Accenture StormTest Framework Guide

Accenture StormTest Development Center

Document ID: ST-12034

Revision Date: September 2016

Product Version: 3.3

Contact us at http://www.accenturestormtest.com or stormtest-support@accenture.com

Contents

1 Preface

1.1 Accenture StormTest

Accenture StormTest Development Center is the leading automated test solution for digital TV services. It is designed to reduce the cost of getting high quality digital TV services to market faster.

StormTest Development Center greatly reduces the need for time-consuming, expensive and error-prone manual testing and replaces it with a more accurate and cost-effective alternative. It scales easily to large numbers and types of devices and integrates with existing infrastructure to give much greater efficiency in testing. It can be used to verify and validate services on a virtually every piece of consumer premises equipment (CPE), from set-top boxes to games consoles and from iPads to Smart TVs. It has been specifically designed to meet the needs of developers and testers of these CPE devices and the applications which run on them.

StormTest Development Center consists of:

- A choice of hardware units that can test 1, 4, 16 or 64 devices. Each device under test can be controlled individually and independently and the audio/video from each device can be captured and analysed to determine the outcome of the test. The StormTest hardware can support either Standard Definition (NTSC/PAL) or High Definition (up to 1080p resolution) video inputs.
- Server software that controls all the hardware and STBs in the cabinet as well as managing a central repository of test scripts and a central database of test results.
- A Client API that allows test scripts to interact with the server software
- A number of graphical tools that allow the user to directly control STBs in the StormTest Development Center rack, to schedule tests to run and to view the results of those test runs.

Test scripts can be run from any location – the tester needs only a network connection to the physical StormTest cabinet. Video and audio output from the STBs under test can be streamed over a network to any location, allowing remote monitoring and control of testing, either within a company LAN or across a WAN.

Alternatively, scheduled tests can run directly on the server, negating the need for maintaining continuous network connection to the StormTest server.

For more detail on the StormTest Development Center product line, see the StormTest Development Center Brochure on the S3 Group website, http://www.s3group.com/tv-technology.

1.2 About this document

This document describes how to use the StormTest deployment framework to write powerful, flexible test solutions.

It can be used by a test writer to easily write powerful and extendible test solutions for a set top box platform.

It is expected that the reader has a good knowledge of basic Python scripting, has gone through the StormTest Programmer`s Guide and has familiarised themselves with the Navigator tool. The reader is also expected to understand the concept of a software framework and the basic concepts of object oriented programming.

1.3 Who should read this document

This document is targeted at the following groups:

- Test writers.
- Those running pre-existing tests, who wish to gain an understanding of why a test is written in a particular way. This may aid in the interpretation of test failures.

1.4 Disclosure

This document and its related code are provided as is. Any change made to the framework which results in it not working anymore will not be supported.

1.5 Related Documentation

The complete StormTest Documentation set is an extensive one. However there is a smaller subset that is of more immediate relevance here:

1) Developer Suite User's Manual – a detailed document describing the StormTest Developer Suite. At a minimum a test writer using this framework will require Developer suite to explore and understand the set-box EPG and create Navigator files to capture this behaviour
2) Programmer's Guide - this shows by example the fundamentals of how to use the Client API in a test script. In fact the framework discussed here will normally extend on these examples but nonetheless it is useful to read this document.
3) StormTest Client API – the detailed description of the methods provided by the StormTest Client. These are the foundation methods for most of the Framework's own APIs.
4) StormTest <Customer release> Doxygen package – HTML code documentation created by the Doxygen tool. The code-base scope includes the current framework and (if applicable) any test code. Usefully, the actual code is also embedded in these files. Only the Doxyfile configuration file is distributed see section 9.4 to generate the HTML documentation.

1.6 Applicable StormTest Versions

The Framework described here is based on the capabilities of StormTest release 3.3 and should be used with only with a StormTest facility using version 3.3 onwards.

2 Why use a Framework?

One of the difficulties of automating set top box testing is the time taken to set it up. When writing your first tests you realize that across different tests much of the same code could be reused. It’s also true that a tester may wish run tests in different ways. A tester may also wish to run the same test/tests across multiple set top boxes and may need the facility to make changes as required per set top box.

It is also very important to ensure the maintainability of your code, in case you are required to test new hardware or software.

Taking these facts into account the framework was designed with the following objectives:

- To speed up the customer test development process. Setting up a framework can be a time consuming task and needs to be carefully engineered.
- Ease of use
- Maintainability
- Scalability
- Use of StormTest good practice

3 Framework Overview

3.1 Framework capabilities and what it provides

- **A Test template integrated with StormTest**

The framework is designed so that the focus of test development only needs to be on the EPG actions and verifications specific to the test. Any 'environmental' complexities associated with StormTest such as connecting to the server, reserving a slot and so on are transparent to the test. Consequently to create a test the script developer simply takes the test script template provided by the framework and develops their test-specific code from this point. Such code would normally consist of the core test and verification API calls that the framework provides (see further below) along with any user-developed methods.

Additionally the framework provides a simple invocation environment so that the test can subsequently be started either interactively (directed at a specific server and slot) or within the scheduler context, and ensures that a test result is returned in the proper fashion to StormTest.

- **Integration with StormTest Navigator**

The navigator is a graphical tool allowing one to quickly and intuitively capture the box EPG menu structure and key-press relationships. It is then possible for a test, using only the navigator methods of the StormTest Client API, to traverse this hierarchy. Hitherto this task has always consumed a lot of custom and project-unique effort, involving tasks such as creating screen captures, state tables and so on.

The framework adds a light layer on this API to further support test development. Primarily what it adds are methods to facilitate handling multiple navigator files (as would most easily describe a complex menu), perform screen-specific verifications and OCR captures, map navigator files to specific box types, and provide error handling consistent with the rest of the framework.

- Core STB test and verification methods

S3's extensive experience in test script development has brought the realisation that a significant majority of tests can be written with a relatively small set of methods. The framework identifies and provides such methods – they are in the areas of EPG traversal (aka 'navigation, aka 'open screen'), key press, screen verification and OCR capture.

- **Advanced Utility libraries**

Supplementing the core methods are utilities to handle more complex scenarios. Most of these focus on advanced aspects of OCR capture.

- **Inheritance**

The framework was designed to support the common scenario where a user needs to write tests that must run on a 'family' of boxes. The boxes are likely to be 'essentially the same but not identical'. The framework offers an object-oriented solution to this situation: the 'family' behaviour is described in a base 'stb' class and derived classes are then created for each of the model variants. These derived classes 'inherit' the family characteristics and thus need only describe the differences. The framework provides a series of example templates illustrating this (normally you should just need to copy and rename these) and also has a mechanism to look after the instantiation of the 'correct' stb object.

It is worth noting that with the introduction of the navigator much of this behaviour is now captured in a navigator file anyway making this situation even easier to handle.

- **Configurabilit**y

The framework provides a simple and robust mechanism to support test configuration independent of the actual test script code. In this way test behaviour can be modified to some extent without actually changing the test script code. This is typically useful where a test references something that is likely to change – the sensible approach would normally be to make the reference to a configuration item rather than 'hard-wire' the data in the code.

3.2 Directory Structure

The structure of the framework files is shown below, you should have all those files.

