|  |  |
| --- | --- |
|  |  |

IPL4asp Test Port for TTCN-3 Toolset with TITAN, Description

# Abstract

The IPL4 test port establishes connection between the TTCN-3 test executor and SUT (System under Test) by transmitting and receiving messages and/or streams. It supports the internet layer protocols IPv4 and IPv6 and the transport layer protocols UDP, TCP (with or without SSL) and SCTP.

Contents

[1 Functionality 3](#_Toc513191530)

[1.1 Supported Internet Protocol version 3](#_Toc513191531)

[1.2 Several transport protocols 3](#_Toc513191532)

[1.3 Backward incompatibilities 4](#_Toc513191533)

[1.4 Platform dependencies and limitations 4](#_Toc513191534)

[1.5 System Requirements 5](#_Toc513191535)

[2 Feature list 5](#_Toc513191536)

[2.1 Transport protocol extensibility 6](#_Toc513191537)

[2.2 Multiple IP addresses 6](#_Toc513191538)

[2.3 Use of DNS names and IP addresses 6](#_Toc513191539)

[2.4 Connection handling at the user level 6](#_Toc513191540)

[2.5 Error propagation 6](#_Toc513191541)

[2.6 Run-time test port control 6](#_Toc513191542)

[2.7 Message dissection 7](#_Toc513191543)

[2.8 ASP-based notification of asynchronous events 7](#_Toc513191544)

[2.9 Automatic IP address discovery 7](#_Toc513191545)

[2.10 Connection option handling 7](#_Toc513191546)

[2.11 SSL functionality and support 7](#_Toc513191547)

[2.11.1 Compilation 8](#_Toc513191548)

[2.11.2 Authentication 8](#_Toc513191549)

[2.11.3 Pre-shared key authentication 9](#_Toc513191550)

[2.11.4 Certificate handling 9](#_Toc513191551)

[2.11.5 SSL limitations 9](#_Toc513191552)

[2.11.6 DTLS SRTP limitations 10](#_Toc513191553)

[2.11.7 ALPN support 10](#_Toc513191554)

[2.12 TLS hostname extension 10](#_Toc513191555)

[2.13 DTLS implementation 10](#_Toc513191556)

[2.13.1 Creation of DTLS associations 10](#_Toc513191557)

[2.13.2 Accepting incoming DTLS associations 11](#_Toc513191558)

[2.13.3 Starting TLS layer upon existing sockets 11](#_Toc513191559)

[2.13.4 Query of the local and peer certificate fingerprint (thumbprint) 11](#_Toc513191560)

[2.13.5 Generation on keys and salts for encrypting SRTP 11](#_Toc513191561)

[2.13.6 Generation of key for DTLS over SCTP 11](#_Toc513191562)

[2.13.7 Setting support for DTLS over SCTP 11](#_Toc513191563)

[2.13.8 Setting the supported SRTP profiles 12](#_Toc513191564)

[2.13.9 DTLS data exchange 12](#_Toc513191565)

[2.13.10 Stopping the TLS layer 13](#_Toc513191566)

[2.14 SCTP stack 13](#_Toc513191567)

[2.15 ConnId release 13](#_Toc513191568)

[2.16 Path MTU Discovery 13](#_Toc513191569)

[3 Test Port Usage 14](#_Toc513191570)

[3.1 Overview 14](#_Toc513191571)

[3.2 Installation and building the test port 14](#_Toc513191572)

[3.3 Extending the port 17](#_Toc513191573)

[3.4 Configuration 18](#_Toc513191574)

[3.4.1 General test port parameters 18](#_Toc513191575)

[3.4.2 Parameters for automatic connection during mapping 20](#_Toc513191576)

[3.4.3 Parameters for IP auto-configuration 20](#_Toc513191577)

[3.4.4 Parameters specifying the default connection options 22](#_Toc513191578)

[3.4.5 Parameters specifying SCTP connection options 24](#_Toc513191579)

[3.4.6 EIN SS7 Stack parameters 26](#_Toc513191580)

[3.4.7 Parameters specifying SSL connection options 27](#_Toc513191581)

[3.4.7.1 Parameters for setting PSK 29](#_Toc513191582)

[3.5 Useful functions 29](#_Toc513191583)

[3.6 Functions for IP auto-configuration 30](#_Toc513191584)

[3.7 Error messages 32](#_Toc513191585)

[3.8 Events 33](#_Toc513191586)

[3.9 Examples 34](#_Toc513191587)

[4 Interface description 34](#_Toc513191588)

[4.1 ProtoTuples 34](#_Toc513191589)

[4.2 Send and receive functions 35](#_Toc513191590)

[4.3 IPv6 Link local address handling 36](#_Toc513191591)

[4.4 Test port control functions 37](#_Toc513191592)

[4.4.1 Creating listener 37](#_Toc513191593)

[4.4.2 Creating connection 37](#_Toc513191594)

[4.4.3 SCTP Multihoming 38](#_Toc513191595)

[4.4.4 Setting connection options 38](#_Toc513191596)

[4.4.5 Getting connection options 39](#_Toc513191597)

[The f\_IPL4\_getOpt function only supports mtu\_discover for now. 39](#_Toc513191598)

[4.4.6 Closing connection 39](#_Toc513191599)

[4.4.7 Abnormal close 40](#_Toc513191600)

[4.4.8 Setting user data 40](#_Toc513191601)

[4.4.9 Getting user data 40](#_Toc513191602)

[4.4.10 Getting user details 40](#_Toc513191603)

[4.4.11 Message dissection 41](#_Toc513191604)

[4.4.12 Message dissection function for binary protocols 42](#_Toc513191605)

[4.4.13 Send messages 42](#_Toc513191606)

[4.4.14 Start TLS over existing TCP/UDP connection 42](#_Toc513191607)

[4.4.15 Getting Path MTU 43](#_Toc513191608)

[5 Terminology 43](#_Toc513191609)

[5.1 Abbreviations 43](#_Toc513191610)

[6 References 44](#_Toc513191611)

# Functionality

The IPL4asp is a general purpose, session independent test port providing access to several transport layers over the Internet Protocols in an extensible manner. The most important features are the support of multiple IP addresses of both IPv4 and IPv6 and that the user of the test port can extend the base functionalities with session specific ones using either TTCN or external C++ function.

The IPL4asp test port has been demanded by projects to produce very high load, therefore performance must always be a key factor during the design and implementation.

The IPL4asp test port will replace the earlier UDP [8], TCP [9] with SSL support and SCTP [10] test ports, providing backward compatibility as far as possible, in order to reduce maintenance efforts.

## Supported Internet Protocol version

The IPL4asp test port shall support both currently used Internet Protocol versions, i.e. IPv4 and IPv6. The version of the Internet Protocol shall be hidden at the level of the test port API (see section 2.3).

## Several transport protocols

The IPL4asp test port shall support the UDP, TCP, TCP with SSL and SCTP transport layer (L4) protocols. The test port shall be designed in an extensible manner that other transport protocols can be added without modifying the framework and the test port API.

## Backward incompatibilities

The following backward incompatibilities exist (R1A -> R2A):

* The Result record was modified to include two more optional fields.  
  This may require update of code if templates for Result are used or a variable of Result type is assigned a value.
* The expected behaviour of the callback function, registered with f\_IPL4\_setGetMsgLen, has been slightly modified:  
  If the callback function returns a length bigger then the message part in the octetstring, then this value is stored and used later without calling the callback function again for the given message.
* The definition of Connection ID is modified:  
  It is a test port component level unique identifier of the connection. It is not guaranteed to be a file descriptor. The connection ID identifies also the protocol.

## Platform dependencies and limitations

The following table indicates which functionalities are available on different platforms:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Linux** | **Solaris** | **Cygwin** |
| **IPv6** | **X** | **X** | **X** |
| **Network interface setup** | **X** | **X** |  |
| **IP address discovery** | **X** |  |  |
| **Connection options** | **All** – See chapter 3.4.3 | | |
| **SCTP support** | **X** | **X** |  |

For guidance on building the test port and for compilation options see 3.2.

In order to take advantage of the SCTP support in IPL4 test port the following platforms can be used:

Solaris 10, or

Suse Linux 9.1 and above with SCTP packages installed as in 1.5

On Ubuntu 8.10 one of the standard API calls used in the test port may block the execution for several seconds. This is a known issue with Ubuntu which may be resolved by applying the latest updates to the OS. To overcome this problem a workaround has been implemented. If specific IP addresses are used in a valid dot notation form instead of domain names, no DNS name resolution is needed, and no calls made to the faulty function.

