ETSI TR 102 365 V1.2.1 (2005-06)

Technical Report

Broadband Radio Access Networks (BRAN);
HIPERLAN Type 2;
Application Programming Interface (API) definition for the UDP/IP based testing of HIPERLAN Type 2 protocol prototypes

Reference
RTR/BRAN-00200011R1

Keywords
access, API, broadband, HIPERLAN, IP, radio, testing

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° 7803/88

Contents

Intellectual Property Rights

IPRs essential or potentially essential to the present document may have been declared to ETSI. The information pertaining to these essential IPRs, if any, is publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (http://webapp.etsi.org/IPR/home.asp).

Pursuant to the ETSI IPR Policy, no investigation, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Foreword

This Technical Report (TR) has been produced by ETSI Project Broadband Radio Access Networks (BRAN).

Compared to the version 1.1.1 of TR 102 365, the present document introduces changes to annex D.

1 Scope

The present document presents the results of work to develop a generic solution for inexpensively testing any protocol and a specific implementation of this solution for the HIPERLAN2 DLC protocol. The generic solution provides an inexpensive means to test any protocol implementation. The implementation is software-based but can be hardware as well. The implementation in software on a PC-based platform is a "virtual" test system. The implementation in hardware with radio transport and frequency capabilities is classic radio-based test equipment.

2 References

For the purposes of this Technical Report (TR), the following references apply:

[1] ETSI/Hiperlan2 Global Forum document: "BRAN, Hiperlan2, Data Link Control (DLC) and Convergence Layers, Interoperability Testing Event - July 2002; Test Bed Description".

[2] ETSI TS 101 761-2: "Broadband Radio Access Networks (BRAN); HIPERLAN Type 2; Data Link Control (DLC) Layer; Part 2: Radio Link Control (RLC) sublayer".

[3] ETSI TS 101 823-2-3 (V1.3.1): "Broadband Radio Access Networks (BRAN); HIPERLAN Type 2; Conformance testing for the Data Link Control (DLC) layer; Part 2: Radio Link Control (RLC) sublayer; Sub-part 3: Abstract Test Suite (ATS) specification".

[4] ETSI ES 201 873-5: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI)".

[5] ETSI ES 201 873-6: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 6: TTCN-3 Control Interface (TCI)".

[6] ISO/IEC 9646: "Information technology - Open Systems Interconnection - Conformance testing methodology and framework".

[7] ETSI TR 102 327: "Broadband Radio Access Networks (BRAN); HIPERACCESS; Application Programming Interface (API) definition for the UDP/IP based testing of HIPERACCESS protocol prototypes".

3 Definitions and abbreviations

3.1 Definitions

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

Protocol Layer Tester (PLT): virtual test system for the testing of protocol layers

virtual tester: PC-based test system that replaces hardware components of a sophisticated test equipment with software components

3.2 Abbreviations

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

AP	Access Point
API	Application Programming Interface
ASP	Abstract Service Primitive
ATS	Abstract Test Suite
DLC	Data Link Control
IUT	Implementation Under Test

MAC	Medium Access Control
MT	Mobile Terminal
MTU	Maximum Transmission Unit for IPv4
PA	Platform Adaptor
PCO	Point of Control and Observation
PDU	Protocol Data Unit
PHY	PHYsical
PLT	Protocol Layer Tester
PUT	Protocol Under Test
RLC	Radio Link Control
SA	System Adaptor
SAP	Service Access Point
SAR	Segmentation And Reassembly
SDL	Specification and Description Language
SUT	System Under Test
TCP	Transmission Control Protocol
TE	Test Equipment
TR	Technical Report
TTCN-2	Tree and Tabular Combined Notation version 2
TTCN-3	Testing and Test Control Notation version 3
UDP/IP	User Datagram Protocol over the Internet Protocol

4 The concepts

4.1 The requirement

The Terms of Reference for the present document call for a virtual tester that will run existing test specifications. This virtual tester would consist of the following:

- a subset of the existing test suite;
- an adaptation layer that would map the protocol messages into UDP/IP packets; and
- an Application Programming Interface for UDP/IP based testing with services that the executable test suite could use to transport messages and other information to and from the system under test (SUT).

Such a virtual tester would allow the HiperLAN2 companies to test and debug DLC protocol stacks early in their development stage and would facilitate and speed up the development of a full-fledged radio-based test tool. Such a tool could be used at interoperability events as well to provide a cheap and fast means to conformance test prototypes. Such conformance testing would be useful to determine errors in implementations and identify possible reasons for interoperability failures.

4.2 Virtual tester/Protocol Layer Tester (PLT)

The ETSI Abstract Test Suites (ATS) are designed to test a device to see if it conforms to the base specification. Usually this base specification specifies the device's protocol layers and performance requirements. The test suite usually mirrors these in its organization and function. The layers may be according to the OSI model or per the protocol designers' concept.

The ATS can be executed only if there is test equipment to run it upon. Test equipment does not come "off the shelf" for today's high performance protocols such as those for broadband radio networks. Test equipment for such protocols requires much the same development effort as the implementation itself. Simply said, full-featured conformance test equipment development is very expensive. This leads to a chicken-and-egg problem. On one hand, prototypes and implementations need to be tested to ensure they are conformant and interoperate and give them the chance to win in the marketplace. On the other hand, test equipment with all the required features for conformance testing is too expensive during prototyping and development.

During prototyping and developing, much of the system's design and implementation is done in software. Only when development and debugging are complete should the design become reality in firmware and hardware. If protocol layer conformance testing could be conducted in parallel during design on protocol prototypes in software or implementations, then product development and testing would be cheaper and quicker.

Is there a way to inexpensively conformance test the protocols in development or finalized that normally require expensive test equipment? The work described in the present document shows that there are low-cost off-the-shelf technology-neutral components requiring a minimum of "glue" to make a "virtual tester".

A "virtual tester" is a PC-based test system that replaces the expensive hardware components of sophisticated test equipment with much cheaper software components.

The development of advanced protocols requires testing and the testing equipment to run these tests. Radio protocols complicate these tasks and increase development times and testing costs. For radio protocols, test equipment is usually not available in time during development to test the implementation's behaviour over the air interface. The expensive up-front cost of radio-based test equipment precludes their arrival in time for use during protocol development.

Therefore, some type of relatively inexpensive means to test protocol implementation behaviour during prototyping and development could be of benefit to manufacturers and testers. This testing would, of necessity, not be conducted over the air interface because of the expense of developing such equipment.

Proven wire interfaces are cheaper and more reliable than new air interfaces. Thus, one reason for a virtual tester is to test protocols destined for an expensive interface in their prototyping and/or development stage. The tester would use a substitute wire interface for the lower transport layers. Another reason for a virtual tester is to conformance test any protocol for an expensive or inexpensive interface during design and development. Finally, a virtual tester could be used at interoperability or similar events to conformance test implementations and prototypes.

The Abstract Test Suite used for protocol testing would remain the same whether for a virtual tester or classical test equipment. Thus, no additional costs would be incurred for writing Abstract Test Suites to run over either test equipment.

The present document is concerned only with protocol messages. However, the use of wire transport layers for testing data normally transmitted using radio can apply to other types of data such as frames. The transmission of data in frames is not similar to protocol behaviour, e.g. a MAC protocol. However, the frame data can still be captured and transmitted over any type of wire protocol such as UDP over IP. The present document does not investigate frame testing or any other type of testing other than protocol conformance testing. Subsequent BRAN Technical Reports on UDP/IP testing substituting for radio testing may cover these non-classic protocol types of testing.

Answering the question of "What is being tested?" is important. The present document addresses the testing of MAC/radio link layer type protocols including their behavior and effects upon radio transmission characteristics. The radio link layer protocol can force changes in transmission frequency, channel, and power. Otherwise said, the radio link layer sometimes changes the performance of the physical layer. These effects are included in the Abstract Test Suite. Thus, device behaviour such as signal strength is tested as well as protocol behaviour if such behaviour is directly linked to the protocol function.

In our view, such behaviour is not PHY layer specific but linked intimately with the protocol and included in the radio link layer ATS. One could argue that such tests are PHY layer tests. Our view is that such PHY behaviour, being the result of radio link layer protocol actions, is rightfully included in the link layer ATS. Only that PHY level behaviour that is not a direct result of radio link protocol layer behaviour should be included in a PHY layer ATS, if such exists.

