# White-box Testing with StormTest

*Automated testing of internal product interfaces*

Revision Date: 9 Nov 2010

S3 Group Document ID: CHP-10123

Author: John Maguire

## Contents

## 1 Abstract

StormTest™ is a platform for automating testing of set top boxes and other audio/video devices. This paper describes the way in which the standard StormTest Development Center product can be used to automate white box testing. It provides a summary of the way in which this has been done by both S3 Group and other users of the StormTest platform.

## 2 White Box Testing defined

The term “white box testing” means testing which exercises internal interfaces in the product which are not normally exposed to the end user of the device. These interfaces are typically made available for direct testing only as part of a development cycle of a product and are often interfaces which later are used by other elements of the product during overall system integration. The term is used in contrast to “black box” testing which implies testing of a product without any knowledge of any of the internal interfaces of the product.

## 3 StormTest Hardware support for White Box testing

White box testing typically requires there to be some way to communicate from the outside world into the internals of the product. There needs to be some agent or test application software sitting inside the device which can process commands from the outside world, exercise the appropriate internal interface, and return the result to the external test system. To communicate with this internal agent software requires some physical channel for the two-way communication and StormTest supports a number of interfaces to Set Top Boxes that can be used for this purpose.

### 3.1 RS232 (Serial)

Each StormTest system is shipped with the ability to undertake two-way RS232 (Serial) communication independently with each Set Top Box (STB) under test directly from each test script. The example later in this document uses a serial interface to the STB under test.

StormTest APIs are provided to:

- configure the speed and communication parameters for the serial port
- allow a test script to be notified asynchronously whenever a particular pattern is observed on the serial output.
- put in place a dedicated parser function that should be called with a copy of all data being received over the serial port output as a parameter
- allow a test script to send arbitrary strings (commands) over the serial port in to the STB

### 3.2 Ethernet

Each StormTest system is shipped with the capability to connect an Ethernet cable from each STB under test directly to the StormTest system. This is typically used for systems that need to test content delivery services over Ethernet but in certain STBs is used as a control interface to an embedded agent. Since there is such a wide variety of communication protocols that could be used over the Ethernet interface to an embedded agent no further details of its use are provided in this document however many of the principles discussed for use with the serial port interface should be applicable.

### 3.3 Custom Interface

Due to the extendable nature of StormTest, it is possible to use any physical interface to communicate with an embedded agent once the appropriate Python wrapper libraries are created for use by test scripts to communicate with the STB.

## 4 Example White-box test deployment

### 4.1 Manual operation

The following example is taken from a previous use of StormTest for white box testing.

In the project in question, the white-box testing agent implementation on the STB pre-dated the StormTest automation step and had been typically controlled by a person sitting at a computer terminal running a serial terminal program. The embedded agent communicated using the serial port on the STB which was set to a fixed set of communication parameters (38,400 bps, 8 data bits, No parity, 1 Stop bit).

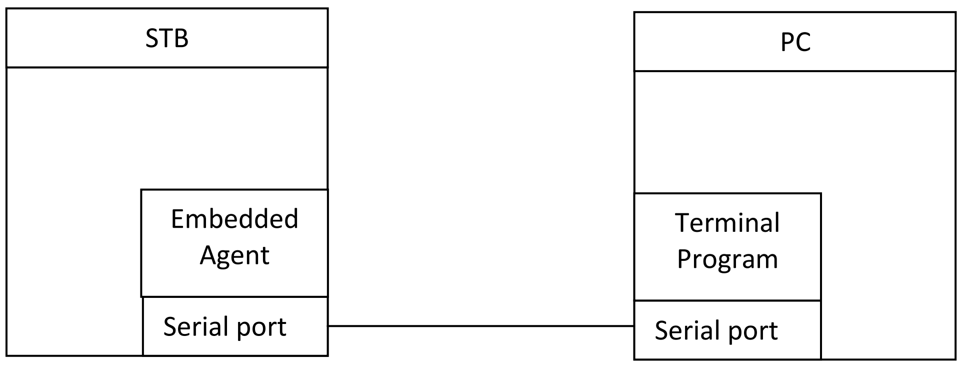

Figure 1 - Manually driven system configuration

Once the PC was connected to the STB via serial and the special test software image on the STB containing the embedded agent was booted, it would present a menu of possible test execution options to the tester via the serial terminal program.

In each case, the options available to the tester were presented as a set of numbered options. Selecting an option required the tester to enter a single numerical digit. In some cases selecting an option led to the printout of another numerical menu list in a hierarchy of menus. Ultimately however a bottom level menu could be reached which were presenting a list of tests and the tester entered a single digit to cause the embedded agent to run the selected test and return a string which contained the test result.