## System Requirements

In order to operate the IPL4asp test port the following system requirements must be satisfied:

* TITAN TTCN-3 Test Executor version R8A (1.8.pl0) or higher installed. For installation guide see [2]. Please note: The usage of TITAN releases earlier than R8A can result a dynamic test case error.
* The following functions from TCC Useful Functions [4] are required:

TCCInterface\_Functions  
(TCCInterface\_Functions.ttcn, TCCInterface.cc)

* The BSD socket interface must be supported, which should be present in most UNIX-like operating systems, such as Solaris, Linux and Cygwin.
* Root privileges are required when using automatic IP address discovery or IP address configuration.
* For SCTP support the following SCTP Linux Kernel implementation packages (or higher version) should be installed:

lksctp-tools-1.0.1-2.i586.rpm

lksctp-tools-devel-1.0.1-2.i586.rpm

* If SSL is used, the same OpenSSL must be installed as used in TITAN. For an OpenSSL installation guide see [5].
* Several functions relays on the OpenSSL library so the used OpenSSL version should support the required functionality. Please consult the OpenSSL manual for the exact versions.
  + The available chippers and SSL/TSL/DTLS versions depends on the OpenSSL versions. Check the OpenSSL manual
  + For DTLS support OpenSSL 1.0.1 or later version must be installed.
  + For DTLS over SCTP the OpenSSL 1.0.1q or later versions is required and the Linux kernel version >= 3.13.0.
  + For PSK support OpenSSL 1.0.2 or later version is required

# Feature list

The features enumerated below have been implemented in the test port.

## Transport protocol extensibility

The test port is designed in an extensible manner that other transport protocols can be added without modifying the framework and the test port API.

## Multiple IP addresses

The IPL4asp test port is handling multiple IPv4 and/or IPv6 addresses, be it available either on virtual or physical interfaces.

The test port provides run-time configuration support to set up virtual interfaces on start and shutting down those interfaces on stop. The physical interface is handled using TCCUsefulFunctions [4].

## Use of DNS names and IP addresses

The IPL4asp test port supports IP addresses in the dotted notation format by default, but DNS names can be used, too.

## Connection handling at the user level

A connection is identified by a connection tuple in connection creation (listen or connect). Later the connection is identified by the connection ID: during message sending receiving, closing, connection property modification and in error notifications. See section 4.4.

As the test port is independent of the upper protocol, the user has to maintain the mapping of connection tuples to any object, e.g. the client in case of the SIP protocol.

## Error propagation

The IPL4asp test port does not generate TTCN error at the point of either protocol or test port error. The immediate error generation behavior can be replaced with a more sophisticated, run-time configurable error handling mechanism, which passes the error to the user who can make decisions how to continue the test execution. See section [4] for details.

## Run-time test port control

The user is able to perform the following operations:

* Open connection
* Modify connection properties
* Close connection

The above calls are implemented as external functions that can achieve the desired effect by calling some public member control functions of the test port. See section 4.4.4 for details.

This strategy does not interfere with the earlier applied ASP-based test port control, as that may be handled in port extension (see provider port in [3]).

## Message dissection

A TTCN-3 or external C++ function can be used to find the message boundary in streams, which is implemented as a function reference registration in the test port.

The function is declared for the test port, but the user shall implement the function depending on what session protocol is used.

The implementation of this feature is replaced with the sliding function prototype (see [3]) as soon as it is supported. See section 4.4.10 for details.

## ASP-based notification of asynchronous events

In the IPL4asp test port, asynchronous events may occur that the user is interested in, e.g. a connection is opened or closed by the peer endpoint.

The test port provides an ASP to inform the user about such events. See section 3.8 for details.

## Automatic IP address discovery

The test supports IP address discovery via DHCP and ARP for IPv4 addresses as functions. The functions return the IP addresses which can be configured using the functions of TCCUsefulFunctions [4].

The parameters for the IP address discovery (such as the number of requested addresses or the name of the lease file) shall be given either as parameters of the function or as test port parameters in the run time configuration file, see 3.4.2.

## Connection option handling

The test port makes it possible to specify connection options during connection creation and for an open connection. See details in section 4.4.4.

## SSL functionality and support

From revision R6A—beside UDP, TCP and SCTP—SSL can also be used as transport channel. The same version of OpenSSL library must be used as in TITAN.

From version R16A, DTLS [11] with DTLS SRTP [12] is supported, with the exception of demultiplexing method described in DTLS SRTP [12] section 5.1.2 Reception, which is not supported. The test port assumes SRTP packets if SRTP is configured for the association.

From version R20A DTLS over SCTP is supported.

From version R30A TLS-PSK and DTLS-PSK is supported.

The supported SSL, TLS and DTLS versions depend on the used OpenSSL library.

The supported SSL/TLS version can be disabled or enabled via test port parameter (see 3.4.6) on test port instance level or via options (see 4.4.4) on connection level.

See sections 3.2, 3.4.3, and 3.4.6.

### Compilation

The usage of SSL and even the compilation of the SSL related code parts are optional. This is because SSL related code parts cannot be compiled without the OpenSSL installed.

The compilation of SSL related code parts can be disabled by not defining the *IPL4\_USE\_SSL* macro in the Makefile during the compilation. If the macro is defined in the Makefile, the SSL code parts are compiled to the executable test code. If it is not defined all SSL related request will cause an error result message about the not supported transport.

### Authentication

The IPL4 test port provides both server side and client side authentication. When authenticating the other side, a certificate is requested and the own trusted certificate authorities’ list is sent. The received certificate is verified whether it is a valid certificate or not (the public and private keys are matching). No further authentication is performed (e.g. whether hostname is present in the certificate). The verification can be enabled/disabled in the runtime configuration file, see 3.4.6.

From server side the test port will always send its certificate and trusted certificate authorities’ list to its clients. If verification is enabled in the runtime configuration file, the server side will request for a client’s certificate. In this case, if the client does not send a valid certificate or does not send a certificate at all, the connection will be refused. If the verification is disabled, the connection will never be refused due to verification failure.

From client side the test port will send its certificate to the server on the server’s request. If verification is enabled in the runtime configuration file, the client will send its own trusted certificate authorities’ list to the server and will verify the server’s certificate as well. If the server’s certificate is not valid, the SSL connection will not be established. If verification is disabled, the connection will never be refused due to verification failure.

The own certificate(s), the own private key file, the optional password protecting the own private key file and the trusted certificate authorities’ list file can be specified in the runtime configuration file, see 3.4.6.

The test port will check the consistency between its own private key and the public key (based on the own certificate) automatically. If the check fails, a warning is issued and execution continues.

### Pre-shared key authentication

The client indicates its willingness to use pre-shared key authentication by including one or more PSK ciphersuites in the ClientHello message, the allowed ciphering suite can be specified (set **ssl\_allowed\_ciphers\_list** to **“PSK”**) the TLS server selects one of the PSK ciphersuites, places the selected ciphersuite in the ServerHello message. It can provide a "PSK identity hint" in the ServerKeyExchange message. The Certificate the CertificateRequest and the CertificateVerify messages are not sent if PSK is used (the parameters related to the certificate should not be set). The TLS handshake is authenticated using the Finished messages as usual. PSK related parameters can be specified in the runtime configuration file see 3.4.7.1.

### Certificate handling

By default, the globally defined certificate is used by all connection. In order to use a connection specific certificate, the test port parameter “TLS\_CERT\_PER\_CONN” should be set to “YES” in the run time configuration file. Once the “TLS\_CERT\_PER\_CONN” is set to yes, the connection specific certificate parameters can be supplied via the options parameter of the connect and listen functions.

### SSL limitations

* The SSL re-handshaking requests are accepted and processed, however re-handshaking cannot be initiated from the test port.
* The usage of SSL session resumption can be enabled/disabled in the runtime configuration file, see 3.4.7.
* The own certificate file(s), the own private key file and the trusted certificate authorities’ list file must be in PEM format. Other formats are not supported.
* The allowed ciphering suites can be restricted in the runtime configuration file, see 3.4.6.

### DTLS SRTP limitations

* The demultiplexing method described in DTLS SRTP [12] section *5.1.2 Reception* is not supported. The test port assumes SRTP packets if SRTP is configured for the association.

### ALPN support

The test port supports the ALPN TLS extension and the ALPN negotiation.

Please note that the ALPN support requires OpenSSL 1.0.2 at least.

## TLS hostname extension

The test port supports the TLS hostname extension as client only.

## DTLS implementation

DTLS support has been implemented only on UDP and SCTP in the test port, UDPLight can be implemented upon request.

Current DTLS implementation supports:

* + Creation of DTLS associations
  + Accepting incoming DTLS associations
  + Starting TLS layer upon existing sockets
  + Query of the peer certificate fingerprint (thumbprint)
  + Generation on keys and salts for SRTP
  + Generation of key for SCTP
  + DTLS data exchange
  + Stopping the TLS layer

### Creation of DTLS associations

Use f\_IPL4\_connect() function with dtls := { udp := {} } or dtls := { sctp := {0,0,0,0} } prototuple to initiate DTLS connection towards the remote peer. The test port will initiate the client-side handshake of the DTLS association, and report AVAILABLE event when it’s finished.

### Accepting incoming DTLS associations

Use f\_IPL4\_listen() function with dtls := { udp := {} } or dtls := { sctp := {0,0,0,0} } prototuple to initiate DTLS listening port. The test port will perform the server-side handshake if a client initiates a TLS handshake.

### Starting TLS layer upon existing sockets

Use f\_IPL4\_StartTLS() function to initiate the client- or server side TLS layer on top of already opened UDP or SCTP sockets.

