Confidential Information

Test Specification

For HbbTV Test Suite Version 9.2

Version 1.0

June 30, 2017

Contents

1 General

1.1 Scope

The scope of this Test Specification is to describe the technical process for verification of HbbTV Devices for conformance with the following HbbTV Specifications:

- ETSI TS 102 796 V1.1.1 (informally known as HbbTV 1.0) + errata
- ETSI TS 102 796 V1.2.1 (informally known as HbbTV 1.5) + errata
- ETSI TS 102 796 V1.3.1 (informally known as HbbTV 2.0)
- ETSI TS 102 796 V1.4.1 (informally known as HbbTV 2.0.1)

This document targets HbbTV Testing Centers and Quality Assurance Departments of HbbTV Licensees.

The HbbTV Test Specification contains five parts:

1) A part that describes the HbbTV Test System and its components.
2) A Test Case creation part that describes the Test Case creation process.
3) HbbTV JavaScript API's and playout definitions used for Test Case implementation
4) Information about the versioning of HbbTV Test specifications.
5) The description of the Test Report format.

The process that is used to define the HbbTV Test Specification, Test Cases and implementation of Test Tools from the HbbTV Specification is depicted in Figure 1.

1) From the HbbTV Specification, Test Cases are defined
2) From the total set of Test Cases the HbbTV Test Specification can be generated
3) From the HbbTV Test Specification the design and implementation of Test Tools will be designed

The grey highlighted elements in Figure 1 are provided by the HbbTV Testing Group.

1.2 Conformance and reference

HbbTV devices shall conform to the applicable parts of the relevant HbbTV Specification.

2 Conditions of publication

2.1 Copyright

2.2 Disclaimer

The information contained herein is believed to be accurate as of the data of publication; however, none of the copyright holders will be liable for any damages, including indirect or consequential from use of the TEST SPECIFICATION for HbbTV or reliance on the accuracy of this document.

2.3 Classification

The information contained in this document is marked confidential and shall be treated as confidential according to the provisions of the agreement through which the document has been obtained.

2.4 Notice

For any further explanation of the contents of this document, or in case of any perceived inconsistency or ambiguity of interpretation, contact:

HbbTV Association

Contact details: Ian Medland (HbbTV Testing Group Chair) testing-list@hbbtv.org

Web Site: http://www.hbbtv.org

E-mail: info@hbbtv.org

3 References

3.1 Normative references

The following referenced documents are required for the application of the present document. For dated references, only the edition cited applies. For non-specific references, the latest edition of the referenced document (including any amendments) applies.

[1] ETSI TS 102 796 “Hybrid Broadcast Broadband TV”; V1.4.1. Including all approved Erratas.

[2] VOID

[3] VOID

[4] ETSI TS 102 809 “Digital Video Broadcasting (DVB); Signalling and carriage of interactive applications and services in hybrid broadcast / broadband environments", V1.2.1

[5] VOID

[6] VOID

[7] VOID

[8] RFC2616, IETF: “Hypertext transport protocol – HTTP 1.1”

[9] VOID

[10] VOID

[11] VOID

[12] VOID

[13] VOID

[14] CI Plus Forum, CI Plus Specification, “Content Security Extensions to the Common Interface”, V1.3 (2011-01).

[15] VOID

[16] VOID

[17] VOID

[18] ETSI EN 300 468 “Specification for Service Information (SI) in DVB systems”, V1.10.1

[19] VOID

[20] VOID

[21] ISO 23009-1 (2012): “Information technology – Dynamic adaptive streaming over HTTP (DASH) – Part 1: Media presentation description and segment formats”

[22] RFC3986, IETF: “Uniform Resource Identifier (URI): Generic Syntax”

[23] RFC4337, IETF: “MIME Type Registration for MPEG-4”

[24] VOID

[25] Document Names for specReference "name" attribute. https://www.hbbtv.org/redmine/projects/hbbtv-ttg/wiki/DocumentNames

[26] Document Names for specReference "name" attribute. https://www.hbbtv.org/redmine/projects/hbbtv-ttg/wiki/AppliesToSpecNames

[27] Test Material Challenging Procedure (TMCP), V2.0

[28] W3C Recommendation (16 January 2014): "Cross-Origin Resource Sharing". Available at: http://www.w3.org/TR/2013/PR-cors-20131205/

[29] Open IPTV Forum Release 1 specification, "CSP Specification V1.2 Errata 1", March 2013. Available from http://www.oipf.tv/specifications

[30] DIAL 2nd Screen protocol specification v1.7 – 19 November 2014

NOTE: Available from http://www.dial-multiscreen.org/dial-protocol-specification

[31] IETF RFC 6455: "The WebSocket protocol"

NOTE: Available at https://tools.ietf.org/html/rfc6455

[32] VOID

[33] ECMAScript Language Specification (Edition 5.1), June 2011

NOTE: Available at http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262%205.1%20edition%20June%202011.pdf

[34] ETSI TS 103 286-2 (V1.1.1): "Companion Screens and Streams; Part 2: Content Identification and Media Synchronisation

3.2 Informative references

[i.1] CEA-2014 revision A, "Web-based Protocol and Framework for Remote User Interface on UPnP™ Networks and the Internet (Web4CE)"

[i.2] ETSI ES 202 130 "Human Factors (HF); User Interfaces; Character repertoires, orderings and assignments to the 12-key telephone keypad (for European languages and other languages used in Europe)", V2.1.2

[i.3] ETSI TS 102 757 "Digital Video Broadcasting (DVB); Content Purchasing API", V1.1.1

[i.4] ETSI TS 101 231 "Television systems; Register of Country and Network Identification (CNI), Video Programming System (VPS) codes and Application codes for Teletext based systems", V1.3.1

[i.5] ISO/IEC/IEEE 9945:2009 Information technology – Portable Operating System Interface (POSIX®) Base Specifications, Issue 7

[i.6] I. Hickson. The WebSocket API. 20 September 2012. W3C Candidate Recommendation.

NOTE: Available at http://www.w3.org/TR/2012/CR-websockets-20120920/

[i.7] Using the DVB CSA algorithm; ETSI. http://www.etsi.org/about/what-we-do/security-algorithms-and-codes/csa-licences

[i.8] Common Scrambling Algorithm; Wikipedia. https://en.wikipedia.org/wiki/Common_Scrambling_Algorithm

[i.9] W3C Candidate Recommendation (11 December 2008): "Web Content Accessibility Guidelines (WCAG) 2.0"

[i.10] JSONP; Wikipedia
https://en.wikipedia.org/wiki/JSONP

[i.11] Wireshark network protocol analyzer
https://www.wireshark.org/

[i.12] Netcat networking utility; Wikipedia
https://en.wikipedia.org/wiki/Netcat

NOTE: Available at http://www.w3.org/TR/2008/REC-WCAG20-20081211/

4 Definitions and abbreviations

4.1 Definitions

For the purposes of the present document, the following terms and definitions apply:

Application data: set of files comprising an application, including HTML, JavaScript, CSS and non-streamed multimedia files

Assertion: Testable statement derived from a conformance requirement that leads to a single test result Broadband: An always-on bi-directional IP connection with sufficient bandwidth for streaming or downloading A/V content

Broadcast: classical uni-directional MPEG-2 transport stream based broadcast such as DVB-T, DVB-S or DVB-C

Broadcast-independent application: Interactive application not related to any broadcast channel or other broadcast data

Broadcast-related application: Interactive application associated with a broadcast television, radio or data channel, or content within such a channel

Broadcast-related autostart application: A broadcast-related application intended to be offered to the end user immediately after changing to the channel or after it is newly signalled on the current channel. These applications are often referred to as "red button" applications in the industry, regardless of how they are actually started by the end user

Conformance requirement: An unambiguous statement in the HbbTV specification, which mandates a specific feature or behaviour of the terminal implementation

Digital teletext application: A broadcast-related application which is intended to replace classical analogue teletext services

HbbTV application: An application conformant to the present document that is intended to be presented on a terminal conformant with the present document

HbbTV test case XML description: The HbbTV XML document to store for a single test case information such as test assertion, test procedure, specification references and history. There exists a defined XSD schema for this document format.

HbbTV Test Specification: Refers to this document.

HbbTV Technical Specification: Refers to [1] with the changes as detailed in [20]

Hybrid terminal: A terminal supporting delivery of A/V content both via broadband and via broadcast

Linear A/V content: Broadcast A/V content intended to be viewed in real time by the user

Non-linear A/V content: A/V content that which does not have to be consumed linearly from beginning to end for example, A/V content streaming on demand

Persistent download[1]**:** The non-real time downloading of an entire content item to the terminal for later playback

Playout: Refers to the modulation and playback of broadcast transport streams over an RF output.

Test Assertion: A high level description of the test purpose, consisting of a testable statement derived from a conformance requirement that leads to a single test result.

[1] Persistent download and streaming are different even where both use the same protocol - HTTP. See clause 10.2.3.2 of the HbbTV specifications, ref [1]

NOTE: Not to be confused with the term “assertion” as commonly used in test frameworks e.g. JUnit.

Test Case: The complete set of documents and assets (assertion, procedure, preconditions, pass criteria and test material) required to verify the derived conformance requirement.

NOTE 1: This definition does not include any test infrastructure (e.g. web server, DVB-Playout…) required to execute the test case.

NOTE 2: For the avoidance of doubt, Test Material must be implemented in a way that it produces deterministic and comparable test results to be stored in the final Test Report of this Test Material. Test Material must adhere to the HbbTV Test Specification as defined by the Testing Group.

Test framework/Test tool/Test harness: The mechanism (automated or manually operated) by which the test cases are executed and results are gathered. This might consist of DVB-Playout, IR blaster, Database- and Webservers.

Test material: All the documents (e.g. HTML, JavaScript, CSS) and additional files (DVB-TS, VoD files, static images, XMLs) needed to execute the test case.

Test Procedure: A high level textual description of the necessary steps (including their expected behaviour) to follow in order to verify the test assertion.

Terminal specific applications: Applications provided by the terminal manufacturer, for example device navigation, set-up or Internet TV portal.

Test Report: A single file containing the results of executing one or more Test Suites in the format defined in section 9. This is equivalent to the “Test Report” referred to in the HbbTV Test Suite License Agreement and HbbTV Full Logo License Agreement.

Test Repository: Online storage container for HbbTV test assertions and Test Cases. Access is governed by the TRAA. Link: https://www.hbbtv.org/pages/about_hbbtv/hbbtv_test_repository.php

Test Suite: This means the collection of test cases developed against a specific version of the HbbTV Specification, including test cases for all possible features.

NOTE 1: The Testing Group specifies and approves which test cases are parts of the HbbTV Test Suite.

NOTE 2: Only Approved HbbTV Test Material shall be a part of the HbbTV Test Suite used for the Authorized Purpose.

NOTE 3: Upon approval of HbbTV, new HbbTV Test Material may be added from time to time at regular time-intervals.

NOTE 4: The HbbTV Test Suite does not include any physical equipment such as a Stream Player/Modulator, IP Server

Test Harness: The Test Harness is a system which orchestrates the selection and execution of Test Cases on the DUT, and the gathering of the Test Case Results for the Test Report.

Standard Test Equipment: The Standard Test Equipment is the collection of all “off the shelf” tools, which are needed to store, serve, generate, and play out the Test Cases on the DUT.

4.2 Abbreviations

For the purposes of the present document, the following abbreviations apply:

AIT	Application Information Table
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CEA	Consumer Electronics Association
CICAM	Common Interface Conditional Access Module
CSS	Cascading Style Sheets

CSS-CII	Interface for Content Identification and other Information [30]
CSS-TS	Interface for Timeline Synchronisation [30]
CSS-WC	Interface for Wall Clock [30]
DAE	Declarative Application Environment
DLNA	Digital Living Network Alliance
DOM	Document Object Model
DRM	Digital Rights Management
DSM-CC	Digital Storage Media – Command and Control
DVB	Digital Video Broadcasting
DUT	Device under Test
EIT	Event Information Table
EIT p/f	EIT present/following
EPG	Electronic Program Guide
GIF	Graphics Interchange Format
HE-AAC	High-Efficiency Advanced Audio Coding
FQDN	Fully Qualified Domain Name
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDTV	Integrated Digital TV
IP	Internet Protocol
JPEG	Joint Photographic Experts Group
JSON	JavaScript Object Notation
MPEG	Motion Picture Experts Group
MSB	Most Significant Bit
OIPF	Open IPTV Forum
PMT	Program Map Table
PNG	Portable Network Graphics
PVR	Personal Video Recorder
RCU	Remote Control Unit
RTP	Real-time Transport Protocol
SSL	Secure Sockets Layer
TLS	Transport Layer Security
TV	Television
UI	User Interface
URL	Uniform Resource Locator
XHTML	Extensible HyperText Markup Language
XML	Extensible Markup Language

4.3 Conventions

MUST	This, or the terms "REQUIRED" or "SHALL", shall mean that the definition is an absolute requirement of the specification.
MUST NOT	This phrase, or the phrase "SHALL NOT", shall mean that the definition is an absolute prohibition of the specification.
SHOULD	This word, or the adjective "RECOMMENDED", mean that there may be valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.
SHOULD NOT	This phrase, or the phrase "NOT RECOMMENDED" mean that there may exist valid reasons in particular circumstances when the particular behaviour is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behaviour described with this label.
MAY	This word, or the adjective "OPTIONAL", mean that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item. An implementation which does not include a particular option MUST be prepared to interoperate with another implementation which does include the option, though perhaps with reduced functionality. In the same manner an implementation which does include a particular option MUST be prepared to interoperate with another implementation which does not include the option (except, of course, for the feature the option provides.)

5 Test system

An HbbTV Test System consists of:

1) A suite of Approved Test Cases, obtained from and maintained by HbbTV Association,

2) A Test Environment used to execute the test cases and record the results of testing in a standard Test Report format. The Environment consists of:

    a. The Device Under Test (DUT)

    b. A Test Harness (TH), which acts as test operation coordinator and manages Test Cases and Execution Results

    c. Standard Test Equipment (STE) required for all Test Environments, as described in 5.2.1

    d. Optional Test Equipment (OTE) which may be used to simplify or allow automation of some Test System tasks, perhaps through proprietary terminal interfaces.

    e. RF and IP connections between the DUT and other elements of the Test Environment

    f. TLS test server used to provide a server-side environment for execution of some TLS related test cases. This is provided on behalf of the HbbTV Association and does not need to be supplied separately.

Figure 1 shows the interconnectivity between these required elements.

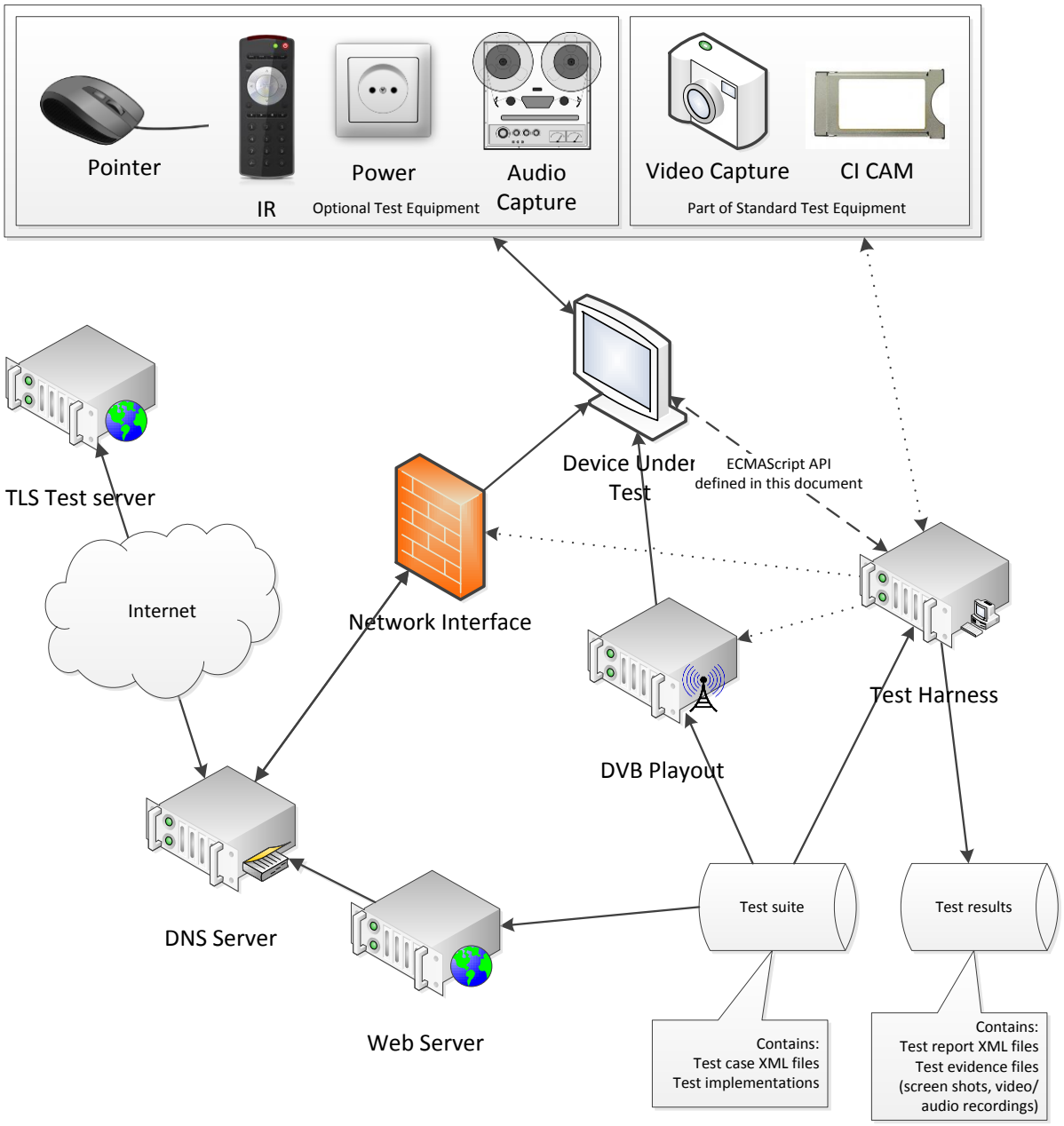


Figure 1: Outline of an example test system overview

5.1 Test Suite

The HbbTV Test Suite consists of a number of Test Cases. Each Test Case is uniquely identified and is intended to verify a specific requirement of the HbbTV specification. The Test Case consists of the following items, mostly stored in a single folder named for the Test Case identifier:

- Test Case XML. This is a multi-purpose document containing:
  - Specification references and applicability
  - Assertion text
  - Procedure and expected results
  - Test Case development history record

- Test implementation files. Where a number of Test Cases use the same files these may be in a shared resource folder.
- Configuration files required to execute the Test Case on a Test Harness.
- License and distribution information.

The Test Cases in a Test Suite can have one of two statuses:

- Approved Test Cases are approved by the HbbTV Testing Group and form part of the requirements for HbbTV compliance
- Additional Test Cases are not approved the HbbTV Testing Group for compliance. They are distributed because they may be of use to developers. Test Cases may be changed from Additional to Approved if they are valid for HbbTV requirements and have been successfully reviewed for approval by the Testing Group.

5.2 Test Environment

The Test Environment for executing the HbbTV Test Suite on a DUT consists of two major components:

- Standard Test Equipment: The Standard Test Equipment is the collection of all "off the shelf" tools, which are needed to store, serve, generate, and play out the Test Cases on the DUT. This includes web servers, and the DVB Playout System. The components of the Standard Test Environment are not included in the Test Suite delivery, but may be provided by commercially available test tools.
- Test Harness: The Test Harness is a system which orchestrates the selection and execution of Test Cases on the DUT, and the gathering of the Test Case Results for the Test Report.

The Test Harness:

1) Uses the information in the HbbTV test case XML description and in the Test Material to initiate the execution of all necessary steps prior to execution (e.g. final test stream generation).
2) initiates the execution of the Test Case on the DUT, causing changes in the environment during execution based on timing points defined in the Test Material and in response to API calls from the Test Case,
3) Collects the test results from the Test Case running on the DUT.

5.2.1 Standard Test Equipment

This chapter further lists the components and requirements for the Standard Test Equipment parts of the HbbTV Test Environment.

The implementation of these components is a proprietary decision. For example, each of these components may or may not be physically installed on a single piece of hardware for each DUT, they may or may not be installed on commonly accessible machine, or they may be virtualized. They may also be already integrated into commercially available tools.

NOTE 1: None of the components defined in this section are delivered with the HbbTV Test Suite.

NOTE 2: There may be more than one DVB Playout device and DUT attached to the Test Harness to allow concurrent testing of multiple devices. There may also be one or more devices (such as a WLAN access point, an Ethernet switch or an IP router) connected between the web server and the DUT. These are implementation decisions which are beyond the scope of this document.

5.2.1.1 Web Server

The Standard Test Equipment shall include an HTTP web server serving the Test Material to the DUT. The web server shall be able to negotiate CORS requests; see [28].

NOTE: There may be test cases in the future that will require the ability to control this operation.

The web server shall accept HTTP requests on port 80. The web server shall be set up in a way that allows reaching the complete test suite under the directory "/_TESTSUITE". For example, the "index.html" file of the test case 00000010 shall be reachable via the URL:
http://hbbtv1.test/_TESTSUITE/TESTS/00000010/index.html

The server shall include a valid installation of the PHP Hypertext Pre-processor to allow dynamic generation of content. Any version of PHP 5.2 or newer may be installed on the server, no special modules are required. Only files with the extension .php shall be processed by the PHP Hypertext Pre-processor.

The Test Harness shall set the PHP Hypertext Pre-processor's "default_mimetype" option to "application/vnd.hbbtv.xhtml+xml; charset=UTF-8".

NOTE 1: This option is usually set in php.ini, but there are often web-server-specific ways to set it too, such as php_admin_value in Apache's mod_php.

NOTE 2: PHP scripts can control the MIME type used for their output, by using a command such as "header('Content-type: video/mp4');". If a PHP script does not specify a MIME type for its output, then PHP uses the MIME type specified by the "default_mimetype" option.Test authors shall ensure that content is served with cache control headers when needed. To set headers, a test will need to use a PHP script to serve the content.

EXAMPLE: when serving a dynamic MPD file for DASH streaming, the PHP script should set the "Cache-Control: no-cache" header.

The file extensions to be served by the web server with their respective MIME types are defined in table 1:

Extension	MIME type
html	application/vnd.hbbtv.xhtml+xml; charset=UTF-8
html5	text/html; charset=UTF-8
txt	text/plain; charset=UTF-8
xml	text/xml; charset=UTF-8
cehtml	application/vnd.hbbtv.xhtml+xml; charset=UTF-8
js	application/x-javascript; charset=UTF-8
css	text/css; charset=UTF-8
aitx	application/vnd.dvb.ait+xml; charset=UTF-8
mp4, m4s	video/mp4
ts	video/mpeg
m4a, mp4a, aac	audio/mp4
mp3	audio/mpeg
bin	application/octet-stream
casd	application/vnd.oipf.contentaccessstreaming+xml; charset=UTF-8
cadd	application/vnd.oipf.contentaccessdownload+xml; charset=UTF-8
mpd	application/dash+xml; charset=UTF-8 [21]
xse	application/vnd.dvb.streamevent+xml; charset=UTF-8
png	image/png
jpg	image/jpeg
gif	image/gif
wav	audio/x-wav
tts	video/vnd.dlna.mpeg-tts
dcf	application/vnd.oma.drm.dcf
ttml	application/ttml+xml; charset=UTF-8
woff	application/font-woff
ttf, cff	application/font-stnt
sub	image/vnd.dvb.subtitle
map	application/octet-stream

Table 1: Web server MIME types

The network connection between the web server and the DUT is controlled by the Test Harness as defined in the playout sets of the Test Cases executed, either directly or via instructions delivered to the operator, to allow disconnection of the network from the DUT, see section 7 for more details.

5.2.1.1.1 Handling fragmented MP4 file requests

The web server shall support special processing for files served from the test suite that end in the extension '.fmp4'.

If the Path section [22] of HTTP requests matches the POSIX basic regular expression[2] [i.5]:

```
^/_TESTSUITE/TESTS/\([a-z0-9][a-z0-9\-]*\(\.[a-z0-9][a-z0-9\-]*\)\{1,\}_[A-Z0-9][A-Z0-9_\-
]*\)/\(.*\.fmp4\)/\(.*\)$
```

And the first capture group:

```
[a-z0-9][a-z0-9\-]*\(\.[a-z0-9][a-z0-9\-]*\)\{1,\}_[A-Z0-9][A-Z0-9_\-]*
```

Matches the ID of a test case in the test suite, and the third capture group:

```
.*\.fmp4
```

Matches the path of a file in that test case's directory (referred to as the container file), then the fourth capture group:

```
.*
```

(Referred to as the segment path) shall be used to extract a subsection of the container file, as follows:

1) The web server shall look in the directory holding the container file for a file named 'seglist.xml'. If this file is not found the server shall return an HTTP response to the client with status 404. This file, if it exists, shall conform to the XSD in SCHEMAS/seglist.xsd. The contents of the seglist.xml file shall not vary during execution of a test. The server shall parse the seglist.xml file, and locate the 'file' element with a 'ref' attribute matching the segment path and a 'video' element matching the container file. If no such element is found the server shall return an HTTP response to the client with status 404. The server shall then read the number of bytes given by the 'size' element from the container file, starting at the offset given by the 'start' element (where an offset of 0 is the first byte in the file.) The HTTP response containing the data shall have a status of 200, and shall have the 'content-type' header [8] set to the value specified by the 'mimetype' element in the XML, or 'video/mp4' if that element is not present [23].

2) If an error occurs reading the file, the server shall return an HTTP response with status 500. If there is not enough data in the file to service the request, the server shall return an HTTP response with status 404.

5.2.1.2 DNS Server

The Standard Test Equipment shall include a DNS (Domain Name System) server that is used by the DUT (either by manual configuration of the DUT or by automatic configuration via DHCP).

The DNS server shall resolve the following domain names:

- hbbtv1.test, hbbtv2.test, hbbtv3.test, a.hbbtv1.test, b.hbbtv1.test, c.hbbtv1.test, shall be resolved to the IP address of the Standard Test Equipment web server.
- server-na.hbbtv1.test shall be resolved to an IP address that is not reachable
- dns-failure.hbbtv1.test shall not be resolved, the DNS server shall reply with NXDOMAIN error.
- External (on Internet) hbbtvtest.org domain and all its sub-domains must be reachable by the DUT and PHP interpreter via both HTTP (port 80) and HTTPS (port 443) protocols
- DUT must be able to access hbbtvtest.org domain via its IP address directly via the HTTP protocol over TLS

---

[2] An equivalent Perl compatible regular expression (PCRE) is;

^/_TESTSUITE/TESTS/([a-z0-9][a-z0-9\-]*(\.[a-z0-9][a-z0-9\-]*)+_[A-Z0-9][A-Z0-9_\-]*)/(.*\.fmp4)/(.*)$

- DUT must be able to access domains specified in Annex C

5.2.1.3 Network Interface

There shall be some mechanism by which the application layer throughput of the network interface connected to the DUT may be "throttled" to a defined maximum bitrate. This shall be controlled by the Test Harness, either directly or via instructions delivered to the operator depending on the implementation of the Test Harness. By default, this restriction shall be 8Mbps, but may be changed by the use of the setNetworkBandwidth() (see 7.4.6) function. At the beginning of each test case, the network throughput shall be returned to the default, regardless of any throttling that was applied during the previous test.

Throughput here is defined as the average number of bits per second passed to the transmitting network interface at the application layer, i.e. the figure excludes all HTTP, TCP/UDP, IP, MAC, LLC, etc. overheads. The restriction only applies to data received by the device under test (i.e. data sent by the test environment). Transmission of data from the device under test shall not be restricted.

NOTE: This functionality may be integrated into a Test Harness; however this is outside of the scope of this document.

5.2.1.4 DVB Playout

The DVB Playout mechanism shall be controlled by the Test Harness, either directly or via instructions delivered to the operator. The transport stream data played out by the DVB Playout mechanism is created from the Playout Set definition of the test that is currently being executed on the DUT.

DVB Playout shall implement TDT/TOT adaptation such that the first TDT/TOT in the base stream is used unaltered the first time that the base stream is played out, and all subsequent TDT/TOTs (including in any repetitions of the base stream) shall be replaced such that the encoded time advances monotonically. For non-integrated playout solutions this means that a playout mechanism which supports TDT/TOT adaptation must be used, and this feature must be activated.

The DVB Playout mechanism shall support up to two DVB multiplexes.

The DVB multiplexes are typically transmitted using multiple modulators (one for each multiplex) and a RF mixer, but may also be transmitted using a single advanced modulator that supports generating multiple signals at once on different RF channels. This specification uses the phrase "modulator channel" to refer to either a modulator outputting a single multiplex, or one channel on an advanced modulator capable of generating multiple signals at once.

When the Test Harness has two modulator channels configured and the Test Harness attempts to execute a test case that specifies two DVB multiplexes, then:

- the first multiplex shall be delivered by the first configured modulator channel;
- the second multiplex shall be delivered by the second configured modulator channel.

When the Test Harness has two modulator channels configured and the Test Harness attempts to execute a test case that specifies a single DVB multiplex, then:

- the multiplex shall be delivered using the first modulator channel configured in the Test Harness;

the second modulator channel shall completely stop outputting a signal.Any two supported modulator settings may be configured in the Test Harness – there is no restriction on, for example, configuring a DVB-S modulator and a DVB-T modulator.

5.2.1.5 Image Capture

When test cases call either of the API calls analyzeScreenPixel() (7.3.3) or analyzeScreenExtended (7.3.4), an image must be taken of the DUT display. Each image taken shall be referenced in the relevant <test case id>.result.xml file in the testStepData element as defined in 9.1.3.4. These images shall be stored within the Test Report as defined in section 9.

The mechanism for this image capture is proprietary and outside of the scope of this specification. It may be automated by the Test Harness, or may be implemented manually by testers using image capture technology, webcam, digital camera or by some other capture mechanism. The format of the image stored is defined in 9.1.3.4.

5.2.1.6 ECMAScript Environment

The standard test equipment shall include an ECMAScript application environment in which tests can be run directly by the Test Harness (referred to as 'harness based tests.') The file containing the code to be run is signalled by the implementation.xml file for the current test. The application environment shall support ECMAScript version 5.1 [33], and shall provide an implementation of the test API for the test to use. It is not required that the ECMAScript environment implement the DAE or DOM.

The file containing the application or the harness based test shall be defined using the harnessTestCode element of the implementation.xml file. If a server side test is defined then the transport stream may include an application, but the application shall not instantiate the test API object or make calls to the test API.

Harness based test applications shall be defined as an ECMAScript application, conforming to the ECMAScript profile defined above. The application shall be contained in a text file encoded in UTF-8. The application environment shall include the test API object as a built-in constructor, which may be instantiated by applications in order to communicate with the test harness. All API calls shall behave as defined for running in the DAE.

When the test harness executes a test with the harnessTestCode element defined, it shall follow the following steps:

1) Commence play out of playoutset 1, if defined

2) Execute application defined in harnessTestCode element

The Test Harness may terminate the test at any time without notifying the test application.

5.2.1.7 CSPG-CI+ Test Module

Several test cases require a custom CI+ CAM ("CSPG-CI+ module") which acts like a standard Content and Service Protection Gateway (CSPG-CI+) module but additionally has an interface with the test harness allowing tests to control certain functions of the module (such as descrambling), and to query the module's status at various points (such as whether the DUT is forwarding content to the CAM for descrambling).This section describes the custom CSPG-CI+ test module functions that need to be implemented (in the CAM) to enable the tests, and other requirements on the test framework.

NOTE: This section focuses on additional functions required of the test CSPG-CI+ module over a normal CSPG-CI+ module.

The CSPG-CI+ module will have to operate with test certificates (not production certificates) because it implements custom test behaviour. This implies that DUTs will need to be configured with test certificates also, during testing.

5.2.1.7.1 Communication between test harness and CSPG-CI+ module

5.2.1.7.1.1 Requirement

Several CI+-related tests require a mechanism for the test harness to:

- configure CAM behaviour (for example put it into a mode where it will respond in a certain way to future DUT/CAM interactions)
- Trigger the CSPG-CI+ CAM to perform an action now (for example send a message informing the DUT of a parental rating change event).

In addition the CSPG-CI+ test CAM needs a mechanism to:

- Indicate to the test harness that, according to its configuration, the host has not behaved as expected/required and hence the test has failed.

5.2.1.7.1.2 Configuration/action required

The following CAM functions need to be controlled by the Test Harness:

1) Do/do not perform CSPG-CI+ discovery on next CAM insertion (i.e. switch between CSPG-CI+ and standard CI+ modes) (accept session establishment to the CI+ SAS resource and OIPF private application)

2) Expect certain DRM messages from the host and, when received, respond to each with a specific configured response

3) Assert that certain DRM messages were indeed received (in a specific order) by the CAM (since some configurable recent time/event)

4) Stop/start descrambling and send a rights_info message to the host

5) Stop/start descrambling and send a parental_control_info message to the host

6) Assert content is/is not currently being received from the host for descrambling

7) Respond to OIPF private message with configured response

8) Set URI and confirm delivery to DUT

5.2.1.7.2 CICAM functionality

The combination of CICAM and harness used with these test APIs shall meet the following requirements, in addition to the requirements implied by the functions specified in 7.4.7.

1) Correct implementation of CI plus [14] and 4.2.3 of [29], except where behaviour needs to be modified to implement other functionality described in this document.

2) Able to decode the CA system as specified in the next Section.

NOTE 1: These requirements apply to the combination of test harness implementation and CICAM, this document does not specify which component within the harness implements the requirements in this document.

NOTE 2: The CICAM may be equipped with either development or production certificates.

5.2.1.7.3 CA System

The CICAM shall support descrambling of content encoded using DVB CSS when the CA_system_ID signalled in the CAT has a value of 4096, 4097 or 4098.

NOTE: The DVB CSA specification is only available to licensees. Information on licensing it is available at [i.7]. Non-licensees may find the Wikipedia page [i.8] helpful.

To descramble a selected service, the CAM shall parse the CA_descriptor from the CAPMT and parse the PID identified by the CA_PID. These PIDs identify the ECMs for the selected service. The format of the ECM (excluding transport stream packet and section headers and stuffing bytes) is shown in the table below:

Syntax	No. of bits	Mnemonic
control_word_indicator	8	bslbf
Reserved	16	
encrypted_control_word	64	bslbf
Reserved	40	

Table 2: CA System ECM format

control_word_indicator – 0x80 indicates that encrypted_control_word is the ‘odd’ key, 0x81 indicates that encrypted_control_word is the ‘even’ key

encrypted_control_word – value to be used as either the odd or even control word, ‘encrypted’ by being XORed with the value 0xAB CD EF 01 02 03 04 05.

