# Component Test Documentation IP-Stack

**Component Version**: 1.03

**Document Version**: 1.3

**Date**: 03/01/2013

**Author/Responsible:** Daniel Baldin (DB), Stefan Grösbrink (SG), Rakesh Garimella (RG), Christoph Topmöller (CT), Alexander Botter (AB)

**Confidentiality Level**: confidential

**Summary**:

This document describes the specification for the testing of the component IP-Stack. Therefore it describes the design of the test cases, the used testing framework and environment as well as the requirements in both hard- and software to execute the tests. It documents the test project and gives and introduction on how to use and adapt the test cases. The reader will be firmly introduced to TTCN3, however, for future adaptations it is suggested to have some basic knowledge about TTCN3.

Table of Contents

[1 Introduction 3](#_Toc344990960)

[2 Test Subject 4](#_Toc344990961)

[2.1. SOC IP-Stack 4](#_Toc344990962)

[2.1.1. TTCN3 4](#_Toc344990963)

[2.1.2. Test Functionality Overview 7](#_Toc344990964)

[3 Test Concept 9](#_Toc344990965)

[3.1. Applied Test Design Techniques 9](#_Toc344990966)

[3.2. Test Case Template 9](#_Toc344990967)

[4 Project Structure 11](#_Toc344990968)

[5 Test Case Implementation 13](#_Toc344990969)

[5.1. Implementation Example 13](#_Toc344990970)

[5.2. Test execution 15](#_Toc344990971)

[5.3. Kernel Module 15](#_Toc344990972)

[6 TCP Test Specification 18](#_Toc344990973)

[6.1. Test Basis 21](#_Toc344990974)

[6.1. Test Results 23](#_Toc344990975)

[7 Appendix 28](#_Toc344990976)

# Introduction

This document is designed to provide a basic understanding and overview of the underlying test concept, the used technologies and the testing environment of the implemented IP-Stack.

The first chapter will introduce the concept of testing by outlining the design techniques used to create the different test cases. It also provides an overview of the template used to document and describe a test case.

In chapter two the test environment with all hardware and software components, including the TTCN-3 language, is depicted. TTCN-3 is a programming language explicitly designed for writing tests and is used for test-automation in this project. Furthermore detailed instructions about how to execute test cases and which requirements have to be met in order to do so, is given.

Finally a full specification of the TCP-Protocol related test cases is provided in the final chapter, including an informal description and the enclosed test steps.

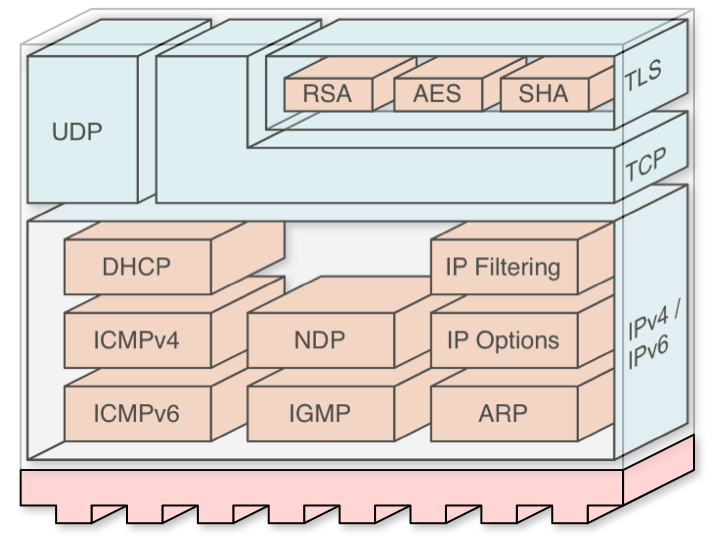
# Test Subject

This chapter introduces the IP-Stack implementation and gives a short overview on how the IP-Stack is tested using the TTCN-3 test suite.

## SOC IP-Stack

The IP-Stack implementation is a small independent implementation of the TCP/IP protocol suite. It is similar to LwIP-Stack (Lightweight TCP/IP stack) [5], which is a full TCP/IP suite implementation. The focus of this project was to reduce resource usage while still having a full scale TCP/IP stack. The IP-Stack is a special implementation for the usage with smart cards. This means the amount of resources is limited.

In Figure 1 you can see the implementation of the IP-Stack. There you can see the structure of all implemented protocols and how they cooperate.



**Figure 1:** IP-Stack implementation

The underlying protocol is IPv4/IPv6. The Internet Protocol suite includes other core protocols. For example ICMP, DHCP or IGMP. The functionality of all these protocols is implemented as well.

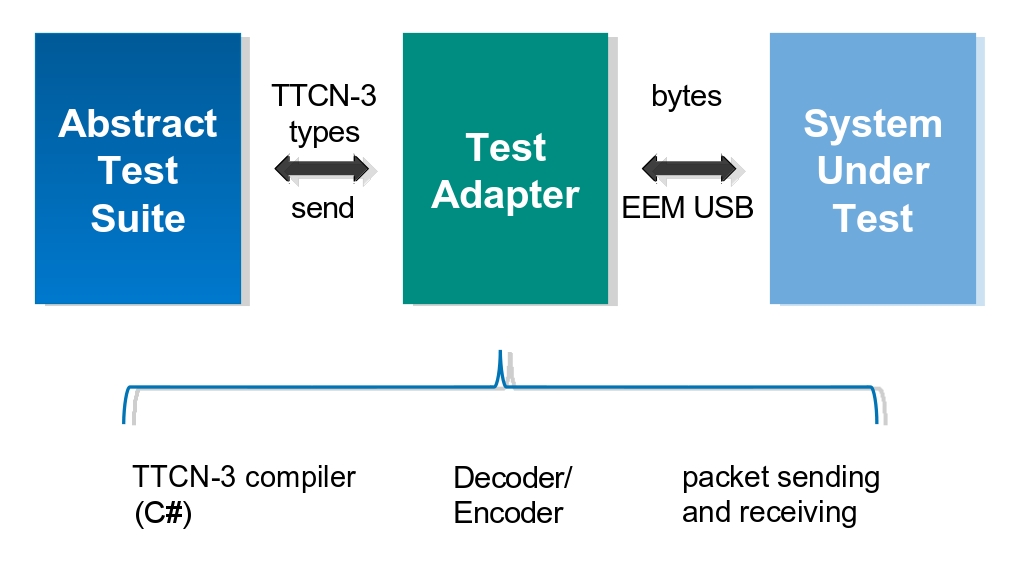
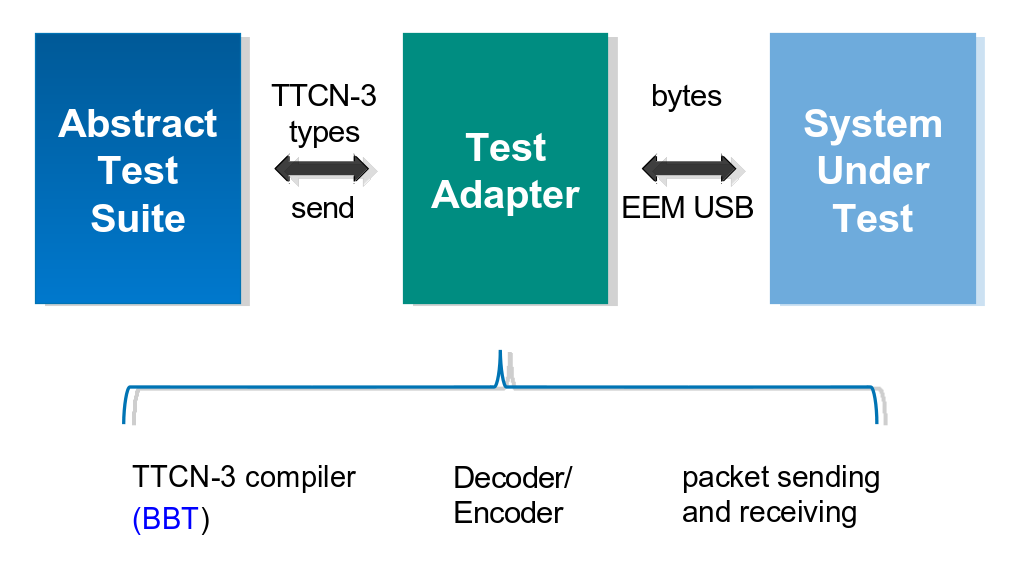
The next layer is the application layer. Here you can find the UDP and TCP protocols. The TCP protocol also includes the cryptographic protocol TLS with the RSA, AES and SHA algorithms. A more detailed description on the implementation and the changes made to the LwIP-Stack can be found in the IP-UICC Implementation Documentation [3].