### Query of the local and peer certificate fingerprint (thumbprint)

Use f\_IPL4\_getLocalCertificateFingerprint() function to query the fingerprint of the local (test port’s) certificate’s fingerprint. If the filename is supplied the fingerprint of the certificate file is returned. Otherwise, the fingerprint of the certificate belongs to the given connID is returned.

Use f\_IPL4\_getPeerCertificateFingerprint () function to query the fingerprint of the peer (remote side) certificate’s fingerprint. This function will return valid fingerprint only if the DTLS association has been established (the DTLS handshake is done).

### Generation on keys and salts for encrypting SRTP

Use the f\_IPL4\_exportSrtpKeysAndSalts() function to generate keys and salts for SRTP encrypting. The function is the implementation of the exporter function described in section 4.2 Key Derivation in [12]. This function will return valid fingerprint only if the DTLS association has been established, because it uses the shared secret agreed during the handshake process.

### Generation of key for DTLS over SCTP

Use the f\_IPL4\_exportSctpKey () function to generate a key for SCTP over DTLS encryption. The function is the implementation of the exporter function described in section 4.2 Key Derivation in [12]. This function will return valid fingerprint only if the DTLS association has been established, because it uses the shared secret agreed during the handshake process.

### Setting support for DTLS over SCTP

In order to run DTLS over SCTP it is necessary to enable *net.sctp.auth\_enable*.

### Setting the supported SRTP profiles

If DTLS is used for SRTP key negotiation, then the supported SRTP protection profiles need to be set prior to the DTLS association is established. Current OpenSSL version (1.0.1g) supports the ‘SRTP\_AES128\_CM\_SHA1\_32’ and ‘SRTP\_AES128\_CM\_SHA1\_80’ protection profiles. The profile names must be separated by colons, ie. ‘SRTP\_AES128\_CM\_SHA1\_32:SRTP\_AES128\_CM\_SHA1\_80’

* Use the f\_IPL4\_setOpt() function and set the *options* / *dtlsSrtpProfiles* field to set the supported SRTP profiles on an existing endpoint (socket). If the connection Id passed to f\_IPL4\_setOpt() is ‘-1’, then all subsequent DTLS handshakes will use the specified selection profiles by default.
* Use the f\_IPL4\_listen() function and set the *options* / *dtlsSrtpProfiles* field to set the supported SRTP profiles for the server endpoint (socket). The specified selection profile will be used in the DTLS handshakes to agree in the SRTP selection profile with the clients.
* Use the f\_IPL4\_connect() function and set the *options* / *dtlsSrtpProfiles* field to set the supported SRTP profiles for the client endpoint (socket). The specified selection profile is used in the DTLS handshake to agree in the SRTP selection profile with the remote peer.

### DTLS data exchange

Use the f\_IPL4\_send() function to send data. If the function is called with UDP prototuple, then the test port will send the data unencrypted (SRTP packets need to be sent this way), otherwise it encrypts as DTLS. In the same way the function will send encrypted messaged over the SCTP stream if the DTLS encryption is enabled.

On incoming data ASP\_RecvFrom is passed to the testcase with the received data. If SRTP selection profile is set on the DTLS association, then test port assumes the incoming data to be unencrypted, and passes it to the testcase without DTLS decryption. Demultiplexing method described in DTLS SRTP [12] section 5.1.2 Reception, which is not supported.

If DTLS over SCTP is enabled, in case of incoming data, the data will be first decrypted by the test port and then passed unencrypted to the test case.

If SRTP selection profile is not set on the DTLS association, then the data is DTLS unencrypted first, and then passed to the testcase.

### Stopping the TLS layer

## SCTP stack

The IPL4 test port can use either the kernel based SCTP stack or the SCTP API of the EIN SS7 stack.

The IPL4 test port supports local multi homing and probing of all IP addresses of the remote side with both SCTP stack.

See section 3.4.5 for configuration file parameters for EIN SS7 stack.

## ConnId release

How to release connId:

1. Traditional way: The connid is released as soon as either:  
   - The test port processed the incoming close event.  
   - or the f\_IPL4\_close was called.  
   Because the connId is released immediately by the test port, the test case code can try to use it, which leads faults. Also the test port can reuse the connId without the knowledge of the test case code
2. Confirmed mode: The connId is released only after the confirmation message. After the connId is ready for release (triggered by either the incoming close or f\_IPL4\_close) the test port put the ASP\_ConnId\_ReadyToRelease into the incoming queue. When the application processes the ASP\_ConnId\_ReadyToRelease it should call the f\_IPL4\_ConnId\_release function to confirm the release.

The confirmed mode can be activated by setting the test port parameter "connId\_release\_mode" to "confirmed".

## Path MTU Discovery

The IPL4 test port can read the Path MTU of a connected socket. See section 4.4.15 for further information.

# Test Port Usage

## Overview

The IPL4asp test port is a general purpose transport layer test port enabling one to use several different transport protocols over IPv4 or IPv6, with individual connection properties. This is achieved by applying the virtual networking host concept, which ensures the use of the whole port number region for each protocol, and the use of each protocol for each IP address representing one virtual networking host. Figure 1 shows one host using N protocols and the same M ports for each protocol. One may use as many of this virtual networking host as needed up to the constraints of the target operating system and hardware.



Figure 1 – The concept of the virtual networking host

The test port is a so called provider port, i.e. the user may define several different session specific test ports based on it, applying the encoding and decoding functions of the session protocols and maybe some more functionalities. For more information on provider ports see [3].

## Installation and building the test port

Since the IPL4asp test port is used as a part of the TTCN-3 test environment, this requires TTCN-3 Test Executor to be installed before any operation of the IPL4asp test port. For more details on the installation of TTCN-3 Test Executor see the relevant section of [2].

The IP addresses to use may be optionally pre-configured, but the run-time configuration of the test port enables one to set up and tear down virtual interfaces, if it is supported. These methods may be combined, too. Dynamic discovery of IPv4 addresses is also possible via function calls.

There are a few IPL4asp specific compilation options to be set for building the test port:

* + - * Platform setting:  
        The platform should be specified by assigning the PLATFORM variable one of the following values: LINUX, SOLARIS, SOLARIS8 or WIN32 (for Cygwin).
      * IP address discovery:  
        To enable this functionality “–DIP\_AUTOCONFIG” has to be added to CPPFLAGS in the Makefile. Currently only Linux is supported. (On other platforms this flag is ignored.)  
        For this functionality the PCAP library is needed, therefore the LINUX\_LIBS variable in the Makefile has to include “-lpcap”.  
        To build the test port on Linux without IP address discovery remove the above two settings.
      * IPv6:  
        The IPv6 parts of the code can be disabled by adding –DNO\_IPV6 to the CPPFLAGS in the Makefile, thus the code can be compiled on those hosts where IPv6 is not supported.
      * SCTP  
        SCTP support can be enabled by adding -DUSE\_SCTP to the CPPFLAGS in the Makefile thus the code can be compiled on those hosts where SCTP is supported.

The IPL4 is able to autodetect the version of the LKSCTP package, so the flags -DLKSCTP\_1\_0\_7 or -DLKSCTP\_1\_0\_9 should not be used, but accepted by the test port.

* + - * Local multihoming with LKSCTP  
        The IPL4 test port supports the multihomed local and remote SCTP endpoints. The local multihomed endpoint support with LKSCTP should be activated –DLKSCTP\_MULTIHOMING\_ENABLED CPPFLAGS in the Makefile

The **–lsctp** linker flag should be added to the linker command in order to compile the test suite with local multihomed SCTP endpoint support.

* + - * SCTP with EIN SS7 stack API [6]  
        The support of the EIN SS7 stack SCTP APIcan be enabled by adding –DUSE\_IPL4\_EIN\_SCTP to the CPPFLAGS in the Makefile thus the code can be compiled on those hosts where EIN SS7 stack API is available.  
        Note: Both SCTP stack can be enabled at the same time, test port parameter determines the actually used SCTP stack.

The **–leinsctp** of **–leinsctp\_r** linker flag should be added to the linker command in order to compile the test suite with EIN SS7 SCTP stack support.

* + - * SSL  
        The compilation of SSL related code parts can be enabled by adding the *IPL4\_USE\_SSL* macro to the CPPFLAGS in the Makefile.

When building the executable test suite the libraries compiled for the OpenSSL toolkit (if the *IPL4\_USE\_SSL* macro is defined) should also be linked into the executable along with the TTCN-3 Test Executor, i.e. the OpenSSL libraries should be set properly into the Makefile generated by the TITAN executor:

**OPENSSL\_DIR =**

Specifies the OpenSSL installation directory. It has to contain the lib/libssl.a file and the include/ directory.

**CPPFLAGS = -D$(PLATFORM) –DIPL4\_USE\_SSL -I$(TTCN3\_DIR)/include -I$(OPENSSL\_DIR)/include**

This line includes the OpenSSL header files and enables SSL code. It shall be used if SSL is used.

If no SSL is used, the generated Makefile by TITAN is suitable.