5.2.1.7.4 Extensions to Implementation XML schema to support configuration of CICAM state

The harness shall have the capability to present to tests a CICAM in each of the following three states:

1) CI+ CAM with OIPF application conforming to 4.2.3 of [29] profiled as in 11.4.1 [1]

2) CI+ CAM without OIPF application

3) CI CAM (i.e. CAM which does not implement CI+)

A harness may do this either using multiple CAMs or by using a single reconfigurable CAM (or by any other suitable mechanism.)

The test's requirements (if any) for use of a CAM shall be declared using an extension to the existing implementation XML syntax (see below).

If the XML element is not present then the test is assumed not to require the use of a CAM and test cases shall not use the functionality described in this section.

5.2.1.7.5 Implementation XML extension

A 'CAM' element is defined, with an optional 'type' attribute. If the element is absent then the harness may assume that no CAM is required by the test. In this case the behaviour of the JS function calls defined in 7.4.7 is undefined.

The 'type' attribute shall have one of the following values:

- cspgcip (default)
- cip
- ci
- none

Where the meanings of the values are as follows:

- cspgcip – the CICAM shall behave as described in Section 5.2.1.7.2 above.
- cip – The CAM shall respond to SAS_connect_rqst APDUs specifying the OIPF application ID (4.2.3.4.1.1 of [29]) with session_status 0x01 (Connection denied - no associated vendor-specific Card application found)
- ci – The CAM shall be configured such then whenever the CAM is powered up or inserted the CAM behaves as a CI, rather than CI plus, device for the purposes of host shunning behaviour (§10 in [14].) There are multiple mechanisms by which this may be achieved, including removal of the 'ciplus' compatibility indicator from the CIS information string, disabling the CA resource, etc.
- none – The test requires that no CAM is inserted in the terminal at the start of the test. This is distinct from the case where the test is not concerned with the presence or absence of a CAM (in which case the 'CAM' element should be omitted altogether.) See also note below.

An example of the XML syntax would be:

```
<?xml version="1.0" encoding="utf-8"?>
<testImplementation id="com.example_00000000"
                    xmlns="http://www.hbbtv.org/2012/testImplementation">

    <playoutSets>
        <playoutSet id="1" definition="playoutset.xml"/>
    </playoutSets>
    <CAM type="cip" />
</testImplementation>
```

NOTE: If a test case requires a CAM to be inserted after the start of the test then the implementation XML shall specify a required CAM configuration using this syntax, and the test shall then request removal and eventual reinsertion of the CAM by calling the 'manualAction' JS-Function (or equivalent) once test execution has commenced. (If the CAM type is 'none' and CAM insertion is requested then the CAM configuration would be indeterminate.)

5.2.1.8 Companion Screen

Several test cases (mainly the tests of the Launcher Application) require a Companion Screen (CS) to be connected to the test network (i.e. on the same network as the DUT, and so that it uses the DNS server provided by the Test Harness). If a Companion Screen is required to run the test this will be indicated via the presence of a 'companionScreen' element in the test's implementation.xml file. If the element is absent then the harness may assume that no Companion Screen is required by the test.

A 'companionScreen' element is defined, with one mandatory 'type' attribute and an optional 'initiallyConnected' attribute.

The 'type' attribute indicates the type of the Companion Screen required to run the test. It shall have one of the following values:

- any – a Companion Screen compatible with the DUT is connected to the test network. It does not matter if the Companion Screen is implemented on iOS, Android, or some other OS,
- iOS - an iOS Companion Screen compatible with the DUT is connected to the test network, or
- Android - an Android Companion Screen compatible with the DUT is connected to the test network.

The "initiallyConnected" attribute indicates if the Companion Screen should be connected to the DUT at the start of the test. It shall have one of the following values:

- true (default) - the Companion Screen shall be connected to the DUT at the start of the test, or
- false - the Companion Screen shall not be connected to the DUT at the start of the test; instead the test will instruct the tester when to connect the Companion Screen.

An example of the XML syntax for the element would be:

```
<companionScreen type="iOS" initiallyConnected="true" />
```

5.2.1.9 Checking media synchronization on a single device

Some of the test cases require precise measurement of the timing of video flashes and audio tones. This is used when the DUT is playing media that should be synchronized. This is the case where the media playback is on a single device, there is no inter-device synchronization involved.

The timing of these flashes and tones shall be measured in the physical domain i.e. as light and sound. However, if the DUT has an analogue audio output that is intended for use with normal headphones (where the headphones are expected to add no delay to the audio signal), then it is acceptable to use that output to detect the audio tones.

If the DUT does not have an integrated display, then the tester should select a suitable display, following any instructions provided by the DUT manufacturer. (E.g. if the DUT's instructions say that, for optimum performance, it should be used with a HDMI 2.0 compliant display, then the tester should ensure that they choose a HDMI 2.0 compliant display. Or if the DUT's instructions include a list of tested TV models then the tester should choose a TV from that list).

The Test Harness shall include sensors to detect and time light emissions from two different places on the display, and to detect and time audio emissions. These sensors shall be synchronised together and each sensor shall have an accuracy of +/-1ms.

NOTE: There are several ways the test harness might accomplish this, including:

- a dedicated device with light sensors and audio inputs

- a high-speed camera set to 1000Hz and pointed at the TV with a genlocked audio recorder, and then manual or automatic analysis of the recordings
- a person looking at a storage oscilloscope with light sensors and audio inputs connected.

There are 2 different requirements:

1) Confirm that a series of tones and flashes across a 15 second period are synchronised (within a tolerance specified by the test case, typically 10ms or 20ms). In this case, it is acceptable to check exact synchronisation (e.g. using an oscilloscope) for only some of those tones and flashes, including at least one near the start of the 15sec period and one near the end of that 15sec period. The rest of the tones and flashes can be checked by an unaided human. (Rationale: if the first and last flashes are synchronized, then the intermediate flashes are likely to either be synchronised or to be wrong by 0.5sec or more).

2) Confirm that the difference between the start times of 2 flashes is a given number of milliseconds, within a tolerance specified by the test case. This is used where the test case cannot precisely synchronize the flashes but it can precisely measure what the offset between the flashes should be.

5.2.1.10 Checking media synchronization when DUT is inter-device synchronisation master

NOTE 1: HbbTV [1] includes protocols for media synchronization between multiple devices [34]. An HbbTV terminal can be put into "master" mode, where it exposes 3 services:

- CSS-WC - Wall Clock. This is a simple UDP-based protocol that allows the Companion Screen to read a clock that ticks at "wall clock" rate. In so doing, the client (the Companion Screen) can measure and compensate for the network latency. It is a bit like NTP, but much simpler.
- CSS-TS - Timeline Synchronisation. This is a simple Websocket-based protocol that allows a Companion Screen to be told the mapping between the CSS-WC wall clock time and one of the timelines in the media that is currently playing on the HbbTV terminal. A Companion Screen can use this information to play out media that is synchronised to the media shown by the HbbTV terminal, e.g. an audio track in a different language, or a video at a different camera angle.
- CSS-CII - Content Identification and other Information. This is a simple Websocket-based protocol that allows a Companion Screen to be told the media that is currently playing on the HbbTV terminal, the timelines that are available, and the URLs for the CSS-TS and CSS-WC protocols.

NOTE 2: The Test Harness includes a partial implementation of the HbbTV inter-device media synchronisation slave client. The part that is needed is specified in the normative parts of this section and section 7.8.3.

The Test Harness shall include clients for the CSS-WC and CSS-TS services [34]. These clients shall be linked together and linked into the light and audio sensors described in the previous section, so that the DUT's emission of light and audio can be timed against the timeline being reported by the DUT.

The APIs for this are defined in section 7.8.3.

NOTE 3: There are several ways the test harness might accomplish this, including:

- Having a dedicated device with light sensors and audio inputs that synchronises (in some proprietary way) to a CSS-WC and CSS-TS client on a PC. (Note: BBC R&D have an example of this).
- Having a dedicated device with light sensors and audio inputs that also implements the CSS-WC client, and communicates with a CSS-TS client on a PC
- Having a dedicated device with light sensors and audio inputs that also implements the CSS-WC and CSS-TS clients

- Having software implementations of CSS-WC and CSS-TS on a PC, which synchronise with a high speed camera and genlocked audio recorder
- Having software implementations of CSS-WC and CSS-TS on a PC, which flashes the PC screen in a way that is synchronised to the timeline. That flashing is detected by an extra light sensor, which is connected to a storage oscilloscope. The light sensors and audio inputs from the DUT are connected to the same oscilloscope.

NOTE 4: Since the CSS-CII service is based on Websockets and is not timing-critical, it can be tested by having a test application connect to it via Websockets using the API defined in section 7.7. There is no need for explicit harness support for testing CSS-CII. The CSS-TS service is also based on Websockets, and is tested using a combination of Websocket-based testing and the Test Harness's client described above.

5.2.1.11 Checking media synchronization when DUT is inter-device synchronisation slave

NOTE 1: A HbbTV terminal that supports the +SYNC_SLAVE option can be put into "slave" mode, where it connects to another "master" terminal using the 3 protocols discussed in the previous section.

NOTE 2: The Test Harness includes a partial implementation of the HbbTV inter-device media synchronisation master server. The part that is needed is specified in the normative parts of this section and section 7.8.4.

The Test Harness shall include servers for the CSS-WC, CSS-TS and CSS-CII protocols [34], with configurability of those servers provided via the Test API function startFakeSyncMaster(). The reported timeline will just advance at a fixed rate.

The Test Harness will also provide a server accessible via a Websocket connection that can be used to retrieve the current timeline time, at the point the request is received from the DUT.

NOTE 3: This uses a Websocket so that the test can ensure the slow set-up of the TCP/IP connection is done in advance. It is expected that once the Websocket connection is set up, sending a Websocket message from the DUT to the Test Harness will be quick. The reply may be slower due to the behaviour of the DUT's task scheduler, or because the DUT happens to be busy processing another JavaScript event at the time the reply is received, but the speed of the reply doesn't affect the accuracy of the test.

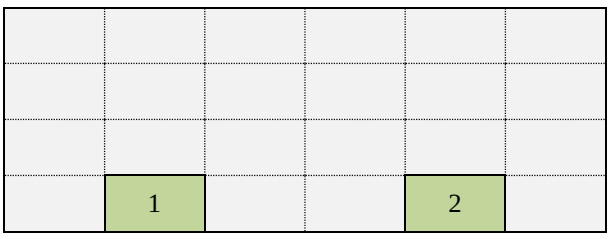
For the tests currently implemented, there is no reason for this to be linked to the light and audio sensors described previously.

The APIs for this are defined in section 7.8.4.

5.2.1.12 Light Sensor positioning for media synchronization tests

As described in section 5.2.1.9, the Test Harness will have sensor(s) that detect flashes of light from 2 different locations. The position of the two detection areas is defined in this section. These detection areas are referred to in this API specification as "light sensor 1" and "light sensor 2". Each of the two defined detection areas may be used for video, subtitles or UI graphics.

The positions of these detection areas are measured on the display that emits light, i.e. after all scaling has been applied. The positions are shown in the following diagram, which shows the whole TV screen (i.e. the entire panel excluding bezel):



(The dashed grid lines are spaced equally, to make it possible to read the position off this diagram).

In numbers, if (0, 0) is the top left of the display and (1, 1) the bottom right, then the co-ordinates of the two defined detection areas are:

- light sensor 1: (1/6, 3/4) to (1/3, 1)
- light sensor 2: (2/3, 3/4) to (5/6, 1)

Test cases using the light sensors must ensure that these detection areas are usually fully black, and a "flash" turns the entire detection area white, then black again. The detection area should be white for approximately 120ms. For measuring synchronization using the analyzeAvSync and analyzeAvNetSync APIs, the time of the "flash" is the mid-point of the period the frame is white. For measuring synchronization with the analyzeStartVideoGraphicsSync API, the time of the "flash" is the instant it changes from black to white.

The Test Harness may choose to monitor any point or area inside the detection area. It does not have to check the entire detection area.

Test cases may display anything they like outside of these detection areas. (E.g.: countdown timer; debug messages; etc).

The Test Harness shall ensure that the display outside the detection area, and ambient or external light sources, do not interfere with its sensors. (For example, by placing the sensors in the middle of the detection area and ensuring the sensitivity is set correctly, or shrouding them with black tape).

For health and safety reasons, test cases using this feature with a single light sensor should not have more than 3 flashes per second, and test cases using this feature with both light sensors at once should not have more than 1 flash per second. Note that some of the APIs defined here impose lower limits. (Rationale: it is desirable to not exceed the photosensitive epilepsy threshold of 3 flashes per second from [i.9]. It is predictable that test cases using two light sensors will sometimes fail due to the content not being synchronised, and in that case the tester will see the two flashes separately, hence the lower limit for such tests. In practise, these limits are not expected to cause any problems for test authors, and the sequence suggested in section 5.2.1.13 is well below these limits).

5.2.1.13 Media requirements for media synchronization tests

The video(s) and/or subtitles must contain a rectangular region that is normally filled with black pixels, but which flashes white at the relevant time(s) (see later). This region is called the “flash area”. This region should be sized and positioned so that, after the HbbTV terminal has applied any applicable scaling, (and, if applicable, the separate display has applied any scaling) it completely covers one of the two detection areas defined in sections 5.2.1.12.

When designing their media and test case, Test Case authors must make allowance for overscan. If the test case is positioning video on a 1280x720 display, then allowing for overscan can be done as follows. First, calculate the detection area positions if there is no overscan:

	Left	Top	Right	Bottom
Screen size	0	0	1280	720
Light Sensor 1	213	540	427	720
Light Sensor 2	853	540	1067	720

Then assume the largest overscan possible (20% horizontal overscan and 10% vertical overscan), and calculate the detection area positions in that case:

	Left	Top	Right	Bottom
Safe area	128	36	1152	684
Light Sensor 1	298	522	470	684
Light Sensor 2	810	522	982	684

Then take the min/max of those two tables to get a flash area which is guaranteed to cover the detection area regardless of the amount of overscan:

	Left	Top	Right	Bottom
Light Sensor 1	213	522	470	720
Light Sensor 2	810	522	1067	720

The audio must be silent, except for short -6dB Full Scale bursts of tone between 1 and 5 kHz. For measuring synchronization, the time of the tone burst is the mid-point of the period the tone burst is audible.

The flashes and tone bursts shall be synchronised.

The duration of each flash and tone should be short, but not so short that it is missed. It is recommended that, for 50Hz terminals, the flash duration is 120ms. This is chosen to be 3 frames of 25fps progressive video, or 3 full frames (6 fields) of interlaced video. It is 6 frames of 50fps progressive video. Choosing a constant duration, instead of a constant number of frames, allows tests to mix 50p and 25p video, and also means that the same audio track can be shared by test cases with different video framerates.

For some tests, there will be a single flash and a single, synchronised tone burst.

For other tests, there will be a repeating pattern of flashes and synchronised tone bursts. When choosing the time between consecutive flashes or consecutive tone bursts, the test case author must consider the analysis criteria documented for the analyzeAvSync() or analyzeAvNetSync() API as appropriate.

NOTE: One pattern that is suitable for these APIs is as follows:

For this pattern, the flashes and synchronised tone bursts should be 120ms in length and start at the following times, where time T is the start of the pattern (which may not be the start of the media):

Flash/tone start time	**Time between flash/tones**
T + 0 sec	
	1 sec
T + 1 sec	
	1 sec
T + 2 sec	
	2 sec
T + 4 sec	
	2 sec
T + 6 sec	
	3 sec
T + 9 sec	
	2 sec

The sequence then repeats with T being 11 seconds larger.

The irregular intervals are chosen so that medium-sized offsets between audio and video can be detected. Seeing 3 consecutive synchronised flashes/tones tells you that the audio and video are synchronised, and that check is guaranteed to detect synchronisation errors of less than 11 seconds. It will detect any synchronisation error that is not a multiple of 11 seconds long, which makes a “false pass” error very unlikely.

5.2.1.14 TLS test server

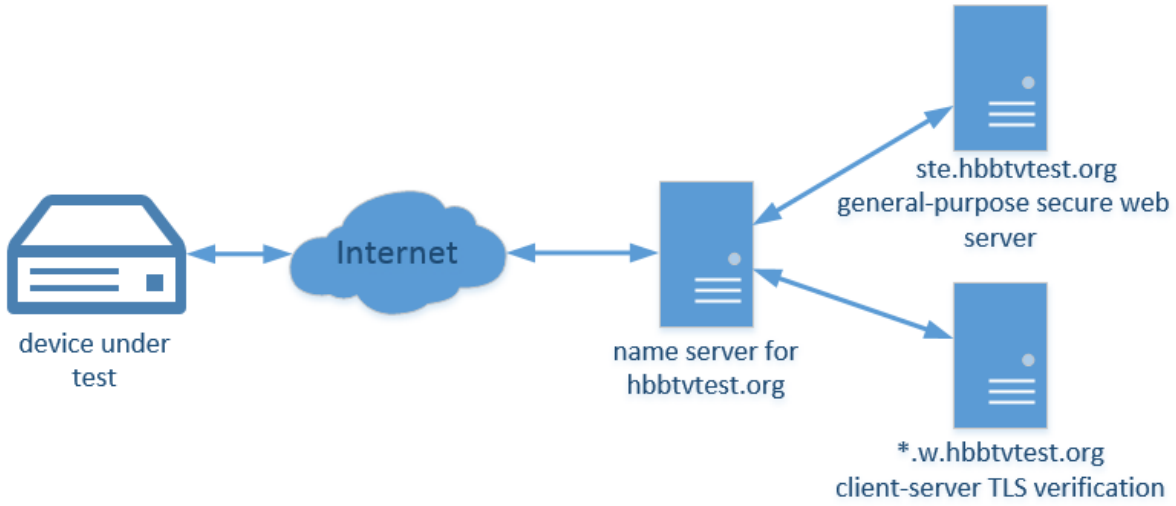
A certain number of TLS test cases require access to low level details of the communication between DUT and remote server when HTTP protocol is used with TLS. The special server used to obtain low level details of the server-client communication is based on a mechanism similar to JSONP [i.10] and its response shall be JavaScript code that has to be evaluated later on.

A separate server is used for test cases that require that the HbbTV application or some other resource be loaded from a https URI, hosted on a server with a valid certificate.

Because of this there are two differently configured virtual servers:

- Server configured for verifying TLS client-server communication
- General-purpose secure web server

Name server for hbbtvtest.org contains multiple NS records, that are sub-domain specific. Requests related to TLS client-server communication will target specific subdomains and the name server will return an IPv4 address of the virtual server configured for verifying TLS client-server communication. For requests targeting the subdomain ste.hbbtvtest.org (where 'ste' stands for standard test environment), name server shall return an IPv4 address of the general-purpose secure web server that can serve static resources over HTTPS. This environment is illustrated on the image bellow.



5.2.1.14.1 Server configured for TLS client-server communication verification

This server runs dedicated web server application based on OpenSSL library that can parse a client's TLS handshake request and verify its content. Server application can also perform an invalid handshake or present an invalid certificate in order to validate the client's behaviour in these kind of scenarios. The domain used for testing purposes is hbbtvtest.org and it provides dedicated subdomains for each use case:

- *.w.hbbtvtest.org - used for verifying parts of the TLS handshake, presents a valid wild-card certificate
- a.hbbtvtest.org - used in for creating an exact match on certificate
- *.host-mismatch.hbbtvtest.org - used for creating a host mismatch
- *.expired.hbbtvtest.org - used for presenting an expired certificate
- *.sha1.hbbtvtest.org - used for presenting a SHA1 certificate
- *.1024.hbbtvtest.org - used for presenting a certificate that has an RSA key with only 1024 bits
- access via IP - used for testing Subject Alternative Name

URLs targeting wildcard subdomain on this server shall be in the following format:

https://LogId-TestId.w.hbbtvtest.org.

LogId part is used as a name of the file in which communication log is stored as a JSON. TestId corresponds to ID of the test case, and server uses it to set up a proper communication context (i.e. to determine which certificate to use or how to handle TLS handshake).

All subdomains used for TLS client-server verification resolve to the same IPv4 address.

Communication log contains low-level details of the communication between server and client, and depending if communication is established or rejected, it is delivered to client in one of two ways:

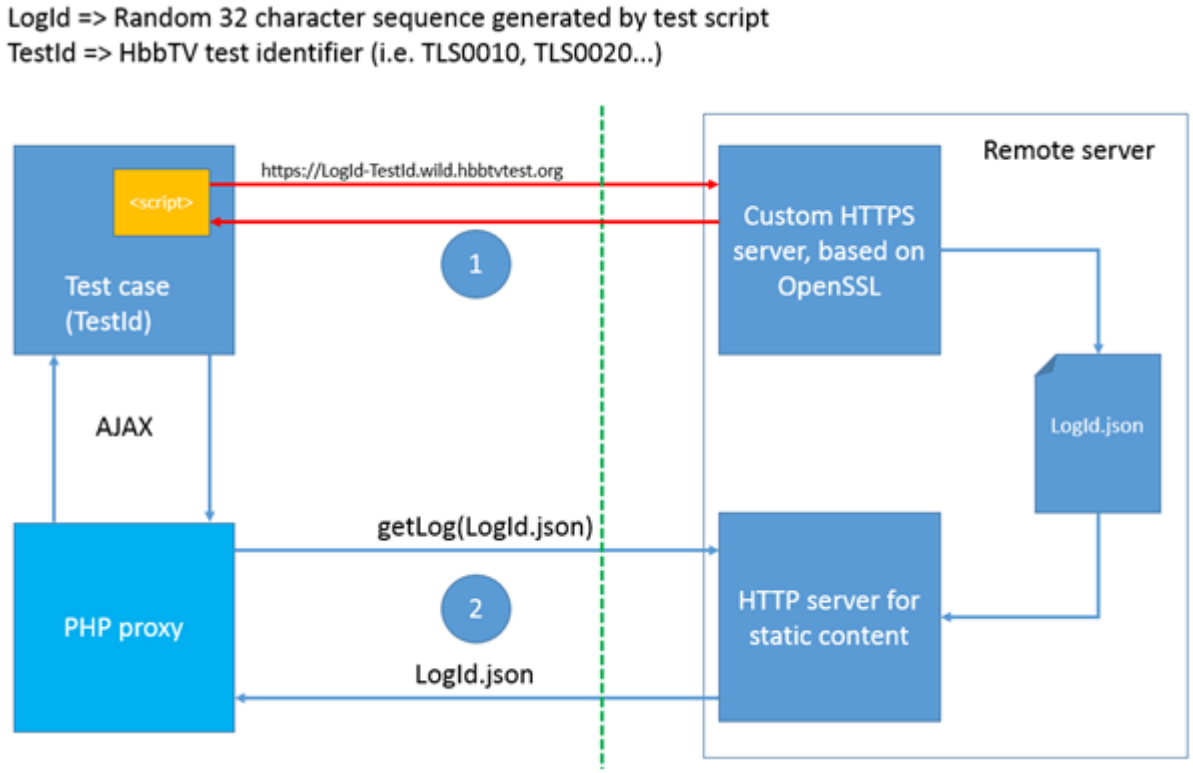
1. If communication is established, script delivers information to test page using mechanism similar to JSONP, where a 'src' attribute is dynamically set to an URL targeting the TLS test server via HTTPS. After the script is successfully loaded and evaluated, test script can use its information to determine the verdict.

2. If communication has failed (script was not loaded), test script uses PHP proxy to get contents of the communication log from remote server via HTTP, using known LogId (generated by test itself, a random 32 character sequence).

Communication logs older than certain time (default will be 60 seconds) are automatically deleted from server. Test case evaluates provided information and determines outcome of the test.

There are test cases which don't require detailed information about the client-server handshake, where test case outcome can be determined by evaluating if script tag was successfully loaded or not.

Block diagram of the system which illustrates both approaches is shown in image provided below.



5.2.1.14.2 General-purpose secure web server

For test cases where an application has to be loaded from an https:URI, or any other resource fetched via https:URI, this web server provides an environment similar to that of a regular Standard Test Equipment HTTP web server, described in section 5.2.1.1. The domain used for testing purposes is ste.hbbtvtest.org.

The web server shall accept incoming HTTP requests on port 80 and HTTPS requests on port 443.

The test suite includes directories matching the pattern:

```
TESTS/<test ID>/on_ste_hbbtvtest_org/v<version number>
```

In this pattern <test ID> is a test case ID and <version number> is an integer version number that starts at 1 for each test case. The web server shall be set up in a way that allows reaching files in those directories under the directory "/_TESTSUITE". For example, the "EME0010.html5" file of the test case org.hbbtv_EME0010 shall be reachable via the following URLs:

```
http://ste.hbbtvtest.org/_TESTSUITE/TESTS/org.hbbtv_EME0010/on_ste_hbbtvtest_org/v1/EME0010.html5

https://ste.hbbtvtest.org/_TESTSUITE/TESTS/org.hbbtv_EME0010/on_ste_hbbtvtest_org/v1/EME0010.html5
```

The first time a Test Case that uses this mechanism is included in the HbbTV Test Suite, the <version number> will be 1. When HbbTV has released a HbbTV Test Suite containing a file matching the pattern "TESTS/<test ID>/on_ste_hbbtvtest_org/v<version number>/", that file will be included unmodified in all future HbbTV Test Suite releases. (This means that the server will continue to work for people running older versions of the HbbTV Test Suite). If HbbTV needs to make a change to such a file, it will be copied to a new "v<version number>" directory using the next higher version number (i.e. 2 for the first time a change is needed, etc).

The server shall include a valid installation of the PHP Hypertext Pre-processor to allow dynamic generation of content. Only files with the extension '.php' shall be processed by the PHP Hypertext Pre-processor.

5.2.1.15 Network analysis tool

For some tests, it is necessary to analyze network activity at the low level in order to check whether expected network issues are happening and if DUT reacts to them as expected.

To make that kind of analysis feasible, test environment must have the Network analysis tool as a part of Standard Test Equipment.

Tests communicate with the Network analysis tool using JS-Function analyzeNetworkLog (see 7.9.1).

Features of the Network analysis tool:

- it has access to the network traffic between DUT and network interface (see Figure 1 in: 5 Test system)
- it is able to record network traffic into the human-readable log file on demand

Some of examples of the off-the-shelf network analysis tools are Wireshark [i.11] and Netcat [i.12].

5.2.2 Test Harness

This chapter further lists the components and requirements for the Test Harness parts of the HbbTV Test Environment.

The implementation of these components is a proprietary decision. For example, each of these components may or may not be physically installed on a single piece of hardware for each DUT, they may or may not be installed on commonly accessible machine, or they may be virtualized. They may also be already integrated into commercially available tools.

5.2.2.1 Test Harness Requirements

The Test Harness is a system which orchestrates the selection and execution of Test Cases on the DUT, and the gathering of the Test Results for the Test Report.

The Test Harness shall implement a mechanism for generation of broadcast transport streams specified by the HbbTV Playout Set as defined in section 7.3.8. This mechanism may be offline (pre-generation of transport streams) or real-time (multiplexing on-the-fly). This is an implementation decision, and is outside the scope of this specification.

The Test Harness shall implement the Test API as defined in section 7 for communication with test applications.

The Test Harness shall store all reportStepResult, analyze[] (e.g. such as analyzeScreenExtended, analyzeScreenPixel etc.) and endTest calls received from the test application(s) during execution. The Test Harness shall use these calls to determine the result of the test case according to the requirements set out in section 6.4, and shall store all of this information in the result xml format defined in section 9.

5.2.2.1.1 OpenCaster Glue Code

The HbbTV Test Suite includes a python script known as the 'OpenCaster Glue Code' which can be used for pre-generation of the transport streams required to execute the test suite. OpenCaster is based on HbbTV

specification v1.1.1 and there are some cases where it is not able to produce streams compatible with the requirements of this specification. In particular:

- Where streams are required to be synchronised with the HTTP server time
- Where streams require the application_recording descriptor to be present.

The OpenCaster Glue Code is a python script released under the Creative Commons license CC BY-SA. It's based on the free toolset called OpenCaster from the company Avalpa (see references below). It reads the information from the Test Case definition XML files of the Test Material and creates a transport stream which later can be played out by the Test Harness and the playout equipment used.

OpenCaster is a free and open source MPEG2 transport stream data generator and packet manipulator developed by Avalpa (www.avalpa.com). It can generate the different tables and convert the table section data to transport streams, filter out PIDs from already multiplexed streams and create object carousels from folders etc.

OpenCaster runs on common Linux© distributions and can be downloaded together with its manual from the Avalpa web page under the Technologies tab.

OpenCaster Glue Code files are located in the Test Suite delivery at TOOLS/OpenCasterGlueCode.

Setup instructions can be found in the TOOLS/OpenCasterGlueCode directory.

5.2.3 Base Test Stream

This section describes the structure of the Transport Stream used by all tests as a basis. For further information on implementation and operation of test cases see 7.5.1.

The base test stream shall be present in the test suite at the path RES/BROADCAST/TS/generic-HbbTV-Teststream.ts.5.2.3.1 Elementary stream structure

The base stream total bitrate is 5,000,000 bps. This is the bitrate before any modification; the rate can be set to a specific value in the playout set XML.

Table 3 below shows the elementary stream allocations in the Transport Stream. Some PID allocations are referenced in the PAT/PMT only – there is no content with these PIDs contained in the stream as distributed.

Table 3: Base test stream PID allocations

PID	Contents
0	PAT
16	NIT
17	SDT
18	EIT (contains both actual present/following and actual schedule EIT tables)
20	TDT/TOT
100	PMT for service 10
101	Video
102	Audio
200	PMT for service 11
201	PMT for service 11 (DSM-CC signalled)
205	AIT for service 11 (see note)
300	PMT for service 12
305	AIT for service 12 (see note)
400	PMT for service 13
405	AIT for service 13 (see note)
500	PMT for service 14
505	AIT for service 14 (see note)
NOTE: PID values marked as AIT are populated in the distributed stream and are referenced in the included PAT/PMT. These shall be used as the insertion points of additional data required by the test implementation (e.g. the AIT for the test case).	

Any of the SI tables above which contain a version number shall use the version number of 1. Streams used by the test harness or by test cases shall not use the version number of 1 unless the SI table is an exact match to the SI table in the base stream. This will ensure that cached SI is not used and tests run as expected.

A test harness may play a stream in between test case runs to reset any cached SI from a test case. SI tables used by this stream which differ from the base stream as defined here shall use a version number of 0. Test cases shall therefore also avoid using a version number of 0 for any SI tables.

This Transport stream has original network ID 99, network ID 99, and transport stream ID 1.

5.2.3.2 Service configuration

The following chart shows the structure of the services in the Transport Stream.

PAT/SDT signalled services			
Service 10	Name	ATE Test10	
	PMT PID	100	
	AIT PID	None	
	Video PID	101	
	Audio PID	102	
	Triplet	63.1.a	
	EIT present	Name	ATE Test10 present
		Description	Present event for service ATE Test10
		Status	0x4 (running)
		Start time	0xd96c112000
		Duration	0x1000
		Language code	eng
	EIT following	Name	ATE Test10 following
		Description	Following event for service ATE Test10
		Status	0x1 (following)
		Start time	0xd96c122000
		Duration	0x1000
		Language code	eng
	EIT schedule	(Table present but empty)	
Service 11	Name	ATE Test11	
	PMT PID	200	
	AIT PID	205	
	Video PID	101	
	Audio PID	102	
	Triplet	63.1.b	
	EIT present	Name	ATE Test11 present
		Description	Present event for service ATE Test11
		Status	0x4 (running)
		Start time	0xd96c112000
		Duration	0x1000
		Language code	eng
	EIT following	Name	ATE Test11 following
		Description	Following event for service ATE Test11
		Status	0x1 (following)
		Start time	0xd96c122000
		Duration	0x1000
		Language code	eng
	EIT schedule	(Table present but empty)	
Service 12	Name	ATE Test12	
	PMT PID	300	
	AIT PID	305	
	Video PID	101	
	Audio PID	102	
	Triplet	63.1.c	
	EIT present	Name	ATE Test12 present
		Description	Present event for service ATE Test12
		Status	0x4 (running)
		Start time	0xd96c112000
		Duration	0x1000
		Language code	eng
	EIT following	Name	ATE Test12 following
		Description	Following event for service ATE Test12
		Status	0x1 (following)
		Start time	0xd96c122000
		Duration	0x1000
		Language code	eng
	EIT schedule	(Table present but empty)	

Service 13	Name	ATE Test13	
	PMT PID	400	
	AIT PID	405	
	Video PID	101	
	Audio PID	102	
	Triplet	63.1.d	
	EIT present	Name	ATE Test13 present
		Description	Present event for service ATE Test13
		Status	0x4 (running)
		Start time	0xd96c112000
		Duration	0x1000
		Language code	eng
	EIT following	Name	ATE Test13 following
		Description	Following event for service ATE Test13
		Status	0x1 (following)
		Start time	0xd96c122000
		Duration	0x1000
		Language code	eng
	EIT schedule	(Table present but empty)	
Service 14	Name	ATE Test14	
	PMT PID	500	
	AIT PID	505	
	Video PID	None (Radio)	
	Audio PID	102	
	Triplet	63.1.e	
	EIT present	Name	ATE Test14 present
		Description	Present event for service ATE Test14
		Status	0x4 (running)
		Start time	0xd96c112000
		Duration	0x1000
		Language code	eng
	EIT following	Name	ATE Test14 following
		Description	Following event for service ATE Test14
		Status	0x1 (following)
		Start time	0xd96c122000
		Duration	0x1000
		Language code	eng
	EIT schedule	(Table present but empty)	
Non-signalled services			
Service 11	Name	ATE Test11	
	PMT PID	201	
	AIT PID	205	
	Video PID	101	
	Audio PID	102	
	DSM-CC PID	206	
	Component/Association Tag	200	
	Carousel id	1	
	EIT present	Name	ATE Test10 present
		Description	Present event for service ATE Test10
		Status	0x4 (running)
		Start time	0xd96c112000
		Duration	0x1000
		Language code	eng
	EIT following	Name	ATE Test10 following
		Description	Following event for service ATE Test10
		Status	0x1 (following)
		Start time	0xd96c122000

		Duration	0x1000
		Language code	eng
	EIT schedule	(Table present but empty)	

5.2.4 Secondary Base Test Stream

This section describes the structure of the secondary Transport Stream used by all tests as a basis where two transport streams are required. For further information on implementation and operation of test cases see 7.5.1.

