #Create enviroment variable
    path_list = []
    print(os.environ.get('PATH'))
    TEST_LOG.write("\n" + str(os.environ.get('PATH')) + "\n")
    path_list.append(systemworkbench_path)
    path_list.append(toolchain_path)
    os.environ["PATH"] += os.pathsep + os.pathsep.join(path_list)
    print(os.environ.get('PATH'))
    TEST_LOG.write(str(os.environ.get('PATH')) + "\n")
    TEST_LOG.flush()

    #Remove previous files and folders if exists
    if os.path.exists(src_root_path):
        print("\nRemove " + src_root_path + "...")
        TEST_LOG.write("\nRemove " + src_root_path + "...\n")
        shutil.rmtree(src_root_path)

    if os.path.exists(binaries_root_path):
        print("Remove " + binaries_root_path + "...")
        TEST_LOG.write("Remove " + binaries_root_path + "...\n")
        shutil.rmtree(binaries_root_path)

    #Create required directories. Required if --skipClean flag is used
    #Create source code root path
    if os.path.exists(src_root_path) is not True:
        os.mkdir(src_root_path)

    #Create binaries root path
    if os.path.exists(binaries_root_path) is not True:
        os.mkdir(binaries_root_path)

    #Create documents root directory
    if os.path.exists(documents_root_path) is not True:
        os.mkdir(documents_root_path)

    #Create test log directory
    if os.path.exists(log_path) is not True:
        os.mkdir(log_path)

    #Create test report directory
    if os.path.exists(report_path) is not True:
        os.mkdir(report_path)

    #Get the latest Mbed OS version (master)
    os.chdir(src_root_path)
    print("\nFetching Mbed OS...")
    TEST_LOG.write("\nFetching Mbed OS...\n")
    TEST_LOG.flush()
    subprocess.call('mbed new .', stdout=TEST_LOG, shell=True)

    #Update Mbed OS version from GitHub
    if args.branch != "latest":
        os.chdir((src_root_path + "/mbed-os"))
        print("Updating Mbed OS to branch " + str(args.branch) + "...")
        TEST_LOG.write("Update Mbed OS to branch " + str(args.branch) + "...\n")
        TEST_LOG.flush()
        subprocess.call('mbed update {}'.format(args.branch), stdout=TEST_LOG, shell=True)

    os.chdir("{}/mbed-os".format(src_root_path))

    #Parse current mbed-os version and git hash
    proc = subprocess.Popen('mbed ls', stdout=subprocess.PIPE, shell=True)
    output = proc.stdout.read()
    temp = output.split(" ")
    temp = temp[4].split(")")
    mbed_version = temp[0]

    TEST_LOG.write("Mbed OS Version: {}\n".format(mbed_version))
    TEST_LOG.flush()
```

```python
    #Parse Mbed OS GitHub SHA hash
    proc = subprocess.Popen('git rev-parse HEAD', stdout=subprocess.PIPE, shell=True)
    mbed_sha = proc.stdout.read()
    TEST_LOG.write("Mbed OS GitHub SHA: {}\n".format(mbed_sha))

    #Copy test application (main.cpp) and to the Mbed OS directory root
    print("Copying main.cpp to Mbed OS...")
    TEST_LOG.write("\nCopy main.cpp to Mbed OS...\n")
    TEST_LOG.flush()
    subprocess.call('cp -v {}/{} {}'.format(script_root_path, "main.cpp", src_root_path), stdout=TEST_LOG, shell=True)

    print("Create timestamp.h file")
    TEST_LOG.write("\nCreate timestamp.h file\n")
    os.chdir(src_root_path)
    #Create timestamp.h header file
    with open("timestamp.h", "w") as header_file:
        header_file.write("#define TIMESTAMP \"{}\"".format(start_time))

    #Check if timestamp.h header file was created
    if os.path.exists(src_root_path + "/timestamp.h"):
        print("timestamp.h created")
        TEST_LOG.write("timestamp.h created\n")
    else:
        TEST_LOG.write("Could not create timestamp.h -> Aborting test...\n")
        sys.exit("Could not create timestamp.h -> Aborting test...")