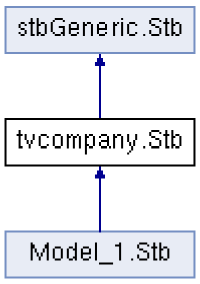
Directories	Comment
`├──extra`	Files to ease the deployment of the framework on your facility. Not test related
`├──s3group_framework`	
`├───stbGeneric`	Generic base class for an STB
`├───utilities`	Framework files and utilities
`├──target_framework`	Folder with your implementation
`├───stbModel`	Model specific classes
`│    ├──freesat`	Represents a 'family' of STBs
`│    │   ├──humax_hdr_1010s`	Specific model
`│    │   └──navigators`	Navigators relating to this family of STBs
`│    └──tvcompany`	
`│        ├──navigators`	
`│        ├──model1`	
`├───tests.examples`	The config file in here determines what STB the tests run on. The values can be overwritten from the command line also.
`│    └──advanced`	Advanced examples. Test must be at this level
`│    └──basic`	Basic examples. Test must be at this level
`│    └──wc`	Warning Center examples. Test must be at this level
`└    └──multi`	Multiple slots test examples

4 Framework Concepts - the Stb class hierarchy

The framework uses an object-oriented approach to facilitate and promote re-use of test code across a 'family' of boxes. This is achieved using a hierarchy of classes as explained in the following section

4.1 Stb class hierarchy

This diagram (you can also see this in Doxygen) shows the framework's Stb class hierarchy template



The example here is of course just a template, a particular deployment would retain the stbGeneric class but rename and extend the tvcompany and Model_1 classes. (The rename process is only a file copy and paste exercise with the caveat that there are some constraints on the file name – see next section)

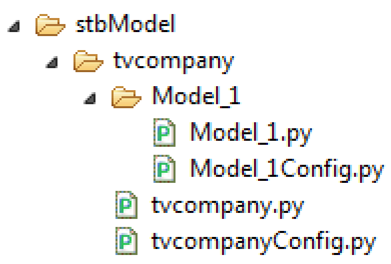
- **stbGeneric** is a generic class with 'methods' that are common to all stb implementations (in fact the core methods we discussed earlier are located here)
- **tvcompany** represents a derived 'family' class which inherits the generic methods and adds family-specific methods
- **Model_1** extends the class with target-specific methods. We described it as a family, typically of course there will be several Model_x classes.

A typical deployment design process will therefore try and identify family methods that can be re-used by all tests (usually these are core PVR activities such as tuning, recording, changing settings, etc), then run and debug these on a reference model before exploring the differences (if any) on the other family models. If there are differences then it is only these that have to be described in the new target class.

Of course the concept of 'model' is flexible – it really is a mechanism to represent any kind of difference between boxes. For example two boxes may have the same EPG and hardware platform but be configured in different operational modes and thus exhibit differences of some sort.

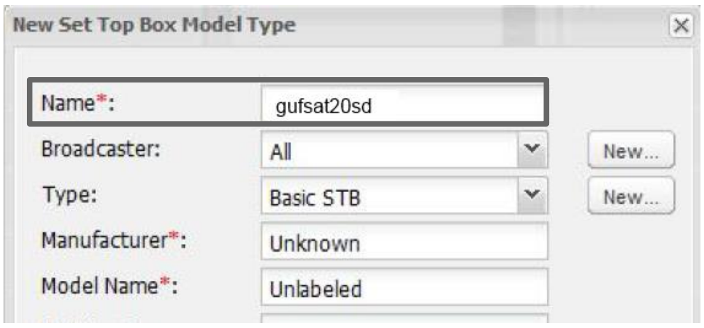
4.2 Adding new Stb classes

This starts by copying and renaming the template files tvcompany.py and Model_1.py and their associated config files. The copy and rename must retain the file hierarchy and naming convention implied in the folder diagram below:


```
stbModel
    tvcompany
        Model_1
            Model_1.py
            Model_1Config.py
        tvcompany.py
        tvcompanyConfig.py
```

Furthermore, and most importantly: **The file prefix of the model-level file ('Model_1' in this example) must have a matching entry in the device name field specified in the StormTest Admin console.**

WARNING: STB model name must not have spaces to work with the framework.


New Set Top Box Model Type

Name*:	gufsat20sd
Broadcaster:	All
Type:	Basic STB
Manufacturer*:	Unknown
Model Name*:	Unlabeled

The template files have a certain amount of mandatory code which must be retained (although you will need to change the class instance names). Thereafter you just extend the class as you see fit.

Here are detailed steps:

Support for the STB must first be added. Use the existing files as examples/templates and follow these steps:

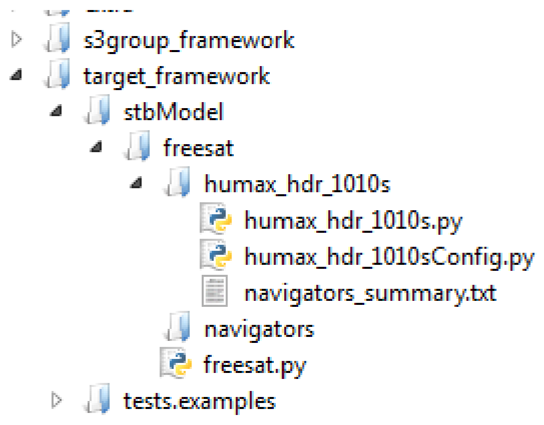
1. Create a 'family' directory under 'stbModel' (e.g : freesat)
2. In that directory, create a python file with the same name – use an existing one as a template(e.g: freesat.py)
3. Under that directory, create another directory for each STB model, using the same name as the model name in the StormTest admin console. (e.g: humax_hdr_1010s)
4. In each of these directories, create a python file with the same name as that directory. Again use an existing file as a template.(e.g: humax_hdr_1010s.py)
5. Create a second file, with an added postfix of 'Config'. This is where configuration data for the tests can be stored, and then accessed in the tests. The sort of data in here is the list of navigators and a list of channels to be tested.(e.g: humax_hdr_1010sConfig.py)
6. Finally, create a directory to hold your navigator files. Reminder: Each navigator file must be listed in the 'Config' python file

Once that is done, create your navigator file(s) and store them in the navigators directory. (e.g: humax_hdr_1010s_lineup.stnav_edit)

Finally, start creating the tests under the 'tests.examples\yourFolderName' directory, using the examples as guides. Tests must be at that level.

You could also use the python script create_model_directory.py in the 'extra' folder to create the file you need.

So if you created a model names: humax_hdr_1010s for the family of DUT: freesat it should look like this:



```
▷ s3group_framework
◢ target_framework
    ◢ stbModel
        ◢ freesat
            ◢ humax_hdr_1010s
                humax_hdr_1010s.py
                humax_hdr_1010sConfig.py
                navigators_summary.txt
              navigators
              freesat.py
    ▷ tests.examples
```

5 Framework Concepts – the anatomy of a test

As already mentioned the framework provides a test template[1] and a set of core methods. You'll need to understand these and use this style as a basis for your own tests.

The following sections discuss this in more detail, starting with the ubiquitous 'hello world' example.

5.1 Inspecting the Hello world code:

You should be able to locate this file at

<delivery_folder>\testFramework\target_framework\tests.examples\basic\hello_world.py

Open it up with your editor of choice and scroll down to the testBody method. You should see something similar to the text box below:

```
def testBody(stb, run_args = []):
    """
        Description:
            This function is the test core function.
        Return:
            test result
    """
    test_options = run_args[2]

    try:

        # MANDATORY
        # a test must be composed of one or more steps
        # using the defineStep method

        #============
        # STEP
        #============

        step_name = "print hello world"
        step_description = "just a print call"
        expected_result =  "the message is printed on the console"
        defineStep(step_name, step_description, expected_result)

        # insert user code here
        print "----------------hello world----------------"


        #============
        # STEP
        #============

        step_name = "print goodbye world"
        step_description = "just a print call"
        expected_result =  "the message is printed on the console"
        defineStep(step_name, step_description, expected_result)

        # insert user code here
```

[1] To be clear there is no explicit template file; example files are provided all adhering to a common template

```
        print "----------------goodbye world----------------"


        # MANDATORY
        #============
        # Test PASS
        #============

        # A test passes by default. An exception must be raised when a failure is
        # detected

        return reportTestPassed ()

    #============
    # Test FAIL
    #============

    except:
        return handleTestException()
```

Although the test is simply printing a message to the console the important points to take away are that the test has a specific structure and has certain mandatory elements.