### 2.1.1. TTCN3

All tests are written in the TTCN-3 language (Testing and Test Control Notation Version 3) [7] which has been used for more than 15 years in standardization as well as industry and is specifically designed for black box testing and certification. The aim of using TTCN-3 is the automation of the tests in the IP-Stack project. TTCN-3 is an internationally standardized testing language designed and developed by the ETSI (European Telecommunications Standards Institute).

TTCN-3 as such is not executable. It requires the following components in order to executable: compiler, adapter and codec implementations (see Figure 2).

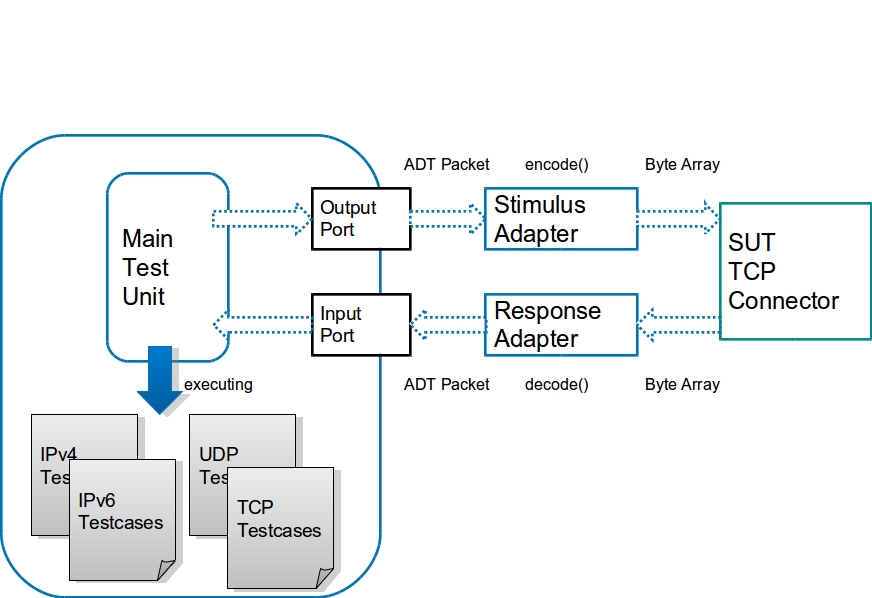
An abstract view of the interaction between the system running the actual test case and the system under test is given here:

**Figure 2:** Test environment components

The *Abstract Test Suite* defines the TTCN-3 test cases and sends messages (packets), that are defined inside the test cases to the *Test Adapter* which is responsible for encoding and decoding messages (The Test Adapter is written in C#) and the SUT for sending/receiving messages. After the sent message has been encoded into a byte-array, it is send to the *System Under Test* *(SUT)* which contains the IP-Stack implementation on the UICC. The SUT receives the message and sends a response in byte-array format according to the message back to the *Test Adapter*. Here the response is decoded and transmitted back to the *Abstract Test Suite*, which then determines whether the test has failed or passed, according to the response.

The compiler used to compile the TTCN-3 modules is the Fraunhofer TTCN3-Studio (ttcn-3.net). This compiler supports the complete TTCN-3 v2.2.1 standard. Since the compiler requires the .NET Framework, it has to run on a Windows machine. The adapters used for encoding and decoding are written in C#.

A more detailed view on the architecture of the test environment is given in the following graphic:



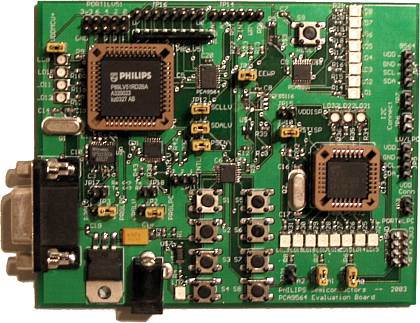
***Figure 3:*** *Abstract Test Environment*

The Figure 3 shows how the different components of the environment communicate with each other.

The test cases executed by the Main Test Unit (MTU) module send packets on outport ports of the MTU and receive response packets from the SUT on input ports. The packets send on the output port are encoded into byte-arrays by the Stimulus Adapter, so that the TCP Connector on the system under test (SUT) can receive and understand them properly.

### 2.1.2. Test Functionality Overview

In this section the connections / interactions between the components contained in the test environment are described.