    return mbed_version, mbed_sha


'''
    BUILD BINARY FILE
    - Export Mbed OS to SW4STM32 IDE
    - Compile and build .bin file using headless build mode
    - Return build result (PASS/FAIL)
'''
def build_binary(board, build_configuration, timestamp):
    build_result = "N/A"

    print("\nCompile and build")
    TEST_LOG.write("\nCompile and build\n")

    print("Working with {}".format(board))
    TEST_LOG.write("Working with {}\n".format(board))
    TEST_LOG.flush()

    #Change to the mbed-os root
    os.chdir("{}/mbed-os".format(src_root_path))

    #Remove Debug and Release directories from the project (mbed-os root)
    if os.path.exists("Debug"):
        shutil.rmtree("Debug")
    if os.path.exists("Release"):
        shutil.rmtree("Release")

    #Change back to source root
    os.chdir(src_root_path)

    #Store data from compile & build phases.
    build_log_file = "{}/{}_build_log.txt".format(log_path, board)
    BUILD_LOG = open(build_log_file, 'w')
    BUILD_LOG.write("Timestamp: {}\n".format(timestamp))
    BUILD_LOG.flush()

    #Export source tree to STM32 System Workbench IDE (SW4STM32)
    retcode = subprocess.call('mbed export -i sw4stm32 -m {}'.format(board), stdout=BUILD_LOG, shell=True)
    if retcode == 0:
        print("Export PASS")
        TEST_LOG.write("Export PASS!\n")
    else:
        print("Export FAIL")
        TEST_LOG.write("Export FAIL!\n")
    TEST_LOG.flush()
```

```
    #Compile and build via headless build
    #Refer to https://gnu-mcu-eclipse.github.io/advanced/headless-builds/, searched 11.4.2018
    retcode = subprocess.call('{}/eclipse \
        --launcher.suppressErrors \
        -nosplash \
        -application org.eclipse.cdt.managedbuilder.core.headlessbuild \
        -data {} \
        -import {} \
        -cleanBuild "Src"/{}'\
        .format(systemworkbench_path, workspace_path, src_root_path, build_configuration), stdout=BUILD_LOG, shell=True)

    if retcode == 0:
        print("Headless build PASS")
        TEST_LOG.write("Headless build PASS!\n")
    else:
        print("Headless build FAIL")
        TEST_LOG.write("Headless build FAIL!\n")

    #Create board specific directory to store binary file
    if not os.path.exists(binaries_root_path + "/" + board + "/" + build_config):
        os.makedirs(binaries_root_path + "/" + board + "/" + build_config)

    BUILD_LOG.close()

    TEST_LOG.write("Copy .bin file to test directory\n")
    TEST_LOG.flush()
    #Copy .bin file to board specific directory
    subprocess.call('cp -v ./{}/Src.bin {}/{}/{}'.format(build_config, binaries_root_path, board, build_config), \
        stdout=TEST_LOG, shell=True)

    #Check if .bin files are generated
    if os.path.exists("{}/{}/{}/Src.bin".format(binaries_root_path, board, build_config)):
        build_result = "PASS"
        print("Success! .bin file generated for board " + board + "\n")
        TEST_LOG.write("Success! .bin file generated for board " + board + "\n")
    else:
        build_result = "FAIL"
        print("Fail! .bin file not found for board " + board + "\n")
        TEST_LOG.write("Fail! .bin file not found for board " + board + "\n")

    TEST_LOG.flush()

    return build_result


'''
    FLASH DEVICE
    - Copy .bin file to device
    - Return flash result (PASS/FAIL)
'''
def flash_device(board, board_dir):
    flash_result = "N/A"

    print("Flashing board " + board + "...")
    TEST_LOG.write("\nFlashing board " + board + "...\n")
    TEST_LOG.flush()

    if args.singleTarget == False:
        #Flash the device by copying binary file to target boards media directory
        for i in range(0, 5, 1):
            TEST_LOG.write("Copy .bin file to {}\n".format(board_dir))
            TEST_LOG.flush()
            retcode = subprocess.call('cp -v {}/{}/{}/Src.bin {}'.format(binaries_root_path, board, build_config, board_dir), \
                stdout=TEST_LOG, shell=True)

            if retcode == 0:
                flash_result = "PASS"
                print("Flash PASS")
                TEST_LOG.write("Flash PASS!\n")
                break
            else:
                flash_result = "FAIL"
                print("Flash FAIL")
                TEST_LOG.write("Flash FAIL! Retries {}/5\n".format(i))

    else:
        #TODO: Add single target flash
        print("Single target flash not implemented")
    return flash_result
```