The secondary base test stream shall be present in the test suite at the path RES/BROADCAST/TS/generic-HbbTV-Teststream_b.ts.

5.2.4.1 Elementary stream structure

The base stream total bitrate is 5,000,000 bps. This is the bitrate before any modification; the rate can be set to a specific value in the playout set XML.

Table 4 below shows the elementary stream allocations in the Transport Stream. Some PID allocations are referenced in the PAT/PMT only – there is no content with these PIDs contained in the stream as distributed.

Table 4: Secondary Base test stream PID allocations

PID	Contents
0	PAT
16	NIT
17	SDT
18	EIT (contains both actual present/following and actual schedule EIT tables)
20	TDT/TOT
100	PMT for service 15
101	Video
102	Audio
105	AIT for service 15(see note)
200	PMT for service 16
205	AIT for service 16 (see note)
300	PMT for service 17
305	AIT for service 17 (see note)
400	PMT for service 18
405	AIT for service 18 (see note)
500	PMT for service 19
505	AIT for service 19 (see note)
NOTE: PID values marked as AIT are populated in the distributed stream and are referenced in the included PAT/PMT. These shall be used as the insertion points of additional data required by the test implementation (e.g. the AIT for the test case).	

Any of the SI tables above which contain a version number shall use the version number of 1. Streams used by the test harness or by test cases shall not use the version number of 1 unless the SI table is an exact match to the SI table in the base stream. This will ensure that cached SI is not used and tests run as expected.

A test harness may play a stream in between test case runs to reset any cached SI from a test case. SI tables used by this stream which differ from the base stream as defined here shall use a version number of 0. Test cases shall therefore also avoid using a version number of 0 for any SI tables.

This Transport stream has original network ID 99, network ID 65281, and transport stream ID 2.

5.2.4.2 Service configuration

The following chart shows the structure of the services in the Transport Stream.

PAT/SDT signalled services			
Service 15	Name	ATE Test15	
	PMT PID	100	
	AIT PID	105	
	Video PID	101	
	Audio PID	102	
	Triplet	63.2.f	
	EIT present	Name	ATE Test15 present
		Description	Present event for service ATE Test15
		Status	0x4 (running)
		Start time	0xd96c112000
		Duration	0x1000
		Language code	eng
	EIT following	Name	ATE Test15 following
		Description	Following event for service ATE Test15
		Status	0x1 (following)
		Start time	0xd96c122000
		Duration	0x1000
		Language code	eng
	EIT schedule	(Table present but empty)	
Service 16	Name	ATE Test16	
	PMT PID	200	
	AIT PID	205	
	Video PID	101	
	Audio PID	102	
	Triplet	63.2.10	
	EIT present	Name	ATE Test16 present
		Description	Present event for service ATE Test16
		Status	0x4 (running)
		Start time	0xd96c112000
		Duration	0x1000
		Language code	eng
	EIT following	Name	ATE Test16 following
		Description	Following event for service ATE Test16
		Status	0x1 (following)
		Start time	0xd96c122000
		Duration	0x1000
		Language code	eng
	EIT schedule	(Table present but empty)	
Service 17	Name	ATE Test17	
	PMT PID	300	
	AIT PID	305	
	Video PID	101	
	Audio PID	102	
	Triplet	63.2.11	
	EIT present	Name	ATE Test17 present
		Description	Present event for service ATE Test17
		Status	0x4 (running)
		Start time	0xd96c112000
		Duration	0x1000
		Language code	eng
	EIT following	Name	ATE Test17 following
		Description	Following event for service

			ATE Test17
		Status	0x1 (following)
		Start time	0xd96c122000
		Duration	0x1000
		Language code	eng
	EIT schedule	(Table present but empty)	
Service 18	Name	ATE Test18	
	PMT PID	400	
	AIT PID	405	
	Video PID	101	
	Audio PID	102	
	Triplet	63.2.12	
	EIT present	Name	ATE Test18 present
		Description	Present event for service ATE Test18
		Status	0x4 (running)
		Start time	0xd96c112000
		Duration	0x1000
		Language code	eng
	EIT following	Name	ATE Test18 following
		Description	Following event for service ATE Test18
		Status	0x1 (following)
		Start time	0xd96c122000
		Duration	0x1000
		Language code	eng
	EIT schedule	(Table present but empty)	
Service 19	Name	ATE Test19	
	PMT PID	500	
	AIT PID	505	
	Video PID	None (Radio)	
	Audio PID	102	
	Triplet	63.2.13	
	EIT present	Name	ATE Test19 present
		Description	Present event for service ATE Test19
		Status	0x4 (running)
		Start time	0xd96c112000
		Duration	0x1000
		Language code	eng
	EIT following	Name	ATE Test19 following
		Description	Following event for service ATE Test19
		Status	0x1 (following)
		Start time	0xd96c122000
		Duration	0x1000
		Language code	eng
	EIT schedule	(Table present but empty)	

6 Test Case specification and creation process

6.1 Test Case creation process

The HbbTV Test Cases are based on the optional and mandatory requirements as defined in the HbbTV Technical Specification. Test Cases are proposed and managed by the HbbTV Test Group.

The process for defining, implementing and accepting HbbTV test cases consists of the steps as depicted in Figure 2.

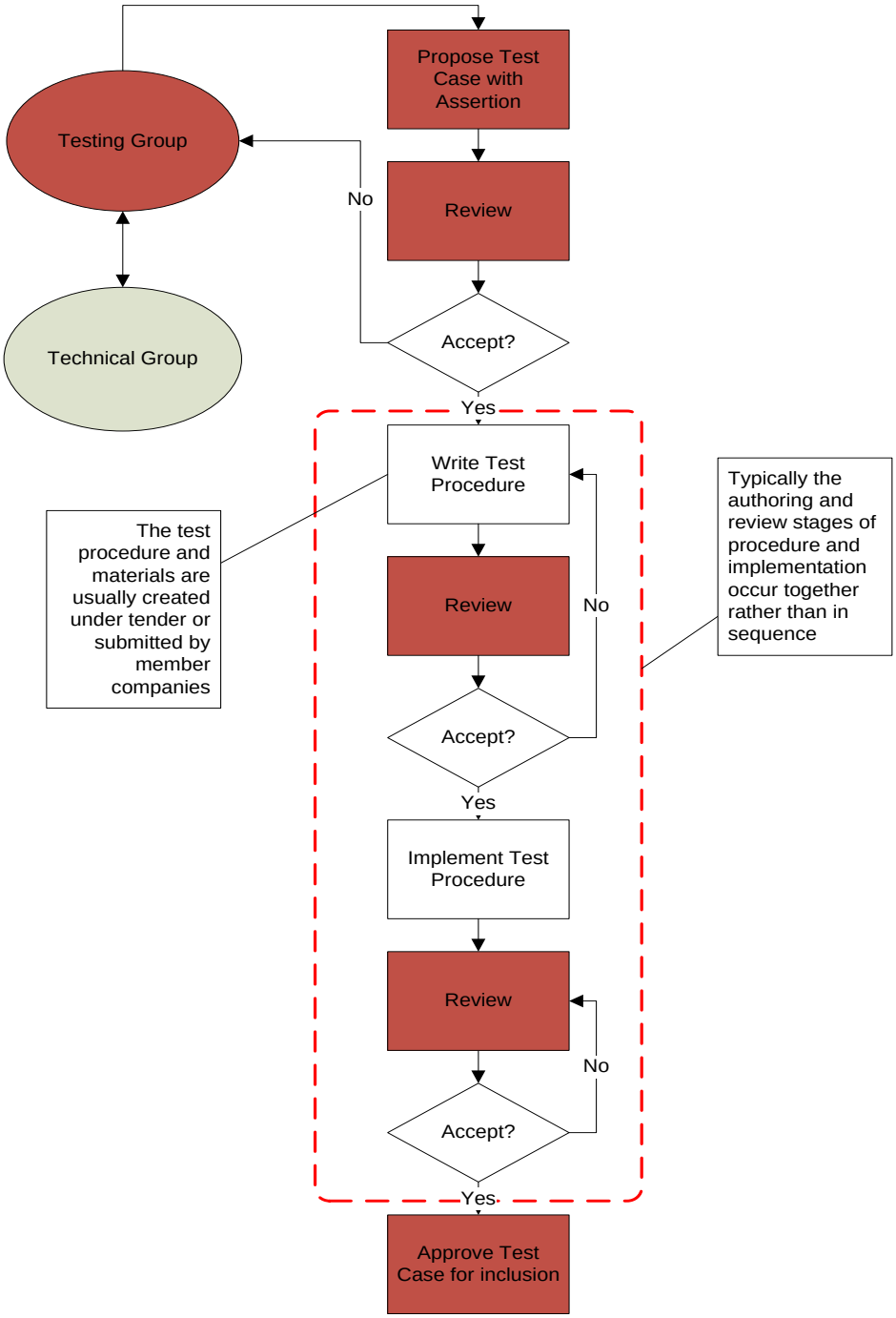


Figure 2: Test creation work flow

6.1.1 Details of the process for creating Test Cases

The detailed process steps for the acceptance of created HbbTV test material is described in the HbbTV Test Material Approval Procedure document which is part of the Test Suite package stored in the Documents folder.

6.2 Test Case section generation and handling

To enable the test process to be handled in a flexible and timely manner the Test Case part of the Test Specification is generated by applying following method:

- Each relevant specification item has been translated into one or more Test Cases.
- Each Test Case is described by a defined set of attributes as listed in section 6.3.
- The Test Case attributes shall be stored in the corresponding XML file which is validated by Schema SCHEMAS/testCase.xsd

NOTE: The use of XML for Test Case definition enables automated processing capabilities. I.e. to use scripting that can generate overviews of existing Test Cases, apply filtering, and allow for flexible generation of various output file formats.

6.3 Test Case Template

Each Test Case consists of a list of attributes, as described below.

6.3.1 General Attributes

The General Attributes uniquely identify the Test Case.

6.3.1.1 Test Case ID

The Test Case ID is a string that uniquely identifies the test case. It contains two parts, a “namespace” and a “Local ID”, separated by an underscore.

For official HbbTV tests, the Test Case IDs will usually be allocated by the testing group. In this case, the namespace shall be “org.hbbtv”. E.g. “org.hbbtv_0000123F”. The testing group must ensure those tests IDs are unique.

It is important that every test has a different Test Case ID. If another organization wants to generate Test Case IDs for its own tests, then it must not use the “org.hbbtv” namespace. Instead, it must take a domain name it controls, reverse it, and use that for the namespace part of the Test Case ID. In this case, the Local ID can be anything permitted by the schema. Organizations should have some internal procedure to allocate Local IDs so that they don’t generate duplicate Test Case IDs.

For example, a company that controls the “example.com” domain could use Test Case IDs like “com.example_FOO”, or “com.example_BAR_BAZ_9876-42”

(To be clear: The domain name used in Test Case IDs must be a real domain name, and must be registered on the Internet in the usual way, using the normal ICANN roots. There is no need for there to be a website there).

See 8.2 for further details on how the Test Case ID is used.

6.3.1.2 Test Case Version

The Test Case Version specifies a specific version of the Test Case and has the following format: <integer> version. See 8.2 for further details on how the Test Case version is used.

6.3.1.3 Origin Information

The Origin Information lists the organizations that own IPR in this Test Case. At least one contributor needs to be specified for each Test Case. However, more than one contributor can be listed. For each contributor, the following attributes exist:

6.3.1.3.1 Part

This is the part of the test case that this organization contributed to. It can be "assertion", "procedure" or "implementation". If the same organization contributes two or three of these parts, they should add a contributor tag for each part.

6.3.1.3.2 Company

The author's company name (mandatory). If the author is an individual who is not associated with a company, this shall be the author's full name.

6.3.1.3.3 Contact Email address

A contact email address for the contributor (mandatory). This should be an address that will be active for many years. It will typically be a generic contact address, not a named individual. This may be used to contact the contributor with questions about licensing, as well as technical questions.

6.3.1.3.4 License information

The license that this contributor uses for the specified part of this test. Valid values are:

- "TMPA Commercial"
- "TMPA CC BY-NC-ND"
- "HbbTV Commercial"

Tests created by other groups may use different values here.

6.3.1.4 Title

A short title to identify this specific Test Case (mandatory).

6.3.1.5 Description

A longer description of what the test does if the title is not sufficient (optional). The format of the Test Case Description is a text field (no limit). Whitespace is not significant.

6.3.2 References

6.3.2.1 Test Applicability

The test specifies which specifications it applies to. Official HbbTV tests shall specify this element and shall use a name of "HBBTV" (case sensitive) and a version of "1.1.1", "1.2.1", "1.3.1" or "1.4.1". Tests that are valid for more than one version of HbbTV shall include tags for all applicable versions. Tests may include additional tags to indicate non-HbbTV specifications that are tested (e.g. OIPF). The master list of specification names is kept on the Wiki page "AppliesToSpecNames" [26]. For each spec element in the <appliesTo> tag, the test case shall:

- be conformant to that specification, and
- test some feature of that specification.

For example, a test which tests an OIPF feature which is also required for both HbbTV 1.1.1 and 1.2.1 would use the following appliesTo element:

```
<appliesTo>
    <spec name="HBBTV" version="1.1.1"/>
    <spec name="HBBTV" version="1.2.1"/>
    <spec name="OIPF" version="1.2"/>
</appliesTo>
```

It is not necessary to include a spec element for every potential regime that could reference HbbTV to use this test. For instance, a country-specific testing regime may require support for HbbTV and seek to use a particular test case - it is not required to include a spec element for that regime.

Every test case shall have an appliesTo element with at least one spec element. It must include a spec element for at least one HbbTV version. It may also include spec elements for other specifications as listed on the "AppliesToSpecNames" Wiki page [26].

6.3.2.2 Specification References

References to the different specification sections or versions. For each version of the HbbTV specification, there is a list of sections covered by this Test Case (top-level). Each top level entry includes references to one or more specification sections (HbbTV sections and optionally external specification sections, e.g. OIPF DAE). Each specification section has the following attributes.

6.3.2.3 Document Name

A short identifier for the document that contains the specification section (e.g. "HBBTV" for the HbbTV specification). While the schema allows any value, HbbTV tests must use the values defined in the wiki page [25]. If you need to reference a spec that isn't listed there, add it to that Wiki page.

6.3.2.4 Chapter

The chapter number within the specified document (a dot separated list of integers or characters without spaces, e.g. 9.3.1)

6.3.2.5 Specification Text

The specific text from the referenced specification section tested by this Test Case (optional). The format of the specification text is a text field (no limit). Whitespace is not significant.

6.3.2.6 Assertion Text

Describes what is tested in this test case (assertion). The format of the Assertion value format shall be a text field (no limit). Whitespace is not significant.

For HbbTV and OIPF tests, there shall be at most one assertion. (Other testing groups that reuse the HbbTV Test Case XML format may relax this limit).

6.3.2.7 Test Object

The textual description of the object under test (specification part or API part). The clause is optional. Where it is included the following values are possible:

- HbbTVClient/AVControlObject
- HbbTVClient/VideoBroadcastObject
- HbbTVClient/ApplicationLifecycle
- HbbTVClient/UserInput
- HbbTVClient/Signalling
- HbbTVClient/DSM-CC
- HbbTVClient/ApplicationManager
- HbbTVClient/Graphics
- HbbTVClient/Security
- HbbTVClient/Browser

If none of the above are suitable then the clause is omitted.

6.3.3 Preconditions

Lists preconditions on the DUT before this test can be run.

6.3.3.1 Required Terminal Options

The terminal options required on the DUT to run this test (if empty, this test is mandatory for all devices). It contains a combined list of option strings as defined in [1], section 10.2.4. The format of the Required Terminal Options value is a text field without spaces.

Available options are +DL for file download functionality, +DRM for DRM functionality, +PVR for PVR functionality, +SYNC_SLAVE for slave operation in inter-device synchronisation, +IPH for "IP delivery Host player mode" as defined in the DVB Extensions to CI Plus ETSI TS 103 205, +IPC for "IP delivery CICAM player mode" as defined in the DVB Extensions to CI Plus ETSI TS 103 205, +AFS for Support for the CICAM Auxiliary File System as defined in the DVB Extensions to CI Plus ETSI TS 103 205. Multiple requirements are concatenated to a single string without spaces in between. They must be listed in alphabetical order. Example: +DL+PVR

You can also use "-" prefixes to indicate the test should only run on devices that do not support the feature. For example, a test with required terminal options set to "-PVR" will not be run on PVRs, but will be run on terminals without PVR functionality.

6.3.3.2 Optional Features

Terminal features which are required on the DUT, in addition to required terminal options, to run this test. Available features are described in table 4.

Table 4: List of optional feature strings

Feature	Description
+2DECODER	Terminal has at least two independent A/V decoders
+CHANNEL_CHANGE_BY_NUMBER	Terminal supports channel changing using number keys from remote control
+CHANNEL_CHANGE_BY_P_PLUS	Terminal supports channel changing by P+/P- or equivalent
+CHANNEL_DELIVERY_BY_DASH	Terminal supports DASH delivery of TV channels
+CI_PLUS	Terminal supports CI Plus
+CICAM_PLAYER_MODE	Terminal supports CI Plus 1.4 CICAM player mode
+CS_APP_LAUNCH	Terminal supports 'launching a CS application from an HbbTV application'
+DASHIF	Terminal supports DASH-IF interoperability guidelines
+DEVICE_ID_DISABLED	User has disabled availability of device ID
+DRM_IN_CICAM	Terminal supports DRM in CI Plus CAM
+DTS	Terminal supports DTSE audio
+DTT	Terminal has a DTT receiver
+DVB_C	Terminal supports the DVB-C delivery mechanism
+DVB_C2	Terminal supports the DVB-C2 delivery mechanism
+DVBMITM	Terminal supports the protection mechanism defined in clause 9 of TS 102 809 V1.3.1
+DVB_PARENTAL_RATING_DESCRIPTOR	The DVB-SI parental rating descriptor is used in the broadcast
+DVB_S	Terminal supports the DVB-S delivery mechanism
+DVB_S2	Terminal supports the DVB-S2 delivery mechanism
+DVB_T	Terminal supports the DVB-T delivery mechanism
+DVB_T2	Terminal supports the DVB-T2 delivery mechanism
+DVBSUB	Terminal supports DVB subtitles in the broadcast and hence also for MPEG-2 TS received by broadband
+EAC3	Terminal supports E-AC3. This can be decoded and sent over analogue output(s), or sent over a digital output (possibly transcoded) or output over internal loudspeakers
+EBUSUB	Terminal supports EBU teletext in the broadcast and

	hence also for MPEG-2 TS received by broadband
+EBUTTD_GT_1_0	Terminal supports a version of EBU-TT-D greater than 1.0
+FDP_RECOVERY	Terminal supports use of IP to recover extra FDP segments
+GRAPHICS_01	Terminal supports graphics performance to level 1
+GRAPHICS_02	Terminal supports graphics performance to level 2
+HDD	Terminal implements a mass storage device
+HEVC_HD_10	Terminal supports HEVC HD at 10 bits per channel
+HEVC_HD_8	Terminal supports HEVC HD at 8 bits per channel
+HEVC_UHD	Terminal supports decoding HEVC at greater than 1920x1080 resolution
+HTML_MIX	Terminal supports multiple simultaneously enabled audio tracks on an HTML5 media element
+HTML5_MEDIA_CONTROLLER	Terminal supports Media Controllers (as defined in the HTML5 specification) for use with HTML5 media elements.
+IPTV_DISCOVERY_APP	Terminal supports IPTV channel discovery via application
+IPTV_DISCOVERY_OSDT	Terminal supports IPTV channel discovery via DVB OSDT
+IPTV_DISCOVERY_SDS	Terminal supports IPTV channel discovery via DVB SD&S
+IPTV_MULTICAST	Terminal supports IP multicast for video delivery
+IPV4	Terminal supports IPv4
+IPV6	Terminal supports IPv6
+MHP	Terminal supports Multimedia Home Platform
+MULTI_CHANNEL_AUDIO	Terminal supports 5.1 or 7.1 channel audio output
+POINTER	Terminal can generate pointer events
+PRIV_CAN_BLOCK_TRACKERS	Terminal is able to block requests to tracking websites
+PRIV_DEFAULT_BLOCK_3RD_PARTY_COOKIES	Terminal does not accept 3rd party cookies in factory default state
+PRIV_DNT_LEGALREGULATORY	Legal or regulatory requirements impact the default for the 'Do Not Track' header
+REMOVEKEYS	Terminal can take away keys from the mandatory key set whilst the application is running
+RLAUNCH_PREAPPROVAL	Terminal manages a list of pre-approved HbbTV applications that can be remotely launched from a CS application. The approval could come from the user or a whitelist provided by the manufacturer
+RLAUNCH_USER_APPROVAL	Terminal has a UI to get user approval for remotely launching HbbTV applications from CS apps that have not been pre-approved
+RLAUNCH_TEMP_UNAVAILABLE	Terminal has temporary states where the feature to remotely launch HbbTV applications is not available
+SPDIF	Terminal provides encoded digital audio via an SPDIF or TOSLink interface
+STEREO	Terminal supports 2.0 channel audio output
+SYNC_BUFFER	Terminal has an internal media synchronisation buffer
+TIMESHIFT	Terminal supports the ability to timeshift live playback
+VK_PLAY	Terminal can generate the VK_PLAY event
+VK_PAUSE	Terminal can generate the VK_PAUSE event
+VK_PLAY_PAUSE	Terminal can generate the combined VK_PLAY_PAUSE event
+WHEEL	Terminal can generate wheel events
+XMLAITSIG	Terminal supports XML AITs with digital signatures

Multiple required features are concatenated to a single string without spaces in between. They must be listed in alphabetical order.

You can also use "-" prefixes to indicate the test should only run on boxes that do not support the feature. For example, a test with optional features set to "-EAC3" will not be run on terminals with E-AC-3 support, but will be run on terminals without E-AC-3 support.

6.3.3.3 Text Condition

A textual description of a precondition. Whitespace is not significant. There are two types of textual precondition, defined below.

6.3.3.3.1 Informative

This description is optional and considered informational for reviewers or implementers (it cannot be assumed that a test operator will see this pre-condition). The format of the textual precondition is a text field (no limit).

6.3.3.3.2 Procedural

This description requires the tester to take specific action (procedural) prior to the execution of the Test Case. If present the execution of this action shall be considered mandatory.

6.3.3.4 TestRun

A reference to zero or more Test Cases that must be passed successfully before this Test Case can run. The Test Cases are referenced by their Test Case ID (see above for format description).

6.3.4 Testing

6.3.4.1 Test Procedure

Steps to perform in the test. The steps mentioned here should describe the major steps that can be found in the implementation code. The actual implementation may consist of more (and probably smaller) steps, but the general layout of the implementation should be described here. Each step has the following elements:

6.3.4.1.1 Index

The numerical sequence index of the test step (describing the order of steps to perform). This starts with index 1 for the first step and increments by 1 for each subsequent step. This attribute is optional.

6.3.4.1.2 Procedure

Contains the textual description of a single step. The format of the description is a text field (no limit). Whitespace is not significant.

6.3.4.1.3 Expected behaviour

Optionally describes the expected behaviour of the DUT when executing this test step.

6.3.4.2 Pass Criteria

Textual description of criteria required to pass the test. The format of the Pass Criteria value is a text field (no limit). Whitespace is not significant.

6.3.4.3 Media Files

Describes a media file used by the test. This is an optional element in the XML, and older tests don't use it. For new tests, it is recommended to use this element to define the media files.

If a test uses multiple media files, this element can be repeated to define each file.

6.3.4.3.1 Name

The file name. This will be the file name on disk. The procedure can use this file name to refer to the media file.

This should just be a file name, not a path. So it shall not contain "/" or "\".

6.3.4.3.2 Description

A human-readable description of the media file. The format of the description value is a text field (no limit). Whitespace is not significant.

6.3.4.4 History

Contains a history of additions and modifications to this test.

6.3.4.4.1 Date

The date of change/update/proposal. The format of the date is YYYY-MM-DD (e.g. 2010-06-10).

6.3.4.4.2 Part

The part of the test that was or is to be updated. Contains one of the following values:

- assertion: for the test metadata (including the assertion)
- procedure: for the test procedure (test steps),
- implementation: for the actual implementation of the test (not included in this document)

6.3.4.4.3 Type

The type of update. Contains one of the following values:

- proposed: Used to indicate the intention to provide the material for the specified part
- submitted: For the initial submission or an update that has to be reviewed
- review_proposed: To indicate the intention to review the specified part
- reviewed: If the update was reviewed and accepted by someone
- accepted: If the update was accepted
- rejected: If the update was rejected and probably should be modified and resubmitted.
- edited: The update provides only editorial changes and has no impact on the test logic. Any changes of this type do not require review.

6.3.5 Others

6.3.5.1 Remarks

Remarks and comments to this test (optional). The format of the remarks is a text field (no limit).

6.4 Test Case Result

6.4.1 Overview (informative)

This document describes the technical process for verification of HbbTV Devices, and as such the choice of pass or fail given by the criteria in 6.4.2 are those that shall be used when generating a conformance report.

A test harness may generate and store other data about tests and use this to present alternative reports to operators. Such information may include indications that although a test would currently be considered as failed by the criteria in 6.4.2, the test may be considered as ‘incomplete’, or passed after further events have taken place (e.g. further test steps or a call to the endTest API method). Such indications and reporting are outside the scope of this specification.

6.4.2 Pass criteria

The result of the test shall be PASS only if all of the following criteria are met:

1) All normative test preconditions were satisfied

2) All step results stored by the test harness have the value ‘true’ for the ‘result’ parameter

3) All calls to analyze API method have been evaluated to give a result and that result is ‘true’

4) All calls to the test API methods that interact with the test environment (e.g. sendKeyCode, changePlayoutSet) succeeded

5) The ‘endTest’ API method was called

If, at a given time, any of the above criteria are not met the result at that time shall be FAIL. The result may change to PASS if these criteria are met at a later time (e.g. an analyze API method is evaluated, or a call to the endTest API method is made.) As step results and analysis results may not be changed, if at a given time the result is PASS then at a later time the result shall not change to FAIL.

7 Test API and Playout Set definition

7.1 Introduction

This section describes the following interfaces:

- A JavaScript API that defines the interface between the Test Harness and DUT.
- A set of XML files that define how the Test Harness should interpret a test case. This allows definition and control of DVB playout required to initiate a test.

By defining these interfaces is it possible for a test case contributor to author test cases that can be run on any HbbTV compatible Test Harness. Similarly, the interface definition allows multiple Test Harnesses to be implemented, with different levels of automation, but still all compatible with test cases that adhere to the interface.

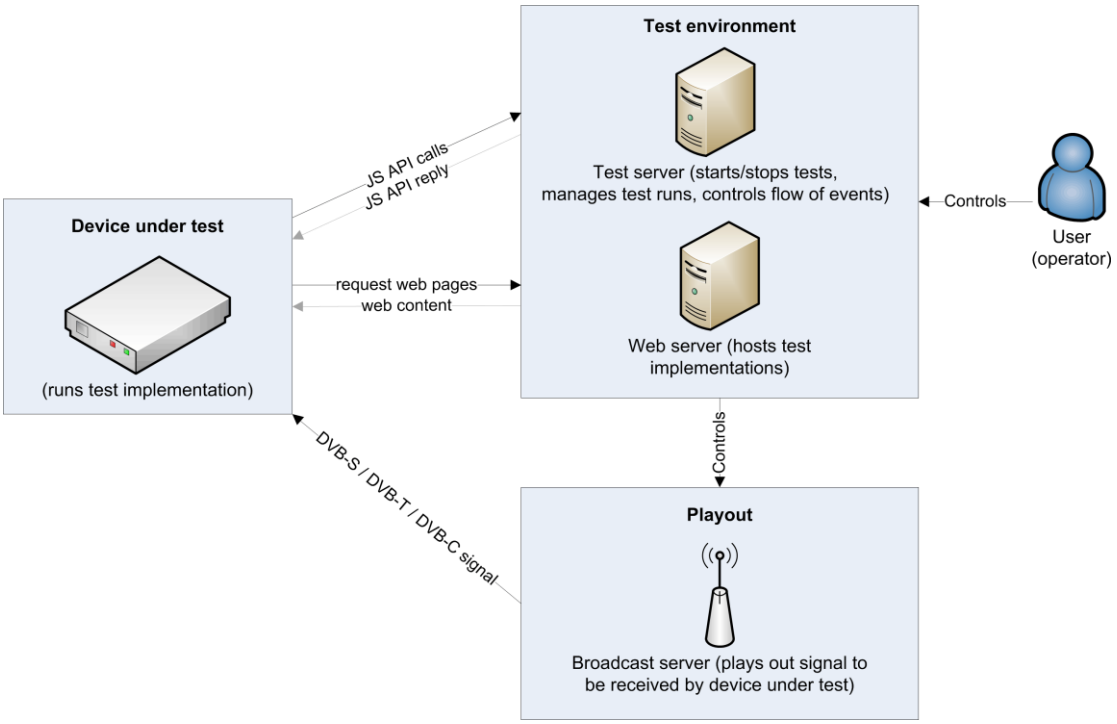


Figure 3: Overview of JS API communication between DUT and Test Harness

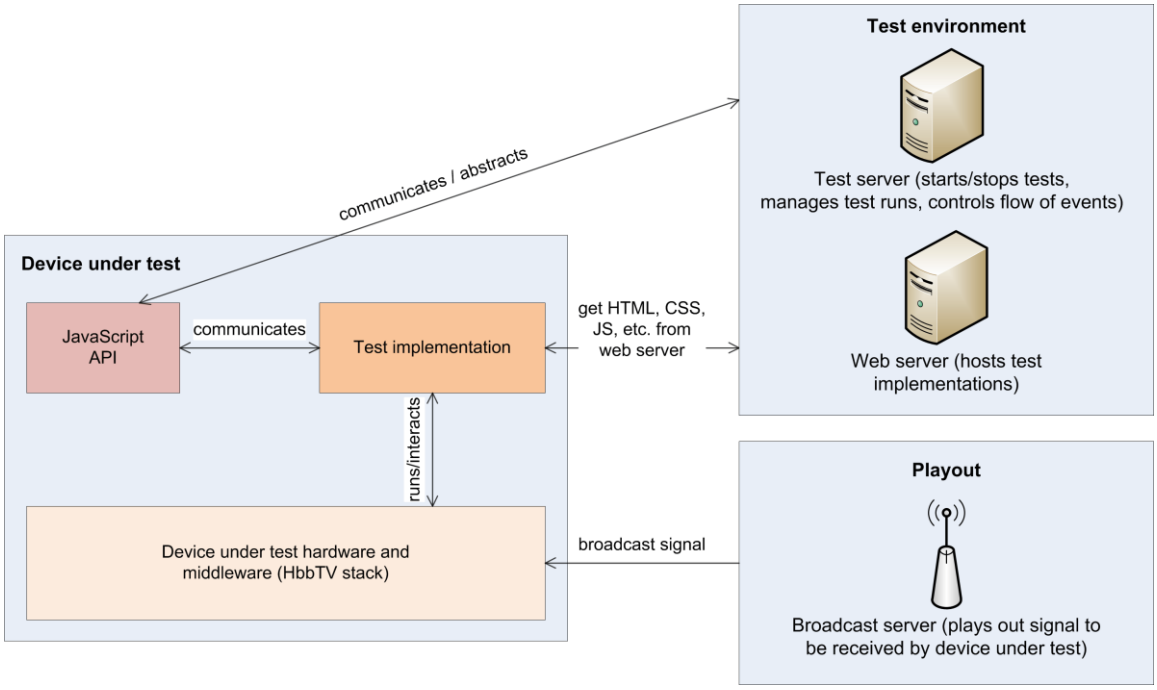


Figure 4: Detailed overview on JS-API abstraction of Test Harness communication

The layout of the APIs described in this document is designed in a way that allows for a high percentage of automation.

There is no necessity for an HbbTV Test Environment to be fully automated. HbbTV JS APIs are therefore designed in such a way that any required test may be operated manually. The implementer of an HbbTV Test Harness may choose for interaction by an operator with the HbbTV Test Environment in a manual way.

The HbbTV JS APIs allow for multiple implementations from different implementers and are designed in such a way that it allows a potential test implementer to implement compatible Test Cases and or Test Harnesses. This allows for combining test cases created by one or more implementers of test cases to create a complete HbbTV (automated) Test Environment.

The JavaScript APIs are divided into three parts:

1) APIs communicating with the Test Environment (see section 7. 2). It informs the Test Environment about the current test's status.

2) APIs communicating with the Device under Test (see section 7.3). This part of the API communicates with the DUT (e.g. send key codes, make screenshots). This can be either be implemented directly by the DUT manufacturer (send commands directly via Ethernet to the DUT), or it can be implemented by someone else (e.g. send commands to an IR sender or a frame grabber).

3) APIs communicating with the Playout Environment (see section 7.4). This part of the API communicates with the Playout Environment (which generates and transmits the DVB-S/-C/-T signal to the DUT). For example, the Playout Environment is responsible for sending the correct AIT to the DUT, such that the test is started on the DUT.

7.1.1 JavaScript strings

Where a String is passed to any JS functions in the HbbTV test API, the following rules apply:

1) It must contain valid UTF-16

2) It must only contain Unicode code points that are allowed by the XML 1.0 specification (http://www.w3.org/TR/2006/REC-xml-20060816/#charsets, section 2.2 Characters).

3) It must not contain the code point U+000D.

I.e. the JavaScript string can only contain the Unicode code points U+0009, U+000A, U+00020-U+D7FF, U+E000-U+FFFD, and U+10000-U+10FFFF.

If a test breaks the rules in the previous paragraph, the test harness shall fail the test.

The test harness must handle any XML escaping necessary when writing characters such as "<" in the test report XML.

7.2 APIs communicating with the Test Environment

7.2.1 Starting the test

The DUT must be in a pre-defined state before a test is started. The pre-defined state is defined in 7.2.2. There are two ways to execute a test:

- The test is started automatically by the DUT as soon as the AIT is parsed and the test application (which is signalled as "AUTOSTART") is detected. This will open the entry URL of the initial test page in the browser on the DUT.
- The test is started by the test harness, executing an application using its resident ECMAScript environment. When the test is started is up to the test harness implementation. This mechanism is used only when the test cannot be practically implemented on the DUT.