For example, on the very first menu, the user was presented with options similar to:

```
1. Run Operating System tests

2. Run video system tests

3. Run audio system tests

4. Run other driver tests

5. Print software version

Please select an option: |
```

In the case where the tester would enter the digit "1" and press return, they would be presented with a second level menu of Operating System related tests similar to:

```
1. Run miscellaneous tests…

2. Run memory tests…

3. Run task test

4. Run semaphore test

5. Run event test

6. Exit

Please select an option: |
```

In this case entering either 1 or 2 would lead to a further menu of options whereas entering the digit "4" and pressing return would immediately execute the embedded agent implemented semaphore test. Tests in certain cases produced intermediate output to the serial terminal but ultimately, once complete, this test would return a response similar to:

```
Semaphore Test Status: PASS
```

At this point it would reprint the options available at the current level of hierarchy within the test navigation command line interface allowing the user to select another test.

### 4.2 Automated operation

It was decided to automate a significant amount of the white-box testing that was being performed on this project. To that end, a number of STBs running the embedded agent code were placed into a standard StormTest Development Center rack and had their serial port interfaces connected so that they could be communicated with directly from the test scripts.

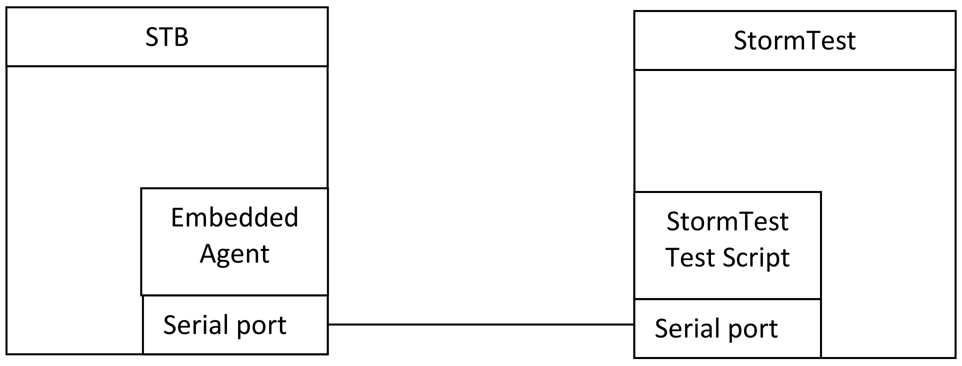

Figure 2 - StormTest driven system configuration

A set of constant command string values were hard-coded into the test script to allow any test to be executed along with the key output result text that needed to be monitored for on the serial output. These hard-coded test execution strings were in effect a concatenation of the individual command line interface (CLI) single digit commands that were required to navigate through the CLI menu structure, delimited by '/n' newline characters.

Setting up the automated system to run a particular set of embedded agent white-box tests was simply a matter of telling it to run a sequence of individual test cases from the set which had previously been hardcoded.

On running each test case, the test script would setup:

- A default parsing function to ensure that all output from the serial port was captured to a log file during test execution.
- A specific handler function to deal with the case where for some reason the software in the STB under test crashed disastrously resulting in an Exception being dumped to the serial port. This was a pretty major event which typically would end the test session.
- A specific handler function to monitor for the return string that was hardcoded as being the expected outcome from running this particular test.

With all of this work done, the hardcoded configuration string required to drive the embedded agent through its CLI navigation tree was written into the serial port of the STB. It would then loop for a predefined maximum amount of time (set to be a maximum of 50 times 10 seconds by default) waiting for one of the handlers to execute and conclude the individual test case execution.

