INTERNATIONAL STANDARD

ISO/IEC 10373-8

First edition
2011-03-15

Identification cards — Test methods —

Part 8: USB-ICC

Cartes d'identification — Méthodes d'essai —

Partie 8: USB-ICC

Reference number
ISO/IEC 10373-8:2011(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 10373-8 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 17, *Cards and personal identification*.

ISO/IEC 10373 consists of the following parts, under the general title *Identification cards — Test methods*:

— *Part 1: General characteristics*

— *Part 2: Cards with magnetic stripes*

— *Part 3: Integrated circuit cards with contacts and related interface devices*

— *Part 5: Optical memory cards*

— *Part 6: Proximity cards*

— *Part 7: Vicinity cards*

— *Part 8: USB-ICC*

— *Part 9: Optical memory cards: Holographic recording method*

Introduction

The USB-ICC is a complex device supporting the USB protocol. The layered structure of the USB protocol involves setting the USB-ICC in different testing configurations when a card manufacturer needs to set forth a Validation Plan. In addition, any USB device belongs to a USB Class. Therefore, the comprehensive testing of any USB device involves carefully developing a Test Plan that includes three different groups of Tests:

1) evaluation of the Electrical, Physical features;

2) effective execution of the USB protocol;

3) execution of Tests designed to prove the compliance of the USB device with its specific Class.

These High-Level Groups of Test are made up of a series of individual Test Scenarios. These scenarios challenge the device, and are designed so that any non-compliance of the card could be disclosed. The final objective is to guarantee the compatibility of the USB-ICC with other USB-compliant devices.

Figure 1 summarizes the Validation Test Framework for the USB-ICC that this part of ISO/IEC 10373 suggests for the USB-ICC.

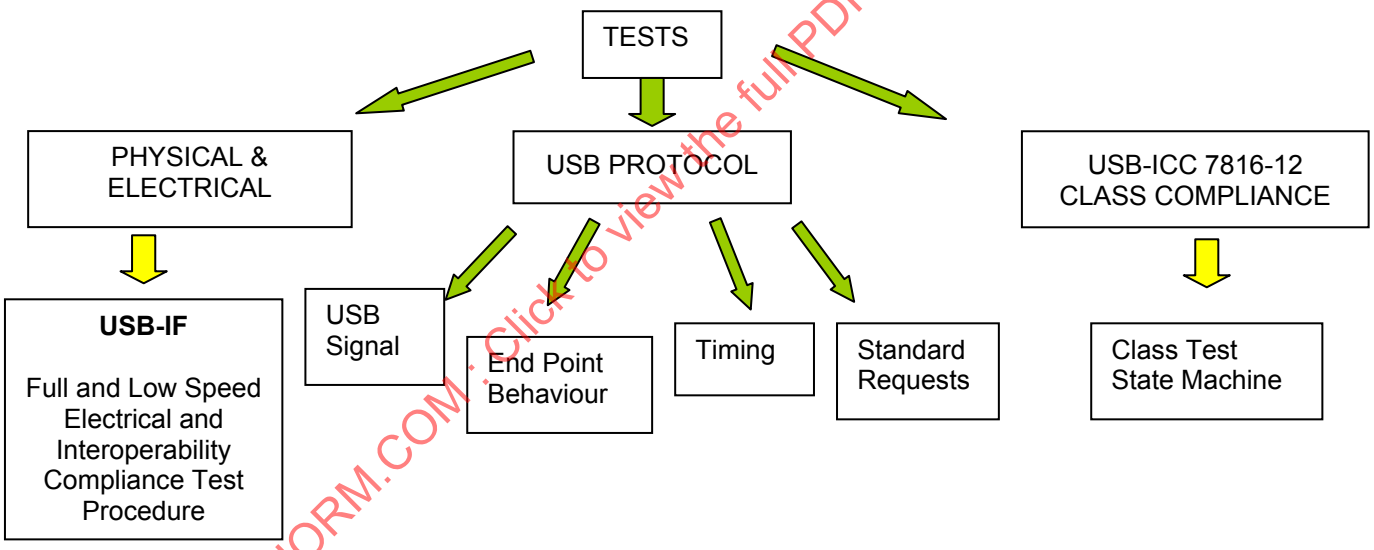


Figure 1 — Compliance test overview

According to ISO/IEC 7816-12, the USB-ICC is required to comply with the USB Specification 2.0 at physical and electrical levels This specification is common to all USB devices; it is published by the USB Implementers Forum, which has also published some Compliance Test Procedures that can be used to test some of the functionalities of the card. Note that ISO/IEC 7816-12 actually describes the USB-ICC Device Class, and testing procedures specific to the USB-ICC Class are not available.

Identification cards — Test methods —

Part 8:
USB-ICC

1 Scope

This part of ISO/IEC 10373 describes a Test Methodology and a list of Test Scenarios to evaluate the compliance of a card with ISO/IEC 7816-12.

Specifically, this part of ISO/IEC 10373:

- addresses USB 2.0 physical layer measurements and electrical compliance testing;
- discusses issues relative to the Test Tools to analyse USB bus traffic and provides guidance for the Test Scenarios given in this part of ISO/IEC 10373;
- proposes a classification of Test Scenarios given in this part of ISO/IEC 10373, along with validation criteria;
- discusses Test Cases for compliance with the USB CCID Class Device.

NOTE Compliance means cards that are called USB-ICC products are designed to match the description in ISO/IEC 7816-12.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 7816-12:2005, *Identification cards — Integrated circuit cards — Part 12: Cards with contacts — USB electrical interface and operating procedures* [RE1 in Test Tags in this part of ISO/IEC 10373]

Universal Serial Bus specification – Revision 2.0, 27 April 2000 [RE2 in this part of ISO/IEC 10373]

3 Terms and definitions

3.1
USB Request
Control Transfer Request
transmission unit for the Control Transfer mode, composed of a **Setup Stage** (3.5), an optional **Data Stage** (3.2) and a **Status Stage** (3.6)

NOTE 1 Used in particular for the **enumeration** (3.4) phase.

NOTE 2 The sum of these stages establishes an applicative protocol layer.

NOTE 3 A Request can be either Standard, Class Specific or Vendor Specific.

3.2
Data Stage
part of the Control Transfer that conveys the data associated with a given Request in one or more Data Bus Transactions

3.3
endpoint
communication channel established between a host and a device

NOTE An endpoint is generally mono-directional, except for the Control Transfer mode, where the endpoint is upstream and downstream at the same time.

3.4
enumeration
Standard Procedure for recognition by the host of the USB-ICC for setting up a communication pipe, during which the host attributes a unique address to the device, and the device driver(s) configure(s) the USB-ICC properly

NOTE 1 It starts when a device is plugged onto a USB port.

NOTE 2 It allows one or more suitable device drivers to be attributed to the device.

3.5
Setup Stage
part of a request (see *Request*) containing the request definition

NOTE The Setup Stage contains the target identification of the request, its direction and the length of the next Data Stage.

3.6
Status Stage
part of a request (see *Request*) standing for global acknowledge of the request

NOTE Any request (except the SetAddress request) is supposed to be terminated before the Status Stage is completed.

3.7
transfer
USB transfer
one or more USB transactions

NOTE A transfer is generally mono-directional (downstream – from host to device, or upstream – from device to host), except for the Control Transfer transfers that are always bi-directional.

3.8
USB mode
mode of transfer used by the USB protocol

NOTE The USB protocol uses four different modes of transfer: The Control Transfer (used for enumeration, and certain interfaces), the Interrupt Transfer (simulating a hardware interrupt behaviour using a polling mechanism), the Bulk Transfer (generally used for non-real-time data transfers) and the Isochronous Transfer (used for real-time data transfers).

3.9
USB transaction
sequence of one, two or three phases: Token, Data, Handshake

4 Symbols and abbreviated terms

NA	Not applicable
ACK	Acknowledged
CCID	Integrated Circuit(s) Cards Interface Device conforming to RE5
NAK	Not acknowledged
STALL	Indicates that a transfer is out of context or wrongly formatted. May require a host intervention.