EEM-USB

TCP-Connection

TCP-Connection

.net

TTCN3-Studio

MTU.ttcn3

TTCN-Studio

MTU.ttcn3

MTU.ttcn3

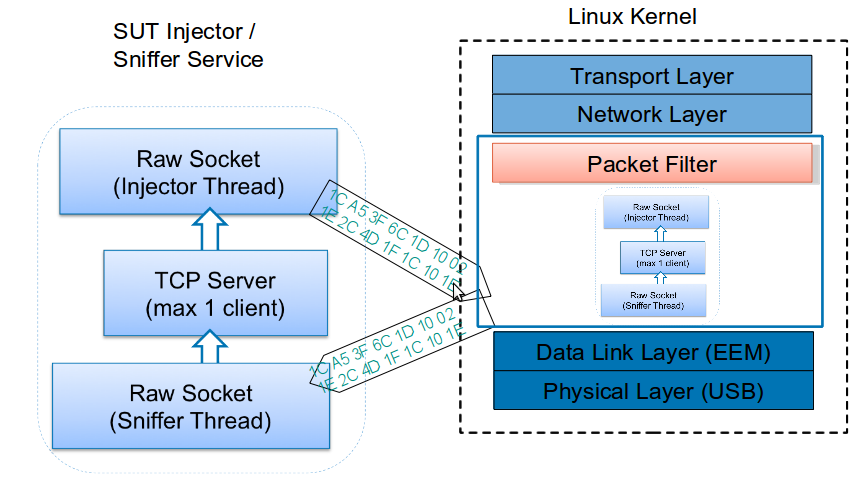
TTCN3-Studio

.net

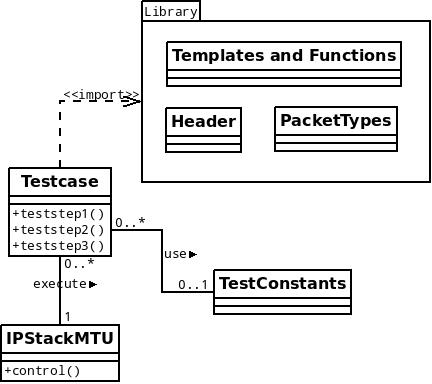
**Figure 4:** The Test-Setup

The actual TTCN-3 test modules are compiled and run inside a Windows machine, since the used TTCN-3 compiler (TTCN3-Studio) requires an installation of the Microsoft .NET Framework. The Windows machine is connected to a Linux machine via TCP-Connection. This connection is used to send properly encoded messages (hexstrings encoded to byte arrays) from the adapter running on the Windows machine to the Linux machine, which then is able to inject the received packet into the SUT as well as receive packets from the SUT using Sockets (c. Figure 2). Therefore a Linux kernel module has been implemented, which is described later on in this section.

The UICC containing the IP-Stack implementation resides on a test board which is connected to a PC running Linux (Ubuntu to be more precise) via USB and an EEM driver (Ethernet Emulation Module). As the internet protocol stack is highly configurable be sure that the configuration (provided functionalities, IP address, etc.) matches the test requirements of the test cases.

**Figure 5:** Packet injection and sniffing at Data-Link Layer

There are several components needed for the MTU to execute a test case. These components and their connection to the MTU are shown in Figure 6.



***Figure 6:*** *Test case and involved components*

A test case contains a certain number of test steps, which are executed when the test case itself is executed. Each test case sends packets to the SUT; these packets have headers which need to be defined. To define the headers of the packets to send in certain test cases Record-Types are used, that are imported from the congruent Library package. The Packet Types module defines complete packets e.g. an ICMPv4 packet consists of an ethernet header, an IPv4 header, an ICMPv4 header and an optional message. Most test cases are using the *TestConstants* module, which contains constants e.g. port numbers, IP-addresses, etc. encoded as hexstrings. These values are needed repeatedly and therefore they are defined in a central module from where they can be reused. Some test cases require the calculation of a CRC (Cyclic Redundancy Check) value, which is done by the *Templates and Functions* module automatically. Finally the test case execution is done by the *IPStackMTU* module inside the control function.

# Test Concept

This chapter describes the general structure of all testcases. It will describe which test cases exist and what different aspects are tested.

Furthermore here you get an overview how to read and understand the testcases and the corresponding test steps.

## Applied Test Design Techniques

The general idea of testing the IP-Stack implementation is to test every supported protocol of the stack. The tests are structured hierarchically by the OSI-model (Open Systems Interconnection model). It starts at the network layer with IP (IPv4/IPv6) and ICMP, then it comes to the transport layer and tests UDP and TCP. In the end it tests the cryptographic protocol TLS in the session layer.

In the following list you can find all the test cases in hierarchical order:

1. **IP\_IPv4**: Testing the functionality of the Internet Protocol in Version 4
2. **IP\_IPv6:** Testing the functionality of the Internet Protocol in Version 6
3. **IP\_ICMP**: Testing a subset of all ICMP protocol functions including the echo(Ping) functionality
4. **IP\_ICMPv6**: Testing the echo(Ping) functionality of ICMP
5. **IP\_UDP\_IPv4**: Testing the UDP protocol with the underlay version of IPv4
6. **IP\_UDP\_IPv6**: Testing the UDP protocol with the underlay version of IPv6
7. **IP\_TCP\_IPv4**: Testing the TCP protocol with the underlay version of IPv4
8. **IP\_TCP\_IPv6**: Testing the TCP protocol with the underlay version of IPv6
9. **IP\_TLS**: Testing the TLS protocol with underlay version of IPv4 and IPv6. Not implemented yet.

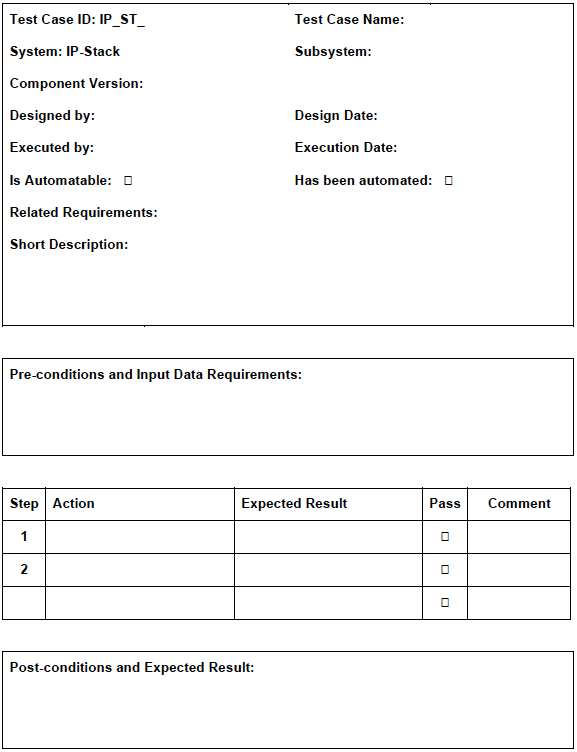
The design of the test cases is very similar. Every test case is divided into different steps. The first step is to establish a valid connection with one protocol of the IP-Stack and sending some data. The next step is to test the header of the different protocols. There are good and bad cases with a variety of correct and incorrect header fields. Whereas the good test cases test expected behaviour and the bad test cases examine which flaws in the implementation could be used for destructive or malicious purposes. Additionally there are also tests that cover most of the options of a protocol which are usually set in the header fields. Another field of testing are the stress tests, where data will continuously be send through an established connection. For example the same data will be send for about 15 minutes or one big data package will be send.

The last step is to test the states and the flow through the states of a protocol. For example the TCP protocol is implemented as a state machine. That means there are test cases which test the normal and valid flow, the good cases, and there are also some bad test cases which are trying to use non-existent paths through the states.