**LINUX\_LIBS = -lssl**

The “-lssl” specifies the OpenSSL runtime library. It shall be used if SSL is used. The best place to include into platform libs. E.g. if LINUX is add it to the LINUX\_LIBS as in the example above.

To compile the source files you will also need the OpenSSL developer toolkit which contains the header files used by the source. If Share Objects (.so) are used in the OpenSSL toolkit, to run the executable, the path of the OpenSSL libraries must be added to the LD\_LIBRARY\_PATH environment variable. For more information see [5].

* Note: There is no longer compilation option to disable (-DNO\_EPOLL) or enable the usage of EPOLL. Usage of EPOLL is implemented in TITAN – if the platform supports it.

Before running the demo please note:

* + - * Parts of the demo – using interface configuration or IP address discovery – can be run successfully only with root privileges.
      * Be careful before running the demo as it tries to reconfigure an Ethernet interface, broadcasts ARP requests and requests IP address leases from a DHCP server
      * The name of the Ethernet interface and the IP address values should be checked and modified in IPL4\_demo.ttcn and IPL4\_demo.cfg as necessary.

## Extending the port

As the port is controlled via the public member functions of the provider port, some special C++ files are required for user extensions of the port that the predefined control functions can be called.

E.g. the f\_IPL4\_close function requires the followings, if the user extended the provider port to user\_PT in the myTest module:

* External function declaration in the user’s ttcn file:  
     
   external function f\_IPL4\_close(  
   inout user\_PT portRef,  
   in ConnectionId id,  
   in ProtoTuple proto := { unspecified := {} }  
   ) return Result;
* External function definition in a C++ file of the user:  
    
   IPL4asp\_\_Types::Result f\_\_IPL4\_\_close(  
   myTest::user\_\_PT& portRef,  
   const INTEGER& id,  
   const IPL4asp\_\_Types::ProtoTuple& proto)  
   {  
   return f\_\_IPL4\_\_close(portRef, id, proto);

}

The demo directory contains the following template files:

* IPL4asp\_User\_CtrlFunct.ttcn  
  Replace the <user port type> tag with your user port type and the <user types module> tag with the module name in which the user port type is declared.
* IPL4asp\_User\_CtrlFunctDef.cc  
  Replace the <user port type> tag with your user port type and the <user types module> tag with the module name in which the user port type is declared. Remember to replace the underscores in the TTCN name with double underscore!

Note that depending on the module in which the control functions are declared, their use may be ambiguous without qualifying the module. E.g. if the default functions shipped with the port should be used in a user module in which also another user port type is defined, use the IPL4asp\_Types module name as follows:

IPL4asp\_Types.f\_IPL4\_listen

Alternatively, one may apply names here depending on their special naming conventions.

Note: In demo directory a script file can be found, called: generate\_control\_functs.sh. This scipt file can be used to automatically generate the files IPL4asp\_User\_CtrlFunct.ttcn and IPL4asp\_User\_CtrlFunctDef.cc and replaces the tags described above. The script should be put in the same directory with IPL4asp\_PortType.ttcn and IPL4asp\_PT.cc because it generates the files from these.

## Configuration

The executable test program behaviour is determined via the run-time configuration file. This is a simple text file, which contains various sections (e.g. [TESTPORT\_PARAMETERS]) after each other. The usual suffix of configuration files is .cfg. For further information on the configuration file see [2].

The IPL4asp test port supports parameters as specified in the following sections.

### General test port parameters

**debug**

Set to “YES” if you need to debug the test port, otherwise “NO”.

The default value is ”NO”.

**connId\_release\_mode**

Controls the connId release method. See 2.15

The default value is “normal”.

**defaultListeningPort**

This shall be used as the default listening port if the user does not specify a port number when opening a listening socket.

The default value is “9999”.

**defaultListeningHost**

This shall be used as the default listening host if the user does not specify a port number when opening a listening socket.

The default value is the IPv4 any address “0.0.0.0”.

**backlog**

This parameter limits the number of connections that can be opened on a listening stream-based socket.

The default value is the system parameter “SOMAXCONN”.

**sockListSizeInit**

This is the initial value of the sockList array. Whenever a new socket cannot be storered in the array, the size is doubled. It is recommended to set it to a value close to the number of connections in order to avoid too many reallocations.

The default value is “2”.

**pureNonBlocking**

The default value for this parameter is: “no”.

You can turn pure non-blocking mode on by setting this parameter to either “yes” or “YES”.

If this mode is on, then the test port will not block your TTCN-3 send statement until the socket can transmit your message. Instead it will return an ASP\_Event ASP containing a Result field with “IPL4\_ERROR\_TEMPORARILY\_UNAVAILABLE” errorCode. As soon as the socket becomes writable the test port sends a notification using the ASP\_Event ASP containing a Result field with “IPL4\_ERROR\_AVAILABLE” errorCode and it’s the user’s responsibility to send the message again.

**extendedPortEvents**

This parameter can be used to turn on extended port events. If extended port events are set to “yes” state, the result of connection open, connection close, listening or various errors will be send in result type port events too.

The default value for this parameter is: “no”.

Note: take care of turning on this parameter. Older Applibs are not prepared for handling these extended events.

**noDelay**

The default value for this parameter is: “no”.

You can turn nodelay mode on by setting this parameter to either “yes” or “YES”.

If this mode is on, then the test port will instruct the kernel socket to immediately send outgoing TCP or SCTP messages without waiting for more data.

If this mode is off, then the kernel will wait for additional messages before sending, in order to optimize the TCP (SCTP) packet sizes.

**lazy\_conn\_id\_handling**

The default value for this parameter is: “no”.

If this parameter is set to “yes” then the connId fields of the outgoing messages or function calls can be -1, which value translated to the real connection id inside the test port.

The connId -1 is accepted only if there is only one connection.

### Parameters for automatic connection during mapping

**map\_behavior**

Controls the behavior of the test port during mapping:

* “none”, The default value. No outgoing connection created, no listening port opened.
* “connect” Outgoing connection is established during map operation. The protocol is determined by the “map\_protocol” parameter. The local address is specified by “defaultListeningHost"” and “defaultListeningPort”. The remote address is specified by “RemoteHost” and “RemotePort”

**map\_protocol**

Controls the protocol used for the connection/listening port opened during map. Possible values:

* “tcp” The default value
* “tls”
* “sctp”
* “udp”

**RemotePort**

The remote port number to connect

**RemoteHost**

The remote host to connect

### Parameters for IP auto-configuration

**ipAddressDiscoveryType**

If set to “DHCP”, IP addresses will be requested from the DHCP server of the network.

If set to “ARP”, the test port itself finds unused IP addresses on network. For this ARP messages are used. In this case the IP address and the network mask of the Ethernet interface must be configured according to the network.

If set to “DHCP\_OR\_ARP”, then IP addresses are requested from the DHCP server. If it is unsuccessful, then ARP messages are used.

**interfaceName**

The name of the Ethernet interface to be used for IP address discovery  
For example: “eth1”

It can be omitted. In this case one from the Ethernet interfaces is selected.

**interfaceIpAddress**

The Ethernet interface to be used for IP address discovery can alternatively be specified with its IP address. This parameter can be omitted.

**excludedInterfaceIpAddress**

It specifies the IP address of the interface to exclude from the search for the Ethernet interface to be used for IP address discovery. It can be omitted.

**ethernetAddressStart**

The format is: “NN-NN-NN-NN-NN-NN” where “N” is a hexadecimal digit.

It is used when IP addresses are requested from the DHCP server.  
This parameter is the Ethernet address to be used for requesting the first IP address. For subsequent IP addresses the Ethernet address is incremented.

If this parameter is omitted an Ethernet address will be generated.

**leaseTime**

It is given is seconds.

IP addresses will be requested from the DHCP server for this duration.

**leaseFile**

This is the path of the lease file. Information about the IP address leases obtained from the DHCP is stored in this file.

This information makes it possible to reuse IP addresses previously requested from the DHCP server. This prevents the possible exhaustion of the IP address space in the server.

It is also used for releasing the requested IP addresses.

**numberOfIpAddressesToFind**

The number of IP addresses either to request from the DHCP server or to find with ARP messages.

**dhcpMsgRetransmitCount**

Maximum retransmit count of DHCP requests. Default is “5”.

**dhcpMsgRetransmitPeriodInms**

Retransmit period of DHCP requests in millisecond. Default is “3000”.

**dhcpMaxParallelRequestCount**

Maximum number of parallel DHCP requests. Default is “25”.

**dhcpTimeout**

DHCP timeout. Default is “infinite”.  
This timeout value is used when some responses (at least one) arrive from the server. If DHCP server is not reachable at all, then shorter timeout is used, which is calculated from the retransmission count and period.

**arpMsgRetransmitCount**

Maximum retransmit count of ARP requests. Default is “3”.

**arpMsgRetransmitPeriodInms**

Retransmit period of ARP requests in millisecond. Default is “1000”.

**arpMaxParallelRequestCount**

Maximum number of parallel ARP requests. Default is “50”.