NOTE: All JS functions defined in this document are defined within the HbbTVTestAPI prototype. This means that e.g. for reportStepResult, you would call "testapi.reportStepResult(...);"

7.2.1.1 Tests started automatically by the DUT

The initial test page includes the JavaScript file "../RES/testsuite.js" which contains the implementation of the JavaScript classes/functions defined in this document. The testing API described in this document is then initialized. To initialize the testing API and to set up the connection to the test harness, the following steps need to be performed by the test application (usually the initial start page referenced in the AIT).

- Include the JavaScript file "testsuite.js" from "RES" directory (either by relative reference "../../RES/testsuite.js" or by absolute reference "http://hbbtv1.test/_TESTSUITE/RES/testsuite.js"), e.g.:

```
<script type="text/javascript" src="../../RES/testsuite.js"></script>
```

- Initialize the test suite by creating a new instance of the HbbTVTestAPI class and calling init() on it. The init() function needs to be called by the initial test page to initialize the connection between the DUT and the server. e.g.:

```
<script type="text/javascript">
var testapi;
window.onload = function() {
     testapi = new HbbTVTestAPI();
     testapi.init();
};
</script>
```

7.2.1.2 Tests started by the harness

The test application includes the JavaScript file which contains the implementation of the JavaScript classes/functions defined in this document. The testing API described in this document is then initialized by creating a new instance of the HbbTVTestAPI class and calling init() on it.

7.2.2 Pre-defined state

Before starting a test, the DUT must be in the state described in this chapter. The method for setting the DUT to the pre-defined state is specific to the testing environment of the Test Harness and the DUT, and is not described in this document. Possible solutions for setting the DUT to the pre-defined state are:

- using a proprietary API provided by the DUT manufacturer
- using an external solution: resetting the power of the DUT and tuning to a pre-defined channel via the remote control (or automated IR sender)

The pre-defined state of the DUT is as follows:

- services scanned and stored in the service list of the DUT. For test cases that only use a single multiplex at a time, it is optional whether the services on transponder 2 are scanned and stored in the service list of the DUT.
    - modulator channel 1 (network id=99, original network id=99, transport stream id=1) with following services:
        - service “ATE test10” (TV service, id=10 – first entry in the service list. This service is not used by any test, but can be used by the test environment to return to a pre-defined state. The content of this service is not defined by the test suite. The content of this service may be changed to any possible content required by the test environment, as long as all other services are not affected).
        - service “ATE test11” (TV service, id=11 – second entry in the service list)
        - service “ATE test12” (TV service, id=12 – third entry in the service list)
        - service “ATE test13” (TV service, id=13 – fourth entry in the service list)
        - service “ATE test14” (Radio service, id=14 – fifth entry in the service list)
    - modulator channel 2 (network id=65281, original network id=99, transport stream id=2) with following services:
        - service “ATE test15” (TV service, id=15 – first entry in the service list)
        - service “ATE test16” (TV service, id=16 – second entry in the service list)
        - service “ATE test17” (TV service, id=17 – third entry in the service list)
        - service “ATE test18” (TV service, id=18 – fourth entry in the service list)
        - service “ATE test19” (Radio service, id=19 – fifth entry in the service list)
- tuned to transponder 1, service “ATE test11”
- no application is running, so the AUTOSTART application signalled on that service will be started automatically.

Additional requirements are also specified in section 7.4.4 of this document.

7.2.3 Callbacks

All API calls defined in this document are asynchronous, as the API may have to interact with the test harness. To know when the API call actually completes, the caller needs to provide a callback function. This function is called as soon as the API call completes (either locally on the DUT or on the server). The first argument of the callback function is always the callback object passed to the API function. This allows the callback function to determine which API call was processed (if the same callback function is used for multiple API calls). The callback function might be called with additional arguments. These additional arguments are defined in the respective API function definition.

The implementation of a function can either be synchronous (e.g., via OIPF debug function, chapter 7.15.5 of OIPF DAE) or asynchronous (e.g. via XMLHttpRequest to the server). The callback function is called in both cases. In a synchronous implementation, this would be done before the function has returned.

Specifying callback functions and callback objects is not required. The callback arguments may be null if the caller is not interested in the callback and the callbackObject may be null if no state is required.

7.2.4 JS-Function getPlayoutInformation()

This function retrieves information on the current playout.

```
void getPlayoutInformation(callback : function, callbackObject : object);
```

ARGS: **callback/callbackObject:** a callback function to invoke when the information is available (also see chapter 7.2.3 Callbacks). The callback function will be called with the following parameters: callback(callbackObject, playoutInformationObject) where the playoutInformationObject is a JavaScript object which has a 'length' property that returns the integer number of multiplexes in the current playout set. It can be indexed using the integers 0 to `length-1` to obtain JavaScript objects giving information about each multiplex in the current playout set. The order shall match the order that the multiplexes were defined in the playout set XML file. For each multiplex object, the following properties shall be available:

- transponderDeliveryType: the idType (integer) for the DVB delivery type (DVB-S(2), DVB-T(2), DVB-C(2)), as specified in OIPF DAE, chapter 7.13.11.1 to be used for createChannelObject() function calls.
- transponderDsd: the delivery system descriptor (tuning parameter) as specified in OIPF DAE, chapter 7.13.1.3 to be used for createChannelObject() function calls, with the corrections from section A.2.4.4 of the HbbTV specification [1] with errata 1 applied
- eitOffset: a non-negative integer that is the 'EIT Offset' for this multiplex as defined in section 7.4.4.7. If the <adjustEit/> tag was not specified for this multiplex then this shall be zero.

If 'length' is 1, then the properties above shall also be available on the playoutInformationObject directly. (I.e. if `length` is 1, then playoutInformationObject.transponderDeliveryType shall be the same as playoutInformationObject[0].transponderDeliveryType, but if 'length' is not 1 then playoutInformationObject.transponderDeliveryType should be undefined).

Test implementers should not call this function when network connection is down, as it may fail when network connection is not available. In this case, this will cause the complete test to fail automatically.

If this method call fails, an empty object with no key/value pairs is returned.

7.2.5 JS-Function endTest()

This function indicates that the test case has completed.

```
void endTest();
```

ARGS: None

This will end the test. No further calls to any API functions after this call shall have any effect on the test result.

The result of the test case is "PASSED" only in the following case:

- there were no calls to reportStepResult with result being *false* during the complete run of the test, and
- the analysis of all analyze function calls succeeded (or there were no analyze calls at all). There may not be any network connection at the time of the function call (or the Test Harness cannot be reached for other reasons), so this function may be implemented asynchronously. Therefore a test implementer must make sure that the network connection is available at the end of the test. (see reportStepResult documentation).

NOTE: This function could be internally implemented by calling reportStepResult with a stepId below 0, but this is not a requirement.

7.2.6 JS-Function reportStepResult()

This function reports a step result (succeeded step or failed step) back to the Test Harness. The Test Harness shall report the stepId, the result, and the comment in the test report. Test implementers should use this function

to report the result of at least each major test step within the test. If no actual test is performed and only an informational message should be delivered, test implementers shall use the reportMessage function.

```
void reportStepResult(stepId : integer, result : boolean, comment : String);
```

ARGS: **stepId:** the step number that has been performed. Shall be called with an integer value >= 0. stepId = 0 indicates that the test application has just started. stepId values shall be unique throughout the execution of each test case. They shall not be repeated. If a Test Harness receives a repeated value of stepId for a test case then the test shall fail.

There is no need for the stepId in the implementation to match the step values in the test specification procedure.

result: *true* if the step has completed successfully, *false* if the step has failed. To send a failure, no endTest() call is required. In this case, the stepId references the failing step. If the result is false, the server may not stop the application immediately (this may take a few seconds or even minutes, depending on the server). Therefore the testing application should ensure that no more reportStepResult calls are executed after this. Furthermore, the Test Harness shall ignore any reportStepResult calls received after a call with *false* result while executing a test.

comment: a comment from the test developer describing what the step actually does (e.g. a reference to the test procedure)

NOTE 1: The step number is usually in ascending order, but this is not a requirement. A test may skip test numbers and/or not report steps in ascending order. This is especially the case if multiple steps are executed in parallel.

NOTE 2: The test source code may have multiple calls to reportStepResult with a particular stepId value, as long as only one of these calls is executed when the test is run.

7.2.6.1 Reporting results when no network connection is available

There may not be any network connection at the time of the function call (or the Test Harness cannot be reached for other reasons), so this function may be implemented asynchronously. In any case, an implementation must make sure that all step results are reported back to the server as soon as network connection is up again in the order they were made. So there might be the following implementations:

EXAMPLE 1: synchronous communication to the server via a non-network communication path (e.g. a serial line connection)

EXAMPLE 2: asynchronous communication with the server (e.g. XmlHttpRequest via network) where all calls are stored in a FIFO queue, that is, one by one, reported to the server. If communication is not possible (e.g. network not available), the JS API implementation will continually try to report the step results in the queue in the background. This is why a test implementer must make sure that the network connection is available at the end of the test.

The following diagram shows the communication (and queuing) with the server in the case of an asynchronous communication with the server:

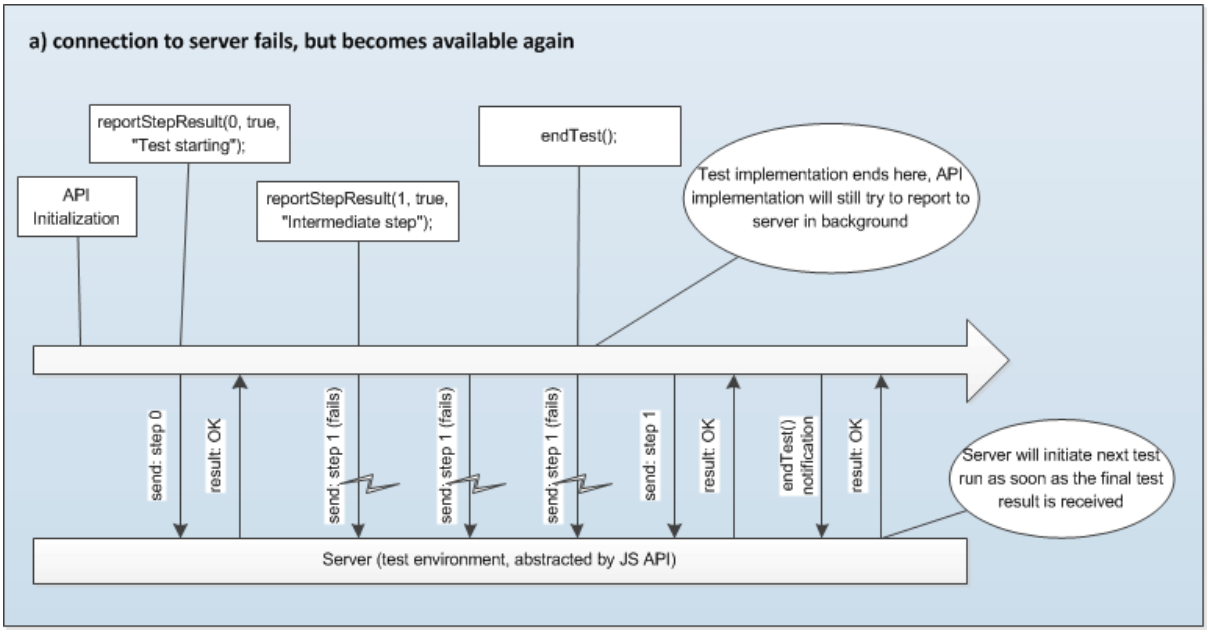


Figure 5: Communication between DUT and Test Harness when network is temporarily unavailable

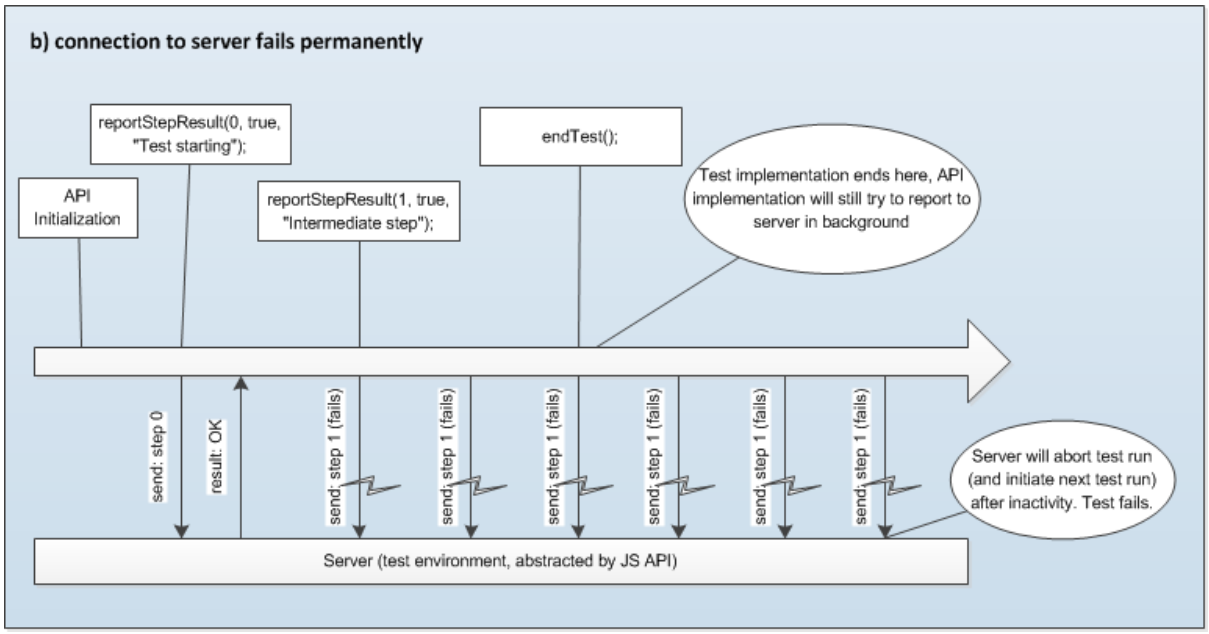


Figure 6: Communication between DUT and Test Harness when network is permanently unavailable

7.2.7 JS-Function reportMessage()

This function reports an informational message to the Test Harness. These messages usually serve as a hint to the test operator (e.g. “waiting 30 seconds”) or give more detailed information when debugging this test. These messages are only informational and do not form part of the test report. It is not specified how the Test Harness should handle these messages: They may be discarded, displayed, logged, etc.

```
void reportMessage(comment : String);
```

ARGS: **comment:** the informational message as String.

NOTE: There may not be any network connection at the time of the function call (or the Test Harness cannot be reached for other reasons), so this function might be implemented asynchronously.

7.2.8 JS-Function waitForCommunicationCompleted()

As the function calls of the API defined in this document are asynchronous, a test implementation might need to make sure that all calls were successfully transmitted to the server and that the queue containing the step results to be reported to the server is now empty. This check should be performed before the network connection is switched off or before the test application destroys itself.

```
void waitForCommunicationCompleted(callback : function, callbackObject : object);
```

ARGS: **callback/callbackObject:** a callback function to invoke when server communication queue is now empty (also see chapter 7.2.3).

NOTE: An implementation of this API which is based on manual interaction may immediately call the callback function when this function is called, as it does not have any communication queue. If a queue is present but empty, the callback function may also be called immediately (but may also be called asynchronously).

7.2.9 JS-Function manualAction()

This function instructs the test operator to carry out an action. This function should only be used if the requested action cannot be achieved using other API methods (e.g. sendKeyCode).

```
void manualAction(check: String, callback : function, callbackObject: object);
```

ARGS: **check:** a textual description of the action required which will be presented to the test operator

callback/callbackObject: a callback function to invoke when the action has been completed by the test operator (also see chapter 7.2.3 Callbacks).

7.3 APIs Interacting with the Device under Test

7.3.1 JS-Function initiatePowerCycle()

This function initiates a power cycle of the DUT. It will first wait a user-defined amount of time, then switch off the DUT, then wait a short period of time, then switch it back on again. The Test Harness shall ensure that the DUT finally returns to the pre-defined state. The period for which the DUT is switched off is not defined, but should be chosen to ensure a clean power cycle of the DUT.

```
void initiatePowerCycle(delaySeconds : int, type : String, callback : function, callbackObject
: object);
```

ARGS: **delaySeconds:** the number of seconds after which the device is switched off. These seconds start counting as soon as the callback function is called, if it is set to 0, the reset may occur while the application is still handling the callback function. If delaySeconds < 0 then the test shall fail.

type: one of the following strings:

- "STANDBY": the device will go to standby mode – if the device does not support standby mode, POWERCYCLE will apply.
- "POWERCYCLE": the device will be physically disconnected from power supply – if the device has a built-in power supply (e.g. battery), the power cycle is defined by the device.

If any other string is received the test shall fail.

callback/callbackObject: a callback function to invoke when the power cycle request was received (also see chapter 7.2.3 Callbacks). The power cycle delay shall start when the callback has completed.

Test implementers should not call this function when network connection is down, as it may fail when network connection is not available. In this case, this will cause the complete test to fail automatically.

Calling this function will cause the current application to be stopped. After reboot, the device needs to be transferred to the pre-defined state (see chapter 7.2.2), so the AUTOSTART application signalled on the initial service will be started.

7.3.2 JS-Function sendKeyCode()

This function requests a key code to be sent to the DUT. This can be implemented by directly sending IR codes to the DUT. Alternatively, this can be implemented by a manufacturer specific API.

```
void sendKeyCode(keyCode : String, durationSeconds : int, callback : function, callbackObject
: object);
```

ARGS: **keyCode:** a string describing the key to send: VK_LEFT, VK_RIGHT, VK_UP, VK_DOWN, VK_ENTER, VK_BACK, VK_RED, VK_GREEN, VK_YELLOW, VK_BLUE, VK_0, VK_1, VK_2, VK_3, VK_4, VK_5, VK_6, VK_7, VK_8, VK_9, VK_PLAY, VK_PAUSE, VK_PLAY_PAUSE, VK_STOP, VK_FAST_FWD, VK_REWIND, VK_EXIT, VK_RECORD. If any other keyCode string is requested then the test shall fail.

durationSeconds: the number of seconds for which the key is held down continuously, or a value of 0 (zero) for a single key press event. If durationSeconds < 0 then the test shall fail.

callback/callbackObject: a callback function to invoke when the key code was actually sent to the receiver (also see chapter 7.2.3 Callbacks).

Test implementers should not call this function when network connection is down, as it may fail when network connection is not available. In this case, this will cause the complete test to fail automatically.

7.3.3 JS-Function analyzeScreenPixel()

This function analyzes the current screen and checks if a specified pixel on that screenshot matches a given reference colour.

```
void analyzeScreenPixel(stepId : integer, comment : String, posx : integer, posy : integer,
referenceColor : String, callback : function, callbackObject : object);
```

ARGS: **stepId:** the step number that has been performed (same as stepId in reportStepResult).

comment: a comment from the test developer describing what the analysis actually does (same as reportStepResult)

posx: the horizontal position of the pixel to analyze within safe border (128-1152)

posy: the vertical position of the pixel to analyze within safe border (36-683)

referenceColor: the reference colour that the pixel should match. The String must start with a hash (#) followed by a 2-digit case insensitive hexadecimal representation of the red, green, and blue colour (e.g. #ff0000 for red colour).

callback/callbackObject: a callback function to invoke when the call completes (also see chapter 7.2.3 Callbacks). The analysis may not have taken place at the time the callback is invoked. The analysis result is not passed back to the application. A failed analysis must cause the complete test to fail, independent on the test result reported back by the reportStepResult function. This is due to the fact that the analysis can also be performed off-line on a taken screen shot.

The method for pixel colour matching is not defined. The following tolerances are defined for analysis:

- The area of pixels analyzed may be up to +/- 10 pixels in each direction of the specified pixel position
- The pixel colour value analyzed may be up to +/- 80/255 of the specified colour value for each colour component

EXAMPLE: This function may be implemented either for manual interaction or for automated processing. During analysis, the implementation may modify the HTML DOM (e.g. add a black layer and a cross hair on top of the screen and removing it after analysis is finished). This might trigger an Application.show() call. If only manual processing is desired, an API application could call analyzeScreenExtended("Is colour of Pixel at Pos. "+posx+"/"+posy+" similar to reference colour "+referenceColor+"?", callback, callbackObject).

Test implementers should not call this function when network connection is down, as it may fail when network connection is not available. In this case, this will cause the complete test to fail automatically.

7.3.4 JS-Function analyzeScreenExtended()

This function analyzes the current screen and performs an extended check on the currently displayed screen content. As this check is described by a string, the check most probably has to be done by a human being, although a test environment may provide an automated implementation of this check. This call should be avoided by test case implementers, if possible.

```
void analyzeScreenExtended(stepId : integer, comment : String, check : String, callback :
function, callbackObject : object);
```

ARGS: **stepId:** the step number that has been performed (same as stepId in reportStepResult).

comment: a comment from the test developer describing what the analysis actually does (same as reportStepResult)

check: a textual description detailing which test to perform. This is the only criteria that shall be used for the assessment of this analysis call.

callback/callbackObject: a callback function to invoke when the analysis was made (also see chapter 7.2.3 Callbacks). The analysis result is not passed back to the application. A failed analysis must cause the complete test to fail, independent on the test result reported back by the reportStepResult function. This is due to the fact that the analysis can also be performed off-line on a taken screen shot.

The same pixel and colour tolerances apply as specified in the analyzeScreenPixel function.

EXAMPLE: This function may be implemented by first taking the screenshot and then performing an offline analysis at a later point in time.

Test implementers should not call this function when network connection is down, as it may fail when network connection is not available. In this case, this will cause the complete test to fail automatically.

7.3.5 JS-Function analyzeAudioFrequency()

This function analyzes the current audio and performs a frequency check on that data.

```
void analyzeAudioFrequency(stepId : integer, comment : String, channelId : integer,
referenceFrequency : int, callback : function, callbackObject : object);
```

ARGS: **stepId:** the step number that has been performed (same as stepId in reportStepResult).

comment: a comment from the test developer describing what the analysis actually does (same as reportStepResult)

channelId: the channel in the referenced audio that is being analyzed. The following channel IDs are supported:

- 1: left channel
- 2: right channel
- 3: centre channel
- 4: rear left channel

- 5: rear right channel

  If the channelId is not implemented (channelId < 1 or channelId > the number of audio channels in the sample) then the test shall fail.

  referenceFrequency: the reference frequency in Hz. Allowed values are 500, 630, 800, 1000, 1250, 1600, 2000, 2500, 3150, 4000. Other values shall cause the test to fail.

  callback/callbackObject: a callback function to invoke when the analysis was made (also see chapter 7.2.3 Callbacks). The analysis result is not passed back to the application. A failed analysis must cause the complete test to fail, independent on the test result reported back by the reportStepResult function. This is due to the fact that the analysis can also be performed off-line on a taken screen shot.

Frequencies should be detected that have a minimum signal level between -50dB and 0dB, assuming no attenuation by the DUT. The tolerance on frequency detected shall be +/- 10%.

The method of audio frequency matching is not defined.

EXAMPLE 1: If the implementation uses a standard one third octave analyzer, ten different one-third octave band frequencies fall into the above defined range (500 Hz, 630 Hz, 800 Hz, 1000 Hz, 1250 Hz, 1600 Hz, 2000 Hz, 2500 Hz, 3150 Hz, 4000 Hz), which should be reasonable distinguishable.

EXAMPLE 2: This function may be implemented either for manual interaction or for automated processing. If only manual processing is desired, an API application could call analyzeAudioExtended("Do you hear a tone with a frequency of about "+referenceFrequency+" Hz on channel "+channelId+"?", callback, callbackObject).

Test implementers should not call this function when network connection is down, as it may fail when network connection is not available. In this case, this will cause the complete test to fail automatically.

NOTE: This function may be implemented by first making an audio capture and performing the analysis at a later point in time.

7.3.6 JS-Function analyzeAudioExtended()

This function analyzes the current audio and performs an extended check on that data (as this check is described by a String, the check most probably has to be done by a human being, although a test environment may provide an automated implementation of this check). This call should be avoided by test case implementers, if possible.

```
void analyzeAudioExtended(stepId : integer, comment : String, check : String, callback :
function, callbackObject : object, duration : integer, delay : integer);
```

ARGS: **stepId:** the step number that has been performed (same as stepId in reportStepResult).

comment: a comment from the test developer describing what the analysis actually does (same as reportStepResult)

check: a textual description detailing which test to perform on the audio data. This is the only criterion that shall be used for the assessment of this analysis call. An empty or null string shall cause the test to fail.

callback/callbackObject: a callback function to invoke when the analysis was made (also see chapter 7.2.3 Callbacks). The analysis result is not passed back to the application. A failed analysis must cause the complete test to fail, independent on the test result reported back by the reportStepResult function. This is due to the fact that the analysis can also be performed off-line on a taken screen shot.

duration: duration of recording in seconds. The argument is optional; if it is not present then the default is 10 seconds.

delay: delay before recording in seconds. The argument is optional; if it is not present then the default is 0 seconds.

This function analyzes (or records for later analysis) the audio over the following period:

- Starting between (delay + 0) and (delay + 2) seconds after the API is called (although this may be delayed if other Test API calls are in progress when this API is called).
- Ending (duration) seconds after the analysis (or recording) started.

The described check shall not require audio data from outside the specified period. This allows the audio to be recorded and then processed later on.

The callback may be called much later than the end of the analysis period, e.g. if the harness is waiting for a user to enter the result, or if the harness is doing non-real-time audio compression or some automated analysis.

The method of audio analysis is not defined.

Test implementers should consider the audio content to be analyzed, assuming there will be a human tester.

E.g., if implementation uses a microphone for audio capture, provided results might not be very exact. Inaccuracy of audio capture may be up to +/- 1000 Hz. Only frequencies should be detected that have a minimum loudness of at least 50% of the maximum allowed loudness.

NOTE: This function may be implemented by first making an audio capture and performing the analysis at a later point in time.

Test implementers should not call this function when network connection is down, as it may fail when network connection is not available. In this case, this will cause the complete test to fail automatically.

7.3.7 JS-Function analyzeVideoExtended()

This function analyzes the current video and performs an extended check on that data. As this check is described by a string, the check most probably has to be done by a human being, although a test environment may provide an automated implementation of this check. This call should be avoided by test case implementers, if possible.

```
void analyzeVideoExtended(stepId : integer, comment : String, check : String, callback :
function, callbackObject : object, duration : integer, delay : integer);
```

ARGS: **stepId:** the step number that has been performed (same as stepId in reportStepResult).

comment: a comment from the test developer describing what the analysis actually does (same as reportStepResult)

check: a textual description detailing which test to perform on the video data. This is the only criteria that shall be used for the assessment of this analysis call. An empty or null string shall cause the test to fail.

callback/callbackObject: a callback function to invoke when the analysis was made (also see chapter 7.2.3 Callbacks). The analysis result is not passed back to the application. A failed analysis must cause the complete test to fail, independent on the test result reported back by the reportStepResult function. This is due to the fact that the analysis can also be performed off-line on a taken recording.

duration: duration of recording in seconds. The argument is optional; if it is not present then the default is 10 seconds.

delay: delay before recording in seconds. The argument is optional; if it is not present then the default is 0 seconds.

This function analyzes (or records for later analysis) the video over the following period:

- Starting between (delay + 0) and (delay + 2) seconds after the API is called (although this may be delayed if other Test API calls are in progress when this API is called).
- Ending (duration) seconds after the analysis (or recording) started.

The described check shall not require video data from outside the specified period. This allows the video to be recorded and then processed later on.

The callback may be called much later than the end of the analysis period, e.g. if the harness is waiting for a user to enter the result, or if the harness is doing non-real-time video compression or some automated analysis.

Test implementers should not call this function when network connection is down, as it may fail when network connection is not available. In this case, this will cause the complete test to fail automatically.

NOTE: This function may be implemented by first making a video capture and performing the analysis at a later point in time.

7.3.8 JS-Function analyzeManual()

This function instructs the test operator to carry out the analysis described in the check parameter, and record the result. This function should only be used if the analysis cannot be achieved using other API methods.

```
void analyzeManual(stepId : integer, comment : String, check: String, callback : function,
callbackObject: object);
```

ARGS: **stepId:** the step number that has been performed (same as stepId in reportStepResult).

comment: a comment from the test developer describing the purpose of the step

check: a textual description of the analysis that must be done which will be presented to the test operator

callback/callbackObject: a callback function to invoke when the power cycle request was received (also see chapter 7.2.3 Callbacks).

7.3.9 JS-Function selectServiceByRemoteControl()

This function requests to select a service by a sequence of (mainly numeric) key codes that is sent to the DUT. This shall be implemented by directly sending IR codes (or equivalent) to the DUT (automated or by a request to the tester to do so manually, similar to reportMessage). The service is identified by its name. The service selection shall switch directly from the current to the new service.

```
void selectServiceByRemoteControl(serviceName: String, callback : function, callbackObject :
object);
```

ARGS: **serviceName:** the name of the service to select, e.g. "ATE Test11". The name shall be the service name as defined in the service descriptor of the target service. If the service is not signalled in the current broadcast then the function shall fail.

callback/callbackObject: a callback function to invoke when the key codes were actually sent to the receiver (also see chapter 7.2.3 Callbacks).

Test implementers should not call this function when network connection is down, as it may fail when network connection is not available. In this case, this will cause the complete test to fail automatically.

This function should not be used to select a radio service when a TV service is selected or vice-versa, as this could require a change of service lists and could involve intermediate service selections (e.g. to the last active service of the new list).

Before calling this API, the test case should ensure no HbbTV applications are requesting the NUMERIC KeySet.

NOTE: Although the naming of this function indicates that the service change must be performed by emulating a remote control (i.e. sending IR commands to an IR receiver), there is no specific requirement to do this. If alternative methods of service selection are available then these are valid.

7.3.10 JS-Function sendPointerCode()

This function requests a pointer device to be moved to a location on the screen of the DUT and optionally a pointer code to be sent to the DUT. This can be implemented by manually moving a pointer device and sending codes to the DUT. Alternatively, this can be implemented by a manufacturer specific API.

```
void sendPointerCode(posx : integer, posy : integer, pointerCode : String, callback :
function, callbackObject : object);
```

ARGS: **posx:** the horizontal position of the pixel to move to (0-1279)

posy: the vertical position of the pixel to move to (0-719)

pointerCode: a string describing the code to send once the pointer is in position: “P_CLICK”, “P_DBLCLICK”, “P_DOWN”, “P_UP”, or “NONE” if no code is to be sent. If any other pointerCode string is requested then the test shall fail.

callback/callbackObject: a callback function to invoke when the pointer has been moved and any requested pointer code was sent to the DUT (also see chapter 7.2.3 Callbacks).

The tolerance on posx and posy is +/- 10 pixels.

7.3.11 JS-Function moveWheel()

This function requests the wheel device to be moved relative to the current position. This can be implemented by manually moving a wheel device. Alternatively, this can be implemented by a manufacturer specific API.

```
void moveWheel(deltax : integer, deltay : integer, deltaz : integer, deltaMode : String,
callback : function, callbackObject : object);
```

ARGS: **deltax:** the relative horizontal position to move by

deltay: the relative vertical position to move by

deltaz: the relative depth position to move by

deltaMode: the measurement mode, one of: “WHEEL_DELTA_PIXEL”, “WHEEL_DELTA_LINE”, “WHEEL_DELTA_PAGE”

callback/callbackObject: a callback function to invoke when the key code was actually sent to the receiver (also see chapter 7.2.3 Callbacks).

7.3.12 JS-Function analyzeScreenAgainstReference()

This function allows comparison of an area of the DUT screen against a reference image. The comment gives additional information on the content of the expected area which may be used for the comparison along with the reference image. Additionally, the reference could be used to perform machine-based comparison of captured DUT output in the specified area of the image.

```
void analyzeScreenAgainstReference (stepId:integer, comment:String, referenceImage:String,
areaOfInterest:Object, callback:function, callbackObject:Object)
```

ARGS: **stepId:** the step number that has been performed (same as stepId in reportStepResult).

comment: a comment describing to the tester content which is expected to appear in the specified area. The area of interest is specified as a separate parameter.

referenceImage: path to a reference image relative to test folder. The reference image should be a complete capture of the screen, although the function will consider only the area of interest. The image shall be in PNG format and the filename shall end in ‘.png’

areaOfInterest: JS object which contains pixel coordinates of the screen region that should be compared (top, left, bottom, right). Pixel coordinates are zero-based.

callback/callbackObject: a callback function to invoke when the call completes (also see chapter 7.2.3 Callbacks). The comparison may not have taken place at the time the callback is invoked. The comparison result is not passed back to the application. A failed comparison must cause the complete test to fail, independent on the test result reported back by the reportStepResult function. This is due to the fact that the analysis can also be performed off-line on a taken screenshot.

7.3.13 JS-Function analyzeTextContent()

This function analyzes text in the specified area (parameter areaOfInterest) searching for specified text (parameter expectedText). In case of manual execution, tester is instructed which text to search if specified area. Additionally, reference image is presented to the tester to further clarify appearance of the text. In case of test automation, OCR algorithm may be used to extract and compare text content from captured DUT output.

```
void analyzeTextContent (stepId:integer, comment:String, expectedText:String,
areaOfInterest:Object, referenceImage:String, callback:function, callbackObject:Object)
```

ARGS: **stepId**: the step number that has been performed (same as stepId in reportStepResult).

comment: a comment from the test developer describing to the tester (if test is performed manually) text which is expected to appear in specified area. Area of interest is specified as a separate parameter.

expectedText: text content expected to be found in given region. If text content is found, test passes.

areaOfInterest: JS object which contains pixel coordinates of the screen region in which text should appear (top, left, bottom, right). Pixel coordinates are zero-based.

referenceImage: path to the image relative to test folder. Reference image should be a complete capture of the screen, although function will consider only area of interest. The image shall be in PNG format and the filename shall end in '.png'. This image may be presented to the tester by the test harness in order to provide further clarification of expected text appearance on the screen.

callback/callbackObject: a callback function to invoke when the call completes (also see chapter 7.2.3 Callbacks). The analysis may not have taken place at the time the callback is invoked. The analysis result is not passed back to the application. A failed analysis must cause the complete test to fail, independent on the test result reported back by the reportStepResult function. This is due to the fact that the analysis can also be performed off-line on a taken screenshot.