## Test Case Template

A test case specification must contain at least the following information:

* test case ID
* meaningful description of the test case objective (short form)
* related requirements
* detailed description (precondition, input data requirements, steps, test idea, post condition,...),
* definition of expected/predicted result
* check boxes for whether the test is automatable and whether it has been automated

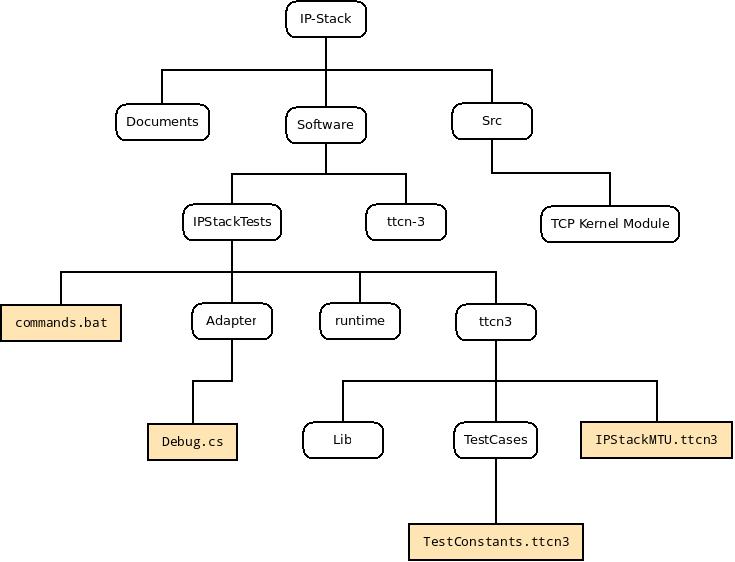
The following template in Figure 7 is used [5]:

**Figure 7:** Testcase specification template

The design of how to describe a test case with the test steps is displayed in Figure 7. The first part is a header with all basic information. For example there you can find the name of the test case and the version of implementation which is tested. The next part contains the preconditions and data requirements. There you get information about necessary conditions before starting a test case. After the precondition there is a list of all test steps. There the information about every test step can be found. In this section you can see what the step is doing, what the expected result is, whether this step has passed and there is also space for a small comment. The last field is for post-conditions which describe the status of the test component after all test steps have been executed.

# Project Structure

This chapter provides a quick overview on how the project is structured and shows some important files needed for test case execution.



**Figure 8:** IP-Stack folder structure

The graphic shows the folder structure of the IP-Stack project. In the following section the folders and their contents are depicted.

**IP-Stack :** The root folder

**Documents :** Contains the documentation of the project

**Src :** Holds the kernel module for the Linux machine

**Software:** : Contains all files associated with test cases

**IPStackTests** : Root folder of all test case related files, also the commands.bat needed for compiling and executing test cases is located here

**Adapter** : The adapter and codec implementations are contained in this folder. Also contains the Debug module, which controls the grain of the log messages for test case execution

**ttcn-3** : Contains library files needed for the compiler

**runtime** : During a test case run, all involved test case modules are copied here

**ttcn3** : Root folder for test case modules and associated libraries. In this folder the IPStackMTU module is located which is the module to execute all other test cases

**Lib** : Contains the libraries used by the test cases including header- and template modules

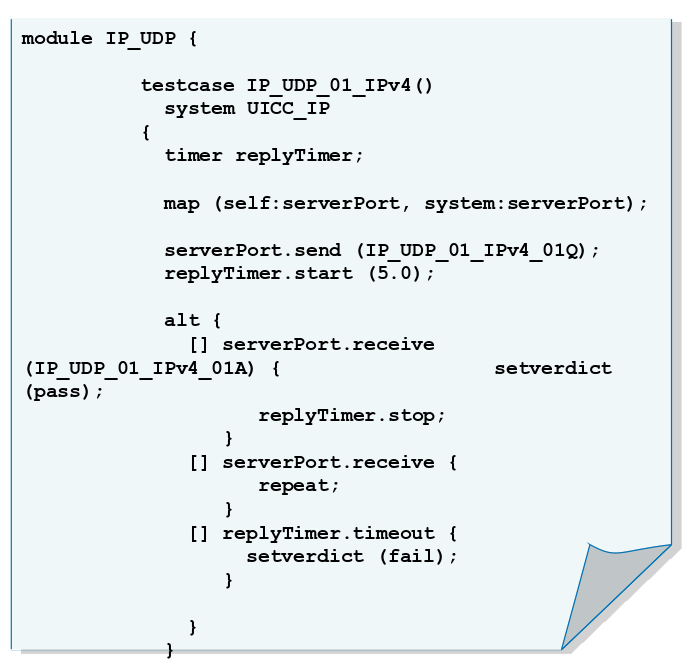
**TestCases** : In this folder all TTCN-3 test case modules including their respective test steps can be found

# Test Case Implementation

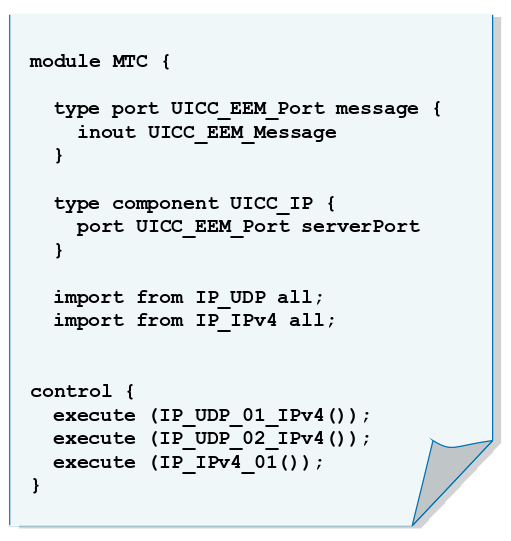
This chapter explains how TTCN-3 test case modules are implemented and executed in previously described test environment.

## Implementation Example

The TTCN-3 test case structure will be explained in detail using the following example:

***Figure 9:*** *TTCN-3 module containing one test case*

A TTCN-3 module typically defines several test cases using the keyword *testcase*. A test case typically maps a system-port and a self-port. The ports are defined in another module. The key operation done inside a test case is the sending of some message. This is done by calling the send function on the port on which the message should be send. In most cases a message and a response type are defined to specify what should be send and what is expected as response (Examples below). After the sending of the message there is statement which defines response alternatives (serverPort.receive). If the response matches the defined response type *IP\_UDP\_01\_IPv4\_01A* the test case has passed (setverdict(pass)). If there is no immediate response, the test case will wait for a certain amount of time. Finally there is a timer which defines how many seconds and milliseconds can pass without a response before a timeout event is triggered and the test case fails (setverdict.fail).

***Figure 10:*** *MTC definition and test case execution*

This module executes all previously defined test case-modules in the control function. Additionally this module defines the port used in the test cases using the keywords type port <portname> message. In the function body there is a statement, which defines what kind of messages can be send and received on this port using the keyword inout. Ports are contained inside components which are used upon test case definitions. The two import statements are needed so that the MTC module has a reference to the modules that contain the test cases to be executed.