The first mandatory element is that your test must consist of one or more 'steps'. The code clearly shows the syntax to define these. A step is meaningful to StormTest as it is used to delineate the log file into reporting-level sections. By using steps you'll therefore be able to readily quantify the extent of how far a test has progressed before any failure.

The last step is implicitly completed by the mandatory call 'reportTestPassed'. Of course this does not mean that all tests will pass. You can probably infer from the next lines that when a failure is detected the reportTestPassed method is not actually called. The framework API methods use python's exception handling mechanism to report such errors to StormTest (see later sections on error handling for more detail on this).

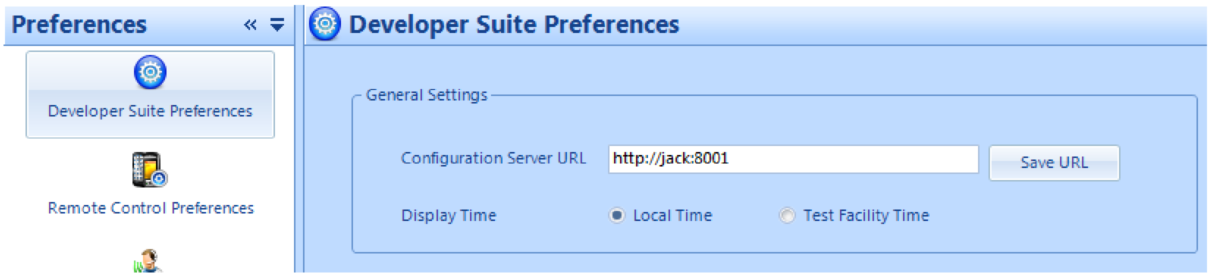
5.2 Running the Hello world code:

Although the test simply prints a message to the console behind the scenes a lot is going on: the test has to connect to the StormTest server and to a particular box on that server before actually executing the body of the test code. (This of course means you will need access to a StormTest server to run the test).

There are two ways this type of script can be run: either as a *scheduled* job or as an *interactive* job. Have a look at the StormTest Developer Suite User's Manual to detail these concepts but for example purposes I'll just show how here how to kick off an interactive job.

- Open a command shell
- CD to <your_folder>\testFramework\target_framework\tests.examples\basic
- Type: "Python hello_world.py <server_name or ip address> <slot number>"

You'll notice that a specific server and slot have to be specified to identify the box of interest. You'll be able to get the server name by opening up Developer Suite and looking under the preferences Tab or from the information presented in the start-up screen (server name is "jack" in this example).


Preferences
Developer Suite Preferences
Remote Control Preferences
Developer Suite Preferences
General Settings

Configuration Server URL: http://jack:8001 [Save URL]

Display Time: [● Local Time] ○ Test Facility Time

If the test is able to run you will get a lot of console output and a video monitor screen. On completion the console output that is of most relevance to the test execution status should be similar to the snippet shown below: (You'll notice that the defineStep definition is expanded out into the log and a number has been associated with each step. This is there of course to assist with log file analysis.)

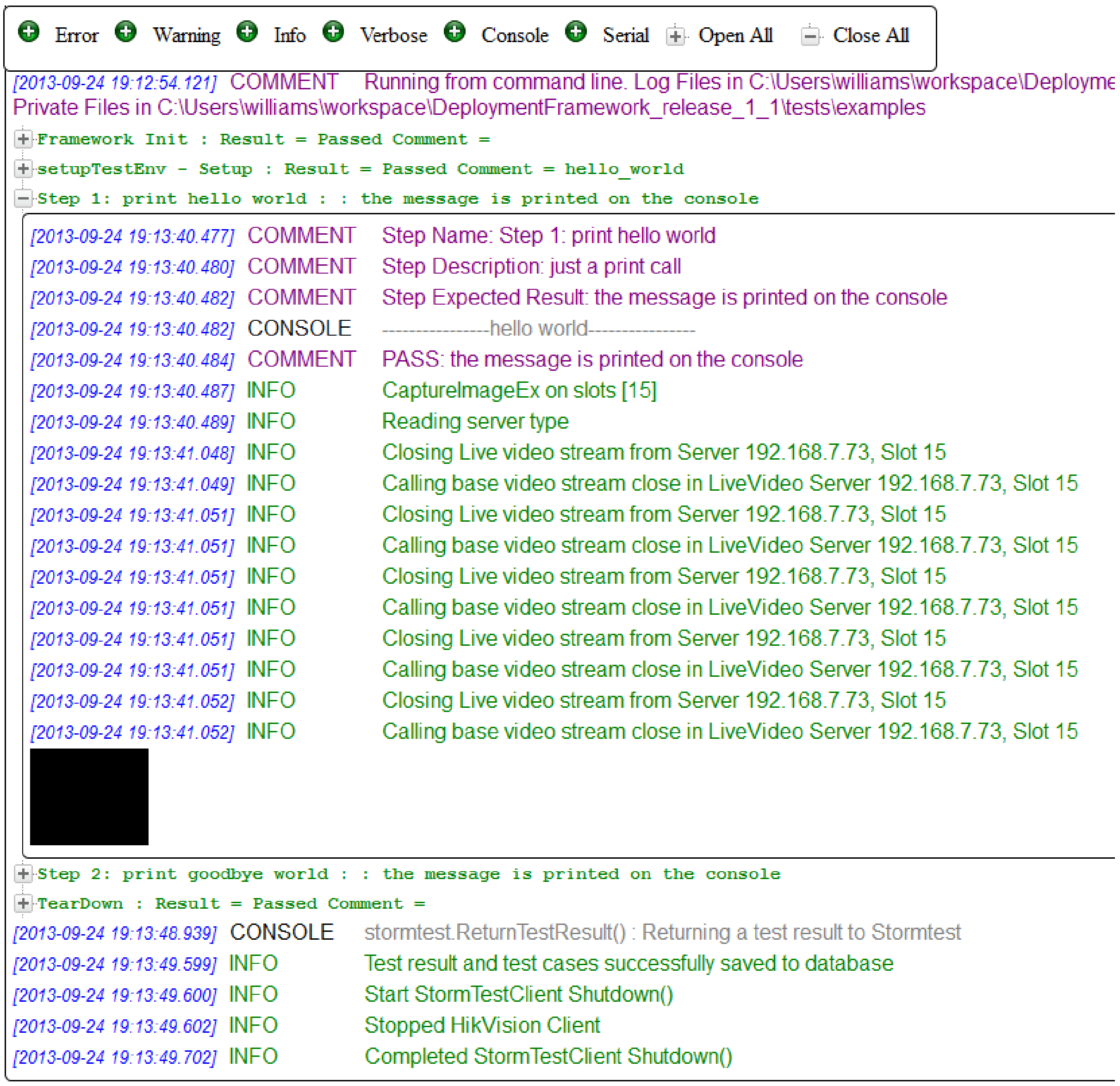
```
Starting test section Step 1: print hello world
7/11/2013 2:39:34 PM COMMENT: Step Name: Step 1: print hello world
7/11/2013 2:39:34 PM COMMENT: Step Description: just a print call
7/11/2013 2:39:34 PM COMMENT: Step Expected Result: the message is printed on the
console
----------------hello world----------------
7/11/2013 2:39:34 PM COMMENT: PASS: the message is printed on the console
7/11/2013 2:39:34 PM INFO    : CaptureImageEx on slots [1]
7/11/2013 2:39:34 PM INFO    : Reading server type
Ending test section Step 1: print hello world Result = Passed Comment = : the
message is printed on the console
Starting test section Step 2: print goodbye world
7/11/2013 2:39:35 PM COMMENT: Step Name: Step 2: print goodbye world
7/11/2013 2:39:35 PM COMMENT: Step Description: just a print call
7/11/2013 2:39:35 PM COMMENT: Step Expected Result: the message is printed on the
console
----------------goodbye world----------------
7/11/2013 2:39:35 PM COMMENT: PASS: the message is printed on the console
7/11/2013 2:39:35 PM INFO    : CaptureImageEx on slots [1]
………
Ending test section Step 2: print goodbye world Result = Passed Comment = : the
message is printed on the console
Starting test section TearDown
```