### Parameters specifying the default connection options

The following parameters give the initial values of default connection options which will be applied when the options are not specified in listen or connect function calls. (Defaults can be changed with function calls on test port component level.)

System settings are not affected by these parameters.

If an option is not specified for a connection and has no test port component level default value, then it is not set. In this case behavior is determined by system wide settings.

Default values for the following parameters are selected so, that backward compatibility is maintained when the parameters are omitted.

**tcpReuseAddress**

It specifies whether SO\_REUSEADDR is set on sockets with TCP protocol. “YES” or “NO” can be given. Default is “YES”.

**sslReuseAddress**

It specifies whether SO\_REUSEADDR is set on sockets with SSL protocol. “YES” or “NO” can be given. Default is “YES”.

**udpReuseAddress**

It specifies whether SO\_REUSEADDR is set on sockets with UDP protocol. “YES” or “NO” can be given. Default is “YES” on Linux, “NO” on other platforms.

**sctpReuseAddress**

It specifies whether SO\_REUSEADDR is set on sockets with SCTP protocol. “YES” or “NO” can be given. Default is “YES” on Linux, “NO” on other platforms.

**tcpKeepAlive**

It enables or disables the keep alive mechanism on TCP. “YES” or “NO” can be given. There is no default.

**tcpKeepCount**

It specifies the count parameter of the keep alive mechanism. (Number of keep alive messages to be sent) There is no default.  
The parameter has effect only on Linux.

**tcpKeepIdle**

It specifies the idle time parameter of the keep alive mechanism. (Number of seconds to wait before sending the first keep alive message) There is no default.  
The parameter has effect only on Linux.

**tcpKeepInterval**

It specifies the interval parameter of the keep alive mechanism. (Time interval between keep alive messages in seconds) There is no default.  
The parameter has effect only on Linux.

**sslKeepAlive**

It enables or disables the keep alive mechanism on SSL over TCP. “YES” or “NO” can be given. There is no default.

**sslKeepCount**

It specifies the count parameter of the keep alive mechanism. (Number of keep alive messages to be sent) There is no default.  
The parameter has effect only on Linux.

**sslKeepIdle**

It specifies the idle time parameter of the keep alive mechanism. (Number of seconds to wait before sending the first keep alive message) There is no default.  
The parameter has effect only on Linux.

**sslKeepInterval**

It specifies the interval parameter of the keep alive mechanism. (Time interval between keep alive messages in seconds) There is no default.  
The parameter has effect only on Linux.

**freebind**

If enabled, this boolean option allows binding to an IP address that is nonlocal or does not (yet) exist.

This option is the per-socket equivalent of the ip\_nonlocal\_bind /proc interface

Please note this option has effect on ipv6 only in Linux kernel 3.3 or above. The option is not supported on SLED/SLES 11.

**dscp**

It is an option to set the DSCP field of the IP headers. There is no default.

**mtu\_discover**

This enumeration option sets the Path MTU behavior. The following values can be assigned to it:

* PMTUDISC\_DONT: Never does Path MTU Discovery.
* PMTUDISC\_WANT: Uses per-route settings.
* PMTUDISC\_DO: Always does Path MTU Discovery.
* MTU: Only for “get” mode! Returns the current Path MTU.

### Parameters specifying SCTP connection options

In the [TESTPORT\_PARAMETERS] section the following parameters can be set for the SCTP support. These parameters are applying to the test port globally.

**sinit\_num\_ostreams**

The parameter is optional, and can be used to determine the number of outbound streams the application wishes to be able to send to. Default is “64”. Allowed values: positive integers.

**sinit\_max\_instreams**

The parameter is optional, and can be used to determine the maximum number of inbound streams the application is prepared to support. Default is “64”. Allowed values: positive integers.

**sinit\_max\_attempts**

The parameter is optional, and can be used to specify how many attempts the SCTP endpoint should make at resending the INIT. Default is “0”. Allowed values: positive integers

**sinit\_max\_init\_timeo**

The parameter is optional, and can be used to determine the largest Time-Out or RTO value (in milliseconds) to use in attempting an INIT. The default value is “0”. Allowed values: positive integers.

Note: the default value of 0 indicates to use the endpoint’s default value. Alteration is not recommended unless you know what you are doing.

**sctp\_data\_io\_event**

The parameter is optional, and can be used to enable the occurrence of the events called: sctp\_data\_io\_event. “YES” or “NO” can be given. Default is “YES”.

**sctp\_association\_event**

The parameter is optional, and can be used to enable the occurrence of the events called: sctp\_association\_event. “YES” or “NO” can be given. Default is “YES”.

**sctp\_address\_event**

The parameter is optional, and can be used to enable the occurrence of the events called: sctp\_address\_event. “YES” or “NO” can be given. Default is “YES”.

**sctp\_send\_failure\_event**

The parameter is optional, and can be used to enable the occurrence of the events called: sctp\_send\_faliure\_event. “YES” or “NO” can be given. Default is “YES”.

**sctp\_peer\_error\_event**

The parameter is optional, and can be used to enable the occurrence of the events called: sctp\_peer\_error\_event. “YES” or “NO” can be given. Default is “YES”.

**sctp\_shutdown\_event**

The parameter is optional, and can be used to enable the occurrence of the events called: sctp\_shutdown\_event. “YES” or “NO” can be given. Default is “YES”.

**sctp\_partial\_delivery\_event**

The parameter is optional, and can be used to enable the occurrence of the events called: sctp\_partial\_delivery\_event. “YES” or “NO” can be given. Default is “YES”.

**sctp\_adaptation\_layer\_event**

The parameter is optional, and can be used to enable the occurrence of the events called: sctp\_adaptation\_layer\_event. In lksctp versions below 1.0.7. this event is called sctp\_adaption\_layer\_event (see 1.5). “YES” or “NO” can be given. Default is “YES”.

**sctp\_authentication\_event**

The parameter is optional, and can be used to enable the occurrence of the events called: sctp\_authentication\_event. This event is defined only in lksctp version 1.0.9 and above (see 1.5). In versions below 1.0.9 setting this parameter is unaffected. “YES” or “NO” can be given. Default is “YES”.

**sctp\_stack**

Selects the used SCTP stack. Possible values:  
“kernel”: the kernel based SCTP stack is used. (default)  
“EIN”: The EIN SS7 Stack SCTP API is used.

**sctp\_path\_mtu\_size**

The parameter is optional, and can be used to specify the PMTU (Path Maximum Transmission Unit) for all SCTP connections. Default is “0”, which means that the kernel routines will determine the PMTU. Allowed values: positive integers. This parameter is not used when the EIN SS7 SCTP implementation is used.

### EIN SS7 Stack parameters

**CPMANAGERIPA**

This parameter sets the IP Address:port of the EINSS7 stack CP Manager.

**USERID**

This identifies the user for the Common Parts. More information can be found on the use of this parameter in the documentation of CP [7].

Possible values of this parameter are: “USER01”, “USER02”,…, “USER20”, “USER21”,…, “USER30” and “40”, “41”,…, “59”,“190”,…,”199”

If several IPL4 test ports are used at the same time each must have different value for the USERID parameter.

**sctpInstanceId**

This parameter sets the instance (back end process) to which the port connects, when the EIN stack is used in Horizontal Distribution mode.

**userInstanceId**

This parameter sets the instance ID of the EINSS7 stack user.

### Parameters specifying SSL connection options

These parameters are only available if IPL4\_USE\_SSL macro is defined during compilation.

**ssl\_verify\_certificate**

The parameter is optional, and can be used to tell the test port whether to check the certificate of the other side. If it is defined “yes”, the test port will query and check the certificate. If the certificate is not valid (i.e. the public and private keys do not match), the connection fails and it will return with the corresponding error in the result message. If it is defined “no”, the test port will not check the validity of the certificate. The default value is “no”.

**ssl\_use\_session\_resumption**

The parameter is optional, and can be used to specify whether to use/support SSL session resumptions or not. The default value is “yes”.

**ssl\_certificate\_chain\_file**

It specifies a PEM encoded file’s path on the file system containing the certificate chain. For detailed information see [5]. Mandatory for server socket(s) and optional if only client socket(s) is used. Note that the server side may require client authentication. In this case no connection can be established without a client certificate.

**ssl\_private\_key\_file**

It specifies a PEM encoded file’s path on the file system containing the server’s RSA private key. For detailed information see [5]. Mandatory if server socket is used, optional if only client socket(s) is used.

**ssl\_private\_key\_password**

The parameter is optional, and can be used to specify the password protecting the private key file. If not defined, the SSL toolkit will ask for it.

**ssl\_trustedCAlist\_file**

It specifies a PEM encoded file’s path on the file system containing the certificates of the trusted CA authorities to use. Mandatory if server socket is used, and mandatory for client sockets if ssl\_verify\_certificate=”yes”.

**ssl\_allowed\_ciphers\_list**

The parameter is optional, and can be used to specify the allowed cipher list. The value is passed directly to the SSL toolkit.

**SSL\_reconnect\_attempts**

This parameter can be used to specify the maximum number of times the connection/server accepts is attempted to be established in SSL reconnect mode. The default value is “5”.