```
'''
    TEST DEVICE
    - Run Timestamp test
    - Run Ping-Pong test
    - Return timestamp and ping-pong test results (PASS/FAIL)
'''
def test_device(serial_port):
    serial_ready = 0
    timestamp_result = "N/A"
    ping_result = "N/A"

    #TODO: Add support for testing only one device

    #Test boards over serial connection
    TEST_LOG.write("\nBegin test sequence:\n")
    try:
        #Create serial connection (COM PORT, BAUD, TIMEOUT)
        with serial.Serial(serial_port, 9600, timeout=2) as ser:
            ser.flush()
            print("\nBegin test at {}".format(serial_port))
            TEST_LOG.write("Begin test at {}\n".format(serial_port))

            #Check if serial connecion has been established. Try opening COM port again if closed
            for i in range(0, 3, 1):
                if ser.is_open != True:
                    print("COM closed")
                    TEST_LOG.write("COM closed\n")
                    ser.open()
                    time.sleep(1)
                else:
                    print("COM open")
                    TEST_LOG.write("COM open\n")
                    break

            #Clear any extra garbage from serial bus. Send character "0" until "0000" buffer is received
            print("Clear serial bus:")
            TEST_LOG.write("Clear serial bus:\n")
            for i in range(0,10,1):
                try:
                    #Write "0" to serial bus
                    ser.write("0")
                    #Read 4 characters from serial bus
                    foo = ser.read(4)
                except:
                    print("Device returned no data (device disconnected or multiple acces on port?)")
                    TEST_LOG.write("Device returned no data (device disconnected or multiple acces on port?)\n")
                    break
                #Received four characters containing "0000"
                if len(foo) == 4 and foo == "0000":
                    print("Serial bus cleared!")
                    TEST_LOG.write("Serial bus cleared!\n")
                    serial_ready = 1
                    break

            #TODO: Add error logging if serial bus couldn't be cleared -> serial_ready != 1

            if serial_ready == 1:
                #Ask for timestamp from DUT
                TEST_LOG.write("Send \"TEST\"\n")
                try:
                    ser.write("TEST")
                    timestamp = ser.read(20)
                    print(timestamp)
                    TEST_LOG.write("Received: {}\n".format(timestamp))

                    if timestamp == start_time:
                        #PASS
                        timestamp_result = "PASS"
                    else:
                        #FAIL
                        timestamp_result = "FAIL"
                except:
                    TEST_LOG.write("Error! Could not write or read serial!")
```

```
                #Initiate PING-PONG test to check if connection is still on
                try:
                    #Clear any extra garbage from serial bus
                    print("Clear serial bus:")
                    TEST_LOG.write("Clear serial bus:\n")
                    for i in range(0,10,1):
                        try:
                            ser.write("0")
                            foo = ser.read(4)
                        except:
                            print("Device returned no data (device disconnected or multiple acces on port?)")
                            TEST_LOG.write("Device returned no data (device disconnected or multiple acces on port?)\n")
                            break

                        if len(foo) == 4 and foo == "0000":
                            print("Serial bus cleared!")
                            TEST_LOG.write("Serial bus cleared!\n")
                            serial_ready = 1
                            break

                    #Run PING-PONG test three times
                    for i in range(0, 3, 1):
                        TEST_LOG.write("Send \"PING\"\n")
                        ser.write("PING")
                        answ = ser.read(4)
                        print(answ)
                        TEST_LOG.write("Received: {}\n".format(answ))
                        if answ == "PONG":
                            #PASS
                            ping_result = "PASS"
                        else:
                            #FAIL
                            ping_result = "FAIL"
                            break
                except:
                    TEST_LOG.write("Error! Could not write or read serial!\n")
    except:
        #Serial connection could not be established
        TEST_LOG.write("Error! Could not establish serial connection!\n")

    print("Test done!\nResults:\nTimestamp: {}\nPING-PONG: {}\n".format(timestamp_result, ping_result))
    TEST_LOG.write("Test done!\nResults:\nTimestamp: {}\nPING-PONG: {}\n\n".format(timestamp_result, ping_result))
    TEST_LOG.flush()
    return timestamp_result, ping_result