7.4 APIs communicating with the Playout Environment

7.4.1 Playout definition

Each test must have at least one definition of a playout set. A playout set comprises the following information:

- Transport streams to play out (independent of delivery type: e.g. DVB-S, DVB-C, DVB-T)
- Definition for AIT tables (optional, if not defined, they must be inside transport streams)
- Definition of DSM-CC carousels
- Other settings (e.g. network connection down)

The playout set definition with id "1" is the initial playout set and shall be active when the test starts. The referenced AIT should start the actual test application. Some tests may require switching between multiple playout sets. The switching is either done after a specified amount of time (timeout attribute in a playout set specifies after how many seconds to automatically switch to the next playout set) or after an API call from the test (see changePlayoutSet function below).

Each test must have a XML definition file called "implementation.xml" residing in the test directory defining all the requirements of the test:

```
<testimplementation id="<testcaseid>">
    <playoutsets>
        <playoutset id="<number>" definition="<rel_filename>"
```

```
        [timeout="<seconds>"] />+
    </playoutsets>
</testimplementation>
```

The XML file “implementation.xml” must validate against the “testImplementation.xsd” XML schema as defined in /SCHEMAS/testImplementation.xsd of the Test Suite.

When the timeout occurs (playout is played out for the specified time in seconds and no changePlayoutSet() call was made), the next playout set to be played out is determined as follows: the id of the current playout set is incremented by one, and the playout set with that new id is played out. If no such playout set exists, the playout set with id 1 is played out.

When changing playout sets (especially the ones containing audio/video), no seamless switching is required: continuity counter errors, PCR continuity problems, incomplete sections and/or tables may occur shortly after time of switching. However, the overall signal should never be interrupted, and the version numbers of changed tables need to be modified to indicate the change to the DUT. As audio/video might show some artefacts, test case implementers should not perform any broadcast audio/video checks up to 5 seconds after a switch occurs.

7.4.2 Relative file names

File names are always relative to the XML file containing them. When the test is executed, the complete test directory is available on the web server in a directory called “TESTS”. On the same level, there is a “RES” directory including the general resource files from the test suite. In addition, the “RES” directory will also contain a file called “testsuite.js” containing the implementation of the test suite API defined in this document.

7.4.3 Playout set definition

A playout set as defined in the test definition must be an XML file. This file defines all the transport stream files, AITs, DSM-CCs that need to be multiplexed to the final test signal played out. The transport streams are included as MPTS files referenced by relative file names.

```
<playoutsetdefinition>
    <transportstream
     file="<rel_filename>" file="<bitspersec>">+
        <pid src="<pid0-8190>" dst="<pid0-8190>"
             description="<text>" />*
        <!-- a PID in the transport stream will only be played
             out if it is listed here. src is the PID within
             the transport stream file. dst is the PID played
             out. Note: the test itself does not include any
             NIT play out. A modulation-type specific NIT is
             inserted later on by the playout server. -->
    </transportstream>
    <generatedData>
        <nit src="<rel_filename>" bitrate="<bps>">
        <ait pid="<pid0-8190>" src="<rel_filename>"
             bitrate="<bps>" version="<0-7>" />+
        <dsmcc pid="<pid0-8190>" association_tag="<0-255>"
               source_folder="<dir>" bitrate="<bps>"
               version="<0-255>" carousel_id="<0-255>">*
            <directory src="<file>" dst="<name>" />*
            <file src="<file>" dst="<name>" />*
            <streamEvent src="<file>" dst="<name>" />?
        </dsmcc>    </generated-data>
    <networkconnection available="YES|NO" />
</playoutsetdefinition>
```

The playout set definition XML files must validate against the “playoutsetDefinition.xsd” XML schema as defined in /SCHEMAS/playoutsetDefinition.xsd of the Test Suite. The referenced NIT XML file must validate against the “nit.xsd” XML schema as defined in /SCHEMAS/nit.xsd of the Test Suite. The syntax of the NIT XML file is defined in 7.4.4.3.3. The referenced AIT table must validate against the AIT XML format as described in TS 102 809 [4], chapter 5.4. The referenced StreamEvent description file must validate against the StreamEvent XML format as described in TS 102 809 [4], chapter with amendments in HbbTV Technical Specification section 9.3.1 [1] [20]. All other referenced data is in binary format.

When creating an AIT, you can assume that the test is available via a pre-defined URL. For more information, see chapter 5.1 Test Environment of this document.

NOTE: The played out AIT version number is not necessarily identical to the version number specified in the playout set definition XML file, as the test harness may add an offset (0, 8, 16, or 24) to that version number, depending on the test run. However, the offset is constant during the run of a single test case.

All bitrates are specified in bits per second (as integer values). Data should be played out with continuous bitrates (no bursts). Due to packaging reasons, the bitrate may vary +/- 1504 bits on a specific second (1504 bits per second is one TS packet per second). The overall specified bitrate should be achieved as closely as possible.

A playout set with network connection set to NO must have a timeout value set to ensure that the Test Harness will terminate this playout set and start a new one with network connection set to YES.

As reportStepResult might require a network connection, the test implementer must make sure that the network connection is available at the end of the test, so the last playout set in a playout set definition shall always have network connection set to YES.

To support the OpenCaster stream generator, StreamEvent objects need the extension ".event" to be interpreted as and create a proper StreamEvent object in the DSM-CC. Without the ".event" extension the behaviour is undefined. So, for example the following is required in the playout set XML streamEvent element:

```
<streamEvent dst="eventObject.event" src="ste.xml"/>
```

7.4.4 Transport stream requirements

The test harness shall support the generation of transport streams as defined by valid playout set definitions as described in 7.4.3. The transport stream shall be valid according to the requirements of [18].

7.4.4.1 Overview (informative)

Generated transport streams are composed of transport stream packets taken from one or more files provided as part of the test suite (as defined by the transportStream element of the playout set definition), optionally combined with transport stream packets generated by the harness (as defined by the generatedData element of the playout set definition.) The packets generated by the harness shall define either an AIT table or a DSM-CC object carousel, both as defined in [4].

7.4.4.2 Static transport stream components

For each transportStream element in the playout set definition the file listed in the file attribute (the original transport stream) shall be repeatedly multiplexed into the generated transport stream.

Each transport stream packet from the original transport stream shall only be multiplexed into the generated transport stream if the PID of the packet is equal to the src attribute of one of the pid elements contained in the transportStream element. In this case, the PID in the packet shall be replaced with the value in the dst attribute of the pid element with a matching src attribute. The value of 16 is not permitted for dst (see 7.4.4.4); if it is present then the transport stream shall not be generated.

In the final generated transport stream packets originating from the original transport stream shall occur at the same frequency at which they occurred in the original transport stream. The frequency of packets in the original transport stream shall be determined by reference to the location of the packets in the transport stream and the bitrate attribute, in bits per second, of the transportStream element.

7.4.4.3 Dynamic transport stream components

7.4.4.3.1 AIT

For each ait element contained in the generatedData element of the playout set definition the harness shall generate transport stream packets and multiplex them into the final generated transport stream such that the generated bits occur at the rate, in bits per second, given by the bitrate attribute of the ait element. If no bitrate attribute is defined then the default value of 5000 bits per second shall be used. The generated transport stream packets shall have their PID set to the value defined by the pid attribute of the ait element.

The version field of the generated transport stream packets containing the AIT shall be set to the value of the version attribute of the ait element, or 0 if no version attribute is defined. The harness may optionally implement the behaviour described in 7.4.4.5.1.

The test harness shall read the file identified by the src attribute of the ait element. The harness shall encode the data read from the XML definition into the generated transport stream packets as defined in clause 7.2.3.1 of [1]. If the file identified by the src attribute does not contain a well-formed XML encoding of an AIT, as defined in 5.4 of [4] (including the <ParentalRating> extension defined in clause 7.2.3.2 of [1]), then the test harness shall not generate the transport stream.

7.4.4.3.2 DSM-CC

For each dsmcc element contained in the generatedData element of the playout set definition the harness shall generate transport stream packets and multiplex them into the final generated transport stream such that the generated bits occur at the rate, in bits per second, given by the bitrate attribute of the dsmcc element. If no bitrate attribute is specified then the default value of 100000 bits per second shall be used. The generated transport stream packets shall have their PID set to the value defined by the pid attribute of the dsmcc element.

The version field of the generated transport stream packets containing the DSM-CC shall be set to the value of the version attribute of the dsmcc element, or 0 if no version attribute is defined. The harness may optionally implement the behaviour described in 7.4.4.5.2.

The test harness shall generate a DSM-CC object carousel as defined in clause 7 of [4] containing a file system. The contents of the directory identified by the source_folder attribute of the dsmcc element shall be included at the root of the generated file system, i.e. Files contained directly in the indicated directory shall be at the root level of the carousel, subdirectories of the indicated directory shall be subdirectories of the carousel, etc. The directory identified by the source_folder attribute may be empty.

For each directory element contained in the dsmcc element, the contents of the directory indicated by the src attribute shall be available in the generated file system at the location indicated by the dst attribute. The location specified by dst is a path relative to the root of the generated file system.

For each file element contained in the dsmcc element, the file indicated by the src attribute shall be available in the generated file system at the location given by the dst attribute.

For each streamEvent element contained in the dsmcc element, the stream event described by the contents of the file identified by the src attribute shall be available in the generated file system at the location given by the dst attribute. If the file identified by the src attribute is not a valid XML document according to the schema defined in 8.2 of [4] then the transport stream shall not be generated.

The location specified by the dst attribute of the directory, file and streamEvent elements is a path relative to the root of the generated file system. All the parent directories of a location specified in a dst attribute must have been defined by either the directory structure identified by the source_folder attribute of the parent dsmcc element, or by the directory structure resulting from a sibling directory element.

For all paths in the generated file system referred to in file and directory elements the test harness shall treat the character '/' (ASCII 47 / Unicode U+002F) as a path separator. The test harness may also treat the character '\' (ASCII 92 / Unicode U+005C) as a path separator. The test harness shall support relative paths, and shall support the referencing of files from any location within the test suite associated with the playout set definition.

If the test harness is unable to locate any one of the indicated file system assets then the test harness shall not generate the transport stream. If the contents of the dsmcc element result in an ambiguous definition for the structure of the constructed carousel (e.g. the dst attribute of a file element refers to a path already defined by the contents of the source_folder attribute) then the transport stream shall not be generated. The test harness may choose not to generate the transport stream if src or source_folder attributes reference file system locations outside the test suite associated with the playout set definition.

The generated DSM-CC carousel shall use the following attributes of the dsmcc element as specified:

- association_tag: this value shall be used for the DSM-CC elementary stream's association tag. If the value is outside the legal range then the transport stream shall not be generated.

- carousel_id: this value shall be used for the DSM-CC carousel ID. If the value is outside the legal range then the transport stream shall not be generated.
- Version: this value shall be used for the version number of the module/DII. If not specified 0 shall be used. The harness may optionally implement the behaviour described in 7.4.4.5.1. If the value is outside the legal range then the transport stream shall not be generated.

7.4.4.3.3 NIT

For nit element contained in the generatedData element of the playout set definition the harness shall generate transport stream packets and multiplex them into the final generated transport stream such that the generated bits occur at the rate, in bits per second, given by the bitrate attribute of the nit element. Only a single NIT is supported per generated TS.

By default, the Test Harness generates a NIT and inserts it into the DVB broadcast that is being played out, as described in 7.4.4.4. This generated NIT may not be suitable for all tests, so the test case can specify the NIT in XML format, and the Test Harness will generate the specified NIT. The NIT XML tag goes in the <generatedData> block in the playoutset XML file, as the first thing in that block.

The XML format includes a placeholder where the delivery system descriptor should be inserted. When generating the NIT, if that placeholder was used, the Test Harness will automatically insert the correct delivery system descriptor(s) containing the selected DVB playout parameters. This allows the same test to work in DVB-T, C, and S.

<nit src="<rel_filename>" bitrate="1504">

The “src” attribute specifies the path to the NIT XML file. It is relative to the directory containing the playoutset XML file.

The optional “bitrate” attribute specifies the NIT bitrate. This includes all the TS packet overhead. The default is 1504 bits/sec (equal to one TS packet per second).

NOTE: For a NIT that fits in one 188-byte TS packet and is repeated at the DVB specified minimum rate of every 10 seconds the required bitrate = 188 byes * 8 bits/byte / 10 seconds = 150.4 bits per second. The bitrate has to be specified as an integer, so you can round that, making sure to round up to stay inside the 10 second limit, giving 151 bits/sec.

The NIT XML defines a customized NIT to be generated by the Test Harness,

```
<nit nid="<0-65535>" version="<0-31>">
  <network>
    <networkNameDescriptor><text></networkNameDescriptor>
  </network>
  <transportStream onid="<0-65535>" tsid="<0-65535>">+
    <autoDeliverySystemDescriptor multiplex="<multiplex>"/>
    <serviceListDescriptor>
      <service sid="<0-65535>" type="<serviceType>"/>+
    </serviceListDescriptor>
    <privateDataSpecifierDescriptor value="<value>"/>
    <rawDescriptor tag="<0-255>">
        <hex_bytes >
    </rawDescriptor>
       <linkageDescriptor type="<0-7><32-255>" onid="<0-65535>" tsid="<0-65535>"
       sid="<0-65535>"/>
  </transportStream>
</nit>
```

All numbers in this file are decimal, because that is the standard XML Schema method of representing a number. Exception: The descriptor bytes that are the contents of the <rawDescriptor> element, described later, are specified in hex. (But the “tag” attribute of <rawDescriptor tag="..."> is decimal).

The <autoDeliverySystemDescriptor> tag inserts the correct delivery system descriptor for the current multiplex in the playoutset (i.e. the multiplex we’re inserting the NIT into), taking into account the configured DVB type and the configured DVB modulation settings.

The delivery system used, and hence the content of the delivery system descriptors is implementation dependent. The inserted delivery system descriptors shall be the same as those returned by the getPlayoutInformation test API defined in 7.2.4.

To refer to other multiplexes, there is an optional parameter on that tag. <autoDeliverySystemDescriptor multiplex="0"> inserts the correct delivery system descriptor for the first multiplex in the playoutset XML file, multiplex="1" refers to the DSD for the second multiplex, etc.

The descriptors inserted by <autoDeliverySystemDescriptor> are:

Modulation	**Descriptor(s)**
DVB-T	terrestrial_delivery_system_descriptor
DVB-T2	T2_delivery_system_descriptor
DVB-C	cable_delivery_system_descriptor
DVB-S	satellite_delivery_system_descriptor
DVB-S2	satellite_delivery_system_descriptor and S2_satellite_delivery_system_descriptor

<network> and <transportStream> are both descriptor loops, so they can include any of the following descriptor tags an unlimited number of times in any order. The generated NIT will contain the relevant descriptors in the order that they were specified in the XML.

- <networkNameDescriptor>. This inserts a network_name_descriptor. The content of this XML tag is the network name. If possible, this will be encoded into the descriptor using "character code table 00 - Latin alphabet" from ETSI EN 300 468 [18]. Otherwise it will be encoded into the descriptor using UTF-8, which will be correctly signalled as described in Annex A of ETSI EN 300 468 [18].
- <autoDeliverySystemDescriptor>. This causes the relevant delivery system descriptor(s) to be inserted, as described above. By default this refers to the the multiplex we're inserting the NIT into. To refer to other multiplexes, there is an optional "multiplex" attribute. multiplex="0" inserts the correct delivery system descriptor for the first multiplex in the playoutset XML file, multiplex="1" refers to the DSD for the second multiplex, etc.
- <serviceListDescriptor>. This inserts a service_list_descriptor. This XML tag contains a list of <service> tags describing services to encode into the descriptor. The services are encoded into the descriptor in the order they are specified in the XML NIT. Each <service> tag has a "sid" attribute for the service ID, and a "type" attribute for the service type. The "type" can be a decimal integer in range 0-255 inclusive, or a short human-readable code for the common types:
  - "mpeg2-sd-tv" = 0x1
  - "radio" = 0x2
  - "teletext" = 0x3
  - "avc-radio" = 0xa
  - "data" = 0xc
  - "mpeg2-hd-tv" = 0x11
  - "avc-sd-tv" = 0x16
  - "avc-hd-tv" = 0x19
- <privateDataSpecifierDescriptor>. Inserts a private_data_specifier_descriptor. The mandatory "value" parameter is the 32-bit ID of the organisation, in decimal.
- <rawDescriptor>. This allows inserting any descriptor. Specify the descriptor tag, in decimal, as the mandatory "tag" parameter. Specify the descriptor payload, if any, in hex as the contents of this tag. The descriptor length will be automatically calculated from the payload. In the XML, the payload can optionally contain whitespace between bytes, but not between nibbles in the same byte. I.e. "123456789abcdef0", "12 34 56 78 9a bc de f0", and "12 34 56 78 9abcDEF0" are all legal payloads and all mean the same thing, but
  "1 23456789abcdef0" is not legal. This allows you to document the payload by breaking it across lines and using XML comments.
- <linkageDescriptor>. This inserts a linkage_descriptor. All the numeric fields are decimal. The "type" attribute specifies the linkage_type field in the generated descriptor. Note that this tag only supports simple usage of the descriptor: it does not support types 8, 13 or 14, and it does not support including any private_data_byte. If you need to generate a linkage_descriptor with private_data_byte, use <rawDescriptor> instead.

See ETSI EN 300 468 [18] for the specification of the generated binary NIT and descriptors.

It is an error to specify a <serviceListDescriptor> or <rawDescriptor> with a descriptor payload larger than 255 bytes.

The generated NIT is limited to a single section. It is an error to specify a NIT larger than that.

The NIT will be signalled as a “NIT actual”. The “current_next_indicator” will always indicate that the NIT is current. “NIT other” or “next” NIT are not supported.

7.4.4.4 Construction of final generated transport stream

The final transport stream is generated by multiplexing together the dynamic and static transport stream components, which shall have been generated as described in 7.4.4.2 and 7.4.4.3, such that the components have the bitrates specified in their definitions.

If the first (or only) DVB multiplex doesn't specify an XML NIT then it behaves as if the following XML NIT was specified:

```
<?xml version="1.0" encoding="utf-8"?>
<nit xmlns="http://www.hbbtv.org/2016/nit"
     nid="99" version="0">
  <network>
    <networkNameDescriptor>HBBTV_A</networkNameDescriptor>
  </network>
  <transportStream onid="99" tsid="1">
    <autoDeliverySystemDescriptor />
    <serviceListDescriptor>
      <service sid="10" type="mpeg2-sd-tv"/>
      <service sid="11" type="mpeg2-sd-tv"/>
      <service sid="12" type="mpeg2-sd-tv"/>
      <service sid="13" type="mpeg2-sd-tv"/>
      <service sid="14" type="radio"/>
    </serviceListDescriptor>
    <privateDataSpecifierDescriptor value="40"/>
  </transportStream>
</nit>
```

If there is a second DVB multiplex and it doesn't specify an XML NIT then it behaves as if the following XML NIT was specified:

```
<?xml version="1.0" encoding="utf-8"?>
<nit xmlns="http://www.hbbtv.org/2016/nit"
     nid="65281" version="0">
  <network>
    <networkNameDescriptor>HBBTV_B</networkNameDescriptor>
  </network>
  <transportStream onid="99" tsid="2">
    <autoDeliverySystemDescriptor />
    <serviceListDescriptor>
      <service sid="15" type="mpeg2-sd-tv"/>
      <service sid="16" type="mpeg2-sd-tv"/>
      <service sid="17" type="mpeg2-sd-tv"/>
      <service sid="18" type="mpeg2-sd-tv"/>
      <service sid="19" type="radio"/>
    </serviceListDescriptor>
    <privateDataSpecifierDescriptor value="40"/>
  </transportStream>
</nit>
```

The generated transport stream should have a data rate matching that required by the parameters of the inserted delivery descriptors.

7.4.4.5 Recommended test harness behaviour

In order to support the widest range of DUTs and their different application caching models the test harness may choose to implement these behaviours. Not implementing these features may result in the test harness not being able to pass some valid implementations of the DUT without an alternative test strategy.

EXAMPLE: Some DUTs may implement an application caching model which relies on the AIT version numbering scheme. If this is not used then the DUT may need to be power cycled or service changed between tests.

7.4.4.5.1 AIT version offset

The version field of the generated transport stream packets containing the AIT should be set so as to be equivalent to the output of the following algorithm:

- test_run is a positive integer incremented by 1 every time the harness starts a test
- base_version is the value of the version attribute of the ait element, or 0 if no version attribute is defined
- version field in transport stream packets = (test_run modulo 4) × 8 + base_version

7.4.4.5.2 DSM-CC version offset

The module/DII version and the version field of the generated transport stream packets containing the DSM-CC should be set so as to be equivalent to the output of the following algorithm:

- test_run is a positive integer incremented by 1 every time the harness starts a test
- base_version is the value of the version attribute of the dsmcc element, or 0 if no version attribute is defined
- version field value = (test_run modulo 2) × 8 + base_version

7.4.4.5.3 PCR regeneration

The test harness may rewrite the PCR fields of the generated transport stream.

7.4.4.6 TOT/TDT synchronization

When the playout set includes a <synchronizeTotTdt/> tag this indicates that the test harness must ensure that the TOT and TDT tables in the Transport Stream are encoded so that the current UTC time is present. The time offset in the TOT shall not be changed. This must be maintained if the Transport Stream is looped during playout.

7.4.4.7 EIT time updating

When the playout set includes a <adjustEit/> tag this indicates that the test harness must update the event start times in the EIT sections in the Transport Stream.

In summary the goal is:

- Given a stream which was recorded at a specific time, the “baseStreamStartTime”, if we’re playing the stream out at a later date then the test harness has to adjust the EIT appropriately so that events that were “one hour after baseStreamStartTime” become “one hour after the test started”.
- It’s usually better if the event start times are a multiple of 1min (or even 5min), so there is a configurable “granularity” which controls the granularity of the adjustment. This defaults to 1min.
- This mechanism also allows the EIT section version numbers to be adjusted, so the RUT detects the change
- The test harness ensures that the generated EIT-schedule data complies with the DVB SI Guidelines. To do this, the EIT-schedule has to be pulled apart and regrouped into sections.

Test authors shall ensure that each playout set using <adjustEit/> uses the same section version number on all EIT-schedule sections relating to a single service. (I.e. a change in EIT-schedule data can only happen when the playout set is changed; but a change in EIT-pf can happen at any point in the stream).

The test harness shall update the EIT using the following process:

1) First calculate the ‘EIT Offset’, which is a positive number of seconds, by:

a) Calculate the number of seconds between the date/time given by the baseStreamStartTime attribute on the <adjustEit/> tag, and the actual time the harness starts running the test. (For tests with more than one playout set, note that this is the time the test is started, not the time the playout set is started).

b) Round that down to a multiple of the value given by the granularity attribute on the <adjustEit/> tag.

2) Prepare the updated EIT-schedule sections as follows:

a) Parse all EIT-schedule sections in the generated TS, and extract the events and section version number for each service. Also remember the maximum number of sections seen in a single TS packet on the EIT PID.

b) For every event extracted in step 1, adjust the start_time value in every event loop inside the EIT by adding ‘EIT Offset’ seconds to the encoded date/time. If the start_time does not encode a valid date/time (e.g. if all bits of the field are set to "1", as explicitly allowed by [18]) then it is not adjusted

c) For each service, increment the version_number by the value given by the incrementVersion attribute on the <adjustEit/> tag. The addition shall be done modulo 32.

d) For each service, generate EIT-schedule sections containing the events, following the rules in ETSI TS 101 211 to sort events and to assign events to sections. Note that calculation of “last midnight” must be based on the time the test is started

3) Update the TS using the following rules:

a) TS packets that are not on the EIT PID are not modified by this algorithm

b) EIT-schedule sections are replaced with the updated EIT-schedule sections from step 2

c) EIT-pf sections are adjusted by:

    a. Adjust the start_time value in every event loop inside the EIT by adding ‘EIT Offset’ seconds to the encoded date/time. If the start_time does not encode a valid date/time (e.g. if all bits of the field are set to "1", as explicitly allowed by [18]) then it is not adjusted

    b. Increment the version_number by the value given by the incrementVersion attribute on the <adjustEit/> tag. The addition shall be done modulo 32.

    c. Adjust the section CRCs accordingly. If the CRC was correct before the changes above, then the CRC shall be correct on the modified section. If the CRC did not match before the changes above, then the CRC shall not match on the modified section.

d) Other sections on the EIT PID are passed through unchanged

e) The repacking of sections into the EIT PID shall respect the maximum number of sections in a single TS packet as measured in step 2a.

f) Note that due to the repacking of sections on the EIT PID, some EIT-pf sections and non-EIT sections will slightly change position in the stream. This is expected to be negligible unless you are using extremely low bitrates for the EIT PID. (In the worst-case, the delay is approximately the size of 2 EIT sections).

7.4.5 JS-Function changePlayoutSet()

This call changes the current playout (and therefore may disable the network connection).

```
void changePlayoutSet(playoutSetId : integer, callback : function, callbackObject : object);
```

ARGS: **playoutSetId:** the id of the playout set to start playing out. If the playout set with this id is not defined for the test then the test shall fail.

callback/callbackObject: a callback function to invoke after the new playout has started (also see chapter 7.2.3).

As audio/video might show some artefacts, test case implementers should not perform any broadcast audio/video checks up to 5 seconds after a switch of playout set occurs.

Test implementers should not call this function when network connection is down, as it may fail when network connection is not available. In this case, this will cause the complete test to fail automatically.

NOTE: To switch from no network to an available network connection, use the timeout parameter in the playout set.

7.4.6 JS-Function setNetworkBandwidth()

This function restricts the maximum incoming application data permitted on the network interface of the device under test.

```
void setNetworkBandwidth(bitrate : int, callback : function, callbackObject : object);
```

ARGS: **bitrate:** the maximum throughput of application data into the device under test, in units of bits per second (bps). Values less than 1bps are not supported, and shall cause the test to fail.

callback/callbackObject: a callback function to invoke when the restriction has been applied (also see chapter 6.2.3 Callbacks).

Throughput is defined as the average number of bits per second passed in to the transmitting network interface at the application layer. I.e. the figure excludes all HTTP, TCP/UDP, IP, MAC, LLC, etc. overheads.

The restriction only applies to data received by the device under test (i.e. data sent by the test harness.) Transmission of data from the device under test shall not be restricted.

Calls to this function set an upper limit on the permitted throughput. The maximum throughput achievable from the test harness may be limited to a lower value by other factors (e.g. bus or CPU saturation.) Test authors should take this into account when deciding appropriate values for bitrate.

7.4.7 CICAM related JS functions

The behaviour of the functions defined below is undefined if the CAM element is not present in the implementation XML. In this case harnesses may cause tests to fail.

The table below shows which functions may be used for each configurable state of the CICAM (these functions are marked ●). Behaviour of functions if they are called other than as permitted by Table 5 is undefined, and the harness may abort the test.

JS-Function	Value of 'type' attribute of 'CAM' element		
	cspgcip	cip	ci
clearAuthentication	●	●	
setScramblingEnabled	●	●	
getScramblingState	●	●	
sendURI	●	●	
sendReply	●		
sendParentalControlInfo	●		
sendRightsInfo	●		
sendSystemInfo	●		
waitForMessage	●		

Table 5: Functions permitted for different CICAM configuration states

7.4.7.1 Error Handling

If an error occurs during the interaction between the harness and the CICAM then the harness shall halt execution of the test. The harness may display an error message to the operator, and may add a meaningful human readable error to the test report. Situations that will trigger this error handling shall include, but are not limited to:

- Invalid parameters in function calls

- Use of functions that communicate with the CICAM when the CICAM is not inserted in the host
- Use of functions that communicate with the CICAM when the 'CAM' XML element (see 5.2.1.7.5) is not included in the implementation XML.

7.4.7.2 Serialisation of 'cicam' methods

The functions described in the following sections all return immediately when called. The functionality is then executed asynchronously and the specified callback (if provided) is called on completion (see 7.2.3). The CICAM shall execute API call actions sequentially. As such, if the action of one function is still being carried out when another function is called, the action from the second function shall not be carried out until the first function has completed its interaction with the CICAM. If an API call causing a CICAM action is called while the CICAM is executing a previous action then the harness may cause the JavaScript callback associated with the running action to be executed either before or after execution of the second action is complete.

Otherwise functions shall be processed as normal.

7.4.7.3 Data types

ECMAScript (and JavaScript profiles) only define basic 'Number' and 'String' types. Several of the functions defined in this section require more restricted input. Rather than define new types the functions shall accept JavaScript primitives and may perform validation to assert that they are valid. (Test developers shall not use invalid values.) The types used in this document, the corresponding primitive type to be used in an implementation, and the valid ranges are shown in the table below.

Type in this document	JavaScript primitive type	Valid values
uint8	number	Integers in range 0 ≤ value ≤ 255
uint16	number	Integers in range 0 ≤ value ≤ 65535
hexBinary	string	See below

Table 6: Interpreting CICAM parameter types

If values are not valid then error handling behaviour shall be as defined in 7.4.7.1.

7.4.7.3.1 'hexBinary' type

The hexBinary type is intended to encode arbitrary binary data, with semantics as defined in 4.2.3.4.1.1.2 of [29] (i.e. as defined for xs:hexBinary type used in XML schemas).

7.4.7.4 JS functions

The APIs are all defined as methods on an object 'cicam' which is a property of the object returned by the HbbTVTestAPI constructor.

If a harness does not support use of a CICAM then the 'cicam' property may have the value 'undefined'. If the 'cicam' property does not have the value 'undefined' then this shall not be taken to mean that the harness supports the use of CICAM or that a CICAM is available. Tests shall request the use of a CICAM by the use of the XML syntax defined in 5.2.1.7.5.

The 'cicam' prefix is shown in the title of the following sections, but not in the function definitions.

7.4.7.4.1 JS-Function cicam.clearAuthentication

This function will clear all cached authentication context data, so at next insertion the CICAM shall carry out the full host authentication protocol. It may also clear other cached CICAM state, depending upon the harness implementation.

```
void clearAuthentication(callback : function, callbackObject : object)
```

ARGS: **callback / callbackObject**: callback function invoked if the function call succeeds

In addition to the above behaviour, calling this function is equivalent to calling the following methods:

- cspgcipSetScramblingEnabled(true)

- cspgicpSendURI(copy freely)

7.4.7.4.2 JS-Function cicam.setScramblingEnabled

This function allows the CICAM's descrambling feature to be disabled.

```
void setScramblingEnabled( enabled : boolean, callback : function, callbackObject : object )
```

ARGS: **enabled**: If 'true' then transport stream descrambling will be enabled, subject to the other requirements of the CI+ specification (e.g. even if this has been set to true, if host authentication fails then descrambling shall not take place). If 'false' then transport stream descrambling shall cease if it is currently being carried out, and not restart.

callback / callbackObject: callback function invoked if the function call succeeds

The state set by this function shall not be preserved across CICAM removal / re-insertion and CICAM reboots.

7.4.7.4.3 JS-Function cicam.getScramblingState

This function allows the retrieval of information about the current state of the CICAM.

```
void getScramblingState( callback : function, callbackObject : object )
```

ARGS: callback/callbackObject: a callback function to invoke when the information is available.

The callback function will be called with the following parameters:

```
callback(callbackObject, cspgcipInformationObject)
```
where the cspgcipInformationObject is an associative array containing the following key/value pairs, where each value is a boolean:

- tsDetected: true if a transport stream is being input to the CICAM, otherwise false
- descrambling: true if the CICAM is currently descrambling a transport stream

The value returned for 'descrambling' indicates that the CICAM has authenticated the host, has a transport stream input and has retrieved sufficient CA system information to configure and start its descrambler. It does not guarantee that descrambling is taking place correctly or that the output video is correctly descrambled: if this information is required then it must be confirmed with an analyzeScreenExtended or analyzeVideoExtended, as appropriate.

7.4.7.4.4 JS-Function cicam.sendReply

This function shall cause the CICAM to send a SAS_async_msg APDU to the host with command_id 0x02 (reply_msg [29]) and with contents structured as 4.2.3.4.1.1.4 of [29].

```
void sendReply( oipf_status : uint8, oipf_ca_vendor_specific_information : hexBinary,
ca_system_id : uint16, callback : function, callbackObject : object )
```

ARGS: **oipf_status**: value to be used for the oipf_status of the reply_msg as defined in 4.2.3.4.1.1.4 of [29]

oipf_ca_vendor_specific_information: value for the oipf_ca_vendor_specific_information field of the reply_msg. May be null, in which case no oipf_ca_vendor_specific_information shall be sent to the host.

ca_system_id: value to be used for the ca_system_id of the SAS_async_msg APDU encoding the reply_msg.

callback / callbackObject: callback function invoked if the function call succeeds

7.4.7.4.5 JS-Function cicam.sendParentalControlInfo

This function shall cause the CICAM to send a SAS_async_msg APDU to the host with command_id 0x03 (parental_control_info — 4.2.3.4.1.1.1 [29]) and with contents structured as 4.2.3.4.1.1.5 of [29].

```
void sendParentalControlInfo( oipf_access_status : uint8, oipf_rating_type : uint8,
oipf_rating_value : uint8, oipf_country_code : array, oipf_control_url : string, ca_system_id
: uint16,  callback : function, callbackObject : object )
```

ARGS: **oipf_access_status**: 0 or 1 (see 4.2.3.4.1.1.5 of [29])

oipf_rating_type: as defined in 4.2.3.4.1.1.5 of [29]

oipf_rating_value: as defined in 4.2.3.4.1.1.5 of [29]

oipf_country_code: JavaScript Array containing 0 or more uint16 values as specified in 4.2.3.4.1.1.5 of [29]. If the array length is 0 or the value of the parameter is null then no oipf_country_code shall be sent.

oipf_control_url: either null, or a string (see 4.2.3.4.1.1.5 of [29]). If null then no oipf_control_url shall be sent. The contents of oipf_control_url shall be encoded using UTF-8 by the harness for inclusion in the SAS_async_msg APDU.

ca_system_id: value to be used for the ca_system_id of the SAS_async_msg APDU encoding the reply_msg.

callback / callbackObject: callback function invoked if the function call succeeds

7.4.7.4.6 JS-Function cicam.sendRightsInfo

This function shall cause the CICAM to send a SAS_async_msg APDU to the host with command_id 0x04 (rights_info — 4.2.3.4.1.1.1 [29]) and with contents structured as 4.2.3.4.1.1.6 of [29].

```
void sendRightsInfo( oipf_access_status : uint8, oipf_rights_issuer_url : string, ca_system_id
: uint16,  callback : function, callbackObject : object )
```

ARGS: **oipf_access_status**: 0 or 1 (see 4.2.3.4.1.1.6 of [29])

oipf_rights_issuer_url: either null or a string as defined in 4.2.3.4.1.1.6 of [29]. If null then no oipf_rights_issuer_url shall be sent. The contents of oipf_rights_issuer_url shall be encoded using UTF-8 by the harness for inclusion in the SAS_async_msg APDU.

ca_system_id: value to be used for the ca_system_id of the SAS_async_msg APDU encoding the reply_msg.

callback / callbackObject: callback function invoked if the function call succeeds

7.4.7.4.7 JS-Function cicam.sendSystemInfo

This function shall cause the CICAM to send a SAS_async_msg APDU to the host with command_id 0x05 (system_info — 4.2.3.4.1.1.1 [29]) and with contents structured as 4.2.3.4.1.1.7 of [29].

```
void sendSystemInfo( oipf_ca_vendor_specific_information : hexBinary, ca_system_id : uint16,
callback : function, callbackObject : object )
```

ARGS: **oipf_ca_vendor_specific_information**: data to be sent for oipf_ca_vendor_specific_information data of system_info, as defined in 4.2.3.4.1.1.7 of [29].

ca_system_id: value to be used for the ca_system_id of the SAS_async_msg APDU encoding the reply_msg.

callback / callbackObject: callback function invoked if the function call succeeds

7.4.7.4.8 JS-Function cicam.waitForMessage

This function instructs the CICAM to wait for a SAS_async_msg APDU to be received from the terminal, and optionally respond with a SAS_async_msg encoding a reply_msg as 4.2.3.4.1.1.3 of [29].

```
void waitForMessage( timeout : integer, oipf_status : uint8,
oipf_ca_vendor_specific_information : hexBinary, ca_system_id : uint16, callback : function )
```

ARGS: **timeout**: a time (in milliseconds) after which the CICAM should stop waiting. Values of timeout must be in the range 1-300000. Values outside this range shall cause the test to fail.

oipf_status: If null then no reply_msg shall be sent. Otherwise a value to be used for the oipf_status of the reply_msg as defined in 4.2.3.4.1.1.4 of [29]

oipf_ca_vendor_specific_information: If oipf_status is not null then the value for the oipf_ca_vendor_specific_information field of the reply_msg. May be null, in which case no oipf_ca_vendor_specific_information field will be included in the reply_msg.

ca_system_id: If oipf_status is not null then the value to be used for the ca_system_id of the SAS_async_msg APDU encoding the reply_msg. May only be null if oipf_status is null.

callback/callbackObject: a callback function to invoke when the information is available.