Now if you go to the location:

<your_folder>\ testFramework\target_framework\tests.examples\\logDir

you will also notice a folder named hello_world_YYYYMMDD_HHMMSS_MS

you can open the logFile.html file in a web browser (we recommend that you use Chrome or Firefox) and you should get something like this:



```
[2013-09-24 19:12:54.121]  COMMENT    Running from command line. Log Files in C:\Users\williams\workspace\Deployme
Private Files in C:\Users\williams\workspace\DeploymentFramework_release_1_1\tests\examples
Framework Init : Result = Passed Comment =
setupTestEnv - Setup : Result = Passed Comment = hello_world
Step 1: print hello world : : the message is printed on the console
  [2013-09-24 19:13:40.477]  COMMENT    Step Name: Step 1: print hello world
  [2013-09-24 19:13:40.480]  COMMENT    Step Description: just a print call
  [2013-09-24 19:13:40.482]  COMMENT    Step Expected Result: the message is printed on the console
  [2013-09-24 19:13:40.482]  CONSOLE    ----------------hello world----------------
  [2013-09-24 19:13:40.484]  COMMENT    PASS: the message is printed on the console
  [2013-09-24 19:13:40.487]  INFO       CaptureImageEx on slots [15]
  [2013-09-24 19:13:40.489]  INFO       Reading server type
  [2013-09-24 19:13:41.048]  INFO       Closing Live video stream from Server 192.168.7.73, Slot 15
  [2013-09-24 19:13:41.049]  INFO       Calling base video stream close in LiveVideo Server 192.168.7.73, Slot 15
  [2013-09-24 19:13:41.051]  INFO       Closing Live video stream from Server 192.168.7.73, Slot 15
  [2013-09-24 19:13:41.051]  INFO       Calling base video stream close in LiveVideo Server 192.168.7.73, Slot 15
  [2013-09-24 19:13:41.051]  INFO       Closing Live video stream from Server 192.168.7.73, Slot 15
  [2013-09-24 19:13:41.051]  INFO       Calling base video stream close in LiveVideo Server 192.168.7.73, Slot 15
  [2013-09-24 19:13:41.051]  INFO       Closing Live video stream from Server 192.168.7.73, Slot 15
  [2013-09-24 19:13:41.051]  INFO       Calling base video stream close in LiveVideo Server 192.168.7.73, Slot 15
  [2013-09-24 19:13:41.052]  INFO       Closing Live video stream from Server 192.168.7.73, Slot 15
  [2013-09-24 19:13:41.052]  INFO       Calling base video stream close in LiveVideo Server 192.168.7.73, Slot 15
Step 2: print goodbye world : : the message is printed on the console
TearDown : Result = Passed Comment =
[2013-09-24 19:13:48.939]  CONSOLE    stormtest.ReturnTestResult() : Returning a test result to Stormtest
[2013-09-24 19:13:49.599]  INFO       Test result and test cases successfully saved to database
[2013-09-24 19:13:49.600]  INFO       Start StormTestClient Shutdown()
[2013-09-24 19:13:49.602]  INFO       Stopped HikVision Client
[2013-09-24 19:13:49.702]  INFO       Completed StormTestClient Shutdown()
```

You can see that each step is part of a Node and by using the framework you get logging and screen capture for each step.

5.3 Using the core methods - Simple Navigation example

This example test (tests.examples\basic\simple_navigation.py) shows how the core navigation methods are used to do a simple screen traversal. In this case we want to open the tv guide and check it did actually open.

```
#============
# STEP
#============

step_name = "Check we can open the Guide"
step_description = "the test will navigate to the TV GUIDE"
expected_result = "Guide open and verification(s) pass"
```

```
defineStep(step_name, step_description, expected_result)

stb.guideOpen() # the name must match that defined in the navigator file
```

We want our test to be independent from the device that is used, hence at the script level we are calling methods defined in the model class: (in this case in freesat.py)

```
    def guideOpen(self):
        """
        Description:
            open the guide using the navigator
        """
        section_name = "guideOpen"
        beginTestSection(section_name)

        self.navigateTo("GUIDE_CHANNEL_VIEW")

        endTestSection(section_name, True)
```

Assuming we have already created a navigator file (using the Navigator tool) to define how to open the screen, created some verifications to confirm we got there and assigned a screen name (‘GUIDE_CHANNEL_VIEW) then the call **navigateTo** will essentially replay the required key-presses and perform all the specified screen verifications en-route. This might seem quite ordinary and straightforward but is in fact extremely powerful and a fundamental concept for this framework.

You might notice too that an ‘self’ prefix is required in the call. This indicates that the core methods are members of a device ‘class’, something we mentioned earlier. What this means is that the behaviour of the core methods has a dependency on the stb type that has been instantiated. In the case of the navigateTo method we can reasonably expect that the behaviour relates to the set of navigator files that are selected.

5.4 Using the core methods continued – adding dynamic verifications

The next code sample (stnModel\freesat\freesat.py) introduce more core methods to show how ‘dynamic’ verifications can be performed. What I mean by this is that whilst the Navigator is focussed on traversing static elements of the menu (‘getting you there’) a test will usually then need to perform certain dynamic verifications having got to this point. Typical examples of this are reading an event title in a guide screen or verifying a record icon in a banner – we can’t tell Navigator what to expect on these screens. Essentially what we want to do then is include these screens in the Navigator file (because we can use it to define a verification or OCR) but not actually ever *navigate* to them.

This concept is supported by the validateAt, checkIfAt and readAt methods; the following code sample proceeds to show how to use those methods.

```
    def bannerOpen(self, refresh=True):
    """ """
        num_retries = 3
        section_name = "bannerOpen"
        beginTestSection(section_name)

        banner_open = self.checkIfAt("BANNER_BASIC")
```

In this first sample we want to check if the banner is open. We have a screen defined in one of our navigator that could validate if the banner is open. So what we do here is call the function 'checkIfAt' to check if the screen in our navigator can be validated: to check if the banner is open.

This way we use a screen from the navigator but do not perform any navigation.

Now in that other example we want to OCR some text on the screen. The conventional way to read OCR using StormTest is to use one of the OCR API. Using the framework you can define screens in your navigator to do OCR. Then you can call the readText function:

```
onscreen_channel_name = self.readText("BANNER_BASIC_OCR_TITLE")
```

And if the screen "BANNER_BASIC_OCR_TITLE" contains an OCR, then it will return the text defined in the OCR region in the screen.