'''
    PREPARE JENKINS WORKSPACE
    - Compress and copy log files to Jenkins workspace
    - Copy HTML test report to Jenkins workspace
'''
def prepare_jenkins_workspace():
    #Compress Logs directory and copy it to the Jenkins STM job workspace for archieving
    retcode = subprocess.call(('sudo tar -czvf {}/Logs.tar.gz {}').format(jenkins_job_workspace_path, log_path), stdout=TEST_LOG,
shell=True)
    if retcode == 0:
        print("\"Logs\" directory compressed succesfully!")
        TEST_LOG.write("\"Logs\" directory compressed succesfully!\n")
    else:
        print("\"Logs\" directory compress failed!")
        TEST_LOG.write("\"Logs\" directory compress failed!\n")

    #Copy the HTML report to Jenkins STM32 job workspace
    retcode = subprocess.call(('cp {} {}').format(report_file, jenkins_job_workspace_path), stdout=TEST_LOG, shell=True)
    if retcode == 0:
        print("HTML report copied succesfully")
        TEST_LOG.write("HTML report copied succesfully\n")
    else:
        print("HTML report copy failed!")
        TEST_LOG.write("HTML report copy failed!\n")



'''
    MAIN LOOP
'''
def main():
    print("Test started at {}".format(start_time))
```

```
    TEST_LOG.write("Test started at {}\n".format(start_time))

    #Dictionary to store test results during every run
    test_results = {"Overall": "", "Binary": "N/A", "Flash": "N/A", "Timestamp": "N/A", "PingPong": "N/A"}
    mbed_version = "N/A"
    mbed_sha = "N/A"

    #Detect boards
    nb_devices = detect_boards()

    #Initialize test
    if not args.skipClean:
        temp = initialize_test()
        mbed_version = temp[0]
        mbed_sha = temp[1]

    #Initialize HTML report
    HTML.start_html(report_file)

    #Fill in general test information
    HTML.fill_general_info(report_file, start_time, nb_devices, mbed_version, args.branch, mbed_sha)

    #Begin the test sequence
    for i in range(0, int(nb_devices), 1):
        target = target_list[i][0]
        media_dir = target_list[i][1]
        com_port = target_list[i][2]

        #Build .bin file
        if not args.skipBuild:
            test_results["Binary"] = build_binary(target, build_config, start_time)

        #Flash device
        if not args.skipFlash:
            test_results["Flash"] = flash_device(target, media_dir)

        #Test device
        if not args.skipTest:
            temp = test_device(com_port)
            test_results["Timestamp"] = temp[0]
            test_results["PingPong"] = temp[1]

        #Check overall result. If any test case fails, the overall result is "FAIL". Otherwise "PASS".
        #If N/A was found Overall is set to "N/A"
        if "FAIL" in test_results.values():
            test_results["Overall"] = "FAIL"
        elif "N/A" in test_results.values():
            test_results["Overall"] = "FAIL"
        else:
            test_results["Overall"] = "PASS"

        #Fill test results into HTML report
        HTML.fill_board_info(report_file, target, test_results["Overall"], test_results["Binary"],\
            test_results["Flash"], test_results["Timestamp"], test_results["PingPong"])

        #Reset test results for next run
        test_results = {"Overall": "", "Binary": "N/A", "Flash": "N/A", "Timestamp": "N/A", "PingPong": "N/A"}

        TEST_LOG.write("Progress: {}/{} completed\n\n\n".format((i+1),nb_devices))
        TEST_LOG.flush()

    #End HTML report
    HTML.end_html(report_file)

    #TODO: Shouldn't these be in the prepare_jenkins_workspace() function?
    print("Compress \"Logs\" directory")
    TEST_LOG.write("Compress \"Logs\" directory\n")
    TEST_LOG.flush()

    #Copy HTML report and Log files to Jenkins workspace
    prepare_jenkins_workspace()

    print("Process completed!")
    TEST_LOG.write("Process completed!\n")

    #Get end time
    end_time = datetime.datetime.now()
    end_time = end_time.strftime("%Y-%m-%d_T%H:%M:%S")
```