Type definitions:

There are two TTCN-3 types which are quite important and frequently used.

**Record:**

A record specifies how data types for messages and responses should look like. According to our example: the message type *IP\_UDP\_01\_IPv4\_01Q* could look like this:

**type record *IP\_UDP\_01\_IPv4\_01Q* {**

**HexString6 srcAddr,**

**HexString6 dstAddr,**

**HexString4 message optional**

**}**

**Template:**

A special kind of data value called a template provides parametrization and matching mechanisms for specifying test data to be sent or received over the test ports i.e. A template initializes all defined fields within a record with fixed values or parameters:

**template *IP\_UDP\_01\_IPv4\_01A* (<parameters can be passed to the template here>){**

**HexString6 srcAddr := 'c0a80202'H,**

**HexString6 dstAddr := 'c0a80201'H,**

**HexString4 message := '01020304'H**

**}**

## Test execution

This section describes the test equipment representing the test setup and the specific configuration. The test set-up includes software or hardware tools supporting test aspects like execution, logging and comparisons.

To set up the test environment at least one Windows and one Linux PC as well as one testboard with the IP-Stack implementation on a UICC are required. The Windows PC needs to have an installation of the Microsoft .NET Framework as well as a TTCN-3 compiler (in the current implementation the TTCN3-Studio compiler was used). To use a different compiler the Test Adapter, that handles encoding and decoding, has to be rewritten according to the language the compiler is able to understand. Certainly the Windows PC needs to have the TTCN-3 test modules including the *IPStackMTU.ttcn3*.

The test cases can of course be modified or new test cases / test steps can be added. To edit a test case a simple text-editor is enough, however there is also an Eclipse-based IDE called *TRex* which understands TTCN-3 syntax and is developed by the University of Göttingen [8]. The usage of this IDE adds quite a lot of comfort in editing and writing TTCN-3 files.

The mentioned Linux PC needs to be connected to the SUT (via USB-EEM to the testboard) containing the UICC with the IP-Stack implementation. The Linux PC has to run the modified kernel module, mentioned earlier in order to inject / sniff packets. The communication traffic between the SUT and the Linux PC can be visualized by running the network analyser Wireshark. This has the advantage of making the debugging process easier.

As described in a former section, all test cases are defined within modules. In order to run a test case, the according module containing the test case has to be compiled and executed.

A batch script (commands.bat, see Figure 8 in chapter 4) is used to compile all test cases using the C# compiler and afterwards execute the IPStackMTU.ttcn3 module, which executes all test cases inside the control section sequentially.

After a test case has been executed according log messages are generated in the command prompt, where the commands.bat script has been executed, which tell whether the test case was successful or not. Additional to that the log gives information about which test steps were successful and which failed.

The granularity of these log informations can be modified using several log levels. The log level can be adjusted in the *Debug.cs* (see Figure 8 in chapter 4) file by changing the value of the *debugMode* variable. Currently there are two log levels available:

0 – Only information about which test cases and included test steps have passed is displayed

5 – Full packet information is displayed

## Kernel Module

The kernel module has been developed to perform the functions of injection and sniffing of data packets. This section gives a detailed explanation of the kernel module.

The kernel moudle named “tcpServerKernelModule.ko” is a kernel module written in C. The kernel module when loaded intializes a TCP Server that allows connection of atmost 1 client PC. Here the client PC is the WindowsMachine on which the TTCN-3 test case modules are running. The TCP server accepts the data that has been sent by the client. The data is the packet is encoded in byte array in hex-format. The packet is then injected to the SUT by the injector kernel thread at the data-link layer. The SUT reacts on the injected message and the response from the SUT is sniffed by the sniffer thread. The sniffed data is then sent back to the client.

Apart from injection and sniffing, the TCP kernel module also filters the unneccessary data from the linux machine to the SUT. A firewall has been implemented using netfilters to block all the TCP packets which are destined to the SUT over IPv4 and IPv6 [6, 11]. The implementation of the firewall is possible only inside the kernel space. Packet filtering is not possible at the user level. This is the primary reason the test case adapter was implemented as kernel module.

**Details of the kernel module**

Operating System used: Ubuntu 12.04 LTS (Precise Pangolin) (Should be compatible with all Ubuntu versions having an EEM implementation.)

Kernel version: 3.2.0.14

gcc version used: 4.6.3

**To complile the kernel module**

To compile the kernel module following commands are needed to saved inside the make file of the kernel module named <filename>.

obj-m += <filename>.o

all:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

The compilation of the <filename> using the make file generates a kernel object file, i.e., <filename>.ko

**Loading and Unloading the kernel module**

To load the kernel module into the Linux kernel following command can be used:

**$sudo insmod <filename>.ko**

To unload the kernel module following command can be used:

**$sudo rmmod <filename>.ko**

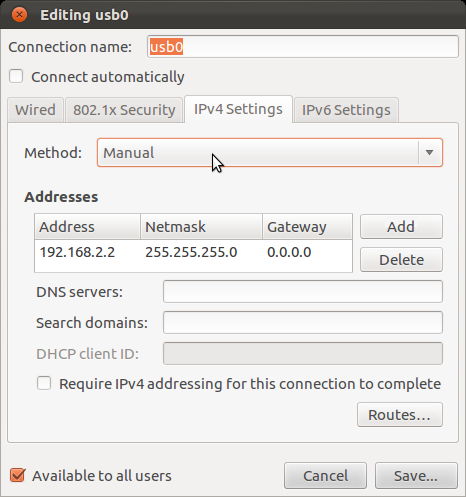
**Setting up the USB0 device**

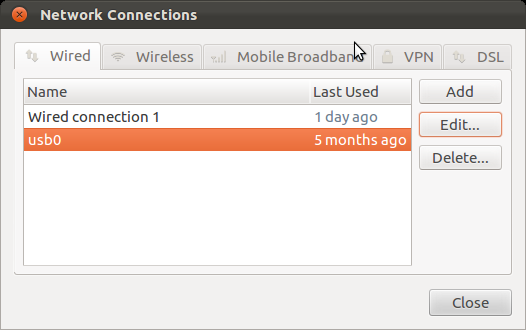
Since the SUT is connected over the USB0 interface using EEM implementation, the Ubuntu gives a random MAC and IP-addresses to the device. For this reason the USB0 device needs to assigned the MAC address and the IP address as used in the test constants file manually every time the linux machine is restarted.

1. To set up the MAC address execute the shell script as a root user: SetUpUSBMacAdd.sh

2. To update the IP address of the SUT go to Ubuntu Network manager > select Edit Connections > Select USB0 inside wired connections and click Edit (refer Figure 11a).

It should open a new window and goto IPv4 tab and assign the IP address as mentioned in the test constants file (refer Figure 11b).