The parameter has no meaning if “pureNonBlocking” is set to “yes”, because in this case the event handler takes care to call the relevant “SSL\_connect” or “SSL\_accept” again when the event happens, and it’s up to the test port or TTCN code how to continue.

Tip: If the SSL connection/server side accepts result is SSL\_ERROR\_WANT\_READ/ SSL\_ERROR\_WANT\_WRITE for all attempts increase this parameter value.

**SSL\_reconnect\_delay**

This parameter can be used to specify the time (in nanoseconds) the test port waits between to SSL reconnection/server accept attempt. The default value is “10000” nanosecond (=0.01 second).

The parameter has no meaning if “pureNonBlocking” is set to “yes”, because in this case the event handler takes care to call the relevant “SSL\_connect” or “SSL\_accept” again when the event happens, and it’s up to the test port or TTCN code how to continue.

Note: Too high value (e.g. 1 second) can cause SSL connection fail.

**SSLv2**

**SSLv3**

**TLSv1**

**TLSv1.1**

**TLSv1.2**

These parameters can be used to disable/enable the support of the specific SSL/TLS version. The default value is “yes”, means enabled. To disable the SSL/TLS version, set the corresponding parameter “no”

**TLS\_CERT\_PER\_CONN**

If set to “YES”, the TLS/SSL certificates parameters can be specified per connection. Otherwise every connection use the same global parameters.

#### Parameters for setting PSK

**psk**\_**identity**

The “PSK identity” is included in the ClientKeyExchange message and transmitted to the server. After the negotiation for “PSK identity” is done, the client and the server can generate their pre-master secrets with the pre-shared key. The parameter is optional.

**psk**\_**key**

The parameter is optional, it is the pre-shared key in hexadecimal representation form.

**psk**\_**identity**\_**hint**

The server can provide a "PSK identity hint" in the ServerKeyExchange message. The parameter is optional.

## Useful functions

The IPL4asp\_Functions TTCN module contains some interface handling functions that may be useful in writing test cases. Each of these functions is based on the TCCInterface\_Functions described in [4].

**f\_setUpInterface**

f\_setUpInterface(

in charstring startIPAddress,

in charstring netmask,

in charstring broadcast,

in integer count,

in charstring ifname,

in integer virtualIfaceStart

)

This function sets up a range of IP addresses, each on a different virtual interface, which happens via IOCTL system calls (as in ifconfig). The starting IP address the netmask, the broadcast address and the name of the interface can be added. The number of IP addresses can be set via the parameter count.

**f\_setDownInterface**

f\_setDownInterface(

in charstring ifname,

in integer count,

in integer virtualIfaceStart

)

This functions tears down the interfaces - set by the parameter ifname - that are possibly set up with the f\_setUpInterface function.

**f\_splitIpAddress**

f\_splitIpAddress(

in charstring addr

) return ro\_integer

Splits a dot format IP address to its segments and returns the values in a record of integer. It supports both IPv4 and IPv6.

**f\_nextIpAddress**

f\_nextIpAddress(

inout ro\_integer addr

)

Based on the input address split with the f\_nextIpAddress function, it returns the next possible IP address, in the same split format. Note that this function neither checks the availability of the address nor skips the network and broadcast addresses.

## Functions for IP auto-configuration

The IPL4asp\_Functions TTCN module contains four functions to help setting up IP addresses automatically.

Parameters for IP address discovery are either taken from the run-time configuration file or as function parameters. The functions give back the found IP addresses and other information needed by the interface handling functions: f\_setIP or f\_setUpInterface.

Parameters related to timing of message sending can only be set in the run-time configuration file.

**f\_findIpAddressesWithDhcp**

f\_findIpAddressesWithDhcp (

inout IPL4asp\_PT portRef,

in charstring expIfName,

in charstring expIfIpAddress,

in charstring exclIfIpAddress,

in charstring ethernetAddress,

in integer leaseTime,

in charstring leaseFile,

in integer nOfAddresses,

out ro\_charstring ipAddresses,

out charstring netMask,

out charstring broadcastAddr,

out charstring ifName

) return boolean

This function requests IP addresses from the DHCP server according to the function parameters.  
The function reads the lease file; reuses the necessary amount of lP addresses; requests additional IP addresses or releases the superfluous ones as necessary; finally it writes the lease file.  
As a result, after successful execution of the function, exactly the specified number of IP addresses will be leased. If the result is successful a return value of true is set.

**f\_findIpAddressesWithARP**

f\_findIpAddressesWithARP (

inout IPL4asp\_PT portRef,

in charstring expIfName,

in charstring expIfIpAddress,

in charstring exclIfIpAddress,

in integer nOfAddresses,

out ro\_charstring ipAddresses,

out charstring netMask,

out charstring broadcastAddr,

out charstring ifName

) return boolean

This function finds IP addresses that can be used in the network with ARP messages. DHCP server is not needed, but the Ethernet interface has to have an IP address and network mask valid on the attached network. If the result is successful a return value of true is set.

**f\_findIpAddresses**

f\_findIpAddresses (

inout IPL4asp\_PT portRef,

out ro\_charstring ipAddresses,

out charstring netMask,

out charstring broadcastAddr,

out charstring ifName

) return boolean

IP addresses are discovered according to the parameters in the run-time configuration file by calling one of the above two functions.

**f\_releaseIpAddressesFromDhcp**

f\_releaseIpAddressesFromDhcp (

inout IPL4asp\_PT portRef

) return boolean

The function releases all the IP addresses requested from the DHCP server. For this purpose requests with 1 second lease time are sent.  
Additionally the lease file is written to contain no IP addresses.  
The function releases IP addresses only if IP addresses were requested with DHCP previously during the same execution.

In these functions the selection of the Ethernet interface is based on three parameters, of which one or none should be given. These are the interface name, the interface IP address or the IP address of the interface cannot be selected. Among the matching interfaces the one configured to be attached to the biggest network is selected.  
It is advised to specify the interface name. If there is only one Ethernet interface it is best to omit all of these parameters.

For DHCP requests each IP address has to have a unique Ethernet address. A continuous range of Ethernet addresses is used. The first Ethernet address may be specified. If this parameter is omitted an Ethernet address is generated.

The lease time specifies the duration of the validity of the IP addresses requested from the DHCP server. The DHCP server might give a different (shorter) lease time.

The lease file stores the necessary information to reuse IP address leases between subsequent calls or to release those.

## Error messages

The IPL4asp test port does not generate TTCN error at the point of either protocol or test port error. The immediate error generation behaviour is replaced with passing the error to the user who can make decisions how to continue the test execution.

The port control functions return the Result type that consists of an optional error code and an optional connection ID. If the result is suspected to be an error, remember to check the test port log (if possible with debug information) for details.  
The Result record has two additional optional fields: os\_error\_code and os\_error\_text. os\_error\_code, when filled, contains the value of errno; and os\_error\_text contains a textual description.

The result of port control functions is successful if the optional error code is not present, otherwise, some error occurred with one of the error codes specified in the following chapters:

**IPL4\_ERROR\_GENERAL**  
General error, the exact reason is either unknown or deemed unworthy to specify.

**IPL4\_ERROR\_INSUFFICIENT\_MEMORY**  
Some memory allocation function, e.g. malloc failed to reserve heap memory.

**IPL4\_ERROR\_INVALID\_INPUT\_PARAMETER**  
One or more of the input parameters were improper.

**IPL4\_ERROR\_UNSUPPORTED\_TRANSPORT**  
The given transport is not yet implemented.

**IPL4\_ERROR\_SOCKET**  
One of the socket handling functions failed.  
If the error occurs on an established TCP or SSL connection, it signs an unrecoverable problem. The corresponding socket will be closed automatically by the testport, and a connClosed event will be also dispatched.

**IPL4\_ERROR\_HOSTNAME**  
The IP address is given in the wrong format or the hostname cannot be resolved.

**IPL4\_ERROR\_INVALID\_CONNECTION**  
No existing connection belongs to the given connection ID. Before using it, the connection must be established, which shall result in the connection ID.

**IPL4\_ERROR\_TEMPORARILY\_UNAVAILABLE**Occurs only in pureNonBlocking mode. If the socket is not writeable thus the message was not sent the test port sends this notification to the user.  
This error code is also used in pureNonBlocking mode when connect cannot establish the connection immediately.  
Result event with error code IPL4\_ERROR\_AVAILABLE is used when the user can send network messages on the connection.

**IPL4\_ERROR\_AVAILABLE**  
Once a formerly “TEMPORARILY\_UNAVAILABLE” (see above) socket becomes writeable again, this notification is sent to the TTCN-3 user. It’s the user’s responsibility to retransmit that message again which was not sent previously because the socket wasn’t available.

## Events

The asynchronous events in the port generate the ASP\_Event type. If the event is suspected to be an error, remember to check the test port log (if possible with debug information) for details.

The event may be of the following types:

type union ASP\_Event {

ConnectionOpenedEvent connOpened,

ConnectionClosedEvent connClosed,

Result result

}

**Result**  
In case of an event, result means error or notification about availability, meaning that the error code is always present. See section 3.7 for the possible error codes and their meaning.