5 Testing USB Physical and Electrical Characteristics

5.1 Introduction

Electrical tests are common to any USB device compliant device. This specification has been published by the USB Implementers Forum. ISO/IEC 10373-3 should refer to this baseline document [RE4].

USB Specification 2.0 defines the following data rates and rise times:

Table 1 — Data rates and rise times

	DATA RATES	RISE TIMES
Low Speed (LS)	1.5 Mbit/s	75ns-300ns
Full Speed (FS)	12 Mbit/s	4ns-20ns
High Speed (HS)	480 Mbit/s	500 ps

USB-ICC manufacturers should build a Validation Test Program based on the USB-IF applicable documentation.

USB 2.0 electrical testing includes:

— Differential signal quality (eye diagram testing, signal rates, EOP width, cross-over voltage range, paired J-K and K-J as well as consecutive jitter and of course rise and fall times);

— In-Rush current check (current drawn by the CUT) when plugged-in;

— Loaded Vbus power line Drop and droop measurements.

For the tests whose compliance guarantees interoperability at electrical level, either the USB-ICC is directly attached to a Host or it is sharing the USB bus with other USB devices.

USB 2.0 specification introduced a 40 times increase in data rates, and therefore a high level of complexity to the chip (Hi-Speed). Specific testing for High-Speed compliant devices include receiver sensitivity, chirp (special signaling during the High-Speed devices speed detection protocol), monotonicity (smooth increase or decrease of Hi-speed signal amplitude without reverse response) and impedance measurements (CUT and cable). The measurement of very short rise times requires use of real-time oscilloscopes with tight requirements for bandwidth, data sample rates and rising and falling times.

The USB connection device shall establish an electrical connection to C1, C5, C4 and C8 only, following the electrical characteristics and protocol given in the USB 2.0 specification.

NOTE 1 Since the publication of ISO/IEC 7816-12, a new relevant USB Specification [RE3] for smart card technology has been released as a supplement to the USB 2.0 Specification. RE3 or Inter-Chip USB specifies the communication between devices operating at different voltage classes (3.0V, 1.8V, 1.5V, 1.2V and 1.0V) using USB Data Transfers. ISO/IEC 7816-12 refers to the USB specification 2.0, requiring the power supply V_{BUS} at 5.0V nominal. However, the configurations and data transfers defined in ISO/IEC 7816-12 may be supported by cards compliant with the Inter-Chip USB. Compliance rules regarding Inter-chip USB products have not been established yet.

NOTE 2 According to ISO/IEC 7816-12, the USB connection device shall establish an electrical connection to C1, C5, C4 and C8 only, following the electrical characteristics and protocol given in the USB 2.0 specification. This document does not address the electrical characteristics of these contacts.

6 Test Set-Up

6.1 Basic Configuration

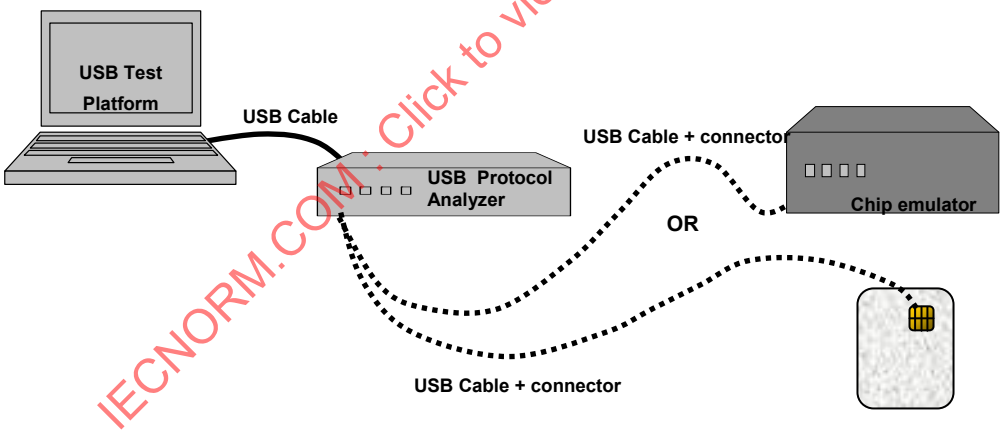
The Basic test set-up enabling the execution of the Test procedures defined in this section typically is made up of three different devices:

1. **The USB Test Platform**, a Host PC, embedding all the necessary software to drive the USB protocol analyzer.

2. **The USB Protocol Analyzer** configured to capture data transfers between the Host PC and the device under test (e.g., the USB-ICC) and display information about the recorded packets. It executes USB software to generate USB traffic.

3. **The USB-ICC** under test or an USB-ICC chip emulator. The tests detailed can be performed either using a chip emulator, or a USB-ICC, or another development system (bond-out for instance). Refer to Clause 6.3 for additional information.

Figure 2 describes the basic Test Configuration:



NOTE Other testing configurations are possible.

Figure 2 — Basic test configuration

6.2 USB Protocol Analyzer features

The USB Protocol Analyzer is configured and controlled by the USB Test Platform to which it is connected through a USB port.

The USB Test Platform configures the USB Protocol Analyzer Unit in traffic generation mode.

Commercial USB Protocol Analyzers usually support enhancements like a *USB Data Transfer.* Within USB, at the lowest level analysis view (packet level) individual packets are representative of the mode of communication taking place. The second higher-level view (Bus Transaction level) combines packets into USB-ICC Bus transactions (Control, Bulk, Interrupt IN), where the actual functional interaction between the Host and the USB-ICC takes place. The highest level view (transfer level) is defined by the device class definitions (CCID class for the USB-ICC).

USB Data Transfer level decoding allows to displays bus interactions between USB devices being tested/analyzed at a level that is more illustrative of the functions being performed

The USB Protocol Analyzer usually includes provisions for on-the-fly detection of and triggering on, numerous events. Such events include specific user-defined bus conditions, packets matching any Packet Identifier (PID), packets matching a Token or Setup transaction, data patterns, and many abnormal (error) bus conditions locate specific data, errors and other desired conditions.

Such pre-selected triggered events may then be recorded and displayed on the USB Test Platform. Real-time detection of events can also be individually enabled or disabled to allow triggering on events as they happen. This includes predefined exception or error conditions, and a user-defined set of search conditions.

6.3 Devices under Test

It is recommended that tests defined in Clauses 8 and 9 be completed using at least two different test platforms:

a) The first one is composed of an emulator of the chip under test; and a card reader interface board configured for USB communication. A PC using the serial port may be used to drive this emulator. The card reader interface board (USB type) can be plugged into the USB Protocol Analyzer record port with a USB connector as shown in Figure 3.

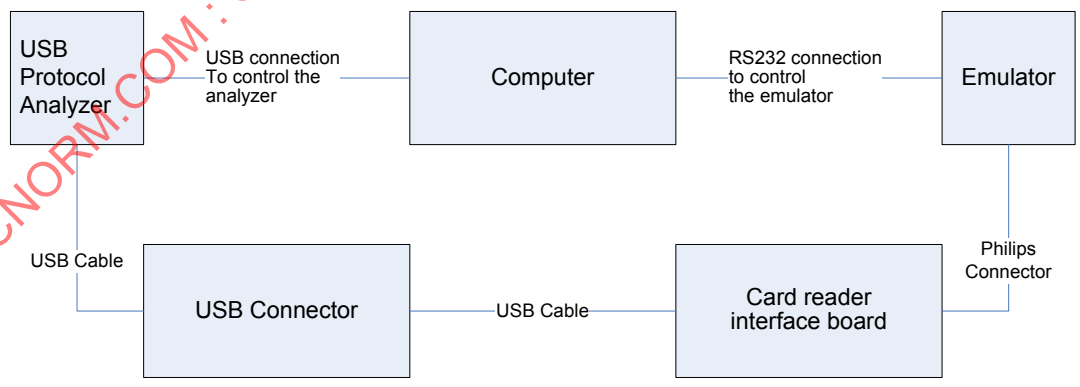


Figure 3 — Test platform 1

b) The second test configuration is made up of a sample smartcard (built with the final component) with a USB connector, plugged into the USB Protocol Analyzer record port with a USB connector as shown in Figure 4.