The callback function will be called with the following parameters:

```
callback( waitState : number, receivedInformationObject : object, callbackObject : object )
```

The callback shall be called when:

- the CICAM has begun to wait for a message to be received. In this case the waitState parameter will have the value 1, and the receivedInformationObject parameter shall be null.
- a message has been received by the CICAM from the host. In this case the waitState parameter will have the value 2, and receivedInformationObject will be an associative array containing the following key/value pairs;
  - ca_system_id will give the value of ca_system_id in the received SAS_async_msg, as a JavaScript number.
  - oipf_ca_vendor_specific_information: the value of oipf_ca_vendor_specific_information from the received send_message, passed as a hexBinary string (see 7.4.7.3.1).
- the timeout expires. In this case the waitState parameter will have the value 3, and receivedInformationObject shall be null.

The callback shall be called a maximum of twice in response to any single call to waitForMessage, with each value of waitState passed at most once. The first call to the callback shall always be with a waitState value of 1.

NOTE: This function only waits for a single SAS_async_msg and only sends a single reply_msg.

7.4.7.4.9 JS-Function cicam.sendURI

This function shall cause the CICAM to initiate the Usage Rules Information (URI) refresh protocol (as specified in 5.7.5.1 of [14]) using the supplied URI.

```
void sendURI( uri : bytestring, callback : function, callbackObject : object )
```

ARGS: **uri**: a URI as defined in 5.7.5.2 of [14]

callback / callbackObject: callback function invoked if the function call succeeds

The URI specified in the function shall be in v2 format.

7.5 Additional notes

7.5.1 Test implementation guidelines

When implementing a test, the test case author should stick to the following guidelines as closely as possible. Failure to adhere to these guidelines may cause the Test Case to be rejected during the implementation review.

7.5.1.1 JS API implementation

Keep in mind that the testsuite.js file needs to be replaced by the test harness provider. Editing or replacing this file is not possible. Packaging this file into a pre-packaged DSM-CC (delivered as transport stream) is not possible. In case a pre-packaged DSM-CC needs to be created, the testsuite.js file needs to be referenced via a HTTP URL, e.g. http://hbbtv1.test/_TESTSUITE/RES/testsuite.js

In this case, the application shall contain a simple_application_boundary_descriptor giving access to the web server (in above case to http://hbbtv1.test/).

7.5.1.2 PASS requires endTest()

A test will only pass if endTest() is called, otherwise the test may time out (Test Harness dependent) or will be unresolved or failed. This makes the implementation of test cases that should pass if an application was killed a bit complicated, but not impossible, as the following solutions exist:

- Have an AIT with an AUTOSTART app that stores a cookie which counts the times that the application was started - doing this will make it possible to verify that the application was started twice, and therefore it must have been killed in between. This solution will only work if cookies are supported (only for broadband connection).
- Have an AIT with an AUTOSTART app that, as soon as it is suitable for the test to give the correct result, changes AIT on the current service (by changing the playout set). The new AIT then carries the following applications:
  - the currently running app as PRESENT, and
  - an AUTOSTART app which will be the app that is fired up once the currently running app has been killed to verify that the previous app got killed.
- By using multiple services (this avoids changing the playout set): service A contains the application 1 as AUTOSTART, which is responsible for performing the test. The application tunes to service B, which contains application 1 as PRESENT and application 2 as AUTOSTART, which will do the endTest() call. The application 1 will keep on running until it is killed, which then will start application 2.

7.5.1.3 Write tests for automation

Avoid using the JS API functions analyzeScreenExtended, analyzeAudioExtended, and analyzeVideoExtended. Whenever possible, design your test to use the functions analyzeScreenPixel and analyzeAudioFrequency instead, which support automation. Ideally Test Cases should not use analyze calls at all, as there are often other means of assessing the pass criteria.

7.5.1.4 Delayed analysis

All JS API analyze functions might be implemented in different ways:

- automated online analysis
- manual online analysis
- manual/automated offline analysis

For manual analysis, always make sure that the changes to the screen/audio/video only happen after the callback is received, which tells you that the analysis is done.

Calls to the analyze functions of the JS API shall be made only when network connection is enabled to allow the Testing API to trigger the capturing of the screen or audio signal. The test might fail otherwise

As analysis may happen offline, the analysis result is not known to the test implementation and will not be reported back in the callback. If analysis fails, the complete test will fail. So if you perform an analyze call (e.g. check whether a specific pixel has a red colour) that fails (pixel is not red), this will make sure that the complete test case fails.

7.5.1.5 Restoring network connection / Playout Set timeout

After changing to a playout set that has the network connection disabled, you should use the playout set's timeout feature to restore the network connection after a specified amount of time by switching to the next playout set after the timeout (which then should have the network connection enabled).

In most other cases, a playout set should not have a timeout in order to allow slow implementations to still pass the test.

Except for very special test cases (with a good reason not to do so), all playout sets shall signal and include all services ATE test 10 – ATE test 14 (see chapter7.2.2).

7.5.1.6 Always include basic SI tables in Playout Set definition

A playout set definition requires the referencing of a PAT, SDT, TDT/TOT. Whenever possible, use the SI table definitions from the default TS file RES/BROADCAST/TS/generic-HbbTV-Teststream.ts.

The playout set definition should NOT include a NIT, which is inserted by the test harness for the tested delivery type (e.g. DVB-S, DVB-C, or DVB-T).

In addition to that, the playout set shall contain the PMTs and referenced elementary streams for the various services that are used by your test.

7.5.1.7 Choose the correct application and organization ID for your application

Test applications should stick to the following ID guidelines:

- The organization ID should be the ID assigned to the HbbTV consortium. This is: 0x70.
- The application ID should be computed using the following algorithm:

1) Start by taking the local part of the test ID.

g) If it cannot be interpreted as a hex number, take the SHA-1 hex digest of the local part

h) Interpret the resulting string as a hex number.

2) When a test case requires more than one application ID, add multiples of 0x100 (256 decimal) for each additional application ID required.

3) While the resulting ID is greater than 0x3fff (16383 decimal), subtract 0x3fff from the ID.

4) If application is intended to be in signed range (for trusted API calls), add the offset 0x4000 to the resulting application ID.

7.5.1.8 Only register the key events you need

Only change the application's keyset object if actual interaction with the remote control is required. When changing the keyset, ensure that you only register the keys that are required for the test. Following these guidelines makes it easier for the test harness to return to the pre-defined state while the test is running.

7.5.1.9 Implementing portable OIPF / HbbTV applications

Where a test case is valid for both HbbTV and OIPF (as identified in the <appliesTo> element), care should be taken to ensure that implementations are compatible with both specification requirements. This means that implementations:

1) Must use the OIPF DOCTYPE. OIPF DAE only permits the 'strict' or 'transitional' XHTML doctypes, as defined in CEA-2014-A (Annex G)

2) Must EITHER have an initial page called index.html or index.cehtml in the test directory, OR must use the OIPF XML extensions in implementation.xml to name the initial page

3) Must not use any HbbTV extensions

7.5.1.10 reportStepResult stepID

The initial reportStepResult upon initialisation of an application should be used to indicate whether that application has started correctly. This should be 'true' except in the case of the application failing to correctly initialise, such as that which may cause an exception to be raised. The initial reportStepResult with stepID 0 should not be used to indicate an expected test failure, such as that caused be the current application being opened in error.

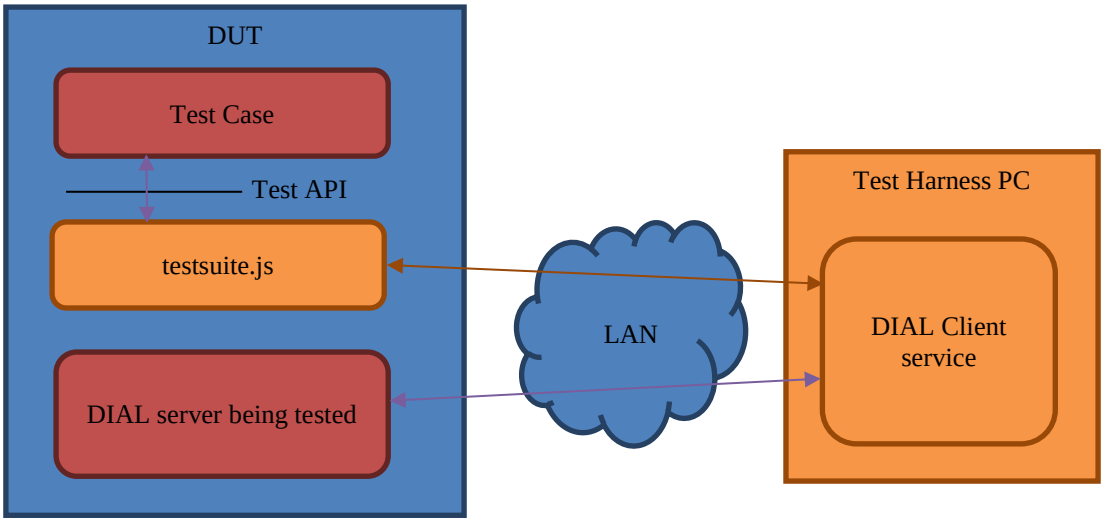
7.5.2 Things to keep in mind

When writing tests, the test implementers should keep the following information in mind:

- Analyze calls may be performed in offline mode. You don't know the analyze result when the callback is performed, you only know that you may continue with your test
- Wait for the callback when using an analyze call before changing the output (e.g. when calling analyzeScreenPixel wait until the callback has returned before changing the screen) to make sure that the analysis can be performed on the correct output.
- Make sure network connection is available when calling any API call except endTest(), reportStepResult(), sendMessage() and waitForCommunicationCompleted().
- If an analyze call fails, this means that the test fails. The test environment then may terminate the test while it continues to run, or it may wait until the test terminates itself
- HbbTV does not require monitoring of the AIT while broadband video is played back. A good test should not change the AIT during playback (only when testing very special cases).
- Before calling endTest(), the test should try to unregister all listeners and stop all broadband video playback. This makes it easier to run the following test.

7.6 APIs for testing DIAL

The Test Harness shall implement a partial DIAL Client, supporting DIAL [30] as profiled, clarified and extended by HbbTV [1]. The APIs in this section allow that DIAL Client to be controlled.



The diagram shows how a Test Case running on the DUT can make calls to testsuite.js to ask for DIAL operations to perform. The Test Harness uses some proprietary protocol (likely an XmlHttpRequest) to pass those requests from testsuite.js to the DIAL Client service that is part of the Test Harness running on the Test Harness PC. The DIAL Client service then actually does the requested DIAL operations (such as DIAL searches and sending DIAL requests). The Test Harness then communicates the result to the Test Case.

The supported DIAL operations include:

- Doing a search for DIAL servers on the network
- Parsing a URL reported by that search to get the URL
- Resolving hostnames in URLs reported by that search
- Getting the UPnP Device Description file from a DIAL server, which gets the URL for DIAL applications
- Getting the HbbTV Application Description from a DIAL server
- Starting an HbbTV Application via DIAL
- Checking that starting an HbbTV Application via DIAL is allowed from a different Origin

These APIs are accessed via the ‘dial’ property on the HbbTVTestAPI object created by the test. The ‘dial’ property is an object with (at least) the methods specified here.

7.6.1 JS-Function dial.doMSearch()

This function causes the test harness to send a SSDP M-SEARCH request to the DUT as defined in section 5.1 of DIAL [30], and get a response.

This function returns immediately. On success, it calls the provided callback function. On failure, the test harness shall cause the complete test to fail automatically and shall not call the callback. Failure can be caused by:

- no SSDP responses,
- multiple different SSDP responses when the harness has not been configured to use a specific one (see next two paragraphs), or
- no LOCATION header or otherwise malformed responses.

As defined in DIAL, the test harness may transmit the SSDP M-SEARCH request multiple times. This may result in multiple SSDP responses for the same device (i.e. with the same LOCATION header). In this case, the test harness shall accept such duplicate SSDP responses.

Optionally, a test harness may support filtering SSDP responses, so only the correct one is returned to the test. E.g. this may be useful if the Companion Screen happens to implement a DIAL server, and the Companion Screen is left on the network when tests using this API are run. If supported, the mechanism for configuring the specific SSDP response to use is harness-dependent and out of scope for this specification. Test harnesses are not required to implement filtering SSDP responses, because there are no tests that both use this API and require a real Companion Screen, so the tester could always turn the Companion Screen off while running tests that use this API.

```
void doMSearch(callback : function, callbackObject : object)
```

ARGS: **callback / callbackObject**: callback function invoked when the harness has a SSDP response as
`callback(callbackObject : object, locationHeader : string)`

where

locationHeader: value of the LOCATION header of the M-SEARCH response (i.e. the URL).

7.6.2 JS-Function dial.resolveIpV4Address ()

This function tries to resolve anything that is a valid hostname into an IPv4 address.

This function returns immediately. When resolving the address is finished, whether successful or not, it calls the provided callback function.

This function shall handle all the hostname formats that are allowed in URIs (see section 3.2.2 of RFC 3986 [22]). This means dotted decimal IPv4 addresses, IPv6 addresses enclosed in square brackets, and DNS names.

```
void resolveIpV4Address (hostname : string, callback : function, callbackObject : object)
```

ARGS: **hostname**: string defining the hostname.

callback / callbackObject: callback function invoked when the harness has finished resolving the IP address as:

```
callback(callbackObject : object, dottedIpv4 : string or null)
```
where,

dottedIpv4: string containing a dotted IPv4 address in the usual format (e.g. "123.245.67.189") if the specified hostname could be resolved, or null if the hostname was not resolvable. This address shall not have leading zeros on any component (i.e. it can be "1.2.0.3" but not "001.002.000.003").

Some examples that are not resolvable and shall call the callback with null include: IPv6 addresses enclosed in square brackets, DNS names that do not exist (i.e. NXDOMAIN response), DNS names that cannot be resolved due to DNS timeouts, and any string that cannot be interpreted as a valid hostname according to RFC 3986 [22].

Test implementers should not call this function when network connection is configured to be down, as it may fail when network connection is not available. In this case, this will cause the complete test to fail automatically.

7.6.3 JS-Function dial.getDeviceDescription()

This function does a HTTP GET for the UPnP device description file at the specified URL. This request shall conform to the DIAL specification [30] (see Section 5.3) as profiled in HBBTV [1] (section 14.7.2), except the harness shall follow up to 10 HTTP redirects. If the request fails (i.e. after following redirects if needed, the last request has anything other than a HTTP 200 response) then the harness shall fail the test automatically and shall not call the callback. If no response is received by the Test Harness within 10 seconds of making the HTTP GET request, the Test Harness shall fail the test automatically and shall not call the callback.

This function shall not do any Same-Origin or CORS [28] security checks. However, the parameters and results of this function are sufficient for a test case to test the DUT's Cross-Origin support (see HBBTV [1], Section 14.8).

```
void getDeviceDescription(url : string, origin : string or null, callback: function,
callbackObject : object)
```

ARGS: **url**: string defining the DIAL LOCATION URL. Typically, this URL will have been obtained from an earlier call to the dial.doMSearch() function.

origin: either null, in which case the request shall not include an Origin header, or else a string, in which case the request shall include an Origin header with the value specified in that string.

callback / callbackObject: callback function invoked when the harness has successfully completed the HTTP request as:

```
callback(callbackObject : object, applicationUrl : string or null, didRedirect :
bool, allowOrigin : string or null)
```

where,

applicationUrl: string containing the value of the Application¬-URL header returned with the UPNP device description file, or null if the DUT did not return that header.

didRedirect: true if the DUT did a HTTP redirect (in violation of the DIAL specification [30]), or false otherwise.

allowOrigin: string containing the value of the Access-Control-Allow-Origin header returned with the UPNP device description file, or null if the DUT did not return that header.

Test implementers should not call this function when network connection is configured to be down, as it may fail when network connection is not available. In this case, this will cause the complete test to fail automatically.

7.6.4 JS-Function dial.getHbbtvAppDescription()

This function does a HTTP GET for the DIAL HbbTV Application description at the specified URL. This request shall conform to the DIAL specification [30] (see Section 6.1.1). If the request fails then the harness shall fail the test automatically and shall not call the callback. HTTP redirects are prohibited and therefore if they are required this shall cause the request to fail. If no response is received by the Test Harness within 10 seconds of making the HTTP GET request, the Test Harness shall fail the test automatically and shall not call the callback.

This function shall not do any Same-Origin or CORS [28] security checks. However, the parameters and results of this function are sufficient for a test case to test the DUT's Cross-Origin support (see HBBTV 1.4.1 [1], Section 14.8).

```
void getHbbtvAppDescription(applicationUrl : string, schemaValidation : bool, origin : string
or null, callback: function, callbackObject : object)
```

ARGS: **applicationUrl**: string defining the DIAL Application Resource URL. Typically, this URL will be based on the DIAL Application URL obtained from an earlier call to the dial.getDeviceDescription() function, plus a trailing slash ('/') character if not already present, and followed by the Application Name 'HbbTV' (see [1], Section 14.7.2).

schemaValidation: if true, the harness shall do XML Schema Validation on the returned XML document, using the HbbTV DIAL Application resource schema (see [1], Section 14.7.2). In that case, if schema validation fails, then the harness shall fail the test automatically and shall not call the callback. Else if false the harness shall not do XML Schema Validation.

origin: either null, in which case the request shall not include an Origin header, or else a string, in which case the request shall include an Origin header with the value specified in that string.

callback / callbackObject: callback function invoked when the harness has successfully completed the HTTP request as:

```
callback(callbackObject : object, app2appUrl : string or null, interdevSyncUrl :
string or null, userAgent : string or null, allowOrigin : string or null)
```
where,

app2appUrl: string containing the value of the X_HbbTV_App2AppURL element from the HbbTV namespace in the DIAL HbbTV Application description, or null if the DUT did not return that element.

interdevSyncUrl: string containing the value of the X_HbbTV_InterDevSyncURL element from the HbbTV namespace in the DIAL HbbTV Application description, or null if the DUT did not return that element,

userAgent: string containing the value of the X_HbbTV_UserAgent element from the HbbTV namespace in the DIAL HbbTV Application description, or null if the DUT did not return that element.

allowOrigin: string containing the value of the Access-Control-Allow-Origin header returned with the DIAL HbbTV Application description, or null if the DUT did not return that header.

Test implementers should not call this function when network connection is configured to be down, as it may fail when network connection is not available. In this case, this will cause the complete test to fail automatically.

7.6.5 JS-Function dial.startHbbtvApp()

This function sends a HTTP POST to the DUT, to request it to start an HbbTV application. This request shall conform to the DIAL specification [30] (see Section 6.1.2). If the POST request cannot be made (e.g. web server rejects connection) then the harness shall fail the test automatically and shall not call the callback. Note that a HTTP response with an error status code shall be reported to the callback, it shall not cause the test to fail automatically. If no response is received by the Test Harness within 10 seconds of making the HTTP POST request, the Test Harness shall fail the test automatically and shall not call the callback.

This function shall not do any Same-Origin or CORS [28] security checks. However, the parameters and results of this function are sufficient for a test case to test the DUT's Cross-Origin support (see HBBTV 1.4.1 [1], Section 14.8).

```
void startHbbtvApp(applicationUrl : string, pathToAitXml : string, origin : string or null,
callback: function, callbackObject : object)
```

ARGS: **applicationUrl**: string defining the DIAL Application Resource URL. Typically, this URL will be based on the DIAL Application URL obtained from an earlier call to the dial.getDeviceDescription() function, plus a trailing slash ('/') character if not already present, and followed by the Application Name 'HbbTV' (see [1], Section 14.7.2).

pathToAitXml: string defining path to AIT XML relative to the directory containing the testcase XML file, using a trailing slash ('/') as a path separator. This XML file is sent to the DUT as the body of this HTTP POST.

origin: either null, in which case the request shall not include an Origin header, or else a string, in which case the request shall include an Origin header with the value specified in that string,

callback / callbackObject: callback function invoked when the harness has successfully completed the HTTP request as:

```
callback(callbackObject : object, returnCode : int, contentType : string or null,
body : string or null, allowOrigin : string or null)
```
where,

returnCode: integer return code in the HTTP response.

contentType: string containing the value of the content type header in the HTTP response, or null if a HTTP response is not received.

body: string containing the value of the body in the HTTP response, or null if a HTTP response is not received.

allowOrigin: string containing the value of the Access-Control-Allow-Origin header returned, or null if the DUT did not return that header.

Test implementers should not call this function when network connection is configured to be down, as it may fail when network connection is not available. In this case, this will cause the complete test to fail automatically.

7.6.6 JS-Function dial.sendOptionsRequest()

This function sends a HTTP OPTIONS request to the DUT, to check that starting an HbbTV app can be done cross-origin (see section 14.8 of [1]). If the OPTIONS request cannot be made (e.g. web server rejects connection, or HTTP error code) then the harness shall fail the test automatically and shall not call the callback. If no response is received by the Test Harness within 10 seconds of making the HTTP OPTIONS request, the Test Harness shall fail the test automatically and shall not call the callback.

This function shall not do any Same-Origin or CORS [28] security checks. However, the parameters and results of this function are sufficient for a test case to test the DUT's Cross-Origin support (see HBBTV 1.4.1 [1], Section 14.8).

```
void sendOptionsRequest(applicationUrl : string, origin : string or null, callback: function,
callbackObject : object)
```

ARGS: **applicationUrl**: string defining the DIAL Application Resource URL. Typically, this URL will be based on the DIAL Application URL obtained from an earlier call to the dial.getDeviceDescription() function, plus a trailing slash ('/') character if not already present, and followed by the Application Name 'HbbTV' (see [1], Section 14.7.2).

origin: either null, in which case the request shall not include an Origin header, or else a string, in which case the request shall include an Origin header with the value specified in that string.

callback / callbackObject: callback function invoked when the harness has successfully completed the HTTP request as:

```
callback(callbackObject : object, allowOrigin : string or null, maxAge : string or
null, allowMethods : string or null, allowHeaders : string or null)
```

where,

allowOrigin: string containing the value of the Access-Control-Allow-Origin header returned, or null if the DUT did not return that header.

maxAge: string containing the value of the Access-Control-Max-Age header returned, or null if the DUT did not return that header.

allowMethods: string containing the value of the Access-Control-Allow-Methods header returned, or null if the DUT did not return that header.

allowHeaders: string containing the value of the Access-Control-Allow-Headers header returned, or null if the DUT did not return that header.

Test implementers should not call this function when network connection is configured to be down, as it may fail when network connection is not available. In this case, this will cause the complete test to fail automatically.

7.7 APIs for Websockets

HbbTV uses Websockets for app2app communications with a Companion Screen, and for media synchronization with a Companion Screen. So the Test Case needs a way to make the Test Harness (in its role as a "fake Companion Screen") open a Websocket to the DUT.

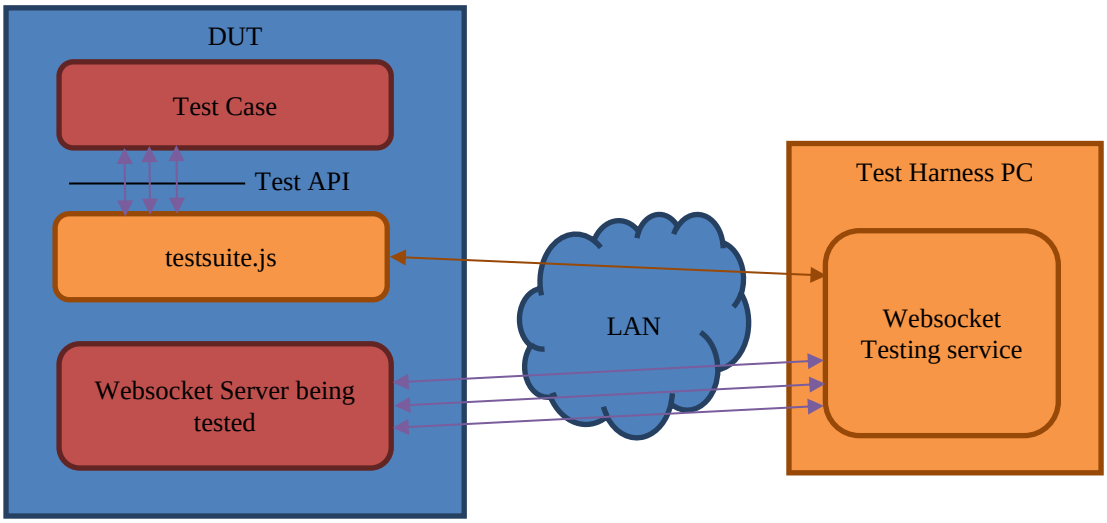
There are 3 reasons why test cases cannot just use the standard W3C Websockets API [i.6], which browsers provide and which is required by the HbbTV specification [1].

First, the W3C Websockets API is insufficient for certain HbbTV Test Cases. There are some extra low-level details that Test Cases need to be able to control and test:

- Sending ‘Ping’ frames and receiving the resulting ‘Pong’ frame
- Websocket extension testing (the ‘Sec-WebSocket-Extensions’ request and reply headers)
- Message fragmentation control for transmitted messages
- CORS headers (setting the ‘Origin’ header to arbitrary values and checking the ‘Access-Control-Allow-Origin’ header)
- Access to the HTTP status code when the WebSocket connection is refused

Second, if the Test Case is running as a harness-based test, the harness-based test environment is not required to provide the W3C Websockets API. Harness based tests must use this API instead.

Thirdly, if the Test Case is running on the DUT, it’s not sufficient to merely open a Websocket directly from the DUT to the media synchronization or app2app endpoints. We need to test that those endpoints are accessible via the network port on the DUT. So this API allows a Test Case to command the Test Harness to open Websockets and forward data to the test case:



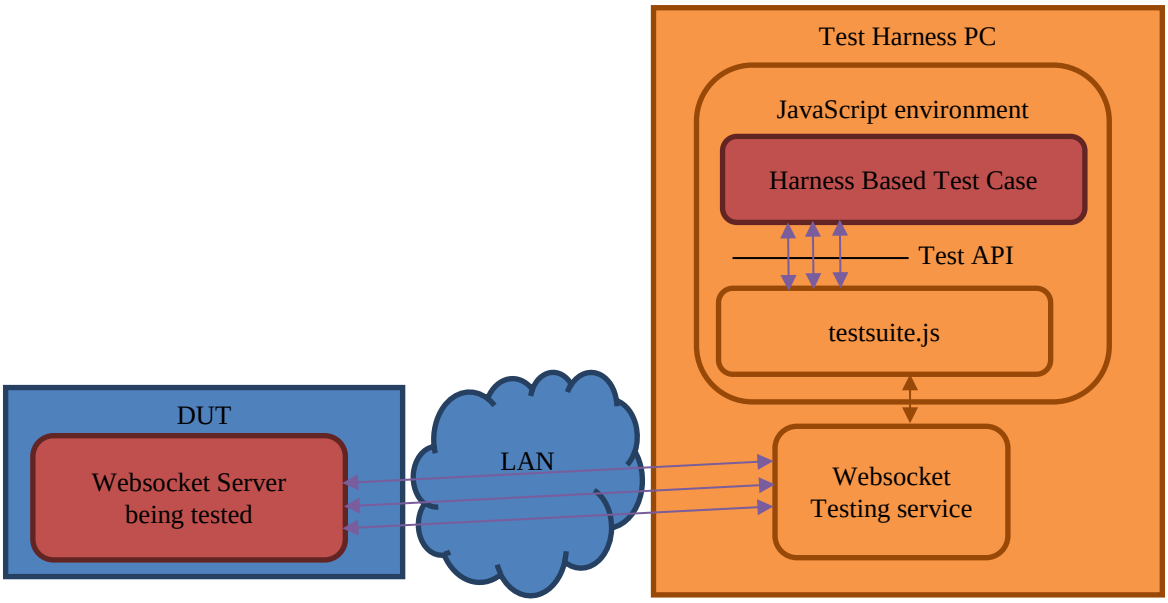
The diagram shows how a Test Case running on the DUT can make calls to testsuite.js to ask it to open Websockets. The Test Harness uses some proprietary protocol to pass those requests from testsuite.js to the Websocket Testing service that is running on the Test Harness PC. The Websocket Testing service then actually opens the Websockets. The Test Harness then forwards messages in both directions between the Test Case and the Websocket Server being tested.

Where this Test API specifies the details of Websocket messages, it refers to the Websocket between the Websocket Testing service and the Websocket Server being tested. It does NOT refer to the “proprietary protocol” part of the system, even if that happens to be implemented using a Websocket.

NOTE 1: The “proprietary protocol” is likely to be a Websocket connection. This should be a single Websocket connection, with all the requests multiplexed down it. It should NOT be one Websocket for each call to openWebsocket(). Using a single Websocket here ensures that, when the Test Case tries to open 400 Websockets at once, the only DUT restrictions that are hit are restrictions in the Websocket Server that’s being tested, not restrictions on how many Websockets you can open in the browser.

NOTE 2: The “proprietary protocol” is just an RPC mechanism – i.e. when the Test Case requests a particular Websocket message be sent with certain options, testsuite.js may just take the message and all the options, and send them as JSON down a Websocket to the Websocket Testing service.

This API can also be used by a test case running as a Harness Based Test:



For security reasons, if the harness follows the design described here, then when there is a new "proprietary protocol" Websocket connection to the the Websocket Testing Service in the Test Harness, that service should check that either the Origin header of the HTTP request is one of the test domains listed in section 5.2.1.2 (i.e. hbbtv1.test etc), or the connection is coming from a harness based test case. See RFC 6454 and the HbbTV specification [1] for details of how a web browser determines the Origin of a page.

NOTE 3: This means that test pages loaded from DSM-CC or from a server on the Internet cannot use the openWebsocket() API. Tests that use the openWebsocket() API must be HTML pages loaded from http://hbbtv1.test/ or from the other DNS names in section 5.2.1.2.

NOTE 4: The Websocket Testing service is a Websocket proxy that allows spoofing the Origin header. This is a powerful tool that we do not want to be used by attackers. The Origin check ensures that only Test Cases can use it.

Without the Origin check, there is the following security vulnerability: Suppose an organization happens to have a Test Harness installed somewhere. Suppose an attacker who wishes to attack that organization knows that they have a Test Harness installed, and knows either the IP address of the Test Harness or a range of IP addresses that will include the Test Harness, and knows the proprietary protocol used by that Test Harness. Further, suppose that the organization has an intranet server inside their firewall that uses Websockets. The attacker first tricks someone at the organization into visiting a malicious webpage (which is trivial for the attacker to arrange via a malicious link in an email). The malicious webpage instructs their desktop web browser to make a Websocket connection to the Test Harness, and then sends commands over that Websocket instructing the Test Harness to connect to a secure internal server using a spoofed Origin header. That allows the attacker to get Websocket access to an internal server (inside the corporate firewall). To prevent this vulnerability, the Test Harness has to reject the initial Websocket connection from the malicious page, based on the Origin header supplied by the browser.

Of course, an Origin check does not preclude connections made by non-browser based clients.

At the start of a testcase execution, there shall not be any Websockets open via this mechanism.

NOTE 5: When a test case stops running, and the terminal destroys the test application, the terminal will close any Websockets open via the normal W3C Websocket API. (See the W3C Websocket API spec for details). If the test harness uses a W3C Websocket in its implementation of this API, then the Websocket Testing service can detect that W3C Websocket being closed and close any related Websocket connections it's created.

For harness-based tests, a similar process can occur but it will be triggered by the harness-based test execution environment being torn down.

It is not possible for a test case to guarantee to close all Websocket connections made with this API in all possible failure cases. Therefore the harness must close them. Because the harness has to close them anyway, it's not worth writing extra code in every test case to close them in the success case and/or certain failure cases.

7.7.1 Encoding of binary data

Where binary data is returned from this API, it is encoded as follows: It is converted to a string by converting each byte in turn using these rules:

- Bytes 0x20 to 0x24 inclusive are mapped to the corresponding characters U+0020 to U+0024 inclusive.
- Bytes 0x26 to 0x7E inclusive are mapped to the corresponding characters U+0026 to U+007E inclusive.
- Other bytes are mapped to the three character sequence consisting of a percent character followed by two uppercase hex characters, e.g. byte 0 maps to "%00" and byte 0xAB maps to "%AB".