**ConnectionOpenedEvent**  
A connection is opened on a listening socket, e.g. a TCP connection is forked from the listener. The ASP contain the connection tuple in addition to the connection ID and the user data that the user can easily manage the connection mappings.  
Note that the userData is the copy that of the listening socket.

**ConnectionClosedEvent**  
A connection is closed by the remote peer. It contains the same data as the ASP of connection opened event. In this case the user may want to remove the corresponding mapping.

**SctpEvent**  
An SCTP specific event is arrived. SCTP specific event can be:

sctp\_data\_io\_event  
sctp\_association\_event  
sctp\_address\_event  
sctp\_send\_failure\_event  
sctp\_peer\_error\_event  
sctp\_shutdown\_event  
sctp\_partial\_delivery\_event  
sctp\_adaptation\_layer\_event  
sctp\_authentication\_event

sctp\_sender\_dry\_event

The arriving of SCTP events can be turned off with the options described in 3.4.4.

## Examples

The demo directory contains examples to use the default user port and to extend the port for SIP.

There is also an old-style port mapping example for TITAN releases before R7.

Example configuration file and a project file for the TITAN GUI are also provided as a starting point.

# Interface description

## ProtoTuples

For each protocol IPL4 testport handles different *Connection tuples* defined in a union called ProtoTuple. The following *Connection tuples* are in use:

UdpTuple {}

This tuple is used in sending and receiving UDP type messages.

TcpTuple {}

This tuple is used in sending and receiving TCP type messages.

SslTuple {}

This tuple is used in sending and receiving SSL type messages.

SctpTuple {

integer sinfo\_stream optional,

integer sinfo\_ppid optional,

SocketList remSocks optional,

AssociationId assocId optional

}

This tuple is used in sending and receiving SCTP type messages. The parameters defined in this tuple can be used to send SCTP specific information.

* sinfo\_stream specifies the stream number of the message
* sinfo\_ppid specifies the information that is passed by the upper layer in the peer application
* remSocks it is used to give multiple remote addresses in case of multihomed connections
* assocId specifies the association ID to identify one specific connection in case of one-to-many connections

DtlsTuple {

UdpTuple udp,

SctpTuple sctp

}

This tuple is used for sending and receiving DTLS type messages.

* udp specifies that underlying layer is UDP
* sctp specifies that underlying layer is SCTP

## Send and receive functions

The message based IPL4asp provider test port has the following declaration:

type port IPL4asp\_PT message {

out ASP\_SendTo

out ASP\_Send

in ASP\_RecvFrom

in ASP\_Event

} with {extension "provider"}

The provider port sends and receives octetstring in each ASP, which may be overridden with a port extension.

ASP\_Send or ASP\_SendTo can be used to send messages over the network depending on whether the socket is connected or not connected, respectively.

type record ASP\_Send {

ConnectionId connId,

ProtoTuple proto optional,

octetstring msg

}

type record ASP\_SendTo {

ConnectionId connId,

HostName remName,

PortNumber remPort,

ProtoTuple proto optional,

octetstring msg

}

* connId: Connection ID – The id returned by f\_IPL4\_listen or f\_IPL4\_connect
* proto: Should be given as omitted or be the transport protocol given in f\_IPL4\_listen or f\_IPL4\_connect
* remName: Remote host name or IP address
* remPort: Remote port number
* msg: Message to be sent in octetstring format

Messages received from the network are sent to the test port user in ASP\_RecvFrom messages.

type record ASP\_RecvFrom {

ConnectionId connId,

HostName remName,

PortNumber remPort,

HostName locName,

PortNumber locPort,

ProtoTuple proto,

UserData userData,

octetstring msg

}

* connId: Connection ID
* proto: Transport protocol identifier
* remName: Remote host name or IP address
* remPort: Remote port number
* locName: Local IP address (it is never converted to name)
* locPort: Local port number
* userData: User defined data (currently an integer – opaque for the test port)
* msg: Received message in octetstring format

Other asynchronous events, such as: connection opened, closed and errors (except error discovered by the synchronous interface) are received in ASP\_Event (see section 3.8). (Note: errors also include notifications about unavailability and availability of a connection for writing.)

## IPv6 Link local address handling

Every IPv6 address, except the unspecified address (::), has a "scope" which specifies in which part of the network it is valid.

In the unicast addressing class, link-local addresses and the loopback address have link-local scope, which means they are to be used in the directly attached network (link). All other addresses, including unique local addresses, have global (or universal) scope, which means they are globally routable, and can be used to connect to addresses with global scope anywhere, or addresses with link-local scope on the directly attached network.

The scope can be specified with prefixes which determine the interface. For example:

fe80::219:b9ff:fef2:fd09%eth3

## Test port control functions

Connection control operations are implemented as functions; and are used to open and close connections, modifying test port behaviour and connection properties.

These functions give back the result of the operation immediately (if there is any) as function return value. Note that in case of connect (in pure non-blocking mode) the operation possibly cannot be completed immediately and a short delay is needed before the first network message sending can be successful. The timing of the first send is supported with an asynchronous event.

Common arguments of the control functions are:

portRef: Test Port reference. This is needed to access some public members of the test port.  
For the description of other common parameters, see section 4.1.

### Creating listener

The f\_IPL4\_listen function can be used to create a listening socket. The connection ID is returned in the Result record.

In case of UDP, the returned connection ID may be used to send messages with ASP\_SendTo. Additionally this connection ID may be used to connect to a specific destination (see section 4.4.2).

external function f\_IPL4\_listen(

inout IPL4asp\_PT portRef,

in HostName locName,

in PortNumber locPort,

in ProtoTuple proto,

in OptionList options := {}

) return Result;

If locName is “”, the default local host name (by default the IPv4 any address) is used that may be changed via run-time configuration (see section 3.4).

If portNum is -1, the default port number (by default 9999) is used may be changed via run-time configuration (see section 3.4).

If options is specified, then the test port level defaults can be overridden. This parameter can be omitted for backward compatibility and simplicity.  
See section 4.4.3.

### Creating connection

The f\_IPL4\_connect function can be used to create a connection. The connection ID is returned in the Result record.

The function may be used also to connect an existing UDP socket created with the f\_IPL4\_listen function (see section 4.4.1). In any other case, the connId argument is ignored and should be -1.

external function f\_IPL4\_connect(

inout IPL4asp\_PT portRef,

in HostName remName,

in PortNumber remPort,

in HostName locName,

in PortNumber locPort,

in ConnectionId connId,

in ProtoTuple proto,

in OptionList options := {}

) return Result;

The default values of locName and locPort are the same as of section 4.4.1.

If portNum is 0 (zero), than the system choose a random available local port number.

Note that in pure no-blocking mode the function returns immediately, possibly without waiting for the connection being successfully established. If the result code is IPL4\_\_ERROR\_\_TEMPORARILY\_\_UNAVAILABLE then an asynchronous Result event carries the result of the operation and the indication that the connection can be used for sending network messages.

If options is specified, then the test port level defaults can be overridden. This parameter can be omitted for backward compatibility and simplicity.  
See section 4.4.3.

### SCTP Multihoming

The IPL4 test port supports local multihoming configuration with the EIN SS7 SCTP stack.

The additional local addresses can be configured via the “HostList” option. The test port supports only IPv4 or IPv6 literal addresses with the SS7 SCTP stack. Both IP literals and host names can be used with kernel SCTP stack.

### Setting connection options

A list of options can be specified when the connection is created in f\_IPL4\_listen or f\_IPL4\_connect and in the f\_IPL4\_setOpt function.

external function f\_IPL4\_setOpt(

inout IPL4asp\_PT portRef,

in OptionList options,

in ConnectionId connId := -1,

in ProtoTuple proto := { unspecified := {} }

) return Result;

The f\_IPL4\_setOpt function can be used to modify test port component level defaults. In this case connId should be omitted. If protocol is specified, then default options for that protocol are modified, otherwise defaults for all applicable protocols are modified.

The f\_IPL4\_setOpt function can also be used to modify options for an opened connection. In this case connId should be specified and proto be omitted.

The default values are selected so that backward compatibility is maintained when options are not set at all. See section 3.4.3.

The currently supported options are ReuseAddress, TcpKeepAlive, SslKeepAlive, sctpAdditionalLocalAddresses, sctpEINConfigGroup, solinger, ssl\_support, and no\_delay, udp\_encap, dscp, mtu\_discover.

ReuseAddress should be specified in connection creation. The optional enable field need only be specified if it is to turn off the option.  
SslKeepAlive and TcpKeepAlive has four fields. (For description see section 3.4.3.) Each can be given independently. For non-defined fields (including enable) defaults are used.

dtlsSrtpProfiles needs to be specified to extend the DTLS handshake with SRTP selection profile negotiation. For details see section 2.12.6.

For IPsec tunnel mode the UDP\_ENCAP option of the UDP socked should be called. The setsockopt should be called with IPPROTO\_UDP, UDP\_ENCAP and the provided value for the option (UDP\_ENCAP\_ESPINUDP\_NON\_IKE, UDP\_ENCAP\_ESPINUDP or UDP\_ENCAP\_L2TPINUDP).