**Figure 11a:** Network Connections **Figure 11b:** Editing USB0

**Figure 11:** Setting up the IP-address for USB0

Alternatively type the following sequence of commands into a terminal:

**$cd /etc/network/**

**$ sudo gedit interfaces** (Ubuntu 12.04)

or

**$ sudo gedit interfaces.d** (Ubuntu older than 12.04)

An editor will open where the IP address can be edited manually. This can be done by adding the following lines:

**auto usb0**

**iface usb0 inet static**

**address 192.168.2.2**

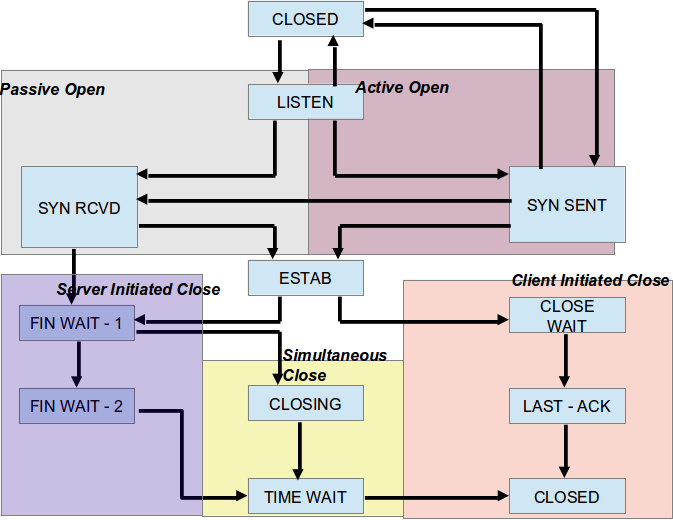
**netmask 255.255.255.0**

**Some known issues:**

1. Some times unloading the kernel module does not release the USB socket inside the kernel. Thus the SUT is not able to connect to the Linux again. A restart of the Linux PC is suggested to solve the problem.
2. The SUT might get disconnected automatically and later reconnected again. Usually done by Ubuntu it releases the USB and then reconnects again,
3. Unloading the kernel module while the test case execution might cause kernel panic.

# TCP Test Specification

In this chapter the test cases related to the TCP Protocol will be introduced to provide a better understanding of the underlying communication, since TCP is more complex than other tested protocols where it was only necessary to send and receive a message. The following documentation does not cover every single test case executed by the framework. However, it gives an overview on the strategy employed for developing the test cases.

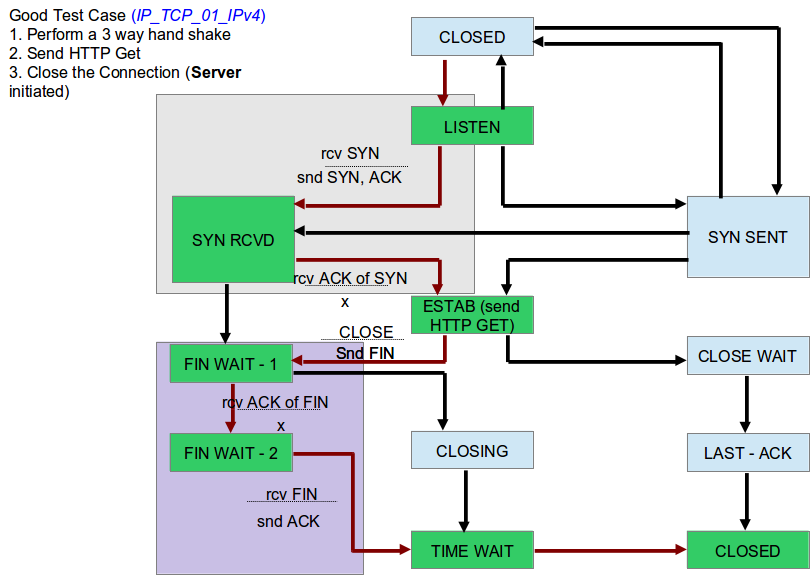


**Figure 12:** TCP Finite State Machine [9,10]

Figure 12 represents the TCP finite state machine. Table 1 explains the state transitions [9, 10].

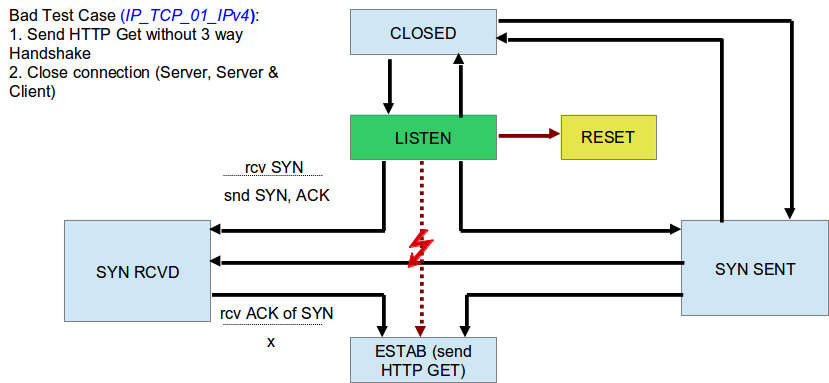
|  |  |  |
| --- | --- | --- |
| **State** | **State Description** | **Events and transitions** |
| ***CLOSED*** | Default state at which each TCP connection starts. | **Passive Open:** Server initialtes the connection on a TCP port and transitioning to *LISTEN* state. |
| **Active** **Open**: Client begins connection setup by *sending* SYN message and transitioning to *SYN-SENT* state. |
| ***LISTEN*** | Server waitinig for a SYN message from any remote client. | **Receive SYN:** Server receives *SYN* message from client. It responds with *SYN+ACK* message and enters into state *SYN RCEIVED* |
| ***SYN-SENT*** | Client has sent a *SYN* message to the server and waiting for the matching *SYN* message | **Receive SYN:** Client receives *SYN* and not an ACK for its *SYN*. Client acknowledges the *SYN* and transitions to *SYN RCEIVED* state. |
| **Receive SYN+ACK:** Client receives *SYN* and *ACK* for its *SYN* from the server, it transitions to *ESTABLISHED* state. |
| ***SYN RCEIVED*** | Both the connecting parties i.e the server and the client have sent *SYN* messages and waiting for *ACK* message to finish connection setup. | **Receive ACK:**  The device receives *ACK* for its *SYN* it transitions to the state *ESTABLISHED*. |
| ***ESTABLISHED*** | Represents open connection. Normal state for data transfer. | **Send FIN:** A device can close the connection by sending message with *FIN flag* set. It transitions to state *FIN-WAIT-1* |
| **Receive FIN:**  A device may receive from partner to close the connection. The device would acknowledge the close request and transition to state *CLOSE-WAIT.* |
| ***CLOSE-WAIT*** | A device receives close request from the partner. | **Send FIN:** The device sends *FIN* message to the partner who has already sent a closing request. After sending the *FIN* message the device transitions to *LAST-ACK* state |
| ***LAST-ACK*** | Device has already acknowledged the FIN it has received. It has set a FIN message and waiting for an acknowledgment. | **Recieve ACK:** The device has received *ACK* for its *FIN* and it moves to *CLOSED* state. |
| ***FIN-WAIT-1*** | A device is waiting for the *ACK* of the *FIN* it has already sent OR waiting for a *connection termination* request from the remote TCP. | **Receive ACK for FIN:** The device receives the *ACK* for the *FIN* it has sent previously. It transitions to *FIN-WAIT-2* |
| **Receive FIN:** Instrad of an *ACK* for the previously set *SYN* the device receives a *FIN* from remote TCP. It acknowledges and transitions to *CLOSING*. |
| ***FIN-WAIT-2*** | Device has received *ACK* for the FIN it has sent and is waiting for *FIN* from the partner. | **Receive FIN:** The device receives FIN, it acknowledges it and moves to TIME-WAIT state. |
| ***CLOSING*** | Device has received *FIN* from the partner and has sent *ACK*. The device however has still not received *ACK* for its own *SYN*. | **Receive ACK:** The device receives *ACK* for its *FIN* and transitions to state *TIME-WAIT*. |
| ***TIME-WAIT*** | Represents waiting for enough time to pass to be sure remote TCP recieved the acknowledgement of its connection termination request. | **Time Expiration:**  After designated wait period, device transitions to *CLOSED* state. |
| ***CLOSED*** | Represents no connetion state at all. |  |