Because the work in the present document specifically address classic protocol layer testing, the virtual tester becomes a "Protocol Layer Tester" (PLT).

The protocol used for the feasibility study is the HiperLAN2 DLC protocol.

4.2.1 A generic, technology-neutral, and inexpensive solution

To be generic, the PLT concept must not be tied to technology that is either hardware or software-expensive. A generic solution should have the following characteristics:

- Apply to any protocol or transfer scheme where environment characteristics can be modelled with binary data; e.g. PDUs, frames, waveforms, transmission frequency, transmission power, received power, etc.

- Abstract Test Suites written in the ETSI-used testing languages TTCN-2 and TTCN-3. However, test suites in TCL, Java, C and its offspring, Perl scripts, etc can be easily incorporated.
- Test execution environments and systems that are either open-source, low-cost, or available from several vendors. Forcing a test execution environment to come from a specific vendor increases the probability of high costs.
- Common and low-cost wire interfaces such as UDP/IP/Ethernet, TCP/IP/Ethernet, etc.
- No/low-cost programming tools for making the PLT's software components and "gluing" them together with APIs.
- Testing of protocols regardless if based on ISO/IEC 9646 [6], another standard, or a proprietary scheme.
- Protocols that conform or not to the OSI layer model.

Figure 1 shows the relationship between the PLT and Protocol Under Test (PUT) that satisfies these characteristics.

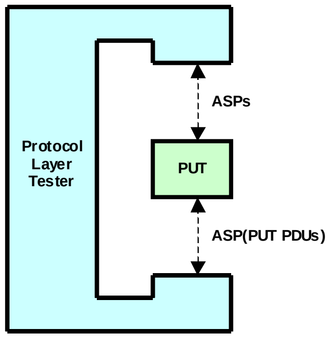


Figure 1: PLT and PUT

In general, the Protocol Layer Tester (PLT) exchanges Abstract Services Primitives (ASP) or Protocol Data Units (PDUs) on the upper and lower end of the Protocol Under Test (PUT).

Figure 2 shows the basic components of both the PLT and PUT using the same relationship shown above.

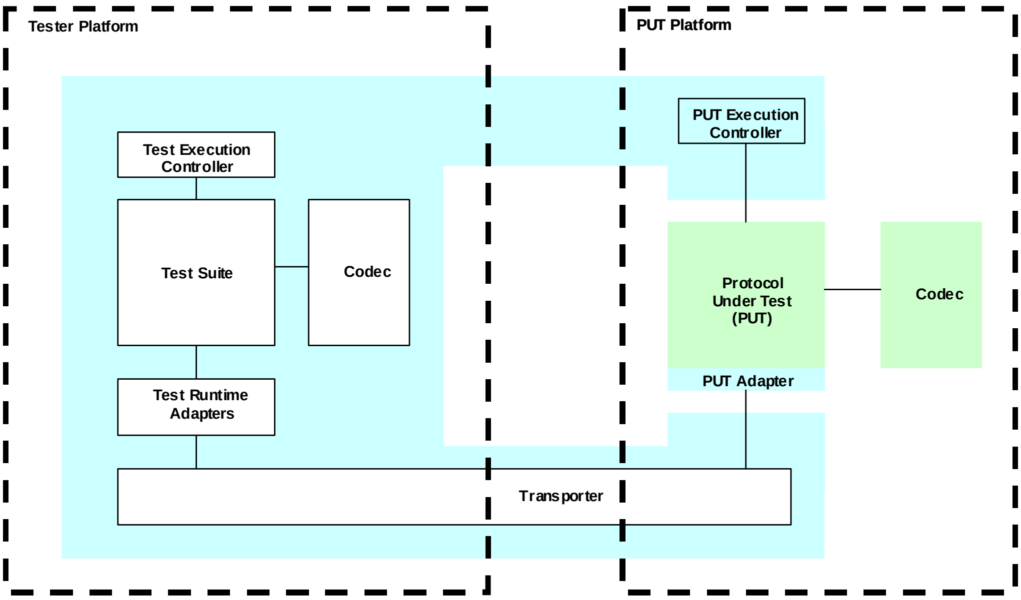


Figure 2: PLT and PUT Components

To be inexpensive, these components cannot be tied to any particular technology with expensive purchase, system, or license costs. The following no/low cost components are part of the PLT:

- Tester Platform: A PC serves as the test equipment's hardware platform with a hard-wired connection, rather than a radio link, from the PC to the PLT.
- Test Execution Controller, Codec, and Test Runtime Adapters: These are components in an off-the-shelf test execution tool.
- Test Suite: (An ATS written for conformance testing of the protocol.) At ETSI where both base specifications and test specifications are usually written for a given product. For the PLT, the ATS used for final product testing is the same as that used for protocol layer prototype testing. Thus, there are no additional test writing costs. The same test execution tool that has the Test Execution Controller, Codec, and Test Runtime Adapter converts the ATS into an executable program (ETS) on the PC hardware platform.
- PUT Platform: This is an implementor decision. It is usually a PC.
- PUT Execution Controller: The controller tells the PUT how to react to certain conditions. This is an implementor decision. This can either be a software module in the form of a script or an operator passing commands usually as primitives to the PUT.
- PUT and Codec: These are what the implementor must develop in any case. The use of a PLT should not increase his development costs for both. The protocol layer implementation is typically assumed to be in software.
- PUT adapter: A low-cost adapter to receive and transmit the PDUs and other required data. This is implementor effort required specifically to run tests against the PLT.
- Transporter: As figure 2 shows, the transporter "glues" the Test Platform to the PUT platform. The development of the virtual tester/PLT primarily centered around this transporter.

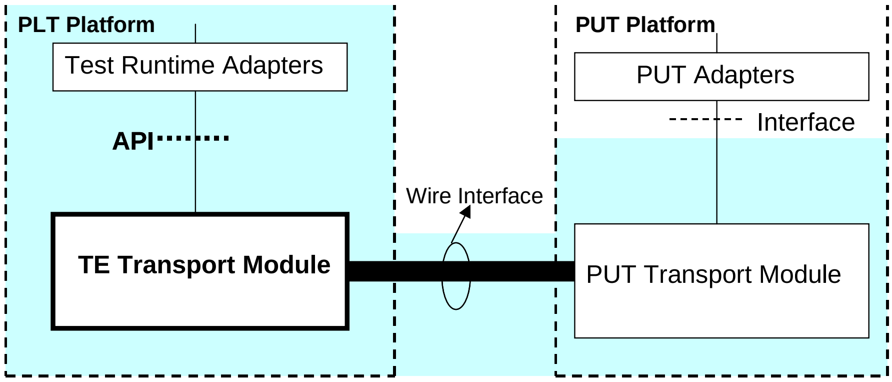


Figure 3: One Concept for the Transporter

Figure 3 is a conceptual diagram of one way to implement the transporter and is that used for work presented in the present document. A following is a brief description of the components.

- Test Runtime Adapters: A component of the test execution tool. They adapt the software elements of the Test Equipment/Abstract Test Suite to the platform and other hardware. For example, the interfaces to platform timers or signal strength measuring apparatus are part of the adapter. The Test Runtime Adapter is provided with the test execution tool and is not specified herein.
- Application Programmer Interface (API): The API that is the subject of the present document is the interface between the Test Runtime Adapter and the TE Transport Module. TR 102 327 [7] specifies a generic API and then instantiates the generic API for the HiperLAN2 Protocol.
- TE Transport Module: This is a software module that basically shuttles data back and forth between the wire interface and the API. It is a software module that uses the calls to the platforms wire input and output ports. Its specification is also included in TR 102 327 [7].

- Wire Interface: This is physical interface that connects the PLT and PUT. It replaces the radio Physical (PHY) layer and is the reason why the PLT is so inexpensive compared to classical radio protocol (and other) test equipment. It can be any wire PHY interface, but the most common by far on PC platforms is Ethernet. The protocols over the Ethernet PHY layer are IP and over them are IP-based protocols such as TCP or UDP. The simplest in use is the latter and was used in this work. TCP would be appropriate if there are large packets for reassembly or if transmission order can vary due to IP routing. Such was not the case in our work.