```
    print("Test ended at {}".format(end_time))
    TEST_LOG.write("Test ended at {}".format(end_time))
    TEST_LOG.close()

if __name__ == "__main__":
    main()
```

Back to STM32_CI general_report

STM32 test report

Test info:

Start time: *2018-04-04_T16:01:44*

No. included boards: *10*

Mbed OS version: *mbed-os-5.8.1*

Git branch: *latest*

Git hash: *addec7ba10054be03849eff58a1d17f157391e7d*

Test results:

Target	Overall result	Create .bin file	Flash device	Timestamp test	Ping-Pong test
NUCLEO_F207ZG	**PASS**	PASS	PASS	PASS	PASS
NUCLEO_F767ZI	**PASS**	PASS	PASS	PASS	PASS
NUCLEO_F429ZI	**PASS**	PASS	PASS	PASS	PASS
NUCLEO_L476RG	**PASS**	PASS	PASS	PASS	PASS
NUCLEO_L073RZ	**PASS**	PASS	PASS	PASS	PASS
NUCLEO_F303ZE	**PASS**	PASS	PASS	PASS	PASS
NUCLEO_F091RC	**PASS**	PASS	PASS	PASS	PASS
NUCLEO_L152RE	**PASS**	PASS	PASS	PASS	PASS
NUCLEO_F401RE	**PASS**	PASS	PASS	PASS	PASS
NUCLEO_F103RB	**PASS**	PASS	PASS	PASS	PASS

```
Test started at 2018-04-13_T09:27:35
Detect mbed devices...
[mbed] Detected "NUCLEO_F207ZG" connected to "/media/espotel/NODE_F207ZG" and using com port "/dev/ttyACM5"
[mbed] Detected "NUCLEO_F767ZI" connected to "/media/espotel/NODE_F767ZI" and using com port "/dev/ttyACM9"
[mbed] Detected "NUCLEO_F429ZI" connected to "/media/espotel/NODE_F429ZI" and using com port "/dev/ttyACM3"
[mbed] Detected "NUCLEO_L476RG" connected to "/media/espotel/NODE_L476RG" and using com port "/dev/ttyACM6"
[mbed] Detected "NUCLEO_L073RZ" connected to "/media/espotel/NODE_L073RZ" and using com port "/dev/ttyACM0"
[mbed] Detected "NUCLEO_F303ZE" connected to "/media/espotel/NODE_F303ZE" and using com port "/dev/ttyACM7"
[mbed] Detected "NUCLEO_F091RC" connected to "/media/espotel/NODE_F091RC" and using com port "/dev/ttyACM1"
[mbed] Detected "NUCLEO_L152RE" connected to "/media/espotel/NODE_L152RE" and using com port "/dev/ttyACM2"
[mbed] Detected "NUCLEO_F401RE" connected to "/media/espotel/NODE_F401RE" and using com port "/dev/ttyACM8"
[mbed] Detected "NUCLEO_F103RB" connected to "/media/espotel/NODE_F103RB" and using com port "/dev/ttyACM4"

Number of devices: 10
Detected devices:
['NUCLEO_F207ZG', '/media/espotel/NODE_F207ZG', '/dev/ttyACM5']
['NUCLEO_F767ZI', '/media/espotel/NODE_F767ZI', '/dev/ttyACM9']
['NUCLEO_F429ZI', '/media/espotel/NODE_F429ZI', '/dev/ttyACM3']
['NUCLEO_L476RG', '/media/espotel/NODE_L476RG', '/dev/ttyACM6']
['NUCLEO_L073RZ', '/media/espotel/NODE_L073RZ', '/dev/ttyACM0']
['NUCLEO_F303ZE', '/media/espotel/NODE_F303ZE', '/dev/ttyACM7']
['NUCLEO_F091RC', '/media/espotel/NODE_F091RC', '/dev/ttyACM1']
['NUCLEO_L152RE', '/media/espotel/NODE_L152RE', '/dev/ttyACM2']
['NUCLEO_F401RE', '/media/espotel/NODE_F401RE', '/dev/ttyACM8']
['NUCLEO_F103RB', '/media/espotel/NODE_F103RB', '/dev/ttyACM4']

/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/snap/bin
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/snap/bin:/home/espotel/Ac6/SystemWorkbench:/home/es
potel/Ac6/SystemWorkbench/plugins/fr.ac6.mcu.externaltools.arm-
none.linux64_1.15.0.201708311556/tools/compiler/bin

Remove /home/espotel/STM32_CI/Test_run/STM32/Src...
Remove /home/espotel/STM32_CI/Test_run/STM32/Binaries...