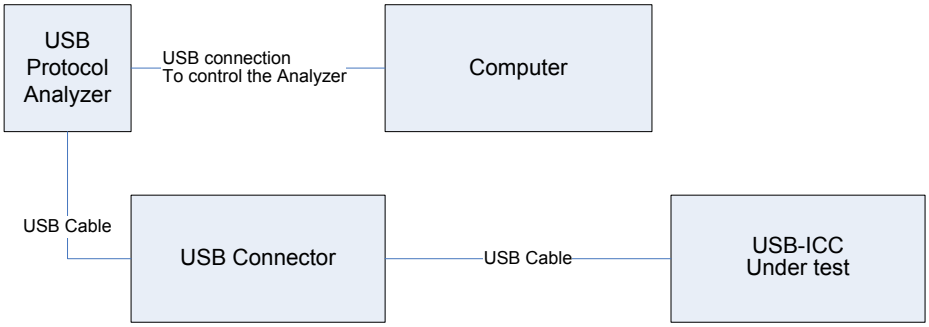


Figure 4 — Test platform 2

7 Test Classification and Validation Criteria

This chapter proposes a classification of the Test Scenarios detailed in Clause 8.

7.1 Test Classification

Each Test Scenario described in Clause 8 is characterized by Type, Target and Criticality as follows:

- **Test Type**: Negative (N) or Nominal (P) , according to the criteria defined in 7.2;
- **Test Target**: Core Feature (C), Prototype (P), Contextual Dependency (D), Security (S) and Side Effect (E) according to the criteria defined in 7.3;
- **Test Criticality**: Critical (C), Major (M) or Minor (m) according to the criteria defined in 7.4.

7.2 Test types

The Test Type refers to the context in which the test is performed compared to the real life conditions. Three types of tests are set out in this section:

- **Negative tests (N):** These tests challenge the ability of the USB-ICC not to run out of control, or to reach an undesired state when accidentally facing a non-standard case. Examples include peer defects, non-compliant USB Hosts devices or drivers. Negative Tests in Clause 8 are tagged with label *N*;
- **Nominal positive tests (P):** They ensure that the card performs properly according to ISO/IEC 7816-12. Positive tests in Clause 8 are tagged with label *P.*

Again, in relation with the Nominal Positive Case tests, two types of tests can be differentiated:

- **Nominal Case Testing:** The Host executes nominal USB Transfers according to USB 2.0 and expects the USB-ICC to respond in an appropriate way;
- **Error Case Testing:** The Host introduces deliberately a given USB Protocol Error in the USB transaction, and expects error detection by the USB-ICC. For instance, a Data Packet error for an OUT Transaction is expected to result in an irresponsive card and the corresponding Host Time-Out (the card doesn't handshake) resulting in a RETRY initiated by the Host.

7.3 Test Targets

Each Test Target refers to a particular part of USB-ICC under test. Five Test Classes are identified in this section:

- **Core features (C):** Such tests run the card in nominal conditions and check the operation of the basic services of the USB-ICC. For instance, an algorithm is executed with several input values and the corresponding output values are checked. These tests are tagged with label *C* in this document;
- **Prototype (P):** Such tests are used to check the boundaries of input parameters of a given command. These tests are tagged with label *P* in this document;
- **Contextual dependency (D):** Such tests ensure that the global context needed to run a command is provided. These tests are tagged with label *D* in this document;
- **Security tests (S):** Such tests check that a given command is executed only if the required security conditions are met. It is however noted, that such tests may interfere with "*contextual dependency"* tests. These tests are tagged with label *S* in this document;
- **Side effects (E):** Such tests have been designed to make sure that a command does not alter its execution context unless it is fully managed. These tests are tagged with label *E* in this document.

7.4 Test criticality

The test criticality provides information about the relative significance and the impact of a failure when executing the test. Three levels can be identified:

- **Critical (C):** A failure when executing a Test characterized as Critical, denotes a serious USB-ICC software bug, leading to an end of the validation process according to this document and possibly requiring the card redesign. These tests are tagged with label *C*;
- **Major (M):** A failure when executing a Test characterized as Major; denotes a significant bug, but which does not prevent other modules of card from working properly. For instance, the discovery of such a bug once the card has been distributed might Expected Result in the design of a softmask to fix the bug. These tests are tagged with label *M;*
- **Minor (m):** A failure when executing a Test characterized as Minor, denotes a less significant bug, meaning that the product is satisfactory operated most of the time, in such a way that the default may be unapparent. These tests are tagged with label *m.*

7.5 Test Acceptance and refusal criteria

In most test scenarios, to successfully pass the Test Scenario, the USB-ICC is required to send an answer as a response to an incoming command.

The only test hereafter where the USB-ICC may not send data back are the Tearing Tests where the power is intentionally cut, so that the card may not have enough time to answer.

In any other case, the status word (SW1-SW2) and the eventual data returned shall match the ones expected by the test scripts.

A Test Scenario shall be considered as successfully *Passed* if the associated test script is run according to the conditions set forth in this section (without the reporting of either status word errors or data errors)

A Test Scenario shall be considered as *Failed,* if the associated test script fails to run according to any of the conditions set forth in this section.

8 USB Protocol Compliance Test Procedure

8.1 USB Protocol Tests Groups

This section documents a series of four Test Groups used to evaluate the USB-ICC operating at Low Speed and/or Full Speed against USB 2.0 requirements:

- The tests relative to specific USB signal management by the card (SE0, Suspend, Resume) are listed in Clause 8.2.1. They cover Speed Identification, USB Reset, Suspend, Resume and Bit Stuffing Management;
- The tests relative to the timing management for the Interpacket Delay are listed in Clause 8.2.2;
- The tests relative to the endpoint behaviour (management of low level protocol and error recovery) are listed in Clause 8.2.3;
- Standard requests management, and associated state machine are listed in Clause 8.2.4;
- Universal Serial Bus specification – Revision 2.0, 27 April 2000 (RE2) shall be used to build the Test Cases.

8.2 Test Description Framework

The Test Cases in this section are identified by a *Test Name* and a *Test Tag*.

The Test Tag shall be built as follows: **T_{***Test Name***}_RE2_{***Section of RE2***}**

RE2 refers in this section to USB Specification 2.0 and *Section of RE2* refers to the USB 2.0 Specification Section, where the feature under test is described.

Test Cases are characterized by its Type, Class and Level of Criticality according to Clause 7.