- PUT Transport Module: This module is on the PUT side and has the same functions as those of the TE Transport Module. Because the PUT Adapter is not identical to the Test Runtime Adapter, and is proprietary, the PUT Transport Module is not identical to the TE's. However the concepts are identical. The difference lies in the function names used by the PUT's adapter. Wire interface function calls may or may not have the same names as the TE's depending on the operating systems used and version numbers. This is not part of PLT per se and, thus, is not technically in the scope of the present document. However, since to test the PLT, the team had to make a PUT, this is included in the work. (There is currently no vendor hardware or software implementation/prototype to test against.)

- Interface (PUT-Side): This interface is defined by the PUT Adapter and, as such, is proprietary. However, it may very well be that the PUT developer may wish to use the very same API and its description for her interface. As will be seen later, the interface contains all the information necessary for testing the protocol using ETSI's ATSs. Thus, there will be many elements that the Interface must use already specified in the PLT-side API. Of course, the PLT-side API is in the public domain and encouraged for use by all.

- PUT Adapters: During development, the PUT designer places an API with primitives under the Protocol in order to drive the lower layers. She may as well have placed an adaptation layer for design reasons. This is proprietary and not a part of the transporter. It may or may not exist. In addition, the designer may decide to combine the PUT Adapters with the Transport Module to make one entity. For our work to make the test PUT, PUT adapters, interface, and Transport Module were one entity.

However, there is a need for an API between the testing executables and the transport module to execute methods and pass data back and forth. For inexpensive development, this API should be simple in concept and practice for both methods and data.

4.2.1.1 The data on the wire interface

For classic protocol test equipment that tests protocols on layer 2 or above, the PHY layer is built into the test equipment. Thus, for radio protocols, a radio PHY layer is part of the conformance test equipment. To do so requires the equivalent of designing and building a radio-based device similar to the IUT for mounting on a test platform. The TE then transmits and receives messages/PDUs/packets/frames or whatever over this PHY interface using the test suite as the criteria for which messages must be sent and should be received.

This is prohibitively expensive unless there is an assured market that will defray the high costs. This is not always the case. The PLT replaces the expensive PHY layer with an inexpensive one.

Layer 2 protocols often control Layer 1 behavior. For example, the BRAN DLC protocols have measures for dynamic frequency selection, transmission power adjustment, frequency shifting, antenna characteristics modification, etc. The ETSI ATSs include observing if this PHY layer behavior conforms with the instructions given by the Layer 2 DLC protocol. For example, say that a DLC protocol function changes the transmitting power to compensate for rain fading with a two-way handshake. Say that the TE is the access point and the IUT is the mobile terminal. The test case would have TE sending the command to increase the power. It would then wait to receive the mobile terminal's acknowledgement plus it would measure the received power both before and after the command to see if the behavior was correct.

Since the PLT is wire-based, it does not measure received power. However, the test case requires the measured power to assign a verdict. Something has to be done. In this case, the measured power is sent on the wire in a data field.

In essence, the data on the wire interface is a snapshot of the environment affected by the protocol's behavior plus the protocol messages sent and received between the PLT and PUT.

What does the PLT do? It transmits and receives "message snapshots".

What is a "message snapshot"? Just as a snapshot of a person is the person and the environment around him at a point in time, a message snapshot is the protocol message (a PDU for example) plus the environment/context at the point in time when the PDU is received or transmitted. The environment includes things like rx/tx signal strength, rx/tx signal frequency, lower layer information such as MAC ID, terminal ID, PHY modes, frame number, timer information, connection IDs, grant information, frame headers, etc. The list could be endless.

The environment/context is the only information needed by the implementation, prototype, or tester to determine the behavior associated with the protocol message in the snapshot. It is the data needed for the TE to test the PUT. For the PUT, it is the data it needs to exhibit the expected behavior. To create an environment/context and then to transform that environment into the needed data is a major reason for the expense of full-featured TE.

For the PLT, on receiving a message snapshot, it determines if the message with its context is expected. If so the test continues or a final Pass verdict is assigned. If not, an Inconclusive or a Fail verdict is assigned. In sending a message snapshot, the PLT forwards a protocol message and that message's context/environment to the Protocol Under Test (PUT) and then determines if the PUT's response conforms to the expected behavior.

The PUT takes the snapshot's message, determines the context from the snapshot, and reacts hopefully in accordance with the base specification. All this is carried on the wire.

The snapshot/context also includes test architecture/configuration information. Simple tester-to-device and concurrent tester-to-devices testing is possible with the PLT and the wire data. Shown below are just two of many testing configurations possible with a PLT.

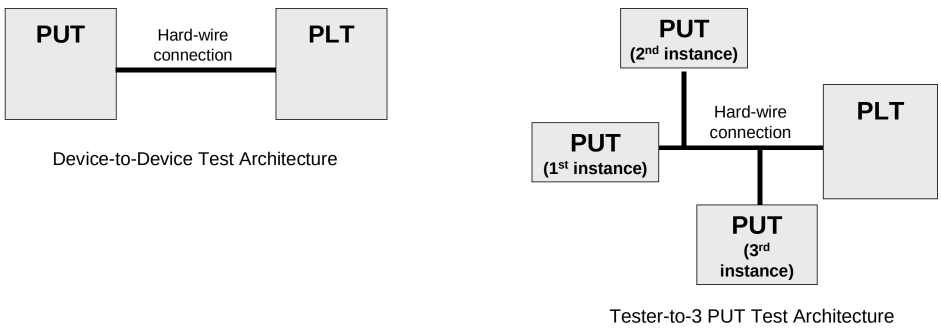


Figure 4: Two test configurations with a PLT

In the left hand, the configuration is straightforward and the wire need not carry configuration information. The right hand case is different. The tester will be sending messages to all three PUT instantiations. The destination of each message has to be indicated in some manner on the wire interface. Similarly, each PUT's response must be tagged with the originating PUT in some way as well.

4.2.2 Advantages and spin-offs

The advantages of the PLT solution with its API and wire interface are:

- Testing of protocol implementations without using expensive test equipment.
- Validation of the ATS's correctness without needing expensive test equipment. This validation also occurs sooner because the waiting for the manufacturing expensive testing equipment is eliminated.
- Protocol development/debugging and the writing/running of tests can occur in parallel thereby reducing the time-to-market and reduced costs because prototype and test debugging occur during the development process.
- Although designed for the test equipment side in conformance testing, the protocol implementor can take the components developed for the PLT and "plug" them, with some modification, into the PUT side in order to run the conformance tests.

A PLT has direct application to protocol development as well.

- Implementors developing the protocol for which a PLT's message snapshot has been already designed can use the snapshot to determine what environmental variables are necessary.
- The PLT's API can possibly be used as part of the PUT's interface to lower protocol layers.

- If the protocol developers want to use the PLT to test their implementation, they can use the hard-wire transport module in the PLT to provide the glue between their implementation and the hard-wire connection (with some modification if the operating system and programming languages are different).

This work also provides spin-offs that, in the long run, could be more beneficial than the above advantages.

- The API is multipurpose. Its intended purpose is for conformance (device to test equipment) testing, but it can also be used in device to device testing at interoperability events. Other possible purposes are providing an interface between an implementation and a simulator or between a simulator and test equipment.

- The results provide an inexpensive and quick way to validate the correctness of the base specification's correctness. The combined use of a protocol prototype "moving" with a base specification's development with tests "moving" along with the base specification's development yields quick results back to the base specification writers.

- The concept can be applied to any entity for which test cases are written. The concept is not limited to protocol layers. One possible application is the testing of abstract representations in software of physical layer characteristics. For example, this concept can be used to test the effects of a specific waveform anomaly that may not be producible by electronics. (In this case, the test equipment side costs are minimal but it may be too expensive to convert a hardware radio frequency interface into a software interface.) "Frame" testing is also another possibility.

4.3 PLT components

The following text describes in detail the generic PLT components requiring development: the Wire Interface Data, the Application Programmer Interface (API), and the TE Transport Module. Clause 5 presents these components as developed for the HiperLAN2 DLC protocol. The Test Runtime Adapters and the Wire Interface components are off-the-shelf and were not developed. The following components were developed for having a PUT to test against: PUT Transport Module, the Interface (PUT-Side), and the PUT Adapters. The effort in their development is not to be counted with that for the PLT.

4.3.1 Wire interface data/wire datagrams

Wire is used to transmit single or multiple PDUs or frames to and from a protocol implementation without the need of concrete lower layer implementations. It also carries the additional information needed by the PLT and PUT entities. The transport service provided by lower layer protocols is provided now by the transporter.