Where binary data is passed into this API, it is passed as a string which is decoded as follows:

- Characters U+0020 to U+0024 inclusive are mapped to the corresponding bytes 0x20 to 0x24 inclusive.
- Characters U+0026 to U+007E inclusive are mapped to the corresponding bytes 0x26 to 0x7E inclusive.
- The three character sequence consisting of a percent character followed by two uppercase hex characters maps to the corresponding byte, e.g. "%00" maps to byte 0 and "%AB" maps to byte 0xAB.
- A percent character that is not followed by two uppercase hex characters means the string is malformed.
- If the string contains a character outside the range U+0020 to U+007E inclusive then the string is malformed

Test Cases shall not pass malformed strings (as defined above) to APIs that are expecting to be able to convert the string to binary data. The handling of malformed strings is test-harness-dependent (but a reasonable choice would be to fail the test with an error).

NOTE 1: This uses a different string encoding from getPlayoutInformation. Experience has shown that test case authors try to print the strings and pass them to test API calls, which does not work for the binary strings returned by getPlayoutInformation. A human-readable string representation is easier to use. Additionally, the string returned by getPlayoutInformation is intended to be passed to an OIPF API that defines the encoding; that restriction does not apply here.

NOTE 2: These APIs uses a string encoding rather than an ArrayBuffer because that is simpler and easier for test case authors to use.

7.7.2 JS-Function openWebsocket()

This function establishes a WebSocket connection (RFC 6455 [31]) to the specified Websockets URL.

```
void openWebsocket(url : string, onConnect : function, onMessage : function, onPong :
function, onClose : function, onFail : function callbackObject : object, originHeader : string
or null, websocketsExtensionHeader: string or null)
```

ARGS: **url**: string defining the Websockets URL. If this URL does not start "ws://" then the connection shall fail, and the onFail callback shall be called.

onConnect: function called when "The WebSocket Connection is Established" as defined in RFC 6455 [31]. If the connection fails then this function will not be called – onFail will be called instead. If the connection succeeds, this is always the first callback that is called.

```
onConnect(callbackObject : object, websocketExtensionHeader : string or null,
websocket : WebsocketClient)
```
where,

websocketExtensionHeader: the value of the Sec-WebSocket-Extensions header sent by the server, or null if that header was not sent. If the server sends multiple Sec-WebSocket-Extensions headers, they are combined into a single value as described in RFC 6455 [31] section 9.1. This is the value sent by the server, regardless of whether the extensions are supported by the test harness or not.

websocket: a new WebsocketClient object.

onMessage: function called when "A WebSocket Message Has Been Received" as defined in RFC 6455 [31]:

```
onMessage(callbackObject : object, data : string or ArrayBuffer, binary : boolean,
websocket : WebsocketClient)
```
where,

data: if the message is a text message, 'data' is the message which has been decoded from UTF-8 into a JavaScript string. If the message is a binary message, then "data" is the message as an ArrayBuffer.

binary: true if the message is a binary message, or false if the message is a text message.

onPong: Called when a Pong frame is received. This may be an unsolicited Pong frame, or it may be a reply to a Ping frame that was sent via WebsocketClient.sendPing().

```
onPong(callbackObject : object, data : ArrayBuffer, websocket : WebsocketClient)
```
where,

data: the data from the Pong frame.

websocket: the WebsocketClient object.

onClose: function called when "The WebSocket Connection is Closed" as defined in RFC 6455 [31], except this is not called if onFail has already been called. Once this method is called, the test must not make any further method calls on this WebsocketClient object, and the harness must not invoke any other callbacks on this WebsocketClient object.

```
onClose(callbackObject : object, statusCode : integer, reason : string, websocket
: WebsocketClient)
```
where,

statusCode: "The WebSocket Connection Close Code" as defined in RFC 6455 [31]

reason: "The WebSocket Connection Close Reason" as defined in RFC 6455 [31]

websocket: the WebsocketClient object.

onFail: function called when the Websocket implementation has to "Fail the WebSocket Connection" as defined in RFC 6455 [31]. Once this method is called, the test must not make any further method calls on this WebsocketClient object, and the harness must not invoke any other callbacks on this WebsocketClient object.

```
onFail(callbackObject : object, statusCode : integer, reason : string, websocket :
WebsocketClient)
```
where,

statusCode: If the Websocket handshake fails due to the HTTP status code not being 101, then this shall equal the HTTP status code that was received (e.g. 503). In all other failure cases it shall equal null.

reason: A human-readable string describing the error. E.g. the test case may choose to fail the test and use this string as part of the failure reason.

websocket: the WebsocketClient object.

originHeader: string to be sent as the value of the WebSockets "Origin" header field. If it is null, or not specified, then that header is not sent.

websocketExtensionHeader: string to be sent as the value of the WebSockets "Sec-WebSocket-Extensions" header field. If it is null, or not specified, then that header is not sent. If this header specifies extensions that are not supported by the test harness, and the server decides to use such an extension, then the test harness shall proceed as if that extension had not been negotiated. (Informative note: In that case, depending on the extension, it is likely that the Websocket connection will be established successfully, but fail with a call to onFail as soon as the server sends data that is not valid according to RFC 6455 [31], but would have been valid according to that extension).

The WebsocketClient object passed to the onConnect() and other callbacks has the methods defined in the following subsections.

NOTE 1: The openWebsocket function does not return a WebsocketClient object.

Tests that use this openWebsocket() API must either be a web page with an Origin that is one of the test domains listed in section 5.2.1.2 (i.e. http://hbbtv1.test/ etc), or be a harness based test case. (See section 7.7 for rationale).

If the test harness receives a ping frame on a Websocket opened with this API, it shall automatically respond with a pong frame as defined in RFC 6455 [31]. Those ping requests and pong responses are not exposed via this API.

The test harness shall not send unsolicited pong frames. The test harness shall not send ping frames unless instructed to do so via WebsocketClient.sendPing().

The test harness is not required to implement any Websockets extensions.

This API shall support having at least 500 simultaneous open Websockets.

NOTE 2: For tests running on the DUT, if the Test Harness provided implementation uses Websockets internally, this implies some sort of multiplexing, because the DUT is only required to support 20 Websockets connections (See HbbTV [1] section 10.2.1).

A test case may call this function multiple times without waiting for a response, e.g. if testing what happens with 400 rapid Websocket connection attempts.

Test implementers should not call this function from a test case running on the DUT when the network connection is configured to be down. Test implementers should not take the network connection down when a test case running on the DUT still has a Websocket open vial this API. Those actions may cause the complete test to fail automatically. These restrictions do not apply to test cases running as a Harness Based Test.

7.7.3 JS-Function WebsocketClient.sendMessage()

This function sends a Websocket message.

```
WebsocketClient.sendMessage(data : string, binary : boolean)
```
or
```
WebsocketClient.sendMessage(data : string, binary : boolean, fragments : array of integers)
```
where,

data: If 'binary' is false, the message as a string, which will be UTF-8 encoded for transmission. If 'binary' is true, the message encoded as specified in section 7.7.1.

binary: true if the message should be sent as a binary message, or false if the message should be sent as a text message.

fragments: if null or not specified, guarantees not to fragment the message. If specified, it shall be an array of strictly positive integers that add up to the size of the message in bytes. The message is fragmented into the requested size fragments.

7.7.4 JS-Function WebsocketClient.sendPing()

This function sends a Ping frame (as per section 5.5.2 of the WebSocket protocol [31]) containing the requested data. If the server correctly replies with a Pong frame, then onPong() will be called.

```
WebsocketClient.sendPing(data : string)
```

Where,

data: string containing the binary data to use as the payload of the Ping frame, encoded as specified in section 7.7.1.

7.7.5 JS-Function WebsocketClient.close()

This function shall "Start the WebSocket Closing Handshake" as defined in RFC 6455 [31], with no code or reason. When the close is complete, the onClose callback will be called if the connection could be closed gracefully, or onFail will be called if the connection fails.

NOTE: It is always possible to close a Websocket connection by closing the underlying socket, so a close cannot fail as such. However, there is a 'graceful close' protocol defined by RFC 6455, that should be followed unless there is an error. In a 'graceful close', both sides send a Close frame and shut down the sending side of their side of their socket, then the socket is closed and onClose is called. If either side resorts to closing the underlying socket without sending a Close frame then onFail is called.

```
WebsocketClient.close()
```

After calling WebsocketClient.close() the test must not make any further method calls on this WebsocketClient object. However, the harness may invoke further callbacks on this WebsocketClient object (e.g. if a message is received before the server's close frame, then the onMessage callback will be called).

7.7.6 JS-Function WebsocketClient.tcpClose()

This function shall cause the harness to close the Websocket without sending a close frame but by simply closing the underlying TCP/IP socket.

```
WebsocketClient.tcpClose(callback: function, callbackObject : object)
```

ARGS: **callback / callbackObject**: callback function invoked when the Test Harness has successfully closed the Websocket's underlying TCP/IP socket.

```
callback(callbackObject : object, websocket : WebsocketClient)
```

where

websocket: the WebsocketClient object.

This function shall return immediately, and arrange for the harness to close the Websocket's underlying TCP/IP socket as soon as possible. When the underlying TCP/IP socket is closed, a callback will be called (see below for which callback).

After calling WebsocketClient.tcpClose() the test must not make any further method calls on this WebsocketClient object. However, the harness may invoke further callbacks on this WebsocketClient object to report events that happened before the socket was actually closed (e.g. if a message is received before the underlying socket was closed, then the onMessage callback will be called).

After tcpClose() is called, then either:

- If the Websocket connection fails or is gracefully closed before the harness could close the TCP/IP socket, then the onFail or onClose callback shall be called as normal. In that case, the callback passed to tcpClose() shall not be called.
- Otherwise, the harness closes the Websocket's underlying TCP/IP socket in response to the tcpClose() call, then calls the callback that was passed to tcpClose().In this case, the onFail and onClose callbacks shall not ever be called.

Once the harness has called onFail, onClose, or the callback passed to tcpClose(), then the harness shall not make any further callbacks for this WebsocketClient object.

7.8 APIs for Media Synchronization testing

7.8.1 JS-Function analyzeAvSync ()

This function checks the synchronization between two or three of: video, second video, subtitles, and audio. It is used with video and subtitles streams containing flashes, and audio streams containing tone bursts, as described in section 5.2.1.13.

```
void analyzeAvSync(
            step_number : integer,
            comment : string,
            checkLight1AgainstAudio : boolean,
            checkLight2AgainstAudio : boolean,
            checkLight1AgainstLight2 : boolean,
            maxDifferenceMillis : integer,
            callback : function,
            callbackObject : object)
```

ARGS: **stepId**: the step number that has been performed (same as stepId in reportStepResult).

comment: a comment from the test developer describing the purpose of the step

checkLight1AgainstAudio: if true, then light sensor 1 is checked against the audio (See section 5.2.1.12 for details of the light sensor positions).

checkLight2AgainstAudio: if true, then light sensor 2 is checked against the audio

checkLight1AgainstLight2: if true, then light sensor 1 is checked against light sensor 2

maxDifferenceMillis: maximum allowed synchronisation error, in milliseconds.

callback/callbackObject: a callback function to invoke when the observation is complete and, if the Test Harness chooses to do the analysis immediately, the analysis is also complete analysis was made successful (also see chapter 7.2.3 Callbacks). The analysis result is not passed back to the application. A failed analysis causes the test harness to fail the complete test, independent on the test result reported back by the reportStepResult function. This is due to the fact that the analysis can also be performed off-line on a taken recording.

The test case must set at least one of the 3 'check' parameters to true, and can set 1, 2 or all 3 of them to 'true'.

NOTE: "Check light sensor 1 against audio, and light sensor 2 against audio, with a 10ms tolerance" is NOT the same as "check all 3 with a 10ms tolerance", since if light sensor 1 flashes 8ms earlier than the audio tone, and light sensor 2 flashes 8ms later than the audio tone, then there is 16ms between light sensor 1 and light sensor 2, so the first example passes and the second example fails.

For a period of 15 seconds, starting within 5 seconds of the time this API is called, this monitors the selected light and audio sensors and checks that all the detected flashes and tones are synchronised within the specified tolerance.

In this description, the time of a flash or tone is the time of the mid-point of the flash or tone. (There is no requirement for flashes or tones to have the same length).

The decision about whether this analysis passes or fails is made as follows:

- If there is a flash or tone that does not have a corresponding flash or tone with a centre within the specified tolerance, and the centre of that flash or tone is more than the specified tolerance plus 100ms from both the start and end of the 15 second observation period, then the test fails. (Note: The special handling of the start and end of the observation period accounts for cases where the synchronization is not perfect but is within the specified tolerance, and the Test Harness starts monitoring just after a flash and just before the corresponding tone, so the tone is detected but the flash is missed).

- If there is no flash or tone detected for any period of 3.5 seconds within the 15 seconds observation period, then that also causes the test to fail. (Note: that means some flashes or tones are not being detected at all, either due to a failure of the DUT - perhaps it's not playing the media at all - or a failure in the test harness – perhaps the light sensor fell off the screen).

- If two flashes or two tones are detected from the same source within a 400ms window, that also causes the test to fail. (Note: that means spurious flashes have been detected, either due to a failure of the DUT or a failure in the test harness).

- If none of the above conditions apply, then this step passes.

NOTE: The algorithm above is designed to work with media files containing the repeating pattern of flashes and tones described in section 5.2.1.13. However, other patterns of flashes and tones can also be used.

The entry/entries in the Test Report XML for this step shall include the maximum measured difference in milliseconds between the flash/tone.

NOTE: The format of the entry in the Test Report XML is not defined. It is for visual reference only. Possible locations for the entry are the testStepComment or testStepData tags.

Test implementers should not call this function when network connection is configured to be down, as it may fail when network connection is not available. In this case, this will cause the complete test to fail automatically.

NOTE: This function checks the mid-point of the flash. Experimental results show that typical TV "picture improvement" processing can affect the detection of the flash start and end times. Experiments suggest that the start time of the flash is affected more than the mid-point is. Particularly when testing audio/video sync, choosing the mid-point makes the tests more robust.

7.8.2 JS-Function analyzeStartVideoGraphicsSync ()

This function checks the synchronization between graphics and either video or subtitles.

```
void analyzeStartVideoGraphicsSync(
            stepId : integer,
            comment : string,
            callback : function,
            callbackObject : object)
```

ARGS: **stepId**: the step number that has been performed (same as stepId in reportStepResult).

comment: a comment from the test developer describing the purpose of the step

callback/callbackObject: a callback function to invoke when the analysis was started (also see chapter 7.2.3 Callbacks). The callback function will be called with the following parameters: callback(callbackObject, analyzeFinishedVideoGraphicsSync) where analyzeFinishedVideoGraphicsSync is a function that can be called as defined below.

The analyzeFinishedVideoGraphicsSync function is provided via the callback, it is NOT a function on the HbbTVTestAPI class. (This design allows Test Harness implementers to return a closure that links the 'finished' call back to the corresponding 'start' call).

```
void analyzeFinishedVideoGraphicsSync(
            expectedDeltaMillis : integer,
            maxDifferenceMillis : integer,
            callback : function,
            callbackObject : object)
```

ARGS: **expectedDeltaMillis**: the number of milliseconds that the flash on light sensor 1 is expected to be ahead of the flash on light sensor 2. May be negative if light sensor 2 is expected to detect a flash first.

maxDifferenceMillis: maximum allowed error in the measured delta, in milliseconds.

callback/callbackObject: a callback function to invoke when the observation is complete and, if the Test Harness chooses to do the analysis immediately, the analysis is also complete (also see chapter 7.2.3 Callbacks). The analysis result is not passed back to the application. A failed analysis must causes the test harness to fail the complete test, independent on the test result reported back by the reportStepResult function. This is due to the fact that the analysis can also be performed off-line on a taken recording.

When calling these functions, the typical procedure is:

1) The Test Case arranges for a video player to be positioned so it completely covers light detection area 1, and a black graphics rectangle to be positioned so it completely covers light detection area 2. The video player may be larger than the light detection area, e.g. if the video contains other information to help debugging as well as the flash area.

2) The Test Case starts playback of a video with a single video flash at a known timeline position. (In this section 7.8.2, "timeline" should be read generally as referring to any way that the Test Case can know the video playback position; it does not refer to any particular way of doing that).

3) The Test Case calls analyzeStartVideoGraphicsSync()

4) That causes the Test Harness to start analysing the two light sensors

5) The Test Harness then calls the callback passed to analyzeStartVideoGraphicsSync(), to indicate that analysis has started.

6) The Test Case checks the video's timeline time. If the call to analyzeStartVideoGraphicsSync() took too long, and the video flash has been missed, then the Test Case calls reportStepResult() to fail the test case, and terminates.

7) The Test Case regularly monitors the video's timeline time. When the video's timeline time gets 'close' to the video flash time, the Test Case records the current video timeline time, initiates a graphics flash by turning the graphics rectangle white, and then records the current video timeline time again. (Note: It's not possible for an HbbTV application to reliably change the graphics at a particular timeline time, so we cannot use analyzeAvSync. The best we can do is 'close', but this procedure carefully measures and compensates for this error).

8) A short time later (typically 2-5 video frames later), the Test Case clears the graphics flash by turning the graphics rectangle black.

9) The Test Case waits for sufficient time for both the following to happen:

    a. the graphics changes to get through all output buffering and actually be displayed as light

    b. the video flash to happen, (remembering to add the acceptable synchronization inaccuracy to the time it's expected to happen), and that flash to get through all output buffering and actually be displayed as light.

10) The Test Case calculates the average of the two timeline times around when it actually initiated the graphics flash, and subtracts that from the timeline time the video flash is encoded, and converts the result to milliseconds. It may then need to apply some corrections (e.g. see the Note below). This is the expectedDeltaMillis.

11) The Test Case converts to milliseconds and adds up: the allowed tolerance as specified in the assertion, half the difference between the two timeline times measured around when it actually initiated the graphics flash, and any other corrections (e.g. see the Note below). This is the maxDifferenceMillis.

12) The Test Case calls the analyzeFinishedVideoGraphicsSync function provided by the Test Harness. The Test Case passes the calculated expectedDeltaMillis and maxDifferenceMillis values.

13) That causes the Test Harness to stop analysing the two light sensors

14) The Test Harness either:

    a. Determines that each light sensor detected a single flash, and the difference between the start times of the flashes was the expected value (within the specified tolerance). In this case, the analysis has passed, and the Test Harness calls the callback provided by the Test Case.

    b. Determines that the conditions in (a) do not hold. In this case, the analysis has failed, and the Test Harness shall fail the entire Test Case immediately and shall not call the callback.

    c. Records the information needed to make that determination for later off-line analysis (e.g. if using a video camera to capture the screen and doing manual analysis later), and calls the callback provided by the Test Case. In this case, the Test Case may be failed when the analysis is carried out, which may be long after the Test Case has finished executing.

The Test Case shall ensure that the time that the Test Harness is observing the light sensors (between step 4 and step 13 in the above procedure) is less than 20 seconds. If this limit is exceeded (because the Test Case does not call analyzeFinishedVideoGraphicsSync quickly enough), the Test Harness *may* fail the entire test case. In that case, if/when the Test Case does eventually call analyzeFinishedVideoGraphicsSync, the callback shall not be called. (Rationale: this places a limit on the memory the Test Harness needs to do the capture).

The Test Harness shall ensure that, given reasonable assumptions about the performance of the DUT, a call to analyzeStartVideoGraphicsSync() takes less than 5 seconds, measured from the start of the call until the callback function is called. The time the Test Harness starts analysing the light sensors shall be somewhere between those two points.

The Test Harness shall ensure that, given reasonable assumptions about the performance of the DUT, a call to analyzeFinishedVideoGraphicsSync takes less than 2 seconds, measured from the start of the call until the Test Harness stops analysing the two light sensors. However, there may be a significant delay before the Test Harness calls the callback – perhaps minutes if a human analysis is involved.

(Informative Rationale: the guarantees provided by these 2 paragraphs are needed to allow the Test Case author to choose where in the video file the flash should occur, and so the Test Case author can ensure they meet the 20 second limit. E.g. the Test Case author may place the flash at 12 seconds into the video file, and call analyzeFinishedVideoGraphicsSync at approximately 15 seconds into the video file. If after starting the video file they get a callback immediately to say playback has started, and then analyzeStartVideoGraphicsSync runs instantly, but analyzeFinishedVideoGraphicsSync takes 2 seconds to stop the observation, then they are still well inside the 20 second limit. Conversely, if after starting the video file it takes 5 seconds for the DUT to make the callback to say playback has started, and then 5sec for analyzeStartVideoGraphicsSync to run, then those steps are still complete 2 seconds before the flash).

If either light sensor detects no flashes, or if either light sensor detects more than 1 flash, then the test harness shall treat that as an analysis failure.

See section 5.2.1.12 for details of the light sensor positions. See section 5.2.1.13 for details of media requirements – of the two alternatives for media described in that section, this API requires media with a single flash.

NOTE: When calculating the tolerance to allow for this function, the Test Case should bear in mind that some of the HbbTV APIs give the time of the last video frame that was composited with graphics, and by the time the video/graphics compositor runs again it may or may not be onto the next video frame. (E.g. with 50fps video, a DUT with a 50fps display will always go onto the next video frame, and a DUT with a 200Hz display that chooses to composite graphics with video at 200Hz will only have a 25% chance of going onto the next video frame). If you are particularly unlucky, the video/graphics compositor may be running in parallel with your JavaScript that changes the graphics, so your graphics change may have to wait 2 video frames. To allow for this potential error of -0 / +2 video frames, you may wish to adjust the expected difference by 1 frame and increase the tolerance by 1 frame.

The entry/entries in the Test Report XML for this step shall include the measured difference in milliseconds between observed and expected times.

Test implementers should not call this function when network connection is configured to be down, as it may fail when network connection is not available. In this case, this will cause the complete test to fail automatically.

NOTE: This function checks the start time of the flash, unlike analyzeAvSync/analyzeAvNetSync. Experimental results show that typical TV "picture improvement" processing can affect the detection of the flash start and end times, and when testing audio/video sync, choosing the mid-point makes the tests more robust. However, that does not apply to this API because it is not testing audio; instead it is testing the video and graphics planes which will probably be equally affected. Choosing to test the start of the flash makes the test cases simpler, as the end of the graphics flash does not need to be timed accurately.

7.8.3 JS-Function analyzeAvNetSync()

For a DUT that is in inter-device synchronization master mode, this function checks the synchronization between the timeline reported by the DUT and video, audio and/or subtitles.

```
void analyzeAvNetSync(
            stepId : integer,
            comment : string,
            wcUrl : string,
            tsUrl : string,
            contentIdStem : string,
            timelineSelector : string,
            unitsPerTick : integer,
            unitsPerSecond : integer,
            patternStart : integer,
            patternRepeatLength : integer,
            pattern : integer[],
            checkLight1AgainstTimeline : boolean,
            checkLight2AgainstTimeline : boolean,
            checkAudioAgainstTimeline : boolean,
            maxDispersionMillis : integer,
            maxExtraDifferenceMillis : integer,
            callback : function,
            callbackObject : object)
```

ARGS: **stepId**: the step number that has been performed (same as stepId in reportStepResult).

comment: a comment from the test developer describing the purpose of the step

wcUrl: the URL of the CSS-WC service that the Test Harness shall connect to. The Test Case should obtain this value from the DUT's CSS-CII service using the Websockets API in section 7.7. This must be a "udp://" URL.

tsUrl: the URL of the CSS-TS service that the Test Harness shall connect to. The Test Case should obtain this value from the DUT's CSS-CII service. This must be a "ws://" URL.

contentIdStem: The "contentIdStem" the Test Harness shall specify in the Setup Data when it connects to the CSS-TS service. The Test Case may have this hardcoded, or it may retrieve it from the DUT's CSS-CII service. See [34] section 5.7.3 for the meaning of this parameter.

timelineSelector: The "timelineSelector" the Test Harness shall specify in the Setup Data when it connects to the CSS-TS service. The Test Case may have this hardcoded, or it may retrieve it from the DUT's CSS-CII service. See [34] section 5.7.3 for the meaning of this parameter.

unitsPerTick: The "unitsPerTick" that the Test Harness shall use to interpret the timeline it receives from the CSS-TS service. The Test Case may retrieve this from the DUT's CSS-CII service. See [34] section 5.5.9.5 for the meaning of this parameter.

unitsPerSecond: The "unitsPerSecond" that the Test Harness shall use to interpret the timeline it receives from the CSS-TS service. The Test Case may retrieve this from the DUT's CSS-CII service. See [34] section 5.5.9.5 for the meaning of this parameter.

patternStart: the time the repeating pattern of flashes and tones starts, in timeline ticks. This refers to the mid-point of the first flash and tone in the pattern.

patternRepeatLength: the length of the repeating pattern of flashes and tones, in timeline ticks

pattern: the mid-point times of the repeating pattern of flashes and tones, in timeline ticks

checkLight1AgainstTimeline: if true, then light sensor 1 is checked against the timeline. (See section 5.2.1.12 for details of the light sensor positions).

checkLight2AgainstTimeline: if true, then light sensor 2 is checked against the timeline.

checkAudioAgainstTimeline: if true, then the audio sensor is checked against the timeline.

maxDispersionMillis: maximum allowed CSS-WC dispersion, in milliseconds.

maxExtraDifferenceMillis: maximum allowed synchronisation error, in milliseconds, before accounting for inaccuracies in the CSS-WC time. The Test Harness shall calculate the CSS-WC dispersion and add the dispersion to maxExtraDifferenceMillis to get the maximum allowed synchronization error.

callback/**callbackObject**: a callback function to invoke when the observation is complete and, if the Test Harness chooses to do the analysis immediately, the analysis is also complete (also see chapter 7.2.3 Callbacks). The analysis result is not passed back to the application. A failed analysis causes the test harness to fail the complete test, independent on the test result reported back by the reportStepResult function. This is due to the fact that the analysis can also be performed off-line on a taken recording.

The Test Case must set at least one of the 3 'check…' options, and may set 1, 2, or all 3 of them.

This function shall return immediately.

The Test Harness shall ensure that, given reasonable assumptions about the performance of the DUT, a call to analyzeAvNetSync() starts analysing the light and/or audio sensor within 5 seconds from the start of the call. Note: That 5 seconds includes the time needed to do the initial synchronization of the Test Harness's clock with the CSS-WC server, and the time to establish the CSS-TS connection and get the first timeline information.

The Test Harness shall analyze the light and/or audio sensors for a period of 15 seconds.

The media playing on the DUT shall contain a repeating pattern of tones and flashes. This may be the pattern described in section 5.2.1.13, or it may be a different pattern. The Test Case describes the pattern to the Test Harness by specifying the 'patternStart', 'patternRepeatLength', and 'pattern' parameters. If there are N entries in the 'pattern' array, then the mid-points of the tones and flashes are expected at the following times (measured in timeline ticks):

- patternStart + pattern[0]
- patternStart + pattern[1]
- patternStart + pattern[2]
- …
- patternStart + pattern[N-2]
- patternStart + pattern[N-1]
- patternStart + patternRepeatLength + pattern[0]
- patternStart + patternRepeatLength + pattern[1]
- …
- patternStart + patternRepeatLength + pattern[N-2]
- patternStart + patternRepeatLength + pattern[N-1]
- patternStart + patternRepeatLength * 2 + pattern[0]
- patternStart + patternRepeatLength * 2 + pattern[1]

- …
- patternStart + patternRepeatLength * 2 + pattern[N-2]
- patternStart + patternRepeatLength * 2 + pattern[N-1]
- patternStart + patternRepeatLength * 3 + pattern[0]
- ...

The pattern keeps repeating until at least 20 seconds after this function was called. (The 20 seconds here is calculated from the 5 second limit to start observing the sensors plus the 15 seconds of observation).

All entries in the pattern[] array must be non-negative and strictly less than patternRepeatLength. The pattern[] array must be strictly increasing. I.e.:

0 <= pattern[0] < pattern[1] < pattern[2] …. < pattern[N-1] < patternRepeatLength.

The decision about whether this analysis passes or fails is made as follows:

- If the Test Harness could not obtain the time from the CSS-WC service on the DUT, the test fails.
- If the calculated CSS-WC dispersion, in milliseconds, exceeds maxDispersionMillis the test fails.
- If the Test Harness could not connect to the CSS-TS service or could not obtain the timeline from that service, then the test fails.
- If there is a flash or tone detected, and its mid-point is more than 100ms from both the start and end of the 15 second observation period, then the Test Harness checks the pattern to see if a flash or tone is expected at that time (within the calculated tolerance). If not, then the test fails.
- If there is a flash or tone that is expected, and the time it is expected is more than the calculated tolerance plus 100ms from both the start and end of the 15 second observation period, and there is no flash or tone detected at that time (within the calculated tolerance), then the test fails. (Note: The special handling of the start and end of the observation period accounts for cases where the synchronization is not perfect but is within the specified tolerance, and the Test Harness starts monitoring just before a flash is expected but just after the flash has happened, so the flash is missed).
- If two flashes or two tones are detected from the same source within a 400ms window, that also causes the test to fail. (Note: that means spurious flashes have been detected, either due to a failure of the DUT or a failure in the test harness).
- If none of the above conditions apply, then this step passes.

The Test Harness shall not send any Actual, Earliest or Latest Presentation Timestamp to the DUT's CSS-TS service.

The Test Harness shall close the CSS-TS connection when capture is complete, and before calling the callback.

The entry/entries in the Test Report XML for this step shall include the calculated CSS-WC dispersion, and the maximum observed difference in milliseconds between observed and expected times for a flash or beep.

Test implementers should not call this function when network connection is configured to be down, as it may fail when network connection is not available. In this case, this will cause the complete test to fail automatically.

NOTE: This function checks the mid-point of the flash. Experimental results show that typical TV "picture improvement" processing can affect the detection of the flash start and end times. Experiments suggest that the start time of the flash is affected more than the mid-point is. Particularly when testing audio/video sync, choosing the mid-point makes the tests more robust.

7.8.4 JS-Function startFakeSyncMaster()

Starts a fake inter-device synchronization master running on the Test Harness, which the DUT can connect to.

```
void startFakeSyncMaster(
            ciiData : object,
            timelineStartValue : integer,
            callback : function,
            callbackObject : object,
            timelineStartSeconds : integer or null,
            timelineStartMicroseconds : integer or null)
```

ARGS: **ciiData**: a JavaScript dictionary containing the data to be returned by the CSS-CII service, except the timeline service and wallclock service URLs do not need to be specified (and will be ignored if they are specified). The implementation of this API must convert the ciiData to JSON using the standard JavaScript JSON.stringify() method, with no replacer specified.

timelineStartValue: the initial value of the timeline time reported by the fake CSS-TS service.

callback/**callbackObject**: a callback function to invoke when the fake synchronization master is started (also see chapter 7.2.3 Callbacks). The callback function will be called with the following parameters: callback(callbackObject : object, ciiUrl : string, syncTestUrl : string)

timelineStartSeconds/**timelineStartMicroseconds**: If both these parameters are non-null, they represent a UNIX timestamp in the past. The microseconds value shall be in the range 0-999999 inclusive. This timestamp is interpreted using the same clock as getPlayoutStartTime(). This timestamp specifies when the timeline starts. If both these parameters are null, the timeline starts when the harness processes this API call, which must be after this function is called and before the callback is called. Test cases shall either specify both of these values as null or specify both of them as integers. Test cases shall not pass a time in the future.

In the description below, a timeline time is computed as follows:

- First, we have to determine the wallclock time when the timeline started. If the timelineStartSeconds and timelineStartMicroseconds parameters are non-null, then they specify this time directly. Otherwise, the test harness chooses the wallclock time when the timeline starts; this start time shall be after this function is called and before the harness calls the callback.

- Next, we have to choose a timeline. The possible timelines are listed in the 'timelines' array in the ciiData passed to startFakeSyncMaster(). For a connection to the CSS-TS service, the timeline will be chosen by comparing the 'timelineSelector' in the setup data against the 'timelineSelector' from each of the entries in the 'timelines' array; the matching entry is used. For the syncTestUrl service, the first timeline in the 'timelines' array is always used. A Test Case that does not list any timelines in the 'timelines' array must not use the syncTestUrl.

- Then the timeline time is calculated as follows: measure the time since the timeline started, in milliseconds; multiply that by 'unitsPerSecond', divide by 'unitsPerTick' and divide by 1000; round the result down to an integer (rounding towards negative infinity); add 'timelineStartValue'. For this calculation. 'unitsPerTick' and 'unitsPerSecond' are taken from the matched entry in the 'timelines' array.

The ciiUrl parameter passed to the callback is the URL of the CSS-CII service provided by the Test Harness. The Test Case can use this URL to configure the DUT as a slave.

The Test Case can make a Websocket connection to the Test Harness using the URL passed as the syncTestUrl parameter to the callback. The protocol on this Websocket is as follows: The only valid message from the Test Case to the Test Harness is a text message where the payload is the string "X". Sending that message causes the Test Harness to respond with the timeline time that the X was received, calculated as described above, as a single decimal integer encoded as text in a Websocket message. The Test Harness will not send any other messages over the Websocket. If the Test Case sends any other message over the Websocket then the behaviour of the Test Harness is not defined by this specification. Future versions of this specification may define other messages.

The fake master services provided by the Test Harness behaves as follows:

- CSS-CII: When a client connects, sends a single notification containing the ciiData specified in the call to startFakeSyncMaster(), with the addition of the correct URLs for the fake CSS-TS and CSS-WC services. Never reports any changes (i.e. does not send any other notifications).

- CSS-WC: Reports a wallclock time advancing at the correct rate. This service shall meet the requirements of HbbTV [1] section 13.7.2 and 13.7.3, with the word "terminal" replaced by "Test Harness".

- CSS-TS: Behaves as follows:

    1. On getting a connection, it waits for the client to send setup data

    2. It checks if the 'contentIdStem' in the setup data matches or is a prefix of the 'contentId' specified in the ciiData passed to startFakeSyncMaster(). It also checks if the 'timelineSelector' in the setup data matches the 'timelineSelector' from one of the entries in the 'timelines' array specified in the ciiData passed to startFakeSyncMaster().

    3. If both the conditions from step 2 hold, then: Within 500ms of receiving the setup data it sends a Control Timestamp to the client. This shall contain the current 'wallClockTime'. It shall contain 'timelineSpeedMultiplier' set to 1. It shall also contain a 'contentTime' that is a timeline time calculated as described above. The harness shall ensure that the wallClockTime and contentTime refer to the same instant in time.

    4. If the conditions from step 2 do not both hold, then: Within 500ms of receiving the setup data it sends a Control Timestamp to the client. This shall contain the current 'wallClockTime'. It shall contain a 'contentTime' and 'timelineSpeedMultiplier' which are both null.