This identification information along with the Test Description and other Test-Specific relevant information is summarized in the following table:

Table 2 — Template for test description

Test Name	**Type:** see 7.2	**Target:** see 7.3	**Criticality:** see 7.4	**Doc: RE2 USB Specification 2.0**
	Tag: : **T_{***Test Name***}_RE2_{***Relevant Section of RE2***}**			
	Title: Proposed Title for the Test identified by its Name and Tag			
	Test Conditions: Usually the Speed to be used			
	Description: Test Execution Procedure in a Step by Step Basis			
	Expected Result: Correct Response by the USB-ICC following Test Description on Standard Conditions.			
	Initial context: Initial conditions for the USB-ICC and /or Test Set-Up.			
	Comments: Additional Information on Test Objectives and Relevance			
	Implementation: See Clause 6 Test Set-Up			

8.2.1 USB signals management Testing

8.2.1.1 Speed identification

Table 3 — Speed identification test

SpeedID	**Type:** N	**Class:** C	**Criticality:** C	**Docs: RE2**
	Tag: T_SpeedID_RE1_7.1.5.1			
	Title: Check USB-ICC speed management.			
	Condition of test: The test is performed in Low Speed for a Full Speed device and conversely.			
	Description: 1. Perform the connection procedure of the device. 2. Send a *GetDescriptor(Device)* using the wrong communication speed. 3. Send a *GetDescriptor(Device)* and insure the answer is correct.			
	Expected Expected Result: Correct answer to the second GetDescriptor.			
	Initial context: The USB-ICC shall be disconnected.			
	Comments:			
	Implementation: USB Test Platform			

8.2.1.2 USBReset

Table 4 — USBReset1 test

USBReset1	**Type:** P	**Class:** C	**Criticality:** C	**Docs: RE2**
	Tag: T_USBReset1_RE1_7.1.7.5			
	Title: Check the reset timing management.			
	Condition of test: NA			
	Description: 1. Perform the connection procedure of the device. 2. Send a *SetAddress(0x01)* after 20ms of idle (with SOF). 3. Check the address has been correctly assigned sending a *GetDescriptor(Device)* at the new address. 4. Send an *SE0* with variable length (from 2.5µs to 10ms) and wait for 20ms (with SOF). 5. Check the reset was effective sending a *GetDescriptor(Device)* at address 0x00.			
	Expected Expected Result: The answer to the last *GetDescriptor()* should exist and be correct.			
	Initial context: The device shall be disconnected.			
	Comments:			
	Implementation: USB Test Platform			

Table 5 — USBReset2 test

USBReset2	**Type:** N	**Class:** C	**Criticality:** C	**Docs: RE2**
	Tag: T_USBReset2_RE1_7.1.7.5			
	Title: Check the reset timing management (error case).			
	Condition of test: NA			
	Description: 1. Perform the connection procedure of the device. 2. Send a *SetAddress(0x01)* after 20ms of idle (with SOF). 3. Check the address has been correctly assigned sending a *GetDescriptor(Device)* at the new address. 4. Send an *SE0* for 2μs and wait for 20ms (with SOF). 5. Check the reset was not effective sending a *GetDescriptor(Device)* at address 0x01.			
	Expected Result: The answer to the last *GetDescriptor()* should exist and be correct.			
	Initial context: The device shall be disconnected.			
	Comments:			
	Implementation: USB Test Platform			

Table 6 — USBReset3 test

USBReset3	**Type:** P	**Class:** C	**Criticality:** C	**Docs: RE2**
	Tag: T_USBReset3_RE1_7.1.7.5			
	Title: Check the reset recovery time.			
	Condition of test: NA			
	Description: 1. Perform the connection procedure of the device. 2. Send a *SetAddress(0x01)* after 20ms of idle (with SOF). 3. Check the address has been correctly assigned sending a *GetDescriptor(Device)* at the new address. 4. Send an *SE0* for10ms. 5. Send a *SetAddress(0x01)* after 10ms of idle (with SOF) 6. Check the *SetAddress()* was effective sending a *GetDescriptor(Device)* at address 0x01.			
	Expected Result: The *SetAddress()* should be accepted by the device and the answer to the last *GetDescriptor()* should exist and be correct.			
	Initial context: The device shall be disconnected.			
	Comments:			
	Implementation: USB Test Platform			

Table 7 — USBReset4 test

USBReset4	**Type:** P	**Class:** C	**Criticality:** C	**Docs: RE2**
	Tag: T_USBReset4_RE1_7.1.7.5			
	Title: Check the reset effect on state machine.			
	Condition of test: NA			
	Description: 1. Perform the connection procedure of the device. 2. Send a *SetAddress(0x01)* after 20ms of idle (with SOF). 3. Check the address has been correctly assigned sending a *GetDescriptor(Device)* at the new address. 4. Send a *SetConfiguration(Configuration)* at the new address (where configuration is defined in device definition) and verify the request is acknowledged. 5. Send an *SE0* for10ms. 6. Check the *SE0* was effective sending a *SetConfiguration(Configuration)* at address 0x00 and verifying the *SetConfiguration()* is rejected with a *STALL* (request out of context).			
	Expected Result: The last *SetConfiguration()* should be rejected with a STALL due to the loss of "*addressed*" status.			
	Initial context: The device shall be disconnected.			
	Comments:			
	Implementation: USB Test Platform			

8.2.1.3 Suspend management

Table 8 — USBSuspend test

USBSuspend	**Type:** P	**Class:** C	**Criticality:** C	**Docs: RE2**
	Tag: T_USBSuspend_RE1_7.1.7.6			
	Title: Check the suspend behavior.			
	Condition of test: NA			
	Description: 1. Perform the connection procedure of the device. 2. Send a *SE0* for 10ms and wait 10ms in idle (with *SOF*). 3. Send a *SetAddress(0x01)* at address 0x00 and wait 5ms in idle (with *SOF*). 4. Force a *Suspend* on the bus (traffic is stopped) for 1013.5ms (3.5ms without any traffic to force the device starting the suspend transition + 10ms left to the device to actually by in suspend mode + 1s for measuring the suspend current). 5. Perform a *Resume* sequence for 30ms and wait 20ms in idle (with *SOF*). 6. Check the address is still assigned to the device sending a *GetDescriptor(Device)* at address 0x01. 7. Send a *SetConfiguration(Configuration)* at the new address (where configuration is defined in device definition) and verify the request is acknowledged. 8. Force a *Suspend* on the bus (traffic is stopped) for 1013.5ms (3.5ms without any traffic to force the device starting the suspend transition + 10ms left to the device to actually by in suspend mode + 1s for measuring the suspend current). 9. Perform a *Resume* sequence for 30ms and wait 20ms in idle (with *SOF*). 10. Check the configuration is still assigned to the device sending a *GetConfiguration()* at address 0x01.			
	Expected Result: The last *GetDescriptor()* (point 6) and the *GetConfiguration()* (point 10) should be accepted and should return a correct answer.			
	Initial context: The device shall be disconnected.			
	Comments: This test gets its full meaning only if the current consumed by the device when in suspend mode is measured (see Annex 1). Nevertheless, this test can be used in automatic sessions as it shows whether the suspend mode disturbs the device operations or not. This test should be performed for each configuration during normal operation of the device.			
	Implementation: USB Test Platform			

8.2.1.4 Resume management

Table 9 — USBResume test

USBResume	**Type:** P	**Class:** C	**Criticality:** C	**Docs: RE2**
	Tag: T_USBResume_RE1_7.1.7.7			
	Title: Check the resume behavior.			
	Condition of test: NA			
	Description: 1. Perform the connection procedure of the device. 2. Send a *SE0* for 10ms and wait 10ms in idle (with *SOF*). 3. Send a *SetAddress(0x01)* at address 0x00 and wait 5ms in idle (with *SOF*). 4. Force a *Suspend* on the bus (traffic is stopped) for 1013.5ms (3.5ms without any traffic to force the device starting the suspend transition + 10ms left to the device to actually by in suspend mode + 1s for measuring the suspend current). 5. Perform a *Resume* sequence for 20ms and wait 10ms in idle (with *SOF*). 6. Check the address is still assigned to the device sending a *GetDescriptor(Device)* at address 0x01. 7. Send a *SetConfiguration(Configuration)* at the new address (where configuration is defined in device definition) and verify the request is acknowledged. 8. Force a *Suspend* on the bus (traffic is stopped) for 1013.5ms (3.5ms without any traffic to force the device starting the suspend transition + 10ms left to the device to actually by in suspend mode + 1s for measuring the suspend current). 9. Perform a *Resume* sequence for 20ms and wait 10ms in idle (with *SOF*). 10. Check the configuration is still assigned to the device sending a *GetConfiguration()* at address 0x01.			
	Expected Result: The last *GetDescriptor()* (point 6) and the *GetConfiguration()* (point 10) should be accepted and should return a correct answer.			
	Initial context: The device shall be disconnected.			
	Comments: This test should be performed for each configuration during normal operation of the device.			
	Implementation: USB Test Platform			