The readText call is a core method and essentially runs the navigator OCR capture defined for this screen and returns the OCR result. Note: *This is the most basic OCR call – there are quite a few more sophisticated readXX methods provided in the framework to optimise the OCR capture. You'll get a more detailed overview of these from the Doxygen documentation and from inspecting their application in the delivered tests (if applicable).*

Regarding the navigator screen, I mentioned earlier that the framework handles multiple navigator *files*. In the scenario described here it is in fact typical that the first guide screen would be located in a 'top-level' navigator file and a second file ('guide') would hold the detailed sub-screens, and, as in this case, specific screens to handle dynamic verifications. The framework makes the navigator screen searching transparent to the test – you just tell it the name of the screen.

In the provided navigator file for the freesat STB the 'top_level' navigator would be "humax_hdr_1010s_top" and the second file holding sub-screens is "humax_hdr_1010s_screen".

5.5 Configuration

If we check the service_title_on_banner.py script we can see a typical example of configuration file usage. The configuration file lets you store data specific to your environment or STB model.

Here first is the code to access the data:

```
# compare against the expected title, as specified in the config file
ref_channel_num=  stb.getConfigItem('channels','REFERENCE_CHANNEL','NAME')
```

And the corresponding entry in the config file:

```
config_data = dict (# obligatory name for top-level

channels=dict(

        REFERENCE_CHANNEL=dict  (
                    NAME="BBC 1 London",
                    DCA=101,
                    LINEUP_POS=1,
                    DEF="SD",
                ),
```

The test can then simply use 'reference_channel' ,'Name' in place of hard-wiring 'BBC 1 LONDON'.

Note that the getConfigItem function makes it easy to retrieve information from python dictionaries.

5.6 Error Handling

In many instances the core methods are able to detect a failure and will raise an exception stopping further test execution. However sometimes the decision to fail is best made at the test level. In such instances the test should raise an exception.

In this example (advanced\reboot.py) the test failed to measure the reboot time so an exception is raised by the test code to force an exit (and report a failure):

```
if not avg_reboot:
    raise TestVerificationError("Reboot failed")
```

At this point you should hopefully have a good overview of the core concepts. The next section is going to build on this and look at specific aspects to writing a test under the framework.

6 Test design under the Framework

We'll look here at some other aspects of the Framework and highlight some recommendations that will help maintain test quality

6.1 Use of Steps

As already mentioned you must have at least one step in the test. In fact it is highly recommended that you insert further steps for each significant verification point in the test. Not doing so means you will not be readily able to identify the general area of failure in a test (unless you actually open the detailed log file). Furthermore the step name, description and expected result should have a concise but meaningful entry.

A useful step technique to note too is that it can be used to display any kind of 'analog' result value in the step name. The previous step performs the 'measurement' and the next step simply reports the result. In this way you can get an overview of such results without having to open the full log file.

Also note that step results are stored in the StormTest database, while logs are deleted by default after 21 days. To perform analytics it is therefore recommended to use steps. In addition test points used in the StormTest dashboard are using test step name to do analytics. It is therefore very important to give a meaningful name to your step. More info in the dashboard section

6.2 Navigator design recommendations

Ensure that the verification thresholds are robust and not 'on the edge'. The Navigator trainer tool makes this a very straightforward process. Also make sure that your navigator file containing navigation data can be fully validated

Use resources for region, area, colour and string as this will make your files easier to maintain.

Adopt a screen naming convention. At a minimum it should hint at the verification(s) in place. For example xxx_OCR, xxx_motion, etc.

Use multiple files otherwise the 'map' may become complicated and difficult to read.

Note: For the framework to be able to traverse across multiple files it needs to be assisted in identifying the connection points between the files. This is done by simply duplicating the screen name in each 'connected' pair of files. For example if a top-level navigator file has a screen name called 'parental_control_settings' then to be able to navigate into a detailed parental control navigator file that file also needs to contain an identical screen of the same name.

You can see this applied in *humax_hdr_1010s _top* and *humax_hdr_1010s_lineup*. The two Navigator files share the screen "LINEUP_1".

A specific navigator should be created to define screens for dynamic checks. These file won`t be validated through the StormTest navigator tool. The file *humax_hdr_1010s_screen* is an example of this principle.

6.3 Test Configurability

The framework supports configuration through run time arguments.

The run time arguments are specified in the invocation syntax using "–cust*n* <string>" or "—misc*n* <*int*>" where n is [1-4]. These values are then available to the test as:

`test_options.cust1….. test_options.misc3..`

It's important to note that the run arguments have no general meaning, the test can interpret the values in any way it likes.

The configuration file supports a more extensive and structured approach. It is arranged as a hierarchical python dictionary with configuration items accessed using the core method

```
getConfigItem (key_1, key_2,….key_n)
```

`key_1,..n` represent the hierarchical path/ dictionary 'keys' to the data of interest within the config file. Using meaningful key names is of course encouraged so that the file becomes largely self-documenting.

The configuration file can exist at one or more levels. At a minimum it must be present at the 'family' level (it has no real meaning at the stbGeneric level) and can optionally be included at the model level. The data items at the lowest level take precedence – in other words you can re-define a data item at the model level, or not define it all in which case the 'family' value is used.

There is one mandatory element in every config file - the name of the top-level dictionary must be 'config_data' - thereafter any hierarchy of sub-dictionaries can be used.(see 4.5 for example

6.4 Navigator configuration

The navigator files (normally at least one, and usually more) must be listed in the 'model-level' configuration file as just described. (As we need model-specific Navigator file sets). The following is an example from the template file, 'Model1Config.py'

```
navigator_info=dict(

        INITIAL_SCREEN="LIVE",
        INITIAL_NAVIGATOR="model_1_your_top_level_navigator",
        NAVIGATORS=[     " model_1_your_top_level_navigator",
                         " model_1_navigator_for_a_sub_menu",
                    ],

    ),
```

6.5 Writing portable tests

By this it is meant tests that can run unchanged across a family of related Stbs.

The framework includes several mechanisms to assist this process as has already been noted. To recap on these and their application:

- The Stb class hierarchy. If there are model 'differences' create a new class rather than popping in some 'conditionality'
- The Navigator in very many cases removes the need for custom navigation and verification code. If at all possible keep with this pattern – changes can then be largely managed by loading a different navigator file set through the config file.

7 Framework – How to

7.1 How to continue the test when a test verification error has been raised

Refer to the test lineup_test.py.

In this case, in the script the TestVerificationError that could be raised during step 2 is caught and the test keeps going.

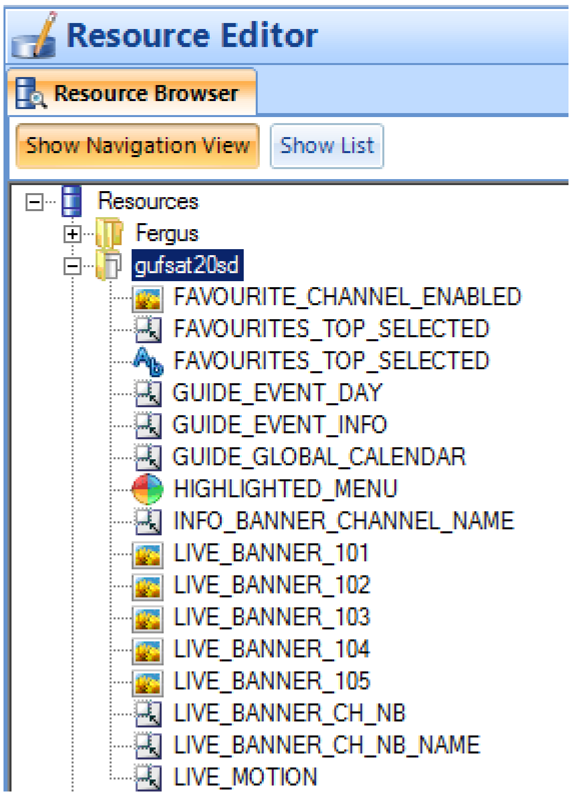
This could be useful if you need steps that should not make your test fail.