The following fragments of the test script code may help in understanding how this was implemented (Note that it has been stripped down to show the critical elements to ease understanding with many of the menu options and serial observers including associated handlers having been removed):

```
##############################################

#White-box testing using StormTest platform

#Example test case from OSY tests

#Copyright © S3 Group 2010

##############################################

#Test menu ID for Operating System (OSY) tests

OSY_TOP_MENU_ID = "1"

#Hardcoded list of options in the OSY menu

osyMiscTestMenu = OSY_TOP_MENU_ID+"\n0"
osyMemoryTestMenu = OSY_TOP_MENU_ID+"\n1"
osyTaskTest = OSY_TOP_MENU_ID+"\n2"
osySemaphoreTest = OSY_TOP_MENU_ID+"\n3"
osyEventTest = OSY_TOP_MENU_ID+"\n4"

#Hard coded list of lowest level tests and expected test result strings

semaphoreTest = [osySemaphoreTest + '\n', "Semaphore Test Status"]

osPrintConcurrencyTest = [osyMiscTestMenu+"\n0" + '\n',"Concurrency Test Status"]
osPrintDisplayTest     = [osyMiscTestMenu+"\n1" + '\n',"Display Test Status"]
timestampTest          = [osyMiscTestMenu+"\n3" + '\n',"Timestamp Test Status"]

memoryTest0 = [osyMemoryTestMenu+"\n0" + '\n',"Memory Test 0 Status"]
memoryTest1 = [osyMemoryTestMenu+"\n1" + '\n',"Memory Test 1 Status"]
memoryTest2 = [osyMemoryTestMenu+"\n2" + '\n',"Memory Test 2 Status"]
memoryTest3 = [osyMemoryTestMenu+"\n3" + '\n',"Memory Test 3 Status"]

taskTest      = [osyTaskTest + '\n',"Task Test Status"]
semaphoreTest = [osySemaphoreTest + '\n', "Semaphore Test Status"]
eventTest     = [osyEventTest + '\n', "Event Test Status"]

# Example function running just the semaphore white-box test
mytestenv.runTest(TestDefs.OSY.Osy.semaphoreTest)

#Function to handle each test case

def runTest(testargs):

    print "myTestEnv - runTest() START"

    #Global variables
    global g_aTestPassed
    global g_aTestFailed
    global g_currentTest
    global g_resultFound
    global g_stbCrashed
    global g_serialData
    global g_currentTest
    global g_rebootRequired

    g_aTestPassed = False
    g_aTestFailed = False

    g_resultFound.clear()
    g_stbCrashed.clear()

    #Call StormTest API to setup an external serial parser to log all serial data received
    server.SetExtParser( serialDataReceived, int(g_myslots[0]) )

    g_currentTest = testargs[1]

    #Call StormTest API to add a handler to catch case of STB crashing
    server.AddObserve("============= EXCEPTION =================", obsCB_stbCrash)

    #Call StormTest API to add a handler to catch expected test result string appearing
    server.AddObserve(testargs[1], currentTestStatus)
    #Clear buffer used to hold all serial data received during test execution
    g_serialData = " "

    #Send the specific white box test execution request out to the serial port
    print "Sending command SendCommandViaSerial(",testargs[0],"None,0,",int(g_myslots[0]),")"
    server.SendCommandViaSerial( testargs[0], None, 0, int(g_myslots[0]) )

    #Clear timers and flags
    waitTime = 0
    waitTimeElapsed = False

    #Wait until we have a result (or timeout which is handled as result) or the STB crashes
    while( (not g_resultFound.isSet()) and (not g_stbCrashed.isSet())):
        server.WaitSec(10)
        if waitTime > 50:
            g_resultFound.set()
            waitTimeElapsed = True
        waitTime += 1

    #Test execution timed out waiting for the test result string, tell the tester
    if waitTimeElapsed == True :
        print "The test string was not found!!"
    #Or else so long as the STB didn't crash call some other function to handle the results
    elif (not g_stbCrashed.isSet()):
        checkResults()

    #Clear buffer used to hold all serial data received during test execution
    g_serialData = " "

    #Remove the registered handlers
    server.RemoveObserve("============= EXCEPTION =================")
    server.RemoveObserve(testargs[1])

    print "myTestEnv - runTest() END"

    #End of runTest()


########## Callback Functions #############################
#Define handler for STB crash exception being seen on serial output
def obsCB_stbCrash(slotNo, a_key, a_string):
    global g_stbCrashed
    print "myTestEnv - callback function obsCB_stbCrash() START"
    global g_currentTest
    print "obsCB_stbCrash -> DEBUG: The STB in", slotNo, "crashed"

    #Open and append to a crash log file
    crashedTests = open('DALTestResults/testsCrashed.txt', 'a')
    crashedTests.write("#"*50+g_currentTest+"#"*50+"\n\n")
    crashedTests.write(a_string.encode("utf-8"))
    crashedTests.write("#"*50+g_currentTest+"#"*50+"\n\n")
    #Close the crash log file
    crashedTests.close()

    #Set the flag for the main routine loop
    g_stbCrashed.set()
    print "myTestEnv - callback function obsCB_stbCrash() END"

#Define handler for case where expected string indicating completion of test is seen
def currentTestStatus(slotNo, a_key, a_string):
    global g_resultFound
    #Set the flag for the main routine loop to finish
    g_resultFound.set()

#Define handler capturing of all serial data seen coming from STB
def serialDataReceived(slotNo, data):
    global g_serialData

    Append all the data to a global buffer
    g_serialData += data
```

## 5 Conclusion

In this technical note we have provided an introduction to the purpose of white-box testing and how it can be automated using the StormTest automated testing platform. We have provided samples of code to indicate how this automated code might be organised. We have also shown how the StormTest APIs can be used to handle the variety of different responses that might asynchronously be issued by an embedded white-box test agent when used with the serial port.

The S3 Group StormTest support team have worked with many customers around the world to address customer specific automation challenges and would be happy to discuss your needs with you.

StormTest Support can be contacted at:

stormtest-support@s3group.com