**Table 1:** Description of TCP state transitions [9,10]



**Figure 13**: A good test case in TCP.

The Figure 13 represents the state transitions for a good test case in TCP. The path flow has been represented by the red lines. The server performs a *passive open*  and waits for client to connect to the TCP port. The server receives SYN message from the client. It responds with a SYN, ACK and transitions to state SYN RCVD. The client sends the ACK for the SYN ACK and transitions to state ESTAB. The data transfer takes place here where the client send the HTTP GET request. The server closes the connection by sending the FIN message and transitions to state FIN-WAIT-1. The client responds with ACK (with FIN flag set inside the ACK message) for the FIN thus moving to state FIN-WAIT-2. The server responds with ACK for the FIN flag and moves to TIME-WAIT. Both server and client for predefined time period before moving to CLOSED state.



**Figure 14:** A bad test case in TCP

Figure 14 represents a bad test case for TCP. Here the Server performs a passive open by opening a TCP port for the client to connect. The client instead of sending a SYN message to initiate TCP connection it tries to send a HTTP GET request. Thus the server responds with a RESET message as no SYN message has been received from client for TCP connection initiation.

## 6.1. Test Basis

The test basis is the set of documents that specify the requirements of the component and on which all test cases of the test-suite are based. The requirement specifications [1, 2] are not detailed enough and have to be supplemented by an analysis of source code and implementation documentation [3].

In this section the list of the RFCs used to define the test cases and steps are provided.

|  |  |
| --- | --- |
| **Protocol** | **RFC (Request For Comments)** |
| UDP (User Datagram Protocol) | RFC 768 <http://www.ietf.org/rfc/rfc768.txt> |
| ICMP (Internet Control Message Protocol version 4) | RFC 792 <https://tools.ietf.org/html/rfc792> |
| ICMPv6 (Internet Control Message Protocol version 6) | RFC 4443 <http://tools.ietf.org/html/rfc4443> |
| IPv4 (Internet Protocol version 4) | RFC 791 <http://www.ietf.org/rfc/rfc791.txt> |
| IPv6 (Internet Protocol version 6) | RFC 2460 <http://www.ietf.org/rfc/rfc2460.txt> |
| TCP (Transfer Control Protocol) | RFC 793 <http://www.ietf.org/rfc/rfc793.txt> |
| TLS (Transport Layer Security) | RFC 2246 (TLS v1.0) <http://www.ietf.org/rfc/rfc2246.txt> |
| RFC 4346 (TLS v1.1)  <http://tools.ietf.org/html/rfc4346> |
| RFC 5246 (TLS v1.2)  <http://tools.ietf.org/html/rfc5246> |

**Table 2:** List of Protocols and associated RFCs

## Test Results

This chapter gives an overview of the current implementation status of the automatic test execution framework. In addition the current test results will be listed. A detailed overview on the implementation and result status of every test case can be found in the corresponding test case specification report inside the printed IP-Stack Test Documents folder.