Fetching Mbed OS...
[mbed] Creating new program "Src" (git)
[mbed] Adding library "mbed-os" from "https://github.com/ARMmbed/mbed-os" at branch/tag "latest"
[mbed] Updating reference "mbed-os" -> "https://github.com/ARMmbed/mbed-
os/#f9ee4e849f8cbd64f1ec5fdd4ad256585a208360"
Mbed OS Version: mbed-os-5.8.2
Mbed OS GitHub SHA: f9ee4e849f8cbd64f1ec5fdd4ad256585a208360


Copy main.cpp to Mbed OS...
'/home/espotel/STM32_CI/Scripts/main.cpp' -> '/home/espotel/STM32_CI/Test_run/STM32/Src/main.cpp'

Create timestamp.h file
timestamp.h created

Compile and build
Working with NUCLEO_F207ZG
Export PASS!
Headless build PASS!
Copy .bin file to test directory
'./Release/Src.bin' -> '/home/espotel/STM32_CI/Test_run/STM32/Binaries/NUCLEO_F207ZG/Release/Src.bin'
Success! .bin file generated for board NUCLEO_F207ZG

Flashing board NUCLEO_F207ZG...
Copy .bin file to /media/espotel/NODE_F207ZG
'/home/espotel/STM32_CI/Test_run/STM32/Binaries/NUCLEO_F207ZG/Release/Src.bin' ->
'/media/espotel/NODE_F207ZG/Src.bin'
Flash PASS!

Begin test sequence:
Begin test at /dev/ttyACM5
COM open
Clear serial bus:
Serial bus cleared!
Send "TEST"
```

```
Received: 2018-04-13_T09:27:35
Clear serial bus:
Serial bus cleared!
Send "PING"
Received: PONG
Send "PING"
Received: PONG
Send "PING"
Received: PONG
Test done!
Results:
Timestamp: PASS
PING-PONG: PASS

Progress: 1/10 completed



Compile and build
Working with NUCLEO_F767ZI
Export PASS!
Headless build PASS!
Copy .bin file to test directory
'./Release/Src.bin' -> '/home/espotel/STM32_CI/Test_run/STM32/Binaries/NUCLEO_F767ZI/Release/Src.bin'
Success! .bin file generated for board NUCLEO_F767ZI

Flashing board NUCLEO_F767ZI...
Copy .bin file to /media/espotel/NODE_F767ZI
'/home/espotel/STM32_CI/Test_run/STM32/Binaries/NUCLEO_F767ZI/Release/Src.bin' ->
'/media/espotel/NODE_F767ZI/Src.bin'
Flash PASS!

Begin test sequence:
Begin test at /dev/ttyACM9
COM open
Clear serial bus:
Serial bus cleared!
Send "TEST"
Received: 2018-04-13_T09:27:35
Clear serial bus:
Serial bus cleared!
Send "PING"
Received: PONG
Send "PING"
Received: PONG
Send "PING"
Received: PONG
Test done!
Results:
Timestamp: PASS
PING-PONG: PASS

Progress: 2/10 completed



Compile and build
Working with NUCLEO_F429ZI
Export PASS!
Headless build PASS!
Copy .bin file to test directory
'./Release/Src.bin' -> '/home/espotel/STM32_CI/Test_run/STM32/Binaries/NUCLEO_F429ZI/Release/Src.bin'
Success! .bin file generated for board NUCLEO_F429ZI

Flashing board NUCLEO_F429ZI...
Copy .bin file to /media/espotel/NODE_F429ZI
'/home/espotel/STM32_CI/Test_run/STM32/Binaries/NUCLEO_F429ZI/Release/Src.bin' ->
'/media/espotel/NODE_F429ZI/Src.bin'
Flash PASS!
```

```
Begin test sequence:
Begin test at /dev/ttyACM3
COM open
Clear serial bus:
Serial bus cleared!
Send "TEST"
Received: 2018-04-13_T09:27:35
Clear serial bus:
Serial bus cleared!
Send "PING"
Received: PONG
Send "PING"
Received: PONG
Send "PING"
Received: PONG
Test done!
Results:
Timestamp: PASS
PING-PONG: PASS

Progress: 3/10 completed



Compile and build
Working with NUCLEO_L476RG
Export PASS!
Headless build PASS!
Copy .bin file to test directory
'./Release/Src.bin' -> '/home/espotel/STM32_CI/Test_run/STM32/Binaries/NUCLEO_L476RG/Release/Src.bin'
Success! .bin file generated for board NUCLEO_L476RG

Flashing board NUCLEO_L476RG...
Copy .bin file to /media/espotel/NODE_L476RG
'/home/espotel/STM32_CI/Test_run/STM32/Binaries/NUCLEO_L476RG/Release/Src.bin' ->
'/media/espotel/NODE_L476RG/Src.bin'
Flash PASS!