8.2.1.5 Bit stuffing management

Table 10 — USBBitStuff1 test

USBBitStuff1	**Type:** N	**Class:** C	**Criticality:** C	**Docs: RE2**
	Tag: T_USBBitStuff1_RE1_7.1.9			
	Title: Check the device detects a Bit Stuffing error.			
	Condition of test: Test shall be performed for each existing endpoint.			
	Description: 1. Perform the connection procedure of the device. 2. Send a *SE0* for 10ms and wait 10ms in idle (with *SOF*). 3. Send a *GetDescriptor(0xFF)* at address 0x00 with a Bit Stuffing error, and check the device does not return any handshake. 4. Send an IN transaction, and verify the device returns a STALL. 5. Send a *GetDescriptor(Device)* at address 0x00 without a Bit Stuffing error, and check the Setup Stage is acknowledged. 6. Send an IN transaction, and verify the device returns a correct answer.			
	Expected Result: The device should ignore the Setup Stage of the request containing the error (and so, reject the following upstream transaction), and accept the following Setup Stage, not containing an error (and so, accept and treat the following upstream transaction).			
	Initial context: The device shall be disconnected.			
	Comments: This test should be performed for each endpoint implemented for the device.			
	Implementation: USB Test Platform			

Table 11 — USBBitStuff2 test

USBBitStuff2	**Type:** P	**Class:** C	**Criticality:** C	**Docs: RE2**
	Tag: T_USBBitStuff2_RE1_7.1.9			
	Title: Check the device manages correctly the Bit Stuffing generation.			
	Condition of test: Test shall be performed for each existing endpoint.			
	Description: 1. Perform the connection procedure of the device. 2. Send a *SE0* for 10ms and wait 10ms in idle (with *SOF*). 3. Send a *GetDescriptor(0xFF)* at address 0x00 with a Bit Stuffing error, and check the device does not return any handshake. 4. Send an IN transaction, and verify the device returns a STALL. 5. Send a *GetDescriptor(Device)* at address 0x00 without a Bit Stuffing error, and check the Setup Stage is acknowledged. 6. Send an IN transaction, and verify the device returns a correct answer.			
	Expected Result: The device should ignore the Setup Stage of the request containing the error (and so, reject the following upstream transaction), and accept the following Setup Stage, not containing an error (and so, accept and treat the following upstream transaction).			
	Initial context: The device shall be disconnected.			
	Comments: This test should be performed for each endpoint implemented for the device.			
	Implementation: USB Test Platform			

8.2.2 Timing management Test

8.2.2.1 Interpacket delay

Table 12 — USBIPDelay1 test

USBIPDelay1	**Type:** P	**Class:** C	**Criticality:** C	**Docs: RE2**
	Tag: T_USBIPDelay1_RE1_7.1.18			
	Title: Check the device response time.			
	Condition of test: Test shall be performed for each existing endpoint.			
	Description: 1. Perform the connection procedure of the device. 2. Send a loop of 100 *GetDescriptor(Device)* request and verify the device response (*ACK* or *DATA0* or *DATA1*) comes less than 7.5 bit times after the end of the packet sent by the host.			
	Expected Result: The Expected Result is obtained by measuring the inter-packet time.			
	Initial context: The device shall be disconnected.			
	Side effects:			
	Comments: This test should be performed for each endpoint implemented for the device.			
	Implementation: USB Test Platform			

Table 13 — USBIPDelay2 test

USBIPDelay2	**Type:** N	**Class:** C	**Criticality:** C	**Docs: RE2**
	Tag: T_USBIPDelay2_RE1_7.1.18			
	Title: Check the device manages the inter-packet time-out.			
	Condition of test: Test shall be performed for each existing endpoint.			
	Description: 1. Perform the connection procedure of the device. 2. Send the Setup Stage of a *GetDescriptor(Device)* with a variable inter-packet delay (from 8 to 20 bit times with an increment of 1 bit time). 3. Reset the device (valid SE0 sequence). 4. Send a regular Setup Stage of a *GetDescriptor(Device)* request, followed by an *IN* transaction. Send the *ACK* of the *IN* transaction after a variable delay (from 8 to 20 bit times with an increment of 1 bit time). 5. Depending on the packet length, send another *IN* transaction or two consecutive *OUT* transactions.			
	Expected Result: The device should repeat the data of first *IN* transaction if the transaction of point 5 is an IN transaction, or the second *OUT* transaction should be acknowledged for the other case.			
	Initial context: The device shall be disconnected.			
	Comments: It is to be noticed that the device behavior for inter-packet delays between 16 and 18 bit times is not specified, meaning that the device behavior shall depend on the implementation for this interval. Note that there are two conditions being tested; these are separated by a reset of the device. This test should be performed for each endpoint implemented for the device.			
	Implementation: USB Test Platform			

8.2.3 Low level protocol and error recovery Testing

8.2.3.1 CRC error detection

Table 14 — USBCRC1 test

USBCRC1	**Type:** N	**Class:** C	**Criticality:** C	**Docs: RE2**
	Tag: T_USBCRC1_RE1_8.7.1			
	Title: Check that a token with a wrong CRC in ignored.			
	Condition of test: Test shall be performed for each existing endpoint.			
	Description: 1. Perform the connection procedure of the device. 2. Send a *GetDescriptor(Device)* request with a CRC error in the *SETUP* token. 3. Send the correct *GetDescriptor(Device)* and check the Expected Result is correct.			
	Expected Result: The device should not return any handshake in the corrupted Setup Stage, and the following *IN* transaction should be rejected with a *STALL*.			
	Initial context: The device shall be disconnected.			
	Comments: This test shall also be performed at least once for all additional endpoints with a suitable transaction.			
	Implementation: USB Test Platform			

Table 15 — USBCRC2 test

USBCRC2	**Type:** N	**Class:** C	**Criticality:** C	**Docs: RE2**
	Tag: T_USBCRC2_RE1_8.7.1			
	Title: Check that a token with a wrong CRC in ignored.			
	Condition of test: Test shall be performed for each existing endpoint.			
	Description: 1. Perform the connection procedure of the device. 2. Send a *GetDescriptor(Device)* request with a CRC error in the *DATA* field of the Data Packet. 3. Send the correct *GetDescriptor(Device)* and check the Expected Result is correct.			
	Expected Result: The device should not return any handshake in the corrupted Setup Stage, and the following *IN* transaction should be rejected with a *STALL*.			
	Initial context: The device shall be disconnected.			
	Comments: This test shall also be performed at least once for all additional endpoints with a suitable transaction.			
	Implementation: USB Test Platform			

8.2.3.2 PID error

Table 16 — USBPID1 test

USBPID1	**Type:** N	**Class:** C	**Criticality:** C	**Docs: RE2**
	Tag: T_USBPID1_RE1_8.7.1			
	Title: Check that a Token packet containing a wrong PID is ignored.			
	Condition of test: Test shall be performed for each existing endpoint.			
	Description: 1. Perform the connection procedure of the device. 2. Send a *GetDescriptor(Device)* request with a PID error in the *SETUP* token (PID value well formatted but not existing). 3. Send the correct *GetDescriptor(Device)* and check the Expected Result is correct. 4. Restart from point 2 with the *SETUP* PID badly formatted. 5. Restart from point 2 with a wrong value badly formatted instead of *SETUP* PID.			
	Expected Result: Each time the PID is wrong, the token should be ignored, and the corresponding transaction should be ignored as well.			
	Initial context: The device shall be disconnected.			
	Side effects:			
	Comments: This test shall also be performed at least once for all additional endpoints with a suitable transaction.			
	Implementation: USB Test Platform			