The concepts follow a message-oriented communication paradigm rather than stream-oriented or operation-oriented paradigms.

The wire interface data replaces the protocol's transport and physical layers.

Proper operation of the protocol layer within the PUT typically depends on additional information that is not included in the PDUs. This information is frequently related to the lower layer protocols. Examples for a layer 3 protocol might include a MAC id. In addition, the API concept must identify PCOs (Point of Control and Observation) and/or SAPs (Service Access Points) if required by the receiving side. Other information optionally included in the API might be text strings for operator instructions.

Typically an PUT is embedded in an environment that offers different types of information including (but not limited to) physical layer and other parameters. Real world protocol implementations might use on this information as is or perform operations upon it. The wire interface data is this information between the PLT and PUT or between PUTs in the case of interoperability testing.

EXAMPLE: A test for an (imaginary) protocol might be for a change between two different physical modes. The tester indicates to the PUT that it should change the physical mode; the PUT acknowledges the change using the old physical mode; PUT starts using the new physical mode and indicates that hand-over has taken place using the new physical mode.

Just from this short example it can be seen that both entities, the PUT and the PLT, need write and read access to environmental information like the actual physical mode used. The PUT in this example would instruct the underlying layer that a physical mode change should occur (write access). The PLT would have some kind of external operation that accesses the underlying layer and queries the actual physical mode used.

If a real system would be tested instead of a PUT, the tested protocol would communicate with its lower layer. The real test device would implement the external operations by accessing the lower layers and reading this information out. However in a PLT/PUT scenario, this necessary information must be communicated between PUT and PLT. This is the wire interface data.

The transmitted data must have a structure so that modules can access, manipulate, and store values. This structure includes both the messages and the snapshot/context/environment. We call encoded structure with the message and message's environment the Wire Datagram. The Wire Datagram provides the framework to transmit this information between the PLT and PUT.

The Wire Datagram

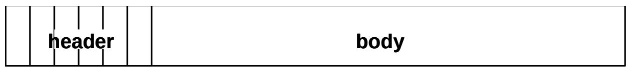


Figure 5: Structure of a Wire Datagram

The Wire Datagram consists of two parts, its header and the body or payload (figure 5). The header contains the snapshot/context (PUT-related lower layer information, PCO/SAP information and possibly additional information) while the body/payload carries the encoded PDUs, frames, etc. The PDU is encoded according to the protocol specification with either standardized methods (e.g. ASN.1 PER) or custom-made transfer syntax.

Write access of a PUT/PLT environment/context would update the header of the Wire Datagram. Read access by a PLT/PUT would use the results to make a decision.

For the purpose of this work we assume that the Wire Datagram is transmitted and received via buffers of byte arrays. However, its type specification is independent of the means used to store, transmit, and receive the data. We have defined the following buffer implementation for the Wire Datagram.

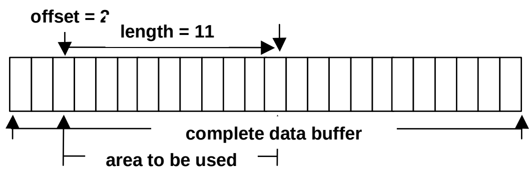


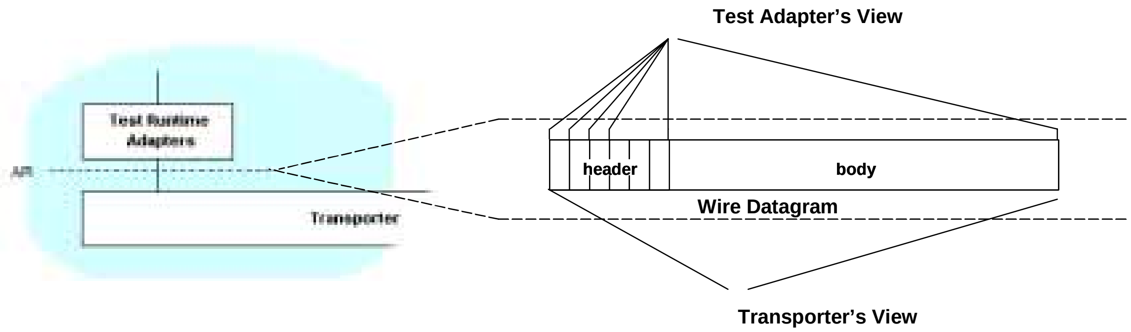
Figure 6: Possible internal structure of a DatagramPacket

According to this specification only the data area as described by the parameters offset and length are used. However, without specifying any particular parameters for offset and length the complete data buffer will be used to store information.

The API focuses on the communication of Datagrams. Within the API, Wire Datagrams are further specialized in order to provide access to the header information independently from their encoding. The encoding of the header information depends on the field of application and can vary from standardized encoding rules to self-defined encoding rules. As the framework is focusing on an early stage of the test suite and protocol specification phase, it may be necessary to change the encoding over time.

Wire Datagram requirements are summarized as follows:

- The transporter relies on a given functionality independent of the specific contents.
- The PLT/PLU implementation requires an abstract access to the contents of the Wire Datagram independent of the encoding and independent of the transporter implementation.



NOTE: It is understood that the transporter accesses also other data from a DatagramPacket, in particular socket addresses and port number. However this visualization is omitted for readability reasons.

Figure 7: Different views via specialized interfaces

Figure 7 shows how different actors use a Wire Datagram. While the Test Runtime Adapter accesses (read/write) individual elements of the datagram, the transporter perceives the datagram only as payload for the underlying transport mechanism, e.g. UDP. The upper view is always specific to a particular protocol or technology. Clause 4.3.1 describes the Wire Datagram for the HiperLAN2 DLC protocol.

Use of the Header

The header fields represent the environment that the PUT and the Test Equipment need for operation.

For example, the base specification requires specific behaviour when the received power level does not meet certain conditions. In actual deployment, rain fading can cause decreased received power requiring the protocol to adjust power levels. To ensure that an IUT conforms to the standard in such events, testing an SUT or a PUT requires in some way those reduced power levels that are part of its environment.

It is difficult to make rain in a test environment. In testing an SUT, the Test Equipment may have some built-in function to reduce transmission power to provoke the expected behaviour. In testing an IUT, there may be a different way to provoke the behaviour. Also, the test writer does not usually know how the tester will use the Test Equipment to run the test. For example, to reduce power levels one tester may reduce transmitting power from the Test Equipment; another may set up a grid connected to ground between the Test Equipment and the SUT; and another may simply take the SUT and walk away far enough from the Test Equipment. To cover these possibilities, the test writer usually creates a "stub" or a hook to the environment that controls the transmission power. The Test Equipment manufacturer and the tester are left to their own devices on how they want to control the power. The interface between their method and the test suite is this "stub" or hook. In TTCN-2, this interface is a TSO (Test Suite Operation).

To invoke the protocol actions for adjusting power levels, the PUT's controller must detect the change and then command the PUT to start the transaction to adjust the power level. This is where the header field comes in. The header field "rxPower" carries the information that the PUT's controller needs to determine if the power levels need adjustment or not.

One can see from this example the heuristic why TSOs are very good indicators of header fields.

Clause 4.3 discusses the HiperLAN2 header in detail.

4.3.2 The API

The well-known socket concepts have been adopted for defining the generic interface structure shown in figure 8.