The dscp option can be specified for an opened connection to set the DSCP field of the IP header.

### Getting connection options

It is also possible to read the value of a given socket option. You can specify an Option, and the current value will be returned in an Extended\_Result, at the msg field.

external function f\_IPL4\_getOpt(

inout IPL4asp\_PT portRef,

in Option option,

in ConnectionId connId := -1,

in ProtoTuple proto := { unspecified := {} }

) return Extended\_Result;

### The f\_IPL4\_getOpt function only supports mtu\_discover for now.

### Closing connection

Connections are closed with the f\_IPL4\_close function. Note that a connection may be disconnected by the remote peer, in which case a notification shall be received as described in section 3.8.

external function f\_IPL4\_close(

inout IPL4asp\_PT portRef,

in ConnectionId id,

in ProtoTuple proto := { unspecified := {} }

) return Result;

### Abnormal close

In order to force the abnormal closure of TCP or SCTP connection the SO\_LINGER option should be enabled and set to 0 by calling f\_IPL4\_setOpt function.

Example:

f\_IPL4setOpt(IPL4port,{{solinger:={l\_onoff:=1,l\_linger:=0}}},connID,{ sctp:={ omit,omit,omit,omit}})

f\_IPL4\_close(IPL4port, connID)

### Setting user data

Each connection may be associated with some user specified data that may help the user to handle connection mappings.

The user data (currently and integer) is opaque for the test port.

external function f\_IPL4\_setUserData(

inout IPL4asp\_PT portRef,

in ConnectionId id,

in UserData userData

) return Result;

### Getting user data

If the user associated data with a connection, then it can be retrieved with the following function:

external function f\_IPL4\_setUserData(

inout IPL4asp\_PT portRef,

in ConnectionId id,

in UserData userData

) return Result;

### Getting user details

If the user needs various connection details, then it can be retrieved with the following function:

external function f\_IPL4\_getConnectionDetails (

inout IPL4asp\_PT portRef,

in ConnectionId id,

in IPL4\_Param IPL4param,

out IPL4\_ParamResult IPL4paramResult

) return Result;

With this function user can get local address and local port, the remote address and remote port, the used protocol, user data or the parent connection ID. Supported connection details:

IPL4\_LOCALADDRESS,

IPL4\_REMOTEADDRESS,

IPL4\_PROTO,

IPL4\_USERDATA,

IPL4\_PARENTIDX

### Message dissection

In stream-based protocols (e.g. TCP, SSL), only the upper protocol may know how to find message boundaries in the stream of bytes. In order to perform this task in the test port independently from any session protocols, a callback function may be registered for each connection in the test port. This way the user will receive complete messages, even in case of stream-based protocols.

external function f\_IPL4\_setGetMsgLen(

inout IPL4asp\_PT portRef,

in ConnectionId id := -1,

inout f\_IPL4\_getMsgLen f,

in ro\_integer msgLenArgs

);

If is the reference of a callback function of the following type:

type function f\_IPL4\_getMsgLen(

in octetstring stream,

inout ro\_integer args

) return integer;

The callback function takes an octetstring as one of its arguments. It contains the bytes of the message received so far.  
The callback function has to return the length of the message if completely received. It has to return -1 if the length cannot be determined. If the message is incomplete, but the length can be determined, then the function should return the length. In this case the callback function will not be called again for the given message – possibly increasing the performance. Alternatively the function may always return -1 when the message is incomplete.

msgLenArgs is record of integer stored for each connection. It is not modified by the test port and opaque for that. Its purpose is to support efficient implementation of the callback function. (In SIP, for example, it could store the length value from the CONTENT-LENGTH header.)

The default operation is to return all bytes which were momentarily received.

If id in the function is omitted (or given as -1), the default function is changed, which will be used for new connections and newly opened listening sockets.

The forked sockets of a listening socket will by default inherit the message dissection function and the msgLenArgs record of the parent.

Note: If id in the function is omitted (or given as -1), function change will **not** take effect in already existing connections and listening sockets. New connections opened by an already existing listening socket will use the message dissection function inherited from its parent listening socket (so they will use the old message dissection function).

### Message dissection function for binary protocols

A predefined message dissection function is provided by the IPL4 test port for binary protocols with fixed placed and constant sized length fields, such as DIAMETER, ICR.

external function f\_IPL4\_fixedMsgLen(in octetstring stream, inout ro\_integer args) return integer;

The args should be a list of 5 integer values

**args[0]:** The offset of the length field from the beginning of the message in octets

**args[1]:** The size of the length field in octets

**args[2]:** The offset of the value of the length fields

**args[3]:** The multiplier of the length field value. The actual length is the multiplication of the multiplier and the length field value in octets

**args[4]:** The endianess of length field. 1-Little endian, 0-Big endian

### Send messages

Besides the ASP\_Send or ASP\_SendTo the f\_IPL4\_send and f\_IPL4\_sento can be used to send messages over the network depending on whether the socket is connected or not connected, respectively.

Using the functions instead of ASPs to send messages provides a reliable control of the send operation in a non-blocking mode.

The function returns the result of the send operation and the number of the sent octets. In case of the congestion the application is able to resend the unsent octets after the socket become writeable.

external function f\_IPL4\_send(

inout IPL4asp\_PT portRef,

in ASP\_Send asp,

out integer sent\_octets

) return Result;

external function f\_IPL4\_sendto(

inout IPL4asp\_PT portRef,

in ASP\_SendTo asp,

out integer sent\_octets

) return Result;

### Start TLS over existing TCP/UDP connection

The test port is able to initiate the TLS negotiation over the existing TCP/UDP connection.

The f\_IPL4\_StartTLS function can be used to initiate the TLS negotiation. The connId refers to the existing TCP/UDP connection on which the TLS negotiation should be started. The server\_side parameter controls whether the test port initiates or accepts the TLS negotiation.

external function f\_IPL4\_StartTLS(

inout IPL4asp\_PT portRef,

in ConnectionId connId,

in boolean server\_side:=false

) return Result;

### Getting Path MTU

You can read the Path MTU value of a connected socket using f\_IPL4\_getConnectedPathMTU. It returns an Extended\_Result.

external function f\_IPL4\_getConnectedPathMTU(

inout IPL4asp\_PT portRef,

in ConnectionId connId := -1,

in ProtoTuple proto := { unspecified := {} }

) return Extended\_Result;

The f\_IPL4\_getOpt function only supports mtu\_discover for now.

# Terminology

Connection tuple: Set of parameters that unambiguously identify a transport connection. It consists of the protocol (TCP, UDP, SCTP or SSL), the local IP address and port and the remote IP address and port.  
Note that even though UDP is not connection oriented, the term connection is used in order to hide the details of different transports at the test port API level as much as possible.

Connection ID: Test port component level unique identifier of the connection.

OpenSSL: The OpenSSL Project is a collaborative effort to develop a robust, commercial-grade, full-featured, and open source toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols as well as a full-strength general purpose cryptography library. For more information on the OpenSSL project see [5].

## Abbreviations

API Application Programming Interface

ARP Address Resolution Protocol

ASP Abstract Service Primitive

DHCP Dynamic Host Configuration Protocol

DNS Domain Name System

DSCP Differentiated Services Codepoint

DTLS Datagram Transport Layer Security

MTU Maximum Transmission Unit

PMTU Path MTU

SCTP Streaming Control Transport Protocol

SIP Session Initiation Protocol

SRTP Secure Real-time Transport Protocol

SSL Secure Socket Layer

TLS Transport Layer Security

TCP Transmission Control Protocol

TTCN-3 Testing and Test Control Notation version 3

UDP User Datagram Protocol

# References

1. ETSI ES 201 873-1 v4.3.1 (2011-06)   
   The Testing and Test Control Notation version 3. Part 1: Core Language
2. 1/198 17-CRL 113 200/6 Uen  
   User Guide for the TITAN TTCN-3 Test Executor
3. 2/198 17-CRL 113 200/6 Uen   
   Programmer’s Technical Reference for TITAN TTCN–3 Test Executor
4. 1551-CNL 113 472 Uen  
   TCC Useful Functions for TTCN-3 Toolset with TITAN  
   User Guide
5. OpenSSL toolkit   
   <http://www.openssl.org>
6. 155 19-CAA 901 548 Uen S11  
   SCTP IETF RFC 2960
7. 15517-CAA 201 29 Uen  
   Common Parts Functional Specification
8. 109 21-CNL 113 346-7 Uen  
   UDPasp Test Port for TTCN-3 Toolset with TITAN  
   Product Revision Information
9. 109 21-CNL 113 347-8 Uen  
   TCPasp Test Port for TTCN-3 Toolset with TITAN  
   Product Revision Information
10. 109 21-CNL 113 469-10 Uen  
    SCTPasp Test Port for TTCN-3 Toolset with TITAN  
    Product Revision Information
11. Datagram Transport Layer Security  
    <http://tools.ietf.org/html/rfc4347>
12. Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)  
    <http://tools.ietf.org/html/rfc5764>