7.2 How to use resources within the framework

The stbGeneric class defines the resource path to be used as:

#resources configuration - need to add this in the resources editor
stormtest.SetResourcePath(name)

Which means that you should create a resource folder using your DUT name, like this (following our example DUT) in the resource editor:



```
Resources
  Fergus
  gufsat20sd
    FAVOURITE_CHANNEL_ENABLED
    FAVOURITES_TOP_SELECTED
    FAVOURITES_TOP_SELECTED
    GUIDE_EVENT_DAY
    GUIDE_EVENT_INFO
    GUIDE_GLOBAL_CALENDAR
    HIGHLIGHTED_MENU
    INFO_BANNER_CHANNEL_NAME
    LIVE_BANNER_101
    LIVE_BANNER_102
    LIVE_BANNER_103
    LIVE_BANNER_104
    LIVE_BANNER_105
    LIVE_BANNER_CH_NB
    LIVE_BANNER_CH_NB_NAME
    LIVE_MOTION
```

7.3 How to use the –skip-startup argument- what is it for?

Most of the scripts provided with the framework are here for educational purposes, to show you the different features of the framework.

When you start your project you will quickly see the need to create what we call a startup procedure. The idea of such a step is to prepare your DUT for your test.

You could check the testStartup function in humax_hdr_1010s.py file. This is a good starting point for any start up procedure.

And you can see it in action in the script zapping.py.

A startup procedure could be time consuming (imagine that you have to delete recordings, favourites or do a factory reset). The –skip-startup argument lets you skip that procedure and save time during your script development or execution.

7.4 How to use utilities, stb functions, config file all together

There are several ways to do this. You could for instance import the utilities inside your script and use them directly inside your steps.

However, the approach that we recommend is the one adopted in the script zapping.py

The script only calls functions of the stb class.

In freesat.py a function to get an average zapping time (note this could also be in stbGeneric.py) is there using a utility. This way the test is not DUT specific (which could become the case if the utilities are used inside the test script directly)

7.5 How to differentiate DUTs by using their software version

This can be done by adding the software version to the 'Description' field of the 'DUT Instance' in the Admin Console. Then in the Framework itself, by creating a different folder per software version, this is usually only necessary for the navigator files, on initialisation the software version of the DUT is read and the appropriate configuration files, navigators etc., can be loaded.

7.6 How to run the examples

The samples provided follows the recommendations in this document, hence it use resources and navigator files. You should follow those steps to run the samples:

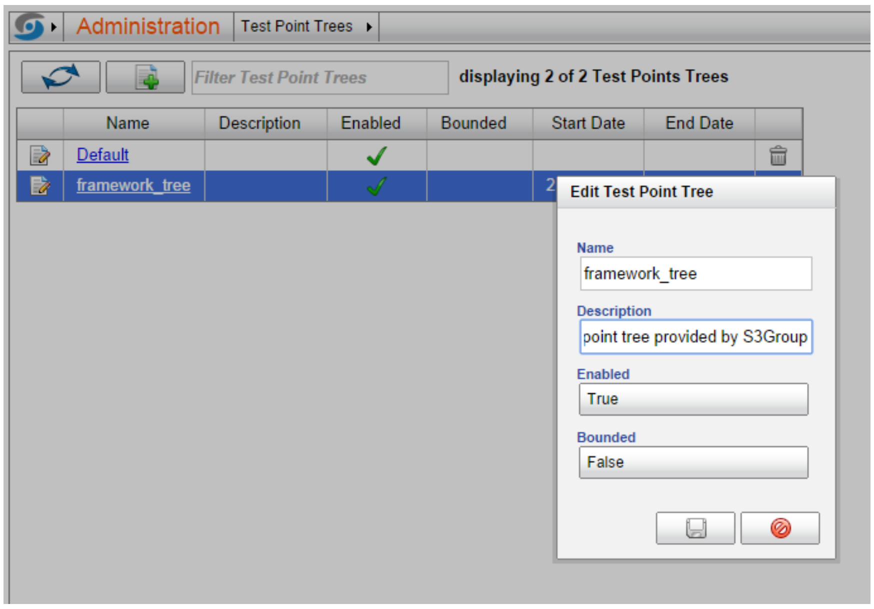
- In the StormTest admin console, create a new model. Model name must be: humax_hdr_1010s
- Add an instance of "humax_hdr_1010s" in desired slot
- In the folder "extra, change the script export_resource_to_server.py so that the variable importServer to the name of your StormTest config server. Then run it with the arguments "–i True"
- In the folder "extra", change the script "export_framework_to_server.py" configuration to match yours and run it.

7.7 How to easily add a STB model in your framework?

In the extra folder refer to the “create_model_directory.py” folder. It will create automatically the files and folder you need.

7.8 How to import provided test points tree?

The framework is provided with a test point tree that matches the example script provided. To import the tree inside your facility you first need to create a tree in the dashboard. (Refer to the dashboard documentation if you don’t know this procedure or get in touch with StormTest support)



displaying 2 of 2 Test Points Trees

Name	Description	Enabled	Bounded	Start Date	End Date
Default		✓			
framework_tree		✓		2	

Edit Test Point Tree

Name: framework_tree

Description: point tree provided by S3Group

Enabled: True

Bounded: False

From your StormTest configuration server, extract the file testPoints.xml from extra\\testPoints.zip

Open a cmd line and go to the directory:

```
c:\program files\apache for stormtest\htdocs\stormtestdashboard\php
```

Test the import:

```
>>>>php.exe importTestPoints.php testPoints.xml framework_tree
```

To commit the change:

```
>>>>php.exe importTestPoints.php testPoints.xml framework_tree commit
```

Upon execution your scripts you can now browse the dashboard test points results and trends.

7.9 How to use the VQA utility?

The example video_analysis.py demonstrate how you could launch a video analysis. **Note that you must have purchased a VQA license for this utility to work.**

The VQA utility use the python library pygal to plot the results in .svg files.

Therefore you need to install pygal wherever you run the VQA utility (your StormTest client and the StormTest server basically)

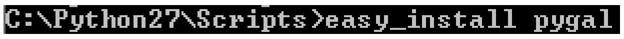
First you will install setuptools which allows you to manage Python packages easily.

Save that file on your StormTest server and/or client and run it:

https://bootstrap.pypa.io/ez_setup.py

Now from a command line simply go to python2.7\script location and run:

easy_install pygal



```
C:\Python27\Scripts>easy_install pygal
```

Done!