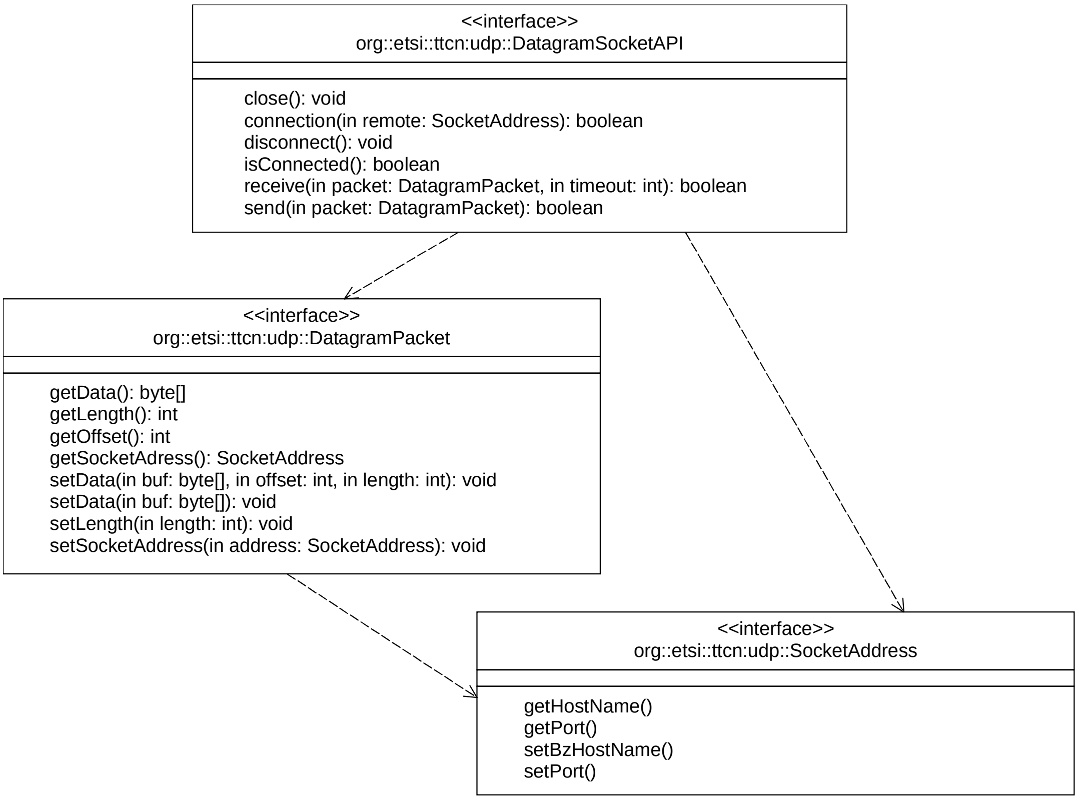


Figure 8: Abstract API description

A `DatagramSocketAPI` uses `DatagramPackets` in order to communicate with a remote peer entity as described via the `SocketAddress`. On this level of abstraction the API defines the means of implementing communication between the PLT and the PUT. In addition does this level of abstraction allow the reuse of the implementation on both sides.

All operations at the interfaces have been defined using UML notation. This clause specifies only interfaces, not any concrete implementations. The operations definitions are defined using the following template.

Signature	`Signature`
In Parameters	Description of data passed as parameters to the operation from the calling entity to the called entity
Return Value	Description of data returned from the operation to the calling entity
Effect	Behaviour required of the called entity before the operation may return.

The `DatagramSocketAPI` defines an interface on how a Test Runtime Adapter can communicate with the Transport Module. Basically, it abstracts from any protocol test suite the relevant additional information in the message header. The `DatagramSocketAPI` together with the `DatagramPacket` interface and the `SocketAddress` interface focuses completely on the communication between PLT and PUT.

Figure 3 highlights the location of the API. The Test Runtime Adapter uses implementation of the `DatagramPacketAPI` from the Transport Module and therefore does not have not to deal with transporting packets to the PUT.

The SocketAddress

A `SocketAddress` defines the host and the port to be used. Within the `DatagramSocketAPI` the `SocketAddress` will be used for described the local as well as the remote addresses.

4.3.3 Wire transport module/Adaptation layer

The Transport Module takes the context/environment information and PDU/message provided in the API, places it into the buffer discussed, and transmits it to the PUT over the wire interface in the Wire Datagram. Simply said, it forms the Wire Datagram given the information provided by the API.

In the other direction, the Transport Module receives the Wire Datagram as transmitted by the PUT, places it into a buffer, determines the API data, and transmits the data via the API to the Test Runtime Adapters.

It is a relatively straightforward module written specifically for the operating system and version. It can be in any programming language.

5 Implementing the PLT for the HiperLAN2 DLC protocol

The idea of using wire for conformance testing the HiperLAN2 DLC came from successful techniques used in the HiperLan interoperability events organized by the ETSI Plugtests™ Service. The interoperability tests involved two IUTs connected via the LAN transmitting the RLC data in UDP datagrams carried over IP. The idea was to replace one of the IUTs with simple conformance test equipment for testing only the protocol layer. This simple equipment would include a PC platform, the ATS already developed for the protocol and the other adaptations required to send and receive the UDP datagrams via the TE.

One of ETSI's test specification goals is to validate a test suite before its publication. A test suite can only be validated if an application (the IUT) is provided by a manufacturer. Better validation is achieved with several IUTs from different manufacturers.

The HiperLAN2 ATS produced in prior work is source of the executable test suite part of the PLT. It is based on the test architectures shown in the following clause.

5.1 Test architecture for the DLC layer

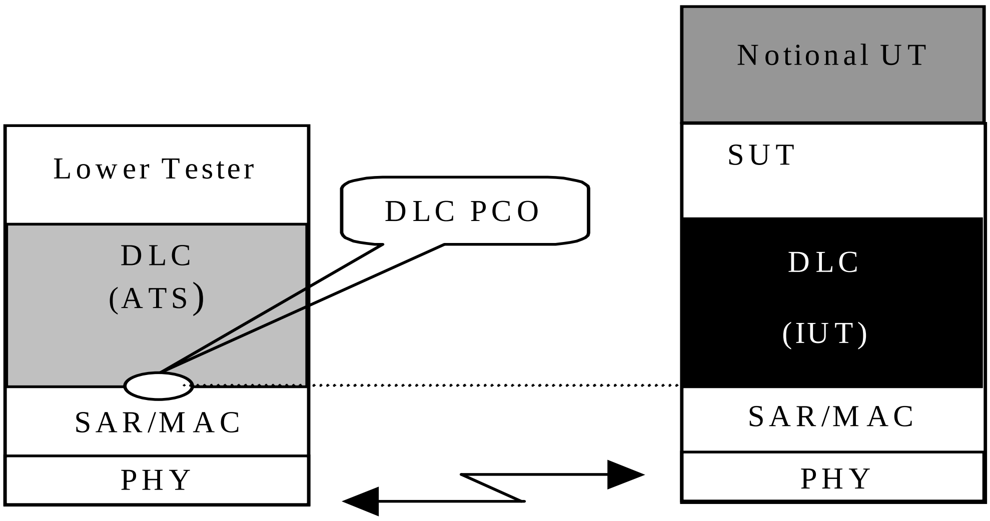


Figure 9: Test architecture for DLC

A single-party testing concept is used that consists of the following abstract testing functions:

Lower Tester: A Lower Tester (LT) is located in the remote BRAN HL2 test system. It controls and observes the behaviour of the IUT.

DLC ATS: A DLC Abstract Test Suite (ATS) is located in the remote BRAN HL2 test system.

DLC PCO: the Point of Control and Observation (PCO) for DLC testing is located at a SAP between the DLC layer and the MAC layer. All test events at the PCO are specified in terms of Abstract testing Service Primitives (ATSP defined in clause 7) containing complete PDUs. To avoid the complexity of data fragmentation and recombination testing, the SAP is defined below these functions.

Notional UT: No explicit upper tester (UT) exists in the system under test. Nevertheless, some specific actions to cover implicit send events and to obtain feedback information are necessary for complete testing. A black box covering these requirements is used in the SUT as a notional UT as defined in ISO/IEC 9646 [6]. This notional UT is part of the test system.

The PLT is situated at the right hand side in the shaded DLC block. The PUT is on the left hand side in the black DLC (IUT) block. The lower SAR/MAC and PHY layers have been replaced by the API and wire data transport module. The radio interface is, of course, replaced by the UDP/IP wire interface. The lower test is part of the test execution system. The Notional UT and SUT are prototype-dependent. They are usually the test engineer running the prototype on a PC.

5.1.1 Test Configurations

5.1.1.1 Test Configurations for MT

Two configurations are defined for MT testing and used in the ATS.

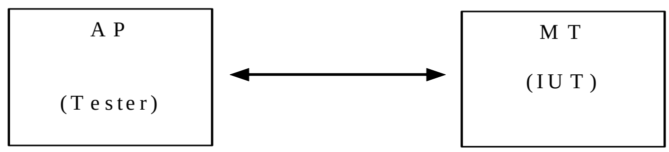


Figure 10: Normal configuration for MT

The normal configuration is for testing the behaviour between the MT and only one AP.