Table 17 — USBPID2 test

USBPID2	**Type:** N	**Class:** C	**Criticality:** C	**Docs: RE2**
	Tag: T_USBPID2_RE1_8.7.1			
	Title: Check that a Data packet containing a wrong PID is ignored.			
	Condition of test: Test shall be performed for each existing endpoint.			
	Description: 1. Perform the connection procedure of the device. 2. Send a *GetDescriptor(Device)* request with a PID error in the *DATA* Token (PID value well formatted but not existing). 3. Send the correct *GetDescriptor(Device)* and check the Expected Result is correct. 4. Restart from point 2 with the *DATA* PID badly formatted. 5. Restart from point 2 with a wrong value badly formatted instead of *DATA* PID.			
	Expected Result: Each time the PID is wrong, the token should be ignored, and the corresponding transaction should be ignored as well.			
	Initial context: The device shall be disconnected.			
	Comments: This test shall also be performed at least once for all additional endpoints with a suitable transaction.			
	Implementation: USB Test Platform			

Table 18 — USBPID3 test

USBPID3	**Type:** N	**Class:** C	**Criticality:** C	**Docs: RE2**
	Tag: T_USBPID3_RE1_8.7.1			
	Title: Check that a handshake packet containing a wrong PID is ignored.			
	Condition of test: Test shall be performed for each existing endpoint.			
	Description: 1. Perform the connection procedure of the device. 2. Send a *GetDescriptor(Device)* request with a PID error in the *HANDSHAKE* token of the first IN transaction of the Data Stage (PID value well formatted but not existing). 3. Send the correct *GetDescriptor(Device)* and check the Expected Result is correct. 4. Restart from point 2 with the *HANDSHAKE* PID badly formatted. 5. Restart from point 2 with a wrong value badly formatted instead of *HANDSHAKE* PID.			
	Expected Result: Each time the PID is wrong, the token should be ignored, and the corresponding transaction should be ignored as well, meaning that the second IN transaction of the Data Stage (that has to be added if the packet size is greater than 18) shall contain the same information as the first one.			
	Initial context: The device shall be disconnected.			
	Comments: This test shall also be performed at least once for all additional endpoints with a suitable transaction.			
	Implementation: USB Test Platform			

8.2.3.3 Data toggling error detection

Table 19 — USBDataToggle test

USBDataToggle	**Type:** P	**Class:** C	**Criticality:** C	**Docs: RE2**
	Tag: T_USBDataToggle_RE1_8.6			
	Title: Check that the data toggle is correctly managed in the device.			
	Condition of test: Test shall be performed for each existing endpoint.			
	Description: 1. Perform the connection procedure of the device. 2. Send a *GetDescriptor(Device)* request with a DATA1 token for the Data Packet instead of a DATA0.			
	Expected Result: The request should be ignored, and no handshake nor data should be returned by the device.			
	Initial context: The device shall be disconnected.			
	Side effects:			
	Comments: This test should also be performed at least once for all additional endpoints with a suitable transaction when possible (OUT endpoints only).			
	Implementation: USB Test Platform			

8.2.3.4 Address error detection

Table 20 — USBAddress test

USBAddress	**Type:** N	**Class:** C	**Criticality:** C	**Docs: RE2**
	Tag: T_USBAddress_RE1_8.7.1			
	Title: Check that a transaction with wrong address is ignored.			
	Condition of test: Test shall be performed for each existing endpoint.			
	Description: 1. Perform the connection procedure of the device. 2. Send a *GetDescriptor(Device)* request to variable addresses (from 0x00 to 0xEF with an increment of 5). 3. Send a *SetAddress(0x5A)*. 4. Restart from point 2 with an address in the range 0x00 to 0xEF.			
	Expected Result: The *GetDescriptor()* should be completely ignored each time the address is wrong.			
	Initial context: The device shall be disconnected.			
	Comments: This test shall also be performed at least once for all additional endpoints with a suitable transaction.			
	Implementation: USB Test Platform			

8.2.3.5 Endpoint error

Table 21 — USBEndP test

USBEndP	**Type:** N	**Class:** C	**Criticality:** C	**Docs: RE2**
	Tag: T_USBEndP_RE1_8.7.1			
	Title: Check that a transaction with wrong endpoint is ignored.			
	Condition of test: Test shall be performed for each existing endpoint.			
	Description: 1. Perform the connection procedure of the device. 2. Send a *GetDescriptor(Device)* request to variable endpoint (from 0x00 to 0x0F and from 0x80 to 0x8F with an increment of 1).			
	Expected Result: The *GetDescriptor()* should be completely ignored each time the endpoint is wrong.			
	Initial context: The device shall be disconnected.			
	Comments: This test shall also be performed at least once for all additional endpoints with a suitable transaction.			
	Implementation: USB Test Platform			

8.2.4 Cancel and Abandon Tests

8.2.4.1 Cancel Request Handling

Table 22 — USBCancel test

USBCancel	**Type:** P	**Class:** C	**Criticality:** C	**Docs: RE2**
	Tag: T_USBCancel_RE1_5.5			
	Title: Check that the device correctly manages the request canceling.			
	Condition of test: NA			
	Description: 1. Perform the connection procedure of the device. 2. Send two consecutive *GetDescriptor(Device)* requests making the Setup Stage of the second request beginning before the Status Stage of the first one.			
	Expected Result: The second *GetDescriptor()* should be treated correctly.			
	Initial context: The device shall be disconnected.			
	Side effects:			
	Comments:			
	Implementation: USB Test Platform			

8.2.4.2 Abandon Request Handling

Table 23 — USBAbandon test

USBAbandon	**Type:** P	**Class:** C	**Criticality:** C	**Docs: RE2**
	Tag: T_USBAbandon_RE1_5.5			
	Title: Check that the device correctly manages the request abandoning.			
	Condition of test: NA			
	Description: 1. Perform the connection procedure of the device. 2. Send two consecutive *GetDescriptor(Device)* requests, the first one being truncated in the sense the Status Stage is sent just after the Setup Stage.			
	Expected Result: The device should accept the Status Stage of the first request and treat correctly the second request.			
	Initial context: The device shall be disconnected.			
	Comments:			
	Implementation: USB Test Platform			

8.2.5 USB State Machine

Table 24 — USBStdReq4 test

USBStdReq4	**Type:** P	**Class:** C	**Criticality:** C	**Docs: RE2**
	Tag: T_USBStdReq4_RE1_9.1			
	Title: Check that the USB states *Default*, *Addressed* and *Configured* are correctly managed.			
	Condition of test: NA			
	Description: 1. Perform the connection procedure of the device. 2. The requests used for this test are *SetAddress()* and *SetConfiguration()*. The transitions (USB States *Default*, *Addressed* and *Configured*) described in the chapter 9.1 shall be tested.			
	Expected Result: Each request shall be accepted and the transition has to be verified using the appropriate mechanism (rejection of a *SetConfiguration(0x01)* for a device in *Default* State for instance).			
	Initial context: The device shall be disconnected.			
	Comments:			
	Implementation: USB Test Platform			

8.3 Standard Request management Tests

8.3.1 Objectives

The aim of this test is to verify that the device correctly supports all the standard requests.

8.3.2 Requirements

It is necessary to verify that each USB standard request in the nominal case is processed correctly (the one that are not supported should be STALLed by the device).

8.3.3 Initial condition

Before each request test, the card (or emulator) is in the proper USB State (i.e. Default State for a GetDescriptor, Addressed State for a SetConfiguration …).