7.10 Running a Warning Center Script

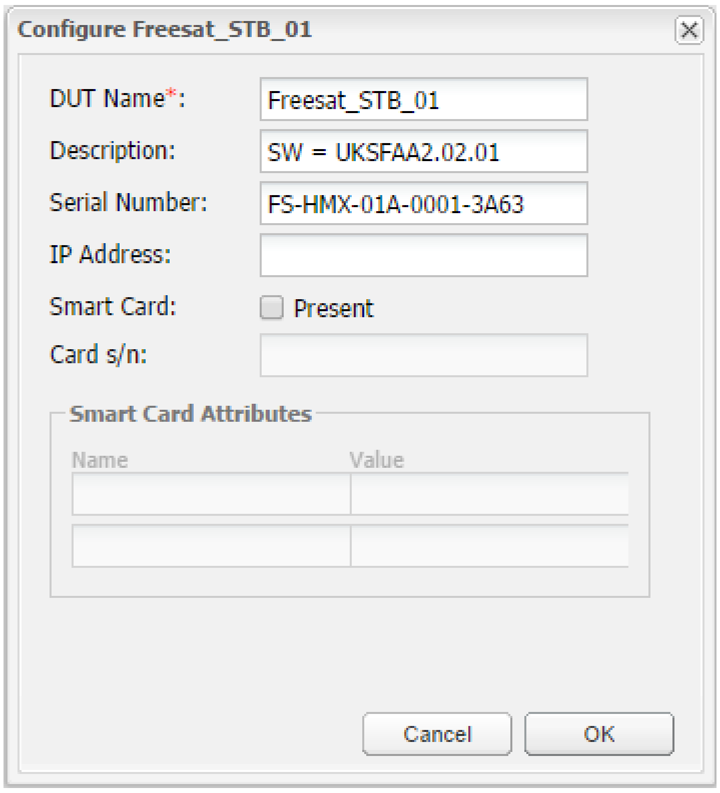
Examples of how a Warning Center script can be run from within the Framework can be found in \tests.examples\wc. The example scripts are `linearServiceMonitor.py` and `linearServiceMonitor_MOS.py`.

7.11 How to get the DUTSW test point to report the actual Software version of my DUT?

This test points, used in the zapping.py examples during the TestSetup is provided as an example on how you could use the description field of your DUT instance in the admin console.

Going to admin console you need to fill the description field like this: (you must respect "SW = " syntax if you don`t want to change the framework code)

The code parsing that info is in stbGeneric.py in the getStbSwVersion() function.



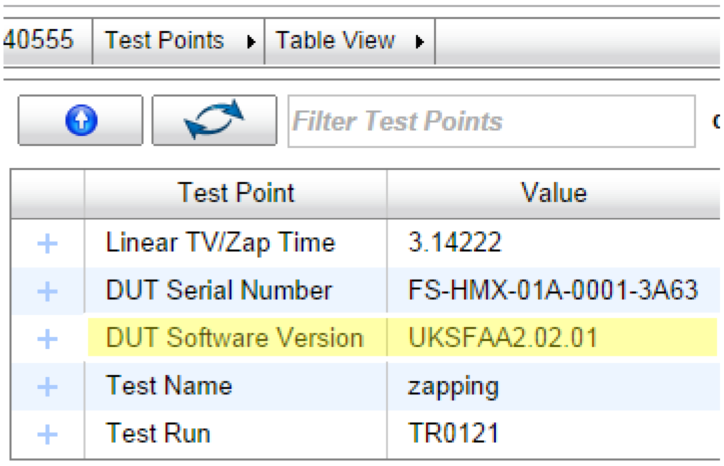
Configure Freesat_STB_01

DUT Name*:	Freesat_STB_01
Description:	SW = UKSFAA2.02.01
Serial Number:	FS-HMX-01A-0001-3A63
IP Address:	
Smart Card:	☐ Present
Card s/n:	

Smart Card Attributes

Name	Value

From the dashboard you will get:



	Test Point	Value
+	Linear TV/Zap Time	3.14222
+	DUT Serial Number	FS-HMX-01A-0001-3A63
+	DUT Software Version	UKSFAA2.02.01
+	Test Name	zapping
+	Test Run	TR0121

8 Running a test

8.1 Invocation from Command line, aka 'interactive job'

A test can be started as follows from the command line:

python testname.py <run_server> <slot> + <further optional arguments>

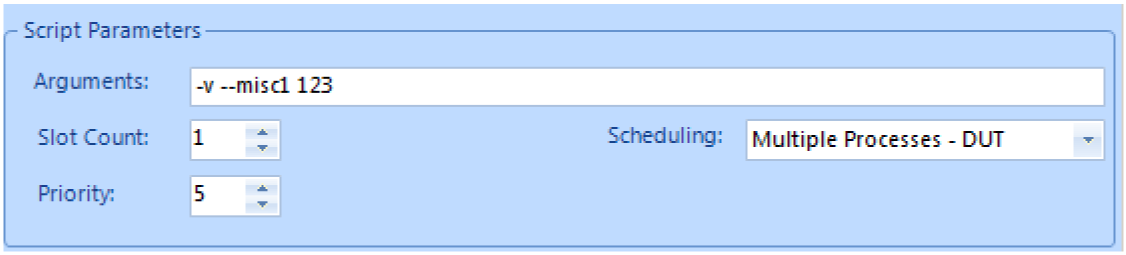
<run_server> and <slot> are **mandatory** arguments and refer respectively to the StormTest server and the slot where the stb model resides.

8.2 Invocation from the StormTest scheduler

Scheduled runs are launched from within Developer suite and in the context of a python executable and a specific server and slot.

In this case there is no need to add any arguments (the configuration file will be used by default and the server name and slot number will be retrieved from the configuration)

In addition If you want to put optional arguments you don`t need to input the <run_server> and <slot> in that case, see example below:



Arguments: -v --misc1 123

Slot Count: 1

Scheduling: Multiple Processes - DUT

Priority: 5

8.3 Configuration file

If no arguments are specified, the arguments will be retrieved from the file in

Tests.examples\\runArgs.cnf

Arguments in this file must be specified in the same way they would be from a command line**. (with a space at the end of the line)**

The configuration file is used only if there is no argument specified (from command line or scheduler).

Note that this file must be present for the framework to run.

8.4 Optional arguments (for both scheduled and interactive jobs)

-c	configuration_server. Not required (and must not be specified) for scheduled runs. Include this option only for an interactive job and only if the configuration server is not the same machine as the run server.
-v	Include this option if you wish to capture a video log for the test run - the default is no video log

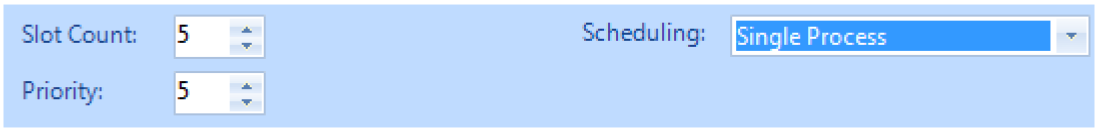
--video_buffer	Enable the video buffer, specify the length in seconds
-l	Language. Include this option to specify the language setting - the default is defined in target init.
-m	Resolution. Include this option to specify the required resolution - the default is 4CIF
-n	Videostandard. Include this option if you wish to set the video standard - default is PAL, other option is NTSC.
--bit_rate	Set the maximum encoding bitrate of the capture card, default is highest value
--proj_dir	absolute or relative path for interactive run logs -do not specify if using Daemon
--log_dir	relative path for interactive run logs
--server_nav	server side navigation, use if you want to use navigator file stored in Public Navigators
--no_window	no video streaming (and no window) from Server to Client (VideoFlag in ReserveSlot)
--frame_rate	Set the Maximum frame rate, limited by the standard. The value is between 1 and 25 for PAL DUTs, 1 and 30 for NTSC DUTs. For HD systems, the valid range is 1 to 60. However, the actual frame rate is limited to 30 on HD systems if the source is 1920 x 1080.
--skip_startup	skip startup step in test body
--test_run	the id of the test runs
--misc1 – misc5	Miscellaneous value

8.5 Multiple slots testing

To run a test that targets multiple slots in a single process the <slot> argument needs to be replaced by a tuple specifying the slot to use, e.g:

```
python testname.py  S15003HV04  [2,3,4]   -W
```

From the scheduler the Scheduling option must be set to single process:


Slot Count: 5
Priority: 5
Scheduling: Single Process

Note that other arguments will be applied to all the slots reserved.