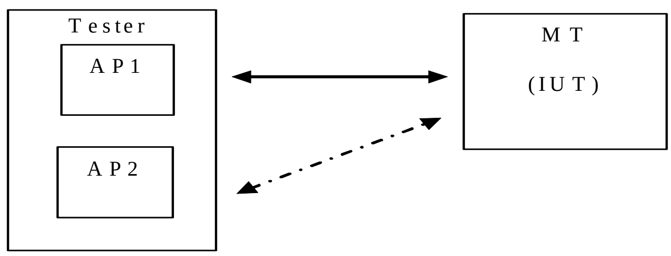


Figure 11: Load levelling configuration for MT

The load-levelling configuration is used when the MT has to interact with two APs. In that case, the two simulated APs are configured to be either a multi-sector AP or two separate APs. Concurrent TTCN functions are used for testing this configuration.

5.1.1.2 Test Configurations for AP

One configuration is defined for AP testing.

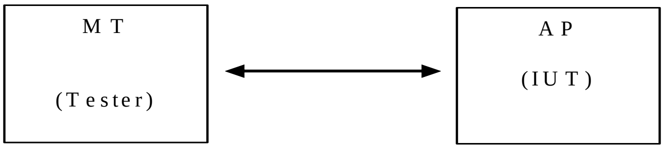


Figure 12: Normal configuration for AP

The normal configuration is for testing the behaviour between the MT and only one AP.

5.2 PLT components

The below discussion uses figure 2 "PLT and PUT Components" of clause 4.2.1 as the basic components diagram.

5.2.1 Existing components

The components described below exist already and were taken "off-the-shelf" and used as such.

5.2.1.1 Test system

The test system is TTCN3-based.

According to [TTCN3 TCI] a test system contains, as a minimum, the following components:

- A test management entity.
- A testing language execution environment.
- One or more codecs.
- And, an adapter to the test system used.

This structure is presented in figure 13.

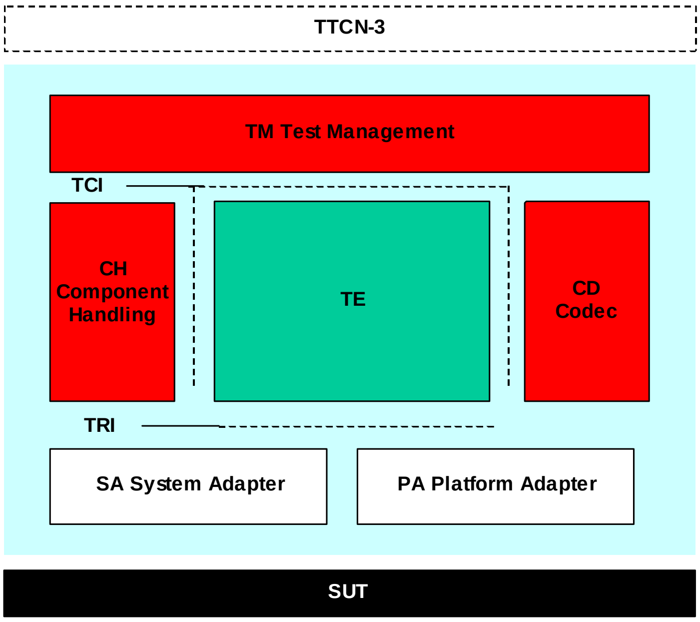


Figure 13: A TTCN-3 Test System

The TE, the TTCN-3 Execution Environment executes the TTCN-3 specification. The communication with the System Under Test (SUT) is performed by the System Adaptor (SA), while the implementation of time and of TTCN-3 external functions is done within the Platform Adaptor (PA). Otherwise said, the System Adapter communicates with PUT. The System and Platform Adaptors are the TTCN-3 equivalent of the Test Runtime Adapters shown in figure 2: "PLT and PUT Components". The Test Execution Control maps to the TTCN-3 Test Management component. Component Handling is not show in figure 2. It is used for concurrent/parallel testing where there is more than one IUT or where two or more components of the test suite are executed at the same time. The interface between the TE and the SA and the TE and the PA is defined within the TRI part of TTCN-3.

TCI defines interfaces for the implementation of codecs that encode and decode data present in the TE according to the specified encoding rules.

The following clauses present the individual components in more detail.

5.2.1.2 Abstract Test Suite (ATS)

The [HL2 ATS] has been developed within ETSI in TTCN-2. A description of the test architecture and the three test configurations necessary for the ATS were presented above.

The running of the ATS against a manufactured implementation usually requires the expensive test equipment associated with conformance testing. However, the present document presents running the ATS against a protocol layer implementation and requires much simpler test equipment-the PLT. It is important to emphasize that for testing with the PLT and for validating/building the test system prototype that **no** modification of the ATS has to be performed.

5.2.1.3 Test System Prototype

The Test System Prototype has been developed on the basis of the TTCN-2 ATS which was then translated automatically into its TTCN-3 equivalent.

The Test System Prototype uses only standardized interfaces and follows a generic test implementation framework derived from the generic test system architecture as presented in clause 5.2.1.1. The following steps are part of the implementation process:

1) Adaptation to the test system.

2) Implementation of codecs.

3) Integration of the test management functions.

Test suite validation should rely on the execution of the test suite. There are other validation methods such as walk-throughs that are useful as well. But nothing is better for validation then executing the test suite against something.

For this work, the test validation process was split into three steps.

Step 1

The implementation of the HiperLAN2 Test System Prototype requires the implementation of a system adapter (SA) and platform adapter (PA) as defined in [TRI]. The purpose of the SA is to implement the communication aspects of the ATS. In other words to implement the sending and receiving of messages. As the implementation of and the access to underlying communication layers vary from test device to test device, this step is referred to as adaptation to the test system. Different test devices are used for different test purposes. In the context of the HiperLAN2 Test System Prototype, the test device is defined to be a PC offering UDP/IP communication. For the implementation of the System Adaptor, built-in operations of the Java SDK have been used to realize a UDP/IP connection. Adapting the Test System Prototype to other lower layers requires changing only the SA implementation.

This step included:

- the implementation of the Test System Adapter on the test tool side by respecting the defined UDP/IP interface;

- the generation of coding/decoding functions from ASN.1 HiperLAN2 protocol specification using PER rules as specified in the DLC Technical Specification. As different projects have shown, the implementation of a codec can constitute a significant amount of time, especially in a Protocol Layer Tester scenario. The required amount of effort heavily depends on the type of encoding (e.g. text based, tabular based, etc) and the notation used to describe this encoding. Codecs are discussed further in a clause below;

- the translation of the test suite from TTCN-2 to TTCN-3. The source ATS was written in TTCN-2. Because of the standardized TRI and TCI for TTCN-3 and the availability of test equipment with these interfaces, the TTCN-2 ATS was converted automatically to an equivalent TTCN-3 ATS;

- the compilation of the TTCN-3 test suite.

While the interfaces for the first step is defined in [TRI], the interfaces for step two and three are defined in [TCI].

Step 2

The task of encoders and decoders (short: codecs) is to translate the abstract data as defined in an abstract test suite into its concrete representation. This concrete representation is referred to as encoding or coding. In general all data that is exchanged with the "real" test system (also the Test System Prototype) has to be encoded. Although the encoding of the same data might be different and depends of the usage of the data, typically the term encoding relates to the encoding related to the peer entity, the IUT. Thus the implementation of the HiperLAN2 Test System Prototype requires an implementation of the encoding as specified in the HiperLAN2 specification, i.e. as specified in bit tables.

Step 3

The last step refers to accessibility of the HiperLAN2 Test System Prototype. Executing an implemented test suite means that for starting and stopping, a test run must be available. This task typically depends on the test device and, therefore, on the test management capabilities offered. The targeted platform for the HiperLAN2 Test System Prototype is a standard PC. Therefore the availability of a test management system can not be guaranteed. However, the used test execution environment TTrun offers the basic functionality of a graphical test management system. Thus only the used test management has to be configured. Migrating the Test System Prototype onto a physical test device would require some additional resources for this task. However this would be in the responsibility of the test solution provider, and had been therefore not considered.

The experimental HiperLAN2 Test System Prototype was built on PC/Windows 2000 using Java 2 SDK with the TTCN-3 runtime environment (TTrun) from Testing Technologies. TTrun implements the TTCN-3 Runtime Interface specified by ETSI [TTCN3 TRI].