    5. Any Actual, Earliest and Latest Presentation Timestamp sent by the client are read and ignored

The Test Harness shall automatically terminate the fake inter-device synchronization master when it stops running the test, before starting the next test.

Test implementers should not call this function when network connection is configured to be down, as it may fail when network connection is not available. In this case, this will cause the complete test to fail automatically.

7.8.5 JS-Function getPlayoutStartTime()

Gets the time that DVB TS playout started. This is the time that the first byte of the relevant DVB TS was sent out of the modulator as a RF signal. The "relevant" DVB TS is the TS generated by the Test Harness from the current playoutset,

```
void getPlayoutStartTime(
            multiplexIndex : integer,
            callback : function,
            callbackObject : object)
```

ARGS: **multiplexIndex**: the index of the multiplex to query. The Test Case must pass zero for this parameter to refer to the first multiplex and must pass one to refer to the second multiplex. If the Test Case specifies an invalid value, the Test Harness may fail the complete test case.

callback/callbackObject: a callback function to invoke with the playout start time (also see chapter 7.2.3 Callbacks). The callback function will be called with the following parameters: callback(callbackObject : object, seconds : integer, microseconds : integer, maxErrorMicroseconds : integer)

This function returns immediately and arranges for the callback to be called as soon as possible.

NOTE 1: Some implementations may involve asking the user to type in the value, so "as soon as possible" may be in a minute or two. If the Test Harness automates TS playout then it may be able to respond in significantly less than a second.

The timestamp passed to the callback specifies a UNIX timestamp, split into an integer number of seconds and 0 to 999999 microseconds. The Test Harness must ensure this time is accurate within +/- 250 milliseconds. The error bound shall be passed to the test as maxErrorMicroseconds, which must be between 0 and 250000 inclusive. This allows the Test Case to make allowance for the error.

NOTE 2: Although this call allows microsecond precision, the value is not expected to be that accurate. If a human is starting the playout, an accuracy of +/- 250 millisecond is achievable. If the Test Harness automates TS playout then it may be able to report the value more accurately.

PHP scripts in the Test Case can access a clock via the PHP time() or microtime() functions. The Test Harness must ensure that time(), microtime(), and getPlayoutStartTime() use the same clock or synchronized clocks. Any potential error due to the clocks not being perfectly synchronized must be accounted for in the value of maxErrorMicroseconds reported by the Test Harness.

NOTE 3: A Test Case may use this to synchronise the availability of DASH segments with the broadcast timeline. To do that:

1. The Test Case calls this JavaScript function, then when the callback is called it passes the timestamp to a first PHP script via XmlHttpRequest, and that first PHP script saves the timestamp in the PHP session.

2. Then the Test Case plays the DASH video, with the MPD URL pointing to a second PHP script. The second PHP script can calculate the segment availability start and end times based on the timestamp in the PHP session and the Test Case's chosen constraints on segment availability, and can include those values in the returned MPD data.

3. The MPD data can contain appropriate timing elements to ensure the DASH player requests the time from the Test Harness via a HTTP call to a third PHP script, which can use time() or microtime() to get the current time. This ensures that the DASH availability start/end times are compared against the correct clock.

4. If necessary, the segment URL in the MPD can point to a fourth PHP script. This PHP script can calculate the segment availability start and end times based on the timestamp in the PHP session, the position of the segment in the DASH video, and the Test Case's chosen constraints on segment availability. This PHP script can then check the current time (obtained using microtime()) against those segment availability times, to determine if the segment should be available or not.

Test implementers should not call this function when network connection is configured to be down, as it may fail when network connection is not available. In this case, this will cause the complete test to fail automatically.

7.8.6 JS-Function analyzeCssWcPerformance()

Causes the Test Harness to make a number of requests to the specified CSS-WC server and check that the terminal responds to each request within a defined time limit.

```
void analyzeCssWcPerformance(
            stepId : integer,
            comment : string,
            cssWcUrl : string,
            numberOfClients : integer,
            numberOfMessages : integer,
            transmitDurationMillis : integer,
            maxDroppedRequests : integer,
            maxResponseTimeMillis : integer,
            callback : function,
            callbackObject : object)
```

ARGS: **stepId**: the step number that has been performed (same as stepId in reportStepResult).

comment: a comment from the test developer describing the purpose of the step

cssWcUrl: URL to the CSS-WC server to be tested. This must be a udp:// URL as used in HbbTV.

numberOfClients: Number of distinct clients to create. Each IP address and port combination is considered a single distinct client. Must be positive and nonzero.

numberOfMessages: Total number of messages to send. Must be positive and nonzero.

transmitDurationMillis: The duration of the "transmit window" when CSS-WC requests are being sent by the Test Harness. Note that the responses to the last few requests will be received after the end of the transmit window.

maxDroppedRequests: The maximum number of CSS-WC requests that can be "dropped" (defined below) if the analysis step is to pass.

maxResponseTimeMillis: If the time taken to receive a CSS-WC response is greater than this value, the analysis step fails. This is measured from the time the CSS-WC request is sent on the network until the CSS-WC response is received on the network. In the case of a 2 part response where both parts are received in the correct order,

callback/**callbackObject**: a callback function to invoke when the analysis is complete (also see chapter 7.2.3 Callbacks). The analysis result is not passed back to the application. A failed analysis causes the test harness to fail the complete test, independent on the test result reported back by the reportStepResult function. If the analysis step fails, then the callback shall not be called.

The test harness shall send **numberOfMessages** CSS-WC requests to the specified CSS-WC server.

If **numberOfMessages** is 1, then the test harness shall immediately send a single CSS-WC request and **transmitDurationMillis** is ignored; in this case the transmit window has a duration of zero.

Otherwise, the test harness shall send a CSS-WC request immediately, and shall send new CSS-WC requests are every (**transmitDurationMillis** / (**numberOfMessages** - 1)) milliseconds. The first **numberOfClients** requests are all sent by distinct clients. After that, they repeat, with the client than sent the Nth message also sending the (N + **numberOfClients**)th message.

All CSS-WC messages sent by the server shall use a 64-bit random number for the originate_timevalue field. This shall be generated by a good-quality random number generator. The test harness shall ensure that all the originate_timevalue values used during a single call to analyzeCssWcPerformance() are unique.

After the end of the transmit window, the Test Harness shall continue to listen for responses for (2 * **maxResponseTimeMillis**) milliseconds.

The Test Harness shall consider the analysis step to have failed if any of the following conditions occur:

- Any UDP packet, received for the IP address and port of any of the CSS-WC clients created by this call, does not match the syntax defined for CSS-WC responses, or does not have a message_type of 1, 2, or 3.
- Any CSS-WC response has an originate_timevalue field that does not match a CSS-WC request sent by that client. This includes the case where the response is received on the wrong IP address or port.
- More than two CSS-WC responses have an originate_timevalue field that match the same CSS-WC request sent by that client.
- Two CSS-WC responses have an originate_timevalue field that match the same CSS-WC request sent by that client, and the first one received by the Harness is not message_type 2 (“response that will be followed by a follow-up response”).
- Two CSS-WC responses have an originate_timevalue field that match the same CSS-WC request by that client, and the second one received by the Harness is not message_type 3 (“follow-up response”).
- If the number of CSS-WC requests that are "dropped" is greater than **maxDroppedRequests**. A request is considered to have been "dropped" if the Test Harness does not receive a CSS-WC response with an ID matching that request, or if the Test Harness receives a response with message_type 2 but does not receive the corresponding follow-up response with message type 3, or if the Test Harness receives a response with message_type 3 but does not receive the corresponding response with message type 2.
- If the "response time" for any CSS-WC request exceeds **maxResponseTimeMillis** milliseconds. Requests that are dropped are not counted. The “response time” for a CSS-WC request is measured

from the time the CSS-WC request is sent on the network until the corresponding CSS-WC response is received on the network. In the case of a 2 part response where both parts are received in the correct order, the arrival time of the second part of the response is measured.

The Test Harness shall consider the analysis step to have passed if none of the above conditions occur.

The entry/entries in the Test Report XML for this step shall include the number of dropped requests, and the maximum CSS-WC response time that was observed during the analysis.

Test harnesses are only required to support the following values for the parameters:

numberOfClients = 5

numberOfMessages = 25

transmitDurationMillis = 1000

maxDroppedRequests = 0

maxResponseTimeMillis = 200

Support for other values for these parameters is optional in a Test Harness. (It may be helpful when debugging test failures, or for future tests).

7.9 APIs for network testing

7.9.1 JS-Function analyzeNetworkLog()

For certain tests, analyzing network log is necessary in order to check whether expected network issues are occurring. This function allows test developers to invoke recording of network traffic and to specify type of analysis that should be performed on data recorded by network analysis tool (described in section 5.2.1.15).

```
void analyzeNetworkLog (
                stepId : integer,
                expectedBehavior : integer,
                URL : String,
                timeout : integer,
                comment : String,
                check : String,
                minLogCount : integer,
                maxLogCount : integer,
                analysisStartedCB : function,
                callback : function,
                callbackObject : object);
```

ARGS: **stepId**: the step number that has been performed (same as stepId in reportStepResult).

expectedBehaviour: Type of behaviour that is expected to see when analyzing network log (integer that represent type of behaviour, see the table below):

Expected behaviour (name of constant)	Integer value	Description
RESOLVED_IP_UNREACHABLE	1	Attempt to access URL results in network log entry showing request to an IP address that is not reachable
DNS_FAIL	2	Attempt to access URL results in network log entry showing a DNS request for the hostname from the URL,

		and a corresponding NXDOMAIN DNS response

If *expectedBehavior* is set to 0, manual analysis according to *check* parameter should be performed.

URL - URL that should be tracked in network log

timeout – Time in milliseconds to log network traffic. Logging of network traffic starts sometime after analyzeNetworkLog() was called but before the callback function analysisStartedCB() is invoked, and runs for this length of time.

comment: a comment from the test developer describing what the analysis actually does (same as reportStepResult)

check: a textual description detailing which checks to perform on the network log. This is the only criteria that shall be used for the assessment of this analysis call. This allows the network log to be recorded (duration specified by timeout parameter) and then processed later on. An empty or null string shall cause the test to fail.

minLogCount: Minimum number of URL occurrences related to expected behaviour found in network log in timeout interval. If number of occurrences is below this number, test automatically fails.

maxLogCount: Maximum number of URL occurrences related to expected behaviour found in network log in timeout interval. If number of occurrences is above this number, test automatically fails.

analysisStartedCB: a callback function which is invoked once that network log capturing/monitoring has been started. Function does not have any parameters. In case of manual analysis, this function is invoked once that tester has started analysis tool and acknowledged to be ready for analysis to test harness through provided UI element. In case of automatic analysis, function is invoked once that harness starts recording/analysis of the network log. This callback function is suitable place in which test implementer should initiate process which causes network activity that needs to be tracked (for example, starts DASH playback).

callback/callbackObject: a callback function to invoke when the analysis was made (also see chapter 7.2.3 Callbacks). The analysis result is not passed back to the application. A failed analysis must cause the complete test to fail, independent on the test result reported back by the reportStepResult function. This is due to the fact that the analysis can also be performed off-line on a taken recording network traffic log.

By using argument ***expectedBehavior***, some of predefined types of network problems may be detected automatically, while ***check*** arguments allows test developers to specify description of the manual log check that should be performed by tester, if automation is not available.

Since analysis or log recording for later analysis may be performed by tester, system needs to allow tester to start and setup network analysis application, as well as to stop it. Test harness first needs to prompt tester to start analysis and wait for tester to confirm that analysis has been started, which leads to invocation of the function ***analysisStartedCB***.

Just before invocation of callback function ***analysisStartedCB***, network log recording/analysis was started, and it should end after ***timeout*** milliseconds. After specified timeout, if analysis is manual, test harness should prompt tester to stop recording/analysis of the network log. Once that tester confirms analysis completion, ***callback*** function is invoked.

Invocation of the callback does not mean that analysis was successful (it may be postponed for later stage). In case when specified network problem occurs less times than specified by ***minLogCount*** or more times than specified by ***maxLogCount***, test harness should automatically fail the test.

8 Versioning

This chapter contains the version control rules for the Test Specification document, the Test Cases (XML files incl. test material) and the Test Suite.

The version control of these parts is affected by the following events:

1) Challenges to existing test cases, see [27].

2) Filling gaps where test material is missing.

3) Areas where the Test Specification isn't detailed enough.

4) New versions of the Technical Specification (system specification)

8.1 Versioning of Technical Specification and Test Specification documents

Each HbbTV Test Specification shall contain the associated HbbTV Technical Specification Version in its title.

Test Specifications under development shall indicate their intermediate status by a draft number.

When a new version of the HbbTV Technical Specification has been created and approved, a new version of the Test Specification document shall be created.

The Test Specification document also includes references to the several parts of the Test Suite. This includes a list containing all necessary Test Cases (stored in the HbbTV Test Repository) which must be successfully performed by the DUT, in order to finish the technical certification process. This list is "List_of_approved_Test_Material__x_xx.txt", where "x_xx" in the filename refers to the Test Suite Version.

During the incremental process of HbbTV testing, above mentioned events 1, 2 or 3 might happen. This means that either:

- New Test Cases and new test material is created,
- Existing Test Cases are refined or
- Existing Test Cases are challenged the therefore excluded from the list of approved Test Material documents.

All this will lead to a new version of the Test Specification document including the list of approved Test Material which refers to the applicable Test Cases stored in the Test Repository.

8.1.1 Initial status of the Test Specification

The first formal released HbbTV Test specification shall start with a version number 1. Figure 7 depicts an example of the version structure of the initial test spec. release.

In this example the HbbTV technical specification has a version 1.1.1, which is bound to a version 1 HbbTV test specification which makes use of Test Cases that shall have a version 1.

Test Specification and Test Case version numbers shall only have integer numbers.

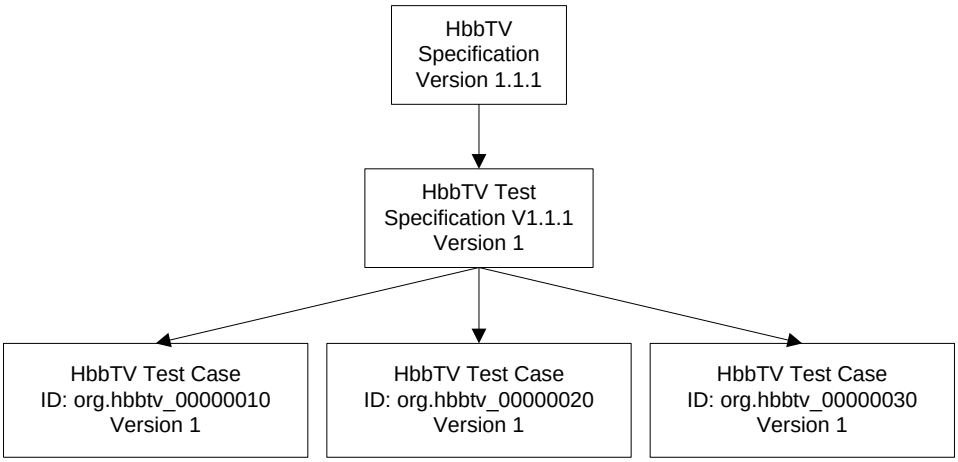


Figure 7: Version structure of the initial Test Specification release

8.1.2 Updating Test Specification, keeping the existing Technical Specification version

Test Cases may be developed to further improve the compliance testing and certification process:

- Test Cases may be improved, upgraded or corrected due to a challenge. In this case the ID number will be kept identical and the version number increases.
- Test Cases may be added to further increase the coverage of testing. In this case a new ID number will be assigned and the version number starts with 1.
- Test Cases may be deleted due to a successful challenge without a replacement.

Existing ID numbers from deleted Test Cases shall not be reused.

In any of the above cases a new version of the Test Specification shall be created and the existing version shall be made obsolete once the new version Test Specification has been formally released.

An example of the creation of a new Test Specification version is indicated in Figure 8.

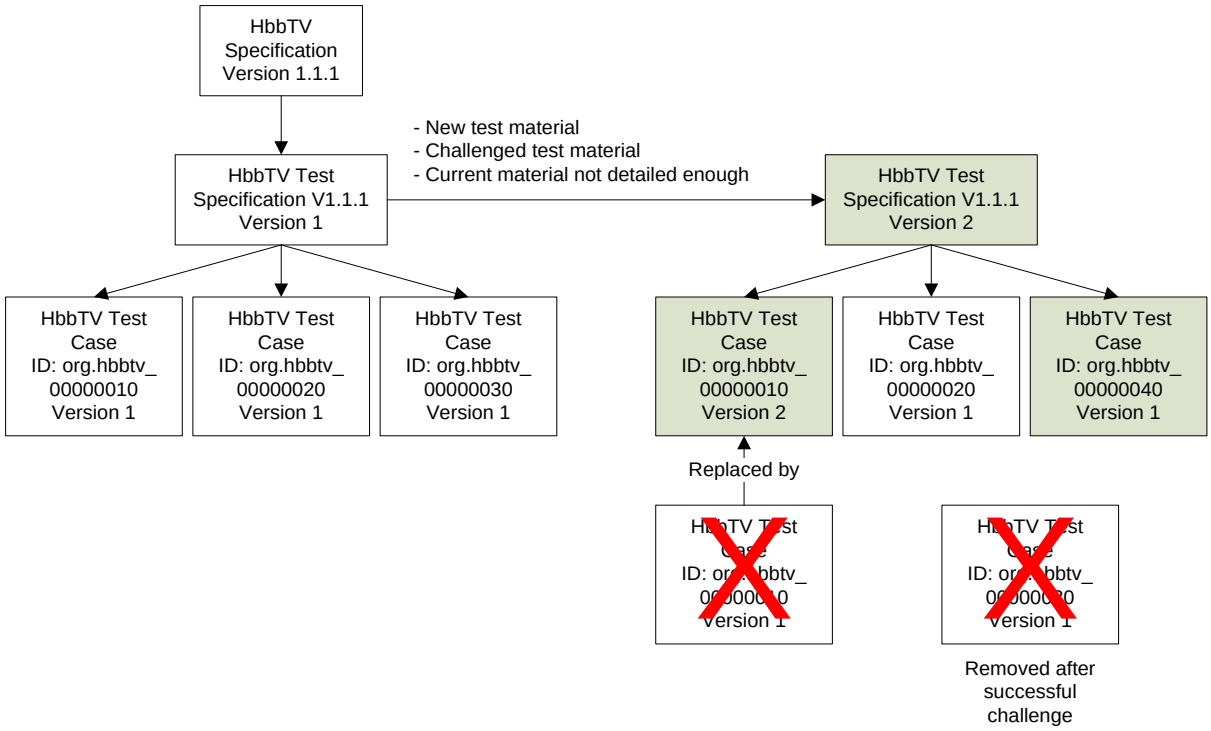


Figure 8: Example of creating a new version Test Specification while keeping the existing version of the Technical Specification

8.1.3 Updating Test Specification after creating a new Technical Specification version.

After a new version of the Technical Specification has been released, a new Test Specification shall also be created and released. New Test Cases and improved Test Cases may be added, as well as deleting obsolete or erroneous Test Cases due to the updated Technical Specification.

- Test Cases may be improved, upgraded or corrected due to a change in the new Technical Specification. In this case the ID number will be kept identical and the version number increases.
- Test Cases may be added due to new technical requirements in the Technical Specification. In this case a new ID number will be assigned and the version number starts with 1.
- Test Cases may be deleted due to obsolete technical requirements.

In any of the above cases a new version of the Test Specification shall be created. The new HbbTV Test Specification shall contain the associated HbbTV Technical Specification Version in its title.

An example of such new Test Specification is given in Figure 9.

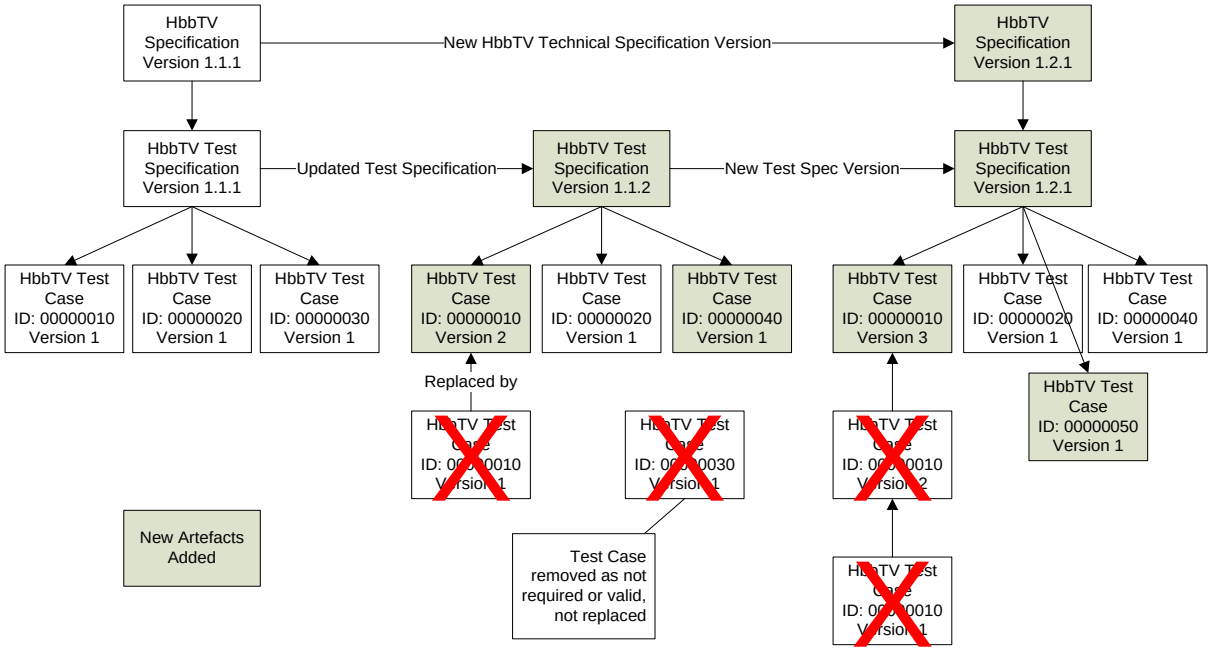


Figure 9: Example of creating a new version Test Specification after creating a new version of the Technical Specification

8.2 Versioning of Test Cases

Central in the Test Case versioning control are the related Test Case items:

- Test Case ID: If the test is changed (e.g. because a problem is identified or an implementation is added), the test case ID stays the same, only the version is increased. If the complete test case must be changed to reflect a modified section of a new HbbTV specification version, a new test case (with a new ID) is created.

- Test Case Version: The version is changed whenever there is a functional change in the test material. This includes changes to Assertion, Procedure or any Test Assets (e.g. HTML-, JavaScript-, CSS and Media-Files etc.). The version need only be changed once per release. Minor typographical edits will not require a version change. It is at the discretion of the Testing Group to decide what constitutes a functional change of a Test Case.

The Test Case ID shall be unique, different from any other HbbTV Test Case.

For Test Case IDs starting “org.hbbtv_”, control of the use of these numbers is administrated by the Test Group.

The Test Case Version number shall have an increased number from the previous version. The numerical increase is one (1) and is administered by the Test Group.

9 Test Reports

9.1 XML Template for individual Test Cases Result

After the execution of each test case against a DUT, the test results for that test case will be collated into an XML document as defined in /SCHEMAS/testCaseResult.xsd of the Test Suite. The Test Case Result contains:

- Test Case ID: Identifier for the test case being reported (as specified in 6.3.1.1).
- Test Case Version: Version of the test case executed (as specified in 6.3.1.2).

The remainder of the Test Case Result consists of a sequence of the following major sections:

- Device Under Test
- Test Performed by
- Test Procedure Output
- Remarks
- Verdict

These are defined in more detail below.

9.1.1 Device Under Test (mandatory)

This lists the information required to identify the DUT that was tested and under which profile.

9.1.1.1 Model

Text string providing a name for the DUT referring to a specific model or a family name of derivative models (e.g. the same platform with different screen sizes).

9.1.1.2 Hardware Version

Text string providing identification of version of hardware typically would include CPU, chassis type, etc.

9.1.1.3 Software Version

Text string providing identification of version of software typically would include software build number, etc.

9.1.1.4 Company

Name of the company for whom the test is being executed. Typically this would be the manufacturer of the DUT.

9.1.1.5 HbbTV Version

ETSI standard reference for the HbbTV Standard supported by the DUT, as a dot separated set of three integers without spaces (e.g. 1.1.1).

NOTE: The HbbTV Version given here doesn't necessarily match the version of the Test suite used. (E.g. a DUT supporting HbbTV 1.2.1 could try to run the tests from the HbbTV 1.1.1 Test Suite).

This field is mandatory if a Test Report is going to be used for HbbTV compliance purposes. (If the XML schema is being reused by another testing group, and they do not require HbbTV compliance, then this field is optional).

9.1.1.6 HbbTV Capabilities

The terminal options tested on the DUT. It contains a combined list of option strings as defined in section 10.2.4 of the HbbTV Specification. The format of the HbbTV Capabilities value is a text field without spaces.

Available options are +DL for download functionality, +DRM for DRM functionality, +PVR for PVR functionality, +SYNC_SLAVE for slave operation in inter-device synchronisation. Multiple requirements are concatenated to a single string without spaces in between. Example: +DL+PVR

NOTE: This should list the options tested, not necessarily those claimed supported.

9.1.1.7 HbbTV Optional Features

Terminal features which are tested on the DUT. Available features are described in 6.3.3.2.

9.1.2 Test Performed By (mandatory)

The name of the test operator who executed this test, in the following format:

9.1.2.1 Name

This should be the test operator's full name.

9.1.2.2 Company

The test operator's company name.

9.1.2.3 Email

The test operator's email address or another contact email address for issues regarding the test result.

9.1.3 Test Procedure Output (mandatory)

This includes the trace of test execution both on the DUT and also on the Test Harness. It also includes the following timestamp information.

9.1.3.1 Start Time

UTC time stamp for time at which the test procedure started for this test

9.1.3.2 End Time

UTC time stamp for time at which the test procedure ended for this test

9.1.3.3 Test Step Output

Results of the execution of a test step, this includes a start and end timestamp of the test execution (as UTC timestamp)

9.1.3.3.1 Index

Index number of the executed test step as defined in the Test Case.

9.1.3.3.2 Start Time

UTC timestamp for the time at which this test step started for this test

9.1.3.3.3 End Time

UTC timestamp for the time at which this test step ended for this test

9.1.3.3.4 Step Result

Either "successful" if the test step completed correctly or "not successful" otherwise.

9.1.3.3.5 Test Step Comment

Comments from the execution of the test step given as a parameter in the reportSetResult() and analyze… calls.

9.1.3.4 Test Step Data (conditional)

This element is mandatory for Test Steps which are triggered by calls to analyzeScreenPixel() and analyzeScreenExtended(). If using either of these calls, the MIME type must be either image/png or image/jpeg. This must be an image of the video output of the RUT.

Specific results from the execution of the test step. This includes the following mandatory attributes:

9.1.3.4.1 id

Index (integer) of the result being reported.

9.1.3.4.2 type

The MIME type of the result being reported. E.g. image/png or text/plain.

9.1.3.4.3 href

Relative hyperlink to the result data in a sub-directory. E.g. "./images/screenshot1.png". This path shall:

- use the "/" character to denote the directory,
- not contain the parent directory ".." indicators,
- use only characters that are valid in filename paths on Windows, Unix/Linux and Apple machines,
- start with either a valid folder name, filename or the "." character, and not with "/", "//" or any OS dependent drive or machine specifier. I.e. it must be a relative path.

9.1.3.5 Test Server Output (mandatory)

This may contain the output of the harness related to the test that was executed for this result. Ideally the server output will also contain timestamp information for the information it is logging.

9.1.3.5.1 Timestamp

UTC timestamp for time at which Test Harness output was started for this test

9.1.3.5.2 Freeform Server Output

Freeform server output in a way that the generated XML Test Case Result remains valid.

9.1.4 Remarks (mandatory)

Remarks and comments on the execution of this test case. The format of the remarks is a text field. It may be empty.

9.1.5 Verdict (mandatory)

Verdict assigned for the test case, based on the criteria as defined in 6.4:

- PASSED: Test met the pass criteria specified in the Test Case.
- FAILED: Test failed to meet the pass criteria specified in the Test Case.

9.2 Test Report

A Test Report contains Test Case Results for one or more Test Cases in one or more Test Suites. Results shall be stored in a ZIP file containing the following directory structure in its root directory:

- The root directory shall contain one or more Test Suite Results directories, where each directory corresponds to results from a single Test Suite. A typical Test Report for the official HbbTV Test Suite shall contain a single directory.
- Each Test Suite Results directory contains only the results of tests from the corresponding Test Suite. Note that the HbbTV official tests are only contained in one Test Suite, therefore there will only be one Test Suite Results directory needed. However, other standards or trademark licensors may require results from multiple Test Suites in their Test Report.
- The name of the Test Suite Results directory is unspecified, but an informative name that contains the version of the Test Suite is recommended e.g. HbbTV-1_2_1-TestSuite-v1_0_0.
- Each Test Suite Results directory shall only contain a Test Case Result directory for each test case that has been executed.
- The name of each Test Case Result directory shall be in the following format, where {test_case_id} is the Test Case ID of the test case that the results pertain to:
  - {test_case_id}
- The Test Case Result directory shall contain one Test Case Result XML document in the format specified in section 9.2.
- The filename of the Test Case Result XML document shall be in the following format:
  - {test_case_id}.result.xml
- All files referenced by Test Step Data shall be stored in the Test Case Result directory where the Test Case Result XML document is located or a subdirectory therein.

Example:

HbbTV-1_2_1-TestSuite-v1_0_0

org.hbbtv_TEST1

org.hbbtv_TEST1.result.xml

org.hbbtv_TEST2

org.hbbtv_TEST2.result.xml

screenshot1.jpg

Trademark Licensor X special test suite-v2

[…]

ANNEX A File validation of HbbTV Test Material (informative)

Purpose of this activity is to check the validity of files that are part of the Test Material in an automatic and objective fashion wherever possible. E.g. is the HTML valid according to the W3C validator, has a Transport Stream been run through a TS analyzer, etc.

The following table lists the file types that are expected to be validated. The right hand column indicates tools that are used. The Test Coordination Group reserves the right to use alternative tools.

File Type	Example tool
JavaScript	JSLint (http://jslint.comcom/)
HTML	W3C Markup Validation Service (http://validator.w3.org/)
CSS	W3C CSS Validation Service (http://jigsaw.w3.org/css-validator/)
XML-Validation	W3C XML Validator (http://www.w3.org/2001/03/webdata/xsv)
MPEG-TS (ETR 101 290)	No recommendation
MP4 file format	No recommendation
MPEG-2/H.264 video content	No recommendation
Image (PNG/JPEG)	No recommendation
MPEG-2 Transport Streams	MPEG-2 Transport Stream packet analyzer: http://www.pjdaniel.org.uk/mpeg/

Table 7: File type validator example tools

ANNEX B Development management tools

To support development and maintenance of the test suite HbbTV hosts a set of collaboration tools.

B.1 Redmine

Redmine (https://hbbtv.org/redmine) is an online project management tool used by HbbTV for many of its specification and testing project management needs. There are currently four projects used by the Testing Group for test suite management:

1) HbbTV Testing is the main group coordination project,
2) HbbTV Test Review is used to track faults identified during test case development and review
3) Test Challenge is used to log challenges on released test cases
4) Test Material Support is used by test suite customers to request help in use of the released test suites.

Members of the Testing Group are eligible to use all of these projects. If you require access please contact support@hbbtv.org.

B.2 Subversion

Subversion is a well-known version control system. It is used to store Test Cases and tools for their development management. This is also the location of schema files for the various standard file formats used for test case creation and test harness configuration. The repository is located at https://hbbtv.org/testing-repo.

B.2.1 Access to the subversion repository

Access to the subversion repository is controlled. Firstly, any member company who wishes to access the repository must complete the Test Repository Access Agreement. Details for obtaining and completing this are to be found at http://www.hbbtv.org/pages/about_hbbtv/hbbtv_test_repository.php. The page also describes how to create login credentials, using htpasswd, to enable you to login.

ANNEX C External hosts

For testing TLS the test system must allow the DUT to access a number of external hosts, as listed in the table below.

https://valid.sfsg2.catest.starfieldtech.com
https://aaacertificateservices.comodoca.com
https://addtrustclass1caroot.comodoca.com
https://addtrustexternalcaroot-ev.comodoca.com
https://addtrustpubliccaroot.comodoca.com
https://addtrustqualifiedcaroot.comodoca.com
https://comodocertificationauthority-ev.comodoca.com
https://comodorsacertificationauthority-ev.comodoca.com
https://securecertificateservices.comodoca.com
https://trustedcertificateservices.comodoca.com
https://usertrustrsacertificationauthority-ev.comodoca.com
https://utnuserfirsthardware-ev.comodoca.com
https://baltimore.omniroot.com
https://ev.omniroot.com
https://assured-id-root.digicert.com/testroot
https://assured-id-root-g2.digicert.com
https://global-root.digicert.com/testroot
https://global-root-g2.digicert.com
https://ev-root.digicert.com/testroot
https://trusted-root-g4.digicert.com
https://2021.globalsign.com
https://2029.globalsign.com
https://2028.globalsign.com
https://valid.gdi.catest.godaddy.com
https://valid.gdig2.catest.godaddy.com
https://valid.sfi.catest.starfieldtech.com
https://valid.sfig2.catest.starfieldtech.com
https://ssltest14.bbtest.net
https://ssltest19.bbtest.net
https://ssltest22.bbtest.net
https://ssltest21.bbtest.net
https://ssltest20.bbtest.net
https://ssltest34.bbtest.net
https://ssltest6.bbtest.net
https://ssltest8.bbtest.net
https://ssltest3.bbtest.net
https://ssltest1.bbtest.net
https://ssltest26.bbtest.net
https://comodoecccertificationauthority-ev.comodoca.com
https://usertrustecccertificationauthority-ev.comodoca.com
https://assured-id-root-g3.digicert.com
https://global-root-g3.digicert.com
https://2038r4.globalsign.com
https://2038r5.globalsign.com
https://ssltest42.ssl.symclab.com
https://ssltest40.ssl.symclab.com
https://ssltest48.ssl.symclab.com

Document history

Document history		
Draft 1	24 February 2016	First draft. Based on the equivalent document for HbbTV 1.3.1, v1.1 draft 5