8.3.4 Nominal cases

Common Test Conditions:

Table 25 — USBStdReq1 test

USBStdReq1	**Type:** P	**Class:** C	**Criticality:** C	**Docs: RE2**
	Tag: T_USBStdReq1_RE1_9.4x			
	Title: Check that all standard requests are correctly managed.			
	Condition of test: NA			
	Description: 1. Perform the connection procedure of the device. 2. The standard requests and corresponding parameters are listed in tables 9-3 to 9-6. The state machine has to be respected for this test.			
	Expected Result: Each request shall be treated correctly (when supported by the device) or rejected with a STALL (when not supported by the device). Special attention shall be paid to the descriptors returned by the device, and checked against the descriptor deduced from the definition of the device in the USB Test Platform.			
	Initial context: The device shall be disconnected.			
	Comments: It should be noted that most of standard requests remain available after enumeration phase. These requests should be tested when the device is in normal operation.			
	Implementation: USB Test Platform			

8.3.5 Test List

Table 26 — Test list

T_USBStdReq1_RE2_9.4.5a	Test of request GetStatus (Device)	Treated correctly
T_USBStdReq1_RE2_9.4.5b	Test of request GetStatus (Endpoint)	Treated correctly
T_USBStdReq1_RE2_9.4.5c	Test of request GetStatus (Interface)	Treated correctly
T_USBStdReq1_RE2_9.4.1	Test of request ClearFeature (For device, Endpoint, Interface)	Stalled ClearFeature requests
T_USBStdReq1_RE2_9.4.9a	Test of request SetFeature (Device)	Stalled
T_USBStdReq1_RE2_9.4.9b	Test of request SetFeature (Endpoint)	Stalled
T_USBStdReq1_RE2_9.4.9c	Test of request SetFeature (Interface)	Stalled
T_USBStdReq1_RE2_9.4.6	Test of request SetAddress ()	Treated correctly
T_USBStdReq1_RE2_9.4.3a	Test of request GetDescriptor (Device)	Treated correctly
T_USBStdReq1_RE2_9.4.3b	Test of request GetDescriptor (Configuration)	Treated correctly
T_USBStdReq1_RE2_9.4.3c	Test of request GetDescriptor (USB-ICC) with a card compliant with ISO7816-12 protocol	Treated correctly
T_USBStdReq1_RE2_9.4.3d	Test of request GetDescriptor (String)	Treated correctly
T_USBStdReq1_RE2_9.4.8a	Test of request SetDescriptor (Device)	Stalled
T_USBStdReq1_RE2_9.4.8b	Test of request SetDescriptor (Configuration)	Stalled
T_USBStdReq1_RE2_9.4.8c	Test of request SetDescriptor (String)	Stalled
T_USBStdReq1_RE2_9.4.8d	Test of request SetDescriptor (Interface)	Stalled
T_USBStdReq1_RE2_9.4.2	Test of request GetConfiguration ()	Treated correctly
T_USBStdReq1_RE2_9.4.7	Test of request SetConfiguration ()	Treated correctly
T_USBStdReq1_RE2_9.4.4	Test of request GetInterface ()	Treated correctly
T_USBStdReq1_RE2_9.4.10a	Test of request SetInterface (Correct Interface)	Treated correctly
T_USBStdReq1_RE2_9.4.10b	Test of request SetInterface (Unknown Interface)	Stalled
T_USBStdReq1_RE2_9.4.11	Test of request SynchFrame ()	Stalled

8.3.6 Error cases

For these tests, each individual standard request is sent to the USB-ICC with:

- wIndex parameter inconsistent;
- wValue parameter inconsistent;
- wIndex and wValue both inconsistent;
- wLength value lower than the real amount of data transmitted;
- wLength value greater than transmitted;
- Error in data transfer direction (in bmRequestType), and finally
- Error in the recipient (in bmRequestType).

It should be noted that for some Standard Requests, the behavior is not defined by [RE2] and the expected result depends on the card implementation.

8.3.7 Common Test Conditions

Table 27 — USBStdReq2 test

USBStdReq2	**Type:** N	**Class:** C	**Criticality:** C	**Docs: RE2**
	Tag: T_USBStdReq2_RE1_9.4x			
	Title: Check that standard requests are correctly rejected when sent with wrong parameters.			
	Condition of test: NA			
	Description: 1. Perform the connection procedure of the device. 2. The standard requests and corresponding parameters are listed in tables 9-3 to 9-6. The state machine has to be respected for this test. The parameters shall be chosen as close as possible to the correct values.			
	Expected Result: Each request shall be rejected would it be supported or not by the device.			
	Initial context: The device shall be disconnected.			
	Side effects:			
	Comments:			
	Implementation: USB Test Platform			

Table 28 — USBStdReq3 test

USBStdReq3	**Type:** N	**Class:** C	**Criticality:** C	**Docs: RE2**
	Tag: ET_USBStdReq3_RE1_9.4			
	Title: Check that the device rejects standard requests out of context.			
	Condition of test: NA			
	Description: 1. Perform the connection procedure of the device. 2. The standard requests and corresponding parameters are listed in tables 9-3 to 9-6. The state machine has to be respected for this test. The parameters shall be chosen as close as possible to the correct values.			
	Expected Result: Each request shall be rejected would it be supported or not by the device.			
	Initial context: The device shall be disconnected.			
	Comments:			
	Implementation: USB Test Platform			

8.3.8 Test List

Table 29 — Test list

Test Reference	Test Scenario	Error in the Request	Expected Result	Behavior specified [RE2]
ET_USBStdReq1_RE2_9.4.5a	GetStatus()	Error in wValue	Treated correctly	No
ET_USBStdReq1_RE2_9.4.5b		Error in wIndex	Stalled	Yes
ET_USBStdReq1_RE2_9.4.5c		Error in wValue and wIndex	Stalled	Yes
ET_USBStdReq1_RE2_9.4.5a		wLength lower	Stalled	No
ET_USBStdReq1_RE2_9.4.5b		wLength greater	Treated correctly	No
ET_USBStdReq1_RE2_9.4.5c		Error in data transfer direction	Stalled	Yes
ET_USBStdReq1_RE2_9.4.5a		Error in recipient	Stalled	Yes
ET_USBStdReq1_RE2_9.4.6a	SetAddress()	Error in wValue	Stalled	No
ET_USBStdReq1_RE2_9.4.6b		Error in wIndex	Treated correctly	No
ET_USBStdReq1_RE2_9.4.6c		Error in wValue and wIndex	Stalled	No
ET_USBStdReq1_RE2_9.4.6d		wLength greater	Stalled	No
ET_USBStdReq1_RE2_9.4.6e		Error in data transfer direction	Stalled	Yes
ET_USBStdReq1_RE2_9.4.3a	GetDescriptor()	Error in wValue	Stalled	Yes
ET_USBStdReq1_RE2_9.4.3b		Error in wIndex	Stalled	No
ET_USBStdReq1_RE2_9.4.3c		Error in wValue and wIndex	Stalled	Yes
ET_USBStdReq1_RE2_9.4.3d		wLength lower	Treated correctly	Yes
ET_USBStdReq1_RE2_9.4.3e		wLength greater	Treated correctly	Yes
ET_USBStdReq1_RE2_9.4.3f		Error in data transfer direction	Stalled	Yes
ET_USBStdReq1_RE2_9.4.2a	GetConfiguration()	Error in wValue	Treated correctly	No
ET_USBStdReq1_RE2_9.4.2b		Error in wIndex	Treated correctly	No
ET_USBStdReq1_RE2_9.4.2c		Error in wValue and wIndex	Treated correctly	No
ET_USBStdReq1_RE2_9.4.2d		wLength lower	Stalled	No
ET_USBStdReq1_RE2_9.4.2e		wLength greater	Treated correctly	No
ET_USBStdReq1_RE2_9.4.2f		Error in data transfer direction	Stalled	Yes
ET_USBStdReq1_RE2_9.4.7a	SetConfiguration()	Error in wValue	Stalled	Yes
ET_USBStdReq1_RE2_9.4.7b		Error in wIndex	Treated correctly	No
ET_USBStdReq1_RE2_9.4.7c		Error in wValue and wIndex	Stalled	Yes
ET_USBStdReq1_RE2_9.4.7d		wLength greater	Stalled	No
ET_USBStdReq1_RE2_9.4.7e		Error in data transfer direction	Stalled	Yes
ET_USBStdReq1_RE2_9.4.4a	GetInterface()	Error in wValue	Treated correctly	No
ET_USBStdReq1_RE2_9.4.4b		Error in wIndex	Stalled	Yes
ET_USBStdReq1_RE2_9.4.4c		Error in wValue and wIndex	Stalled	Yes
ET_USBStdReq1_RE2_9.4.4d		wLength lower	Stalled	No
ET_USBStdReq1_RE2_9.4.4e		wLength greater	Treated correctly	No
ET_USBStdReq1_RE2_9.4.4f		Error in data transfer direction	Stalled	Yes
ET_USBStdReq1_RE2_9.4.10a	SetInterface()	Error in wValue	Stalled	Yes
ET_USBStdReq1_RE2_9.4.10b		Error in wIndex	Stalled	Yes
ET_USBStdReq1_RE2_9.4.10c		Error in wValue and wIndex	Stalled	Yes
ET_USBStdReq1_RE2_9.4.10d		wLength greater	Stalled	No
ET_USBStdReq1_RE2_9.4.10e		Error in data transfer direction	Stalled	Yes

9 Tests for compliance with CCID Class

IMPORTANT — This part of ISO/IEC 10373 only addresses the USB-ICC in Control Transfer Version A mode. Future editions may add further Transmission Modes specified by ISO/IEC 7816-12.