5.2.1.3.1 Codecs

One of the most time and resource consuming task in test suite implementation is to implement the codecs. The codec translates the abstract data as described in TTCN into its concrete representation and vice versa. In general, protocols define either their own encoding scheme by using a so called "tabular" encoding, or they rely upon standardized encoding rules such as ASN.1. Examples for these encoding rules are BER and PER. However, the availability of defined and standardized encoding rules does not solve the problem that that codecs have to be integrated into the test environment.

In a PLT context, codecs for two tasks can be identified. On one hand, codecs that encodes/decodes ASP and PDUs for the peer communication with the PUT are needed. On the other hand codecs for the encoding and decoding of the Wire Datagrams are needed. While the encoding rules for the first one are specified by the appropriate protocol standard, the latter one is defined as part of PLT development. The encoding rules for ASPs and PDUs cannot be modified by the PLT developer while those for the PUT's Wire Datagram are the developer's choice.

5.2.1.3.1.1 PDU and ASP Codecs

HiperLAN2 defines protocol data units using the ASN.1 with [ASN1 PER] encoding rules. For the implementation of the PER encoding rules, different commercial tools and software libraries are available. However, as it already has been stated before, the availability of tools that produce standard compliant encodings does not solve the problem of integrating them into the PLT.

Existing encoding/decoding tools typically offer a value API in order to fill in the tool's proprietary data structures. Afterwards the codec operates on this codec internal data structures in order to generate the encoded representation of this data structure.

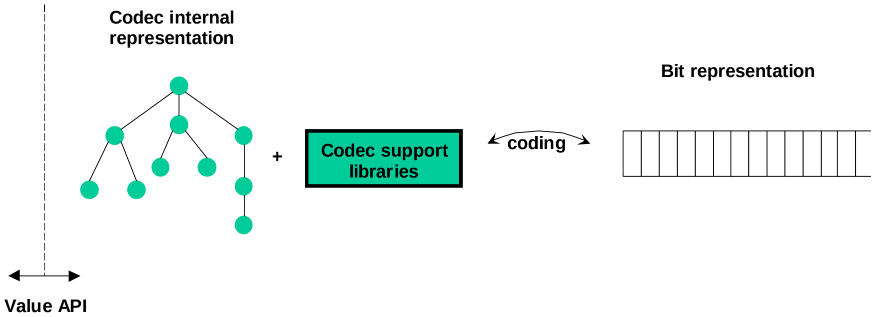


Figure 14: General operation of codecs

Figure 14 shows this process.

Integrating existing codecs into an existing environment like PLT's TTCN-3 conformant runtime environment is reduced to the task of translating an application (i.e. tester) internal data structure into the codecs' internal data structure and vice versa.

As both the PLT's and the codecs' runtime environment are not tailored for a particular purpose, tooling for solving this task generally is available.

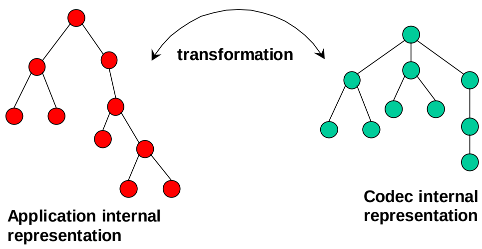


Figure 15: Tree - Usage of encoders

Thus the task of encoding PDUs/ASPs in a PLT can be solved efficiently as long as:

a) Standardized encoding rules have been specified in the protocol standard.

b) Standardized runtime environments are used.

HiperLAN2 defines the usage of standardized encoding rules (PER) instead of transfer syntax tables. Thus, existing coding tools have been used for the implementation of the PDU/ASP codecs. As the test environment offers a standardized coding interface (TCI-CD), the integration of the codec resulted in the application of an available TCI-to-Codec translator. As a result, the resources required for this integration are remarkably low.

However, the fact that off-the-shelf codecs offer only proprietary interfaces limits the applicability of this approach. If a standardized coding interface for codec generators would have been available, PLT implementers or users could have selected codecs according to their interface features, thus increasing acceptance and applicability of the PLT approach in the area of standardized encoding rules.

5.2.1.3.1.2 APIs Datagram Codecs

As described, a PLT exchanges wire datagrams with the PUT. Obviously, this datagram must be encoded and decoded. Clause 4 introduced the API for construction and accessing these datagrams. However this access is at an abstract level. In order to implement a complete PLT the abstract datagram has to be encoded.

The HiperLAN2 wire datagrams have been defined using ASN.1 and the HiperLAN2 API has defined the read and write access on the information that has been defined using ASN.1. PER has been chosen because of the availability of codecs. In fact, the same tools used for PDU encoding have been used to gain time and resources. In reality, the codec's internal structure had to be accessed in order to provide the API the necessary information. The fact that standardized notation (ASN.1) together with standardized encoding rules have been chosen to define the datagrams and their encoding has improved the reliability of the implementations. The manual implementation of codecs is by no means trivial and is, in fact, quite error prone.

5.2.2 Developed components

The following presents the components that had to be developed for PLT. Additionally, PUT components had to be developed since a PUT prototype was unavailable at the time of testing.

5.2.2.1 Wire datagram

The ASN.1 description of the wire datagram is at annex A. Annex A also gives an example of a set of values for one wire datagram.

As explained in clause 4, the wire datagram has two parts: the header containing the context/environment and the body that contains, in the HiperLAN2 case, a PDU.

5.2.2.2 The API for HiperLAN2 DLC

In order to facilitate development, the API has been refined with specialized interfaces that provide useful operations in order to access the API header elements without dealing with any API coding related issues.

Although the API concept could be abstracted from its transporter and implementation technology, a UDP/IP transporter is assumed hereafter.

A UDP/IP based transporter uses UDP communication on both sides, the PUT and PLT, for communicating PDUs or frames (see note 1).

NOTE 1: For readability reasons in the following the term PDU is used whenever data send to and received from a protocol layer is referenced. Depending on the abstraction chosen in the test suite, the relevant data elements might also be frames, multiple PDU, etc.

The API transfers information between the Test System and the IUT(s). The body contains the protocol message units that are, in this case, DLC PDUs. The header contains the information needed both by the IUTs and the Test system.

The Test system information requirements for running the test cases were determined by a hand review of the test cases to determine what additional information over and above the PDUs were required.

This interface between the test system and the UDP datagram was developed and is shown in figure 16.

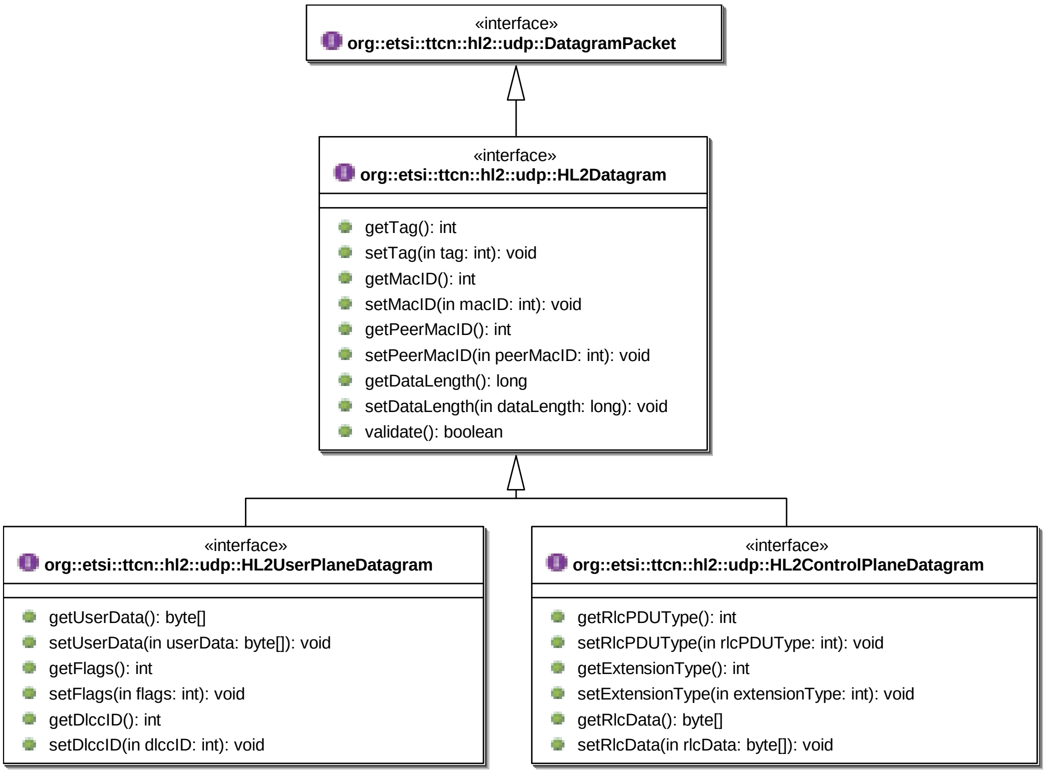


Figure 16: Interface between Test System and UDP datagram

As it has been seen in clause 4.3.1 the DatagramPacket interface provides generic means for the communication via the transporter. But this level of abstraction is insufficient when a test adaptation has to access elements of the API header and body. As it will be shown below the API header contains various types of information where the operation on bit level for providing and retrieving the data is inappropriate.