Begin test sequence:
Begin test at /dev/ttyACM6
COM open
Clear serial bus:
Serial bus cleared!
Send "TEST"
Received: 2018-04-13_T09:27:35
Clear serial bus:
Serial bus cleared!
Send "PING"
Received: PONG
Send "PING"
Received: PONG
Send "PING"
Received: PONG
Test done!
Results:
Timestamp: PASS
PING-PONG: PASS

Progress: 4/10 completed



Compile and build
Working with NUCLEO_L073RZ
Export PASS!
Headless build PASS!
Copy .bin file to test directory
'./Release/Src.bin' -> '/home/espotel/STM32_CI/Test_run/STM32/Binaries/NUCLEO_L073RZ/Release/Src.bin'
Success! .bin file generated for board NUCLEO_L073RZ
```

```
Flashing board NUCLEO_L073RZ...
Copy .bin file to /media/espotel/NODE_L073RZ
'/home/espotel/STM32_CI/Test_run/STM32/Binaries/NUCLEO_L073RZ/Release/Src.bin' ->
'/media/espotel/NODE_L073RZ/Src.bin'
Flash PASS!

Begin test sequence:
Begin test at /dev/ttyACM0
COM open
Clear serial bus:
Serial bus cleared!
Send "TEST"
Received: 2018-04-13_T09:27:35
Clear serial bus:
Serial bus cleared!
Send "PING"
Received: PONG
Send "PING"
Received: PONG
Send "PING"
Received: PONG
Test done!
Results:
Timestamp: PASS
PING-PONG: PASS

Progress: 5/10 completed



Compile and build
Working with NUCLEO_F303ZE
Export PASS!
Headless build PASS!
Copy .bin file to test directory
'./Release/Src.bin' -> '/home/espotel/STM32_CI/Test_run/STM32/Binaries/NUCLEO_F303ZE/Release/Src.bin'
Success! .bin file generated for board NUCLEO_F303ZE

Flashing board NUCLEO_F303ZE...
Copy .bin file to /media/espotel/NODE_F303ZE
'/home/espotel/STM32_CI/Test_run/STM32/Binaries/NUCLEO_F303ZE/Release/Src.bin' ->
'/media/espotel/NODE_F303ZE/Src.bin'
Flash PASS!

Begin test sequence:
Begin test at /dev/ttyACM7
COM open
Clear serial bus:
Serial bus cleared!
Send "TEST"
Received: 2018-04-13_T09:27:35
Clear serial bus:
Serial bus cleared!
Send "PING"
Received: PONG
Send "PING"
Received: PONG
Send "PING"
Received: PONG
Test done!
Results:
Timestamp: PASS
PING-PONG: PASS

Progress: 6/10 completed


Compile and build
```

```
Compile and build
Working with NUCLEO_F091RC
Export PASS!
Headless build PASS!
Copy .bin file to test directory
'./Release/Src.bin' -> '/home/espotel/STM32_CI/Test_run/STM32/Binaries/NUCLEO_F091RC/Release/Src.bin'
Success! .bin file generated for board NUCLEO_F091RC

Flashing board NUCLEO_F091RC...
Copy .bin file to /media/espotel/NODE_F091RC
'/home/espotel/STM32_CI/Test_run/STM32/Binaries/NUCLEO_F091RC/Release/Src.bin' ->
'/media/espotel/NODE_F091RC/Src.bin'
Flash PASS!