Example of multi slot testing are available in test.examples/multi

9 Viewing test results

9.1 Test ran with the Scheduler

If running a test from the storm test scheduler then results can be viewed using the StormTest Developer Suite -> Test manager -> Test Results.

9.2 Test ran from command line

If running from the command line then a results directory should be created per run of each test in the relevant test folder directory. The StormTest Log View application should be used to load the logfile.html file which should exist in the results directory. This will contain video (if enabled) and console logging for the test run.

Note that you can also see the results of tests run from the command line in the Developer Suite, but access to the logs is restricted to the user who launched the test.

9.3 StormTest Dashboard

The StormTest development centre Dashboard is a web application for monitoring a Development centre facility. It provides an extra way to check the results of test run. Being a web application it can be accessed without the StormTest client installer being installed. It provides detailed test run logs and useful statistics on your facility.

The StormTest dashboard is not part of the software provided along with a normal StormTest developer suite installation. Please contact StormTest support team to get it.

The full documentation of the dashboard is available under the Engage portal in the Doc area: here

9.4 Doxygen –

Doxygen documentation can be generated for the framework.

You will find an example DoxyFile in the *testFramework* folder.

Please follow the example code in the framework in terms of adding the doxygen tags to each new method you add.

To run doxygen, install doxygen and run it as follows from the dos prompt (cmd prompt pointing at the testframework folder):

```
Doxygen Doxyfile_config
```

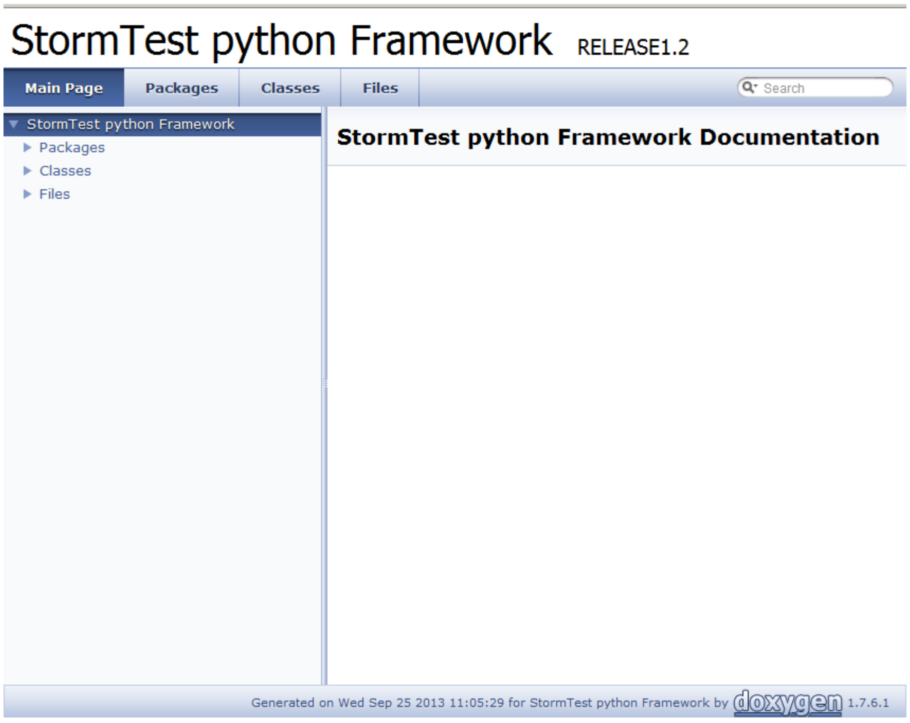
```
e.g: C:\Users\williams\workspace\DeploymentFramework_release_1_2>Doxygen Doxyfile_config
```

You will need to edit the following lines of the Doxyfile_config file to point it to the directory locations on your machine.

```
INPUT               = absolute\path\to\your\directory

OUTPUT_DIRECTORY      = absolute\path\to\your\output\directory
```

Once the output has been generated in the output directory, open the index.html file within the html directory to view the doxygen output. You should get something like this:


StormTest python Framework RELEASE1.2
Main Page
Packages
Classes
Files
Search
StormTest python Framework
Packages
Classes
Files

StormTest python Framework Documentation

Generated on Wed Sep 25 2013 11:05:29 for StormTest python Framework by doxygen 1.7.6.1

The framework methods which can be called from a user's test are listed in the Appendix.

9.5 Work Environment - IDE

The framework is delivered independently of any IDE but has been developed using the spyder IDE and has also been tested under Eclipse using the PyDev plugin.

You can find more information about IDEs on the StormTest forum

9.6 Share your ideas

When extending your framework or developing your test cases, you may face some problems which are common across StormTest users. We invite you to join the **StormTest forum community** to share your thought, python code and experience.

The Software – Developing Test scripts – board is THE PLACE to discuss these issues.

Appendix

File	Method
.\stbGeneric\stbGeneric.py	stbInfo
	getRcHistory
	getSerialParameters
	setSerialParameters
	setStbVideoOffset
	getStbVideoOffset
	updateNavLocation
	validateNavigator
	navigateTo
	validateAt
	validateNotAt
	checkIfAt
	readText
	readContains
	readNumbers
	readTime
	readDate
	pressDigits
	pressKeys
	getConfigItem
	getConfigItemIfAvailable
	getZapMeasurment
	getChannelListDCA
	checkServiceAvailableLinear
	checkVideoQuality
	checkAudioPresent
	checkVideoPresent
	checkVideoMotion
	measureVideoQuality
	tuneToLinearService
	waitFor
	navigateToInitialScreen
	getNavLocation
	setWcResultStatus
	setWcResultScreenshot
	setWcServiceName
	setWcResultValue
	addWcExtraEvent
.\utilities\avUtil.py	setImageMatchPercent
	getImageMatchPercent
	setWaitImageMatchTimeout

	getWaitImageMatchTimeout
	setWaitImageMatchComparisonFreq
	getWaitImageMatchComparisonFreq
	verifyImage
	verifyIcon
	verifyNotIcon
	verifyNotImage
	verifyBackgroundMotion
	verifyMotion
	verifyNoMotion
	verifyImageMatch
	captureScreen
	compareColor
	waitIconMatch
	waitImageMatch
	waitIconNoMatch
	waitImageNoMatch
	detectMotionEx
	compareImageEx
	measureZapping
	waitOnMotionRectangle
	measureAudio
	measureVideoQuality
	measureTransition
.\utilities\environmentUtil.py	launchTest
.\utilities\sdoUtil.py	parseNavigationResult
	parseValidateScreenResult
	waitScreenDefMatch
	waitScreenDefNoMatch
	getScreenDefinitionOCRRegionRect
	parseScreenDefinitionInfo
	navigatorCallback
.\utilities\testUtil.py	defineStep
	reportTestPassed
	beginTestSection
	endTestSection
	endWCStep
	beginWCStep
	printTrace
	setStepStatusFail
	reportTestCompleted
	handleTestException
.\utilities\vqaUtil.py	setVideoLogLimit

	startAnalysis
	stopAnalysis
.\utilities\WcTestHandler.py	disableWCResult
	enableWCResult
	setWcResultStatus
	setWcResultScreenshot
	setWcServiceName
	setWcResultValue
	getWcServiceNumber
	getWcServiceName
	getWcSubtestParam
	addWcExtraEvent
	sendExtraEvent