State: 03.01.2012

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ICMPv6 Test Cases** | | | | | |
| Test Case ID | Test Case Name | Subsystems | Steps | Automated | Passed |
| IP\_ ICMPv6\_01\_IPv6 | ICMP Invalid Type Tests | IPv6 | 7 | 7/7 | 7/7 |
| IP\_ ICMPv6\_02\_IPv6 | ICMP Invalid Code Tests | IPv6 | 13 | 13/13 | 13/13 |
| IP\_ ICMPv6\_03\_IPv6 | ICMP Echo Tests | IPv6 | 6 | 6/6 | 6/6 |
| IP\_ ICMPv6\_04\_IPv6 | ICMP Checksum Tests | IPv6 | 2 | 2/2 | 2/2 |
| IP\_ ICMPv6\_05\_IPv6 | ICMP NDP Tests | IPv6 | 10 | 9/10 | 9/9 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ICMPv4 Test Cases** | | | | | |
| Test Case ID | Test Case Name | Subsystems | Steps | Automated | Passed |
| IP\_ ICMP\_01\_IPv4 | ICMP Invalid Header Field Tests | IPv4 | 2 | 2/2 | 2/2 |
| IP\_ ICMP\_02\_IPv4 | ICMP Checksum Tests | IPv4 | 3 | 3/3 | 3/3 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **IPv6 Test Cases** | | | | | |
| Test Case ID | Test Case Name | Subsystems | Steps | Automated | Passed |
| IP\_ IPv6\_HDR1 | Invalid IPv6 Header Tests | ETH | 5 | 5/5 | 5/5 |
| IP\_ IPv6\_HDR2 | IPv6 Data Tests | ETH | 3 | 3/3 | 3/3 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **IPv4 Test Cases** | | | | | |
| Test Case ID | Test Case Name | Subsystems | Steps | Automated | Passed |
| IP\_ IPv4\_HDR1 | Invalid IPv4 Header Tests | ETH | 6 | 6/6 | 6/6 |
| IP\_ IPv4\_HDR2 | Ipv4 Fragmentation Tests | ETH | 3 | 3/3 | 3/3 |
| IP\_ IPv4\_HDR3 | Ipv6 Data Tests | ETH | 4 | 4/4 | 4/4 |
| IP\_ IPv4\_HDR4 | Ipv6 Options Tests | ETH | 0 | 0/0 | 0/0 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **UDP over IPv4 Test Cases** | | | | | |
| Test Case ID | Test Case Name | Subsystems | Steps | Automated | Passed |
| IP\_ UDP\_01\_IPv4 | UDP Data Transfer IPv4 | IPv4 | 5 | 5/5 | 5/5 |
| IP\_ UDP\_02\_IPv4 | UDP Stability IPv4 | IPv4 | 1 | 1/1 | 1/1 |
| IP\_ UDP\_03\_IPv4 | UDP Multiple Connections IPv4 | IPv4 | 2 | 2/2 | 2/2 |
| IP\_ UDP\_04\_IPv4 | UDP Connection and TCP Connection | TCP, IPv4 | 2 | 0/2 |  |
| IP\_UDP\_05\_IPv4 | Invalid UDP Data Transfer | IPv4 | 3 | 2/3 | 2/2 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **UDP over IPv6 Test Cases** | | | | | |
| Test Case ID | Test Case Name | Subsystems | Steps | Automated | Passed |
| IP\_ UDP\_01\_IPv6 | UDP Data Transfer IPv6 | IPv6 | 5 | 5/5 | 5/5 |
| IP\_ UDP\_02\_IPv6 | UDP Stability IPv6 | IPv6 | 1 | 1/1 | 1/1 |
| IP\_ UDP\_03\_IPv6 | UDP Multiple Connections IPv6 | IPv6 | 2 | 2/2 | 2/2 |
| IP\_ UDP\_04\_IPv6 | UDP Connection and TCP Connection | TCP, IPv6 | 2 | 0/2 |  |
| IP\_UDP\_05\_IPv6 | Invalid UDP Data Transfer | IPv6 | 3 | 2/3 | 2/2 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **TCP over IPv4 Test Cases** | | | | | |
| Test Case ID | Test Case Name | Subsystems | Steps | Automated | Passed |
| IP\_ TCP\_01\_IPv4 | TCP Connection Setup | IPv4 | 6 | 4/6 | 4/4 |
| IP\_ TCP\_02\_IPv4 | TCP Connection Stability | IPv4 | 4 | 2/4 | 2/2 |
| IP\_ TCP\_03\_IPv4 | TCP Connection Close | IPv4 | 6 | 0/6 |  |
| IP\_ TCP\_04\_IPv4 | TCP Checksums | IPv4 | 4 | 4/4 | 4/4 |
| IP\_ TCP\_05\_IPv4 | TCP Multiple Connections | IPv4 | 8 | 0/8 |  |
| IP\_ TCP\_06\_IPv4 | TCP Options | IPv4 | 5 | 0/5 |  |
| IP\_ TCP\_07\_IPv4 | TCP Variable Segment Size | IPv4 | 0 | 0/0 |  |
| IP\_ TCP\_08\_IPv4 -  IP\_ TCP\_12\_IPv4 | TCP Security 1-5 | IPv4 | 0 | 0/0 |  |
| IP\_ TCP\_13\_IPv4 | TCP Security 6 | IPv4 | 4 | 4/4 | 4/4 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **TCP over IPv6 Test Cases** | | | | | |
| Test Case ID | Test Case Name | Subsystems | Steps | Automated | Passed |
| IP\_ TCP\_01\_IPv6 | TCP Connection Setup | IPv6 | 6 | 4/6 | 4/4 |
| IP\_ TCP\_02\_IPv6 | TCP Connection Stability | IPv6 | 4 | 2/4 | 2/2 |
| IP\_ TCP\_03\_IPv6 | TCP Connection Close | IPv6 | 6 | 0/6 |  |
| IP\_ TCP\_04\_IPv6 | TCP Checksums | IPv6 | 4 | 4/4 | 4/4 |
| IP\_ TCP\_05\_IPv6 | TCP Multiple Connections | IPv6 | 8 | 0/8 |  |
| IP\_ TCP\_06\_IPv6 | TCP Options | IPv6 | 5 | 0/5 |  |
| IP\_ TCP\_07\_IPv6 | TCP Variable Segment Size | IPv6 | 0 | 0/0 |  |
| IP\_ TCP\_08\_IPv6 -  IP\_ TCP\_12\_IPv6 | TCP Security 1-5 | IPv6 | 0 | 0/0 |  |
| IP\_ TCP\_13\_IPv6 | TCP Security 6 | IPv6 | 4 | 4/4 | 4/4 |

# Appendix

1. Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Type of change | Team / Author |
| 1.0 | 01.04.11 | * initial draft | Stefan Grösbrink |
| 1.1 | 14.11.12 | * include first ideas, and small modifications of the structure | Christoph Topmöller |
| 1.2 | 06.12.12 | * extension and modification of the content | Alexander Botter |
| 1.3 | 21.12.12 | * Added TCP FSM and corrected few typos | Rakesh Garimella |

Table 3: Revision History

B. References

|  |  |
| --- | --- |
| [1] | Sagem Orga, s-lab; Configurable IP-Stack for Smart Cards, project description, November 3, 2009 |
| [2] | Sagem Orga, s-lab; Konfigurierbarer IP-Stack für Smart Cards, Projektbeschreibung, November 4, 2010 |
| [3] | Sagem Orga, s-lab; IP-Stack on UICC Development Documentation; implementation specification, 2011 |
| [4] | Stefan Grösbrink, s-lab: Test Case Specification Template, 2011. |
| [5] | Adam Dunkels, lwIP – A leightweight TCP/IP Stack; CNA lab Swedish Institute of Computer Science; URL: [http://savannah.nongnu.org/projects/lwip](http://savannah.nongnu.org/projects/lwip/)/ |
| [6] | http://www.roman10.net/linux-kernel-module-basics-and-hello-world/ |
| [7] | ETSI – Testing and Test Control Notation; URL: <http://www.ttcn-3.org/> |
| [8] | University of Göttingen – TRex Refactoring and Metrics Tool; URL: <http://www.trex.informatik.uni-goettingen.de/trac> |
| [9] | <http://www.tcpipguide.com/free/t_TCPOperationalOverviewandtheTCPFiniteStateMachineF-2.htm> |
| [10] | <http://www.ietf.org/rfc/rfc793.txt> |
| [11] | [http://www.netfilter.org](http://www.netfilter.org/) |

C. Abbreviations

CTS Component Test Specification

CFS Component Functional Specification

CTC Component Test Concept

IP Internet Protocol

SUT System under Test

TCS Test Case Specification

UICC Universal Integrated Circuit Card

D. Definitions / Glossary

Test basis The documentation on which the test cases are based

E. Conventions / Notations

**Coding in normal text:**

A hexadecimal value is presented as ‘<value>’, e.g. ‘80’ indicates hexadecimal value 80 (decimal value 128).

A sequence of same hexadecimal values is presented as ‘<value>..<value>’, e.g. ‘FF..FF’ indicates a sequence of several bytes with the same value ‘FF’.

**Coding in tables:**

In tables all given values are hexadecimal, especially:

FF..FF Indicates a sequence of hexadecimal bytes, all with the same content ‘FF’

00..00 Indicates a sequence of hexadecimal bytes, all with the same content ‘00’

CC Indicates any defined constant value of one hexadecimal byte.

XX Indicates a variable content (value) of one hexadecimal byte.

XX XX Indicates a variable content (value) of two hexadecimal bytes.

XX..XX Indicates a sequence of hexadecimal bytes with a variable content (value).

CC..CC Indicates a sequence of hexadecimal bytes, all with the same defined

constant value