Begin test sequence:
Begin test at /dev/ttyACM1
COM open
Clear serial bus:
Serial bus cleared!
Send "TEST"
Received: 2018-04-13_T09:27:35
Clear serial bus:
Serial bus cleared!
Send "PING"
Received: PONG
Send "PING"
Received: PONG
Send "PING"
Received: PONG
Test done!
Results:
Timestamp: PASS
PING-PONG: PASS

Progress: 7/10 completed



Compile and build
Working with NUCLEO_L152RE
Export PASS!
Headless build PASS!
Copy .bin file to test directory
'./Release/Src.bin' -> '/home/espotel/STM32_CI/Test_run/STM32/Binaries/NUCLEO_L152RE/Release/Src.bin'
Success! .bin file generated for board NUCLEO_L152RE

Flashing board NUCLEO_L152RE...
Copy .bin file to /media/espotel/NODE_L152RE
'/home/espotel/STM32_CI/Test_run/STM32/Binaries/NUCLEO_L152RE/Release/Src.bin' ->
'/media/espotel/NODE_L152RE/Src.bin'
Flash PASS!

Begin test sequence:
Begin test at /dev/ttyACM2
COM open
Clear serial bus:
Serial bus cleared!
Send "TEST"
Received: 2018-04-13_T09:27:35
Clear serial bus:
Serial bus cleared!
Send "PING"
Received: PONG
Send "PING"
Received: PONG
Send "PING"
Received: PONG
Test done!
Results:
```

```
Timestamp: PASS
PING-PONG: PASS

Progress: 8/10 completed



Compile and build
Working with NUCLEO_F401RE
Export PASS!
Headless build PASS!
Copy .bin file to test directory
'./Release/Src.bin' -> '/home/espotel/STM32_CI/Test_run/STM32/Binaries/NUCLEO_F401RE/Release/Src.bin'
Success! .bin file generated for board NUCLEO_F401RE

Flashing board NUCLEO_F401RE...
Copy .bin file to /media/espotel/NODE_F401RE
'/home/espotel/STM32_CI/Test_run/STM32/Binaries/NUCLEO_F401RE/Release/Src.bin' ->
'/media/espotel/NODE_F401RE/Src.bin'
Flash PASS!

Begin test sequence:
Begin test at /dev/ttyACM8
COM open
Clear serial bus:
Serial bus cleared!
Send "TEST"
Received: 2018-04-13_T09:27:35
Clear serial bus:
Serial bus cleared!
Send "PING"
Received: PONG
Send "PING"
Received: PONG
Send "PING"
Received: PONG
Test done!
Results:
Timestamp: PASS
PING-PONG: PASS

Progress: 9/10 completed



Compile and build
Working with NUCLEO_F103RB
Export PASS!
Headless build PASS!
Copy .bin file to test directory
'./Release/Src.bin' -> '/home/espotel/STM32_CI/Test_run/STM32/Binaries/NUCLEO_F103RB/Release/Src.bin'
Success! .bin file generated for board NUCLEO_F103RB

Flashing board NUCLEO_F103RB...
Copy .bin file to /media/espotel/NODE_F103RB
'/home/espotel/STM32_CI/Test_run/STM32/Binaries/NUCLEO_F103RB/Release/Src.bin' ->
'/media/espotel/NODE_F103RB/Src.bin'
Flash PASS!

Begin test sequence:
Begin test at /dev/ttyACM4
COM open
Clear serial bus:
Serial bus cleared!
Send "TEST"
Received: 2018-04-13_T09:27:35
Clear serial bus:
Serial bus cleared!
Send "PING"
```

```
Received: PONG
Send "PING"
Received: PONG
Send "PING"
Received: PONG
Test done!
Results:
Timestamp: PASS
PING-PONG: PASS

Progress: 10/10 completed


Compress "Logs" directory
/home/espotel/STM32_CI/Test_run/STM32/Documents/Logs/
/home/espotel/STM32_CI/Test_run/STM32/Documents/Logs/NUCLEO_F767ZI_build_log.txt
/home/espotel/STM32_CI/Test_run/STM32/Documents/Logs/NUCLEO_F429ZI_build_log.txt
/home/espotel/STM32_CI/Test_run/STM32/Documents/Logs/NUCLEO_F091RC_build_log.txt
/home/espotel/STM32_CI/Test_run/STM32/Documents/Logs/NUCLEO_L073RZ_build_log.txt
/home/espotel/STM32_CI/Test_run/STM32/Documents/Logs/NUCLEO_F103RB_build_log.txt
/home/espotel/STM32_CI/Test_run/STM32/Documents/Logs/NUCLEO_L152RE_build_log.txt
/home/espotel/STM32_CI/Test_run/STM32/Documents/Logs/NUCLEO_F207ZG_build_log.txt
```