9.1 Objectives

Demonstrate the compliance of the USB-ICC in its normal mode of operation for its intended usage:

1) Enumeration in compliance with the USB-ICC shall use the card descriptors of ISO/IEC 7816-12:2005, Clause 7.

2) Proper Operation of the ISO/IEC 7816-12 Standard Command Set, for the Data Transfer modes described in ISO/IEC 7816-12:2005, Clause 8 by the execution of

   — a series of Conformity Tests with specific USB requests shall be according to ISO 7816-12 USB protocol configuration;

   — a series of Specific Tests following historical bug discover in USB chips.

3) Correct Execution of APDU exchange procedures shall be according to the standard State Machines described in ISO/IEC 7816-12.

The exchange of data between the Host and USB-ICC may be done using bulk transfers or control transfers. For control transfer, two implementations are possible. They are named by ISO/IEC 7816-12 Version A and Version B. Bulk transfer mode is compliant to the CCID specification, e.g. it uses a subset of the messages/requests as defined in this specification.

The Test Scenarios proposed in this document conform to the following rules:

1) Processing of the Control Transfer A Requests, according to ISO/IEC 7816-12:2005, Clause 8. The USB-ICC functionality is tested in two cases in both: Nominal case and introducing errors in the Request Parameters and verifying that the card recognizes them and performs according to RE2.

2) State Machine Compliance Verification: The USB-ICC is sent a complete set of requests intended to verify if the card changes its internal State according to the transitions defined by the State diagrams of section 8.1.

9.2 Control A Transfer Individual Request Test

9.2.1 Test Description

Test of each ISO/IEC 7816-12 Request in both nominal case and with errors in the Request parameters.

9.2.2 Objectives

1) To verify that the device under test supports the USB requests as per ISO/IEC 7816-12 section 8.2.1.

2) To verify the error detection capabilities of the device under test when the parameters of these requests are wrong.

9.2.3 Requirements

The nominal case of ISO 7816-12 USB requests shall be correctly processed. ISO 7816-12 USB requests containing error in the parameters are processed according to the following rules:

1) The SETUP Stage is always acknowledged, unless there is a packet error (SETUP or DATA0 packets). In this case the USB-ICC remains irresponsive and the Host proceeds to a RETRY.

2) A USB protocol Error during the DATA Stage, shall result in an irresponsive USB-ICC. However, the card still may STALL or NACK a DATA STAGE IN Transaction.

3) In the previous table, “stalled” corresponds to a situation, where the USB-ICC endpoint returns a STALL handshake the STATUS phase of the Data Transfer:

   — As the Handshake packet for an OUT STATUS Stage;

   — Immediately after the IN Token for an IN STATUS Stage.

9.2.4 Initial USB-ICC condition:

The device is in USB Default State.

9.2.5 Testing Set-Up

This test shall be performed with the emulator and the sample card. This test shall be passed with a card or emulator configured for the USB ISO7816-12 protocol (see Clause 2).

9.2.6 Test List: Individual Requests in Nominal and with Error Cases

Table 30 — Test list

Icc_power_off request	**test_icc_power_off_1**	**Nominal case**	**Treated correctly**
	test_icc_power_off_2	Error in wValue	Stalled
	test_icc_power_off_3	Error in wIndex	Stalled
	test_icc_power_off_4	Error in wValue and in wIndex	Stalled
	test_icc_power_off_5	Error in data transfer direction	Stalled
	test_icc_power_off_6	wLength greater	Stalled
	test_icc_power_off_9	bmRequestType as standard request	Stalled
Icc_power_on request	**test_icc_power_on_1**	**Nominal case**	**Treated correctly**
	test_icc_power_on_2	Error in wValue	Stalled
	test_icc_power_on_3	Error in wIndex	Stalled
	test_icc_power_on_4	Error in wValue and in wIndex	Stalled
	test_icc_power_on_5	Error in data transfer direction	Stalled
	test_icc_power_on_7	wLength lower	Treated correctly
	test_icc_power_on_8	wLength = 0	Stalled
	test_icc_power_on_9	bmRequestType as standard request	Stalled
Data_Block request (Status Word case)	**test_data_block_A_1**	**Nominal case**	**Treated correctly**
	test_data_block_A_2	Error in wValue	Stalled
	test_data_block_A_3	Error in wIndex	Stalled
	test_data_block_A_4	Error in wValue and in wIndex	Stalled
	test_data_block_A_5	Error in data transfer direction	Stalled
	test_data_block_A_6	wLength greater	Stalled or Send SW
	test_data_block_A_7	wLength lower	Stalled
	test_data_block_A_8	wLength = 0	Stalled
	test_data_block_A_9	bmRequestType as standard request	Stalled

Data_Block request	**test_data_block_B_1**	**Nominal case**	**Treated correctly**
(Data case)	test_data_block_B_2	Error in wValue	Stalled
	test_data_block_B_3	Error in wIndex	Stalled
	test_data_block_B_4	Error in wValue and in wIndex	Stalled
	test_data_block_B_5	Error in data transfer direction	Stalled
	test_data_block_B_6	wLength greater	Stalled
	test_data_block_B_7	wLength lower	Send packet of short length or null length
	test_data_block_B_8	wLength = 0	Stalled
	test_data_block_B_9	bmRequestType as standard request	Stalled
Get_Icc_Status request	**test_get_icc_status_1**	**Nominal case**	**Treated correctly**
	test_get_icc_status_2	Error in wValue	Stalled
	test_get_icc_status_3	Error in wIndex	Stalled
	test_get_icc_status_4	Error in wValue and in wIndex	Stalled
	test_get_icc_status_5	Error in data transfer direction	Stalled
	test_get_icc_status_6	wLength greater	Stalled
	test_get_icc_status_7	wLength lower	Stalled
	test_get_icc_status_9	bmRequestType as standard request	Stalled
Xfr_block request	**test_xfr_block_A_1**	**Nominal case**	**Treated correctly**
(APDU case)	test_xfr_block_A_2	Error in wValue	Stalled
	test_xfr_block_A_3	Error in wIndex	Stalled
	test_xfr_block_A_4	Error in wValue and in wIndex	Stalled
	test_xfr_block_A_5	Error in data transfer direction	Stalled
	test_xfr_block_A_6	wLength greater	Stalled
	test_xfr_block_A_7	wLength lower	Stalled
	test_xfr_block_A_8	wLength = 0	Stalled
	test_xfr_block_A_9	bmRequestType as standard request	Stalled
Xfr_block request	**test_xfr_block_B_1**	**Nominal case**	**Treated correctly**
(Data case)	test_xfr_block_B_2	Error in wValue	Stalled
	test_xfr_block_B_3	Error in wIndex	Stalled
	test_xfr_block_B_4	Error in wValue and in wIndex	Stalled
	test_xfr_block_B_5	Error in data transfer direction	Stalled
	test_xfr_block_B_8	wLength = 0	Stalled
	test_xfr_block_B_9	bmRequestType as standard request	Stalled