Implementations of the specified interfaces `HL2UserPlaneDatagram` and `HL2ControlPlaneDatagram` hide the need of handling the encoding/decoding of API messages directly within a test adapter. Datagram implementations can therefore be considered to provide the coding for different API messages. The user, i.e. the test adapter implementers can therefore focus on the provisioning of the necessary API information.

Figure 16 displays an extract of the specialized API definitions. For different type of API distinct datagram packets have been defined. As the different API messages share some common functionality a Hiperlan2 "base packet" has been defined (`HL2Datagram`). Special information that is only present in the different API messages is available only in the special packets `HL2UserPlaneDatagram` and `HL2ControlPlaneDatagram`.

As it can be easily concluded implementations of `HL2UserPlaneDatagram` and `HL2ControlPlaneDatagram` perform the encoding/decoding of the API header information, like `DlccID` or `ExtensionType`, and the API message body, like `UserData` or `RlcData`.

Thus the `DatagramPacket` and its specializations perform a central role in the encapsulation of the API message and therefore increase the reusability. One the one hand side the implementations of the API messages (`HL2UserPlaneDatagram` and `HL2ControlPlaneDatagram`) can be potentially reused within the Test Runtime Adapter and PUT Adapter if required. On the other side an implementation of the transporter is completely independent of the API messages it transports and thus reusable for different kind PLTs.

The Test System's components are discussed in the following clauses.

Relation to TRI System Adapters

The DatagramSocketAPI has been designed considering the needs of a TTCN-3 System Adaptation (SA) Layer implementation. According to TRI the SA implements the communications aspects of a TTCN-3 test suite. For the PLT, this means encoding/decoding of API messages and their sending/receiving. Clause 4 introduced the concept of a protocol layer tester (PLT) and described the functionality of Adapters and the Transporter.

For the TRI, the SA implements the Test Runtime Adapters as well as parts of the transporter.

Using the concepts of the `DatagramSocketAPI` a possible test case execution results in the following message exchange between the participating entities.

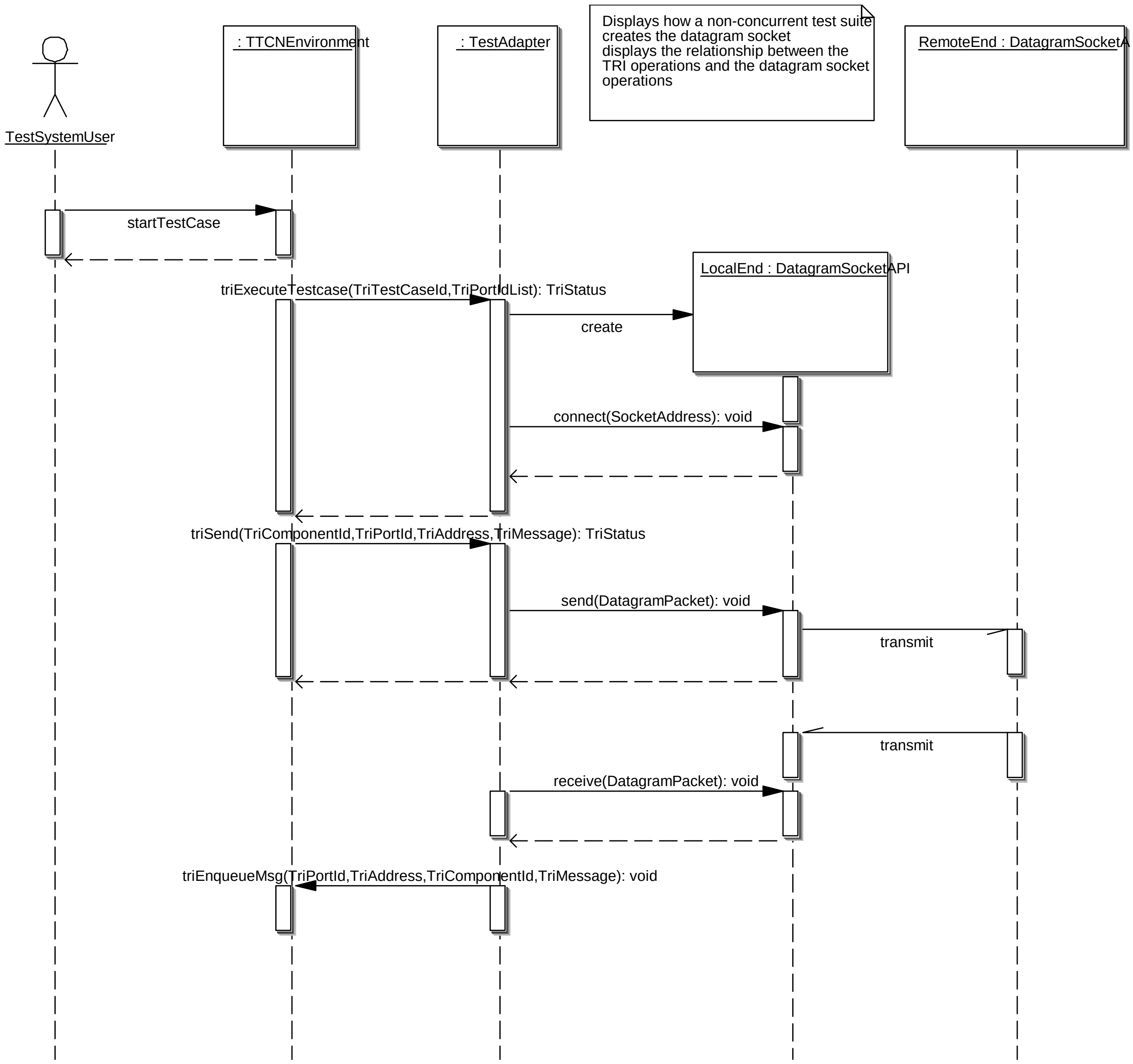


Figure 17: Starting a test case and using the DatagramSocketAPI

Figure 17 displays the necessary steps for a complete send and receive cycle. After the user triggers the execution of a test case at the test environment, a `triExecuteTestcase` operation is triggered within the SA according to TRI. Here it is assumed that the creation of the `DatagramSocketImpl` is performed at this point. In a concurrent test configuration scenario, the creation of the `DatagramSocketAPI` implementation could also be postponed to the occurrences of a `triMap()`. For simplicity reasons, we assume that a non-concurrent test scenario is described by figure 17.

From this point in time the test suite is able to receive messages from the PUT (see note 2).

NOTE 2: The PUT is modelled in this figure in having an analogous implementation of the DatagramSocketAPI. Although this is unnecessary, the PUT will use functionality similar to the ones defined within the DatagramSocketAPI approach within the PUT adapter.

Thus the `DatagramSocketAPI` is connected to the PUT using the connect operation. Sending a message from within the test suite will trigger a triSend operation at the SA. This will result in sending the message within a `DatagramPacket` to the PUT using the send operation on the `DatagramSocketAPI`, after having encoded the test suite PDU/ASP into the API message.

Processing a message received from the PUT will be identified by a successful call to receive on the `DatagramSocketAPI`. The return `DatagramPacket` will contain the API message sent by the PUT. The `DatagramPacket` contains the encoded API message. The PDU/ASP will be extracted and enqueued within the test system using the `triEnqueueMessage()` operation of the TRI interface.

5.2.2.3 Wire transport module

The Wire Transport Module is the software code required to place/extract the API information into/from the Wire Datagram structure and send/receive the datagram on the wire. Annex C presents the Java code for the Wire Transport module.

5.2.3 Clocks and timing

A detailed study was made of clocks and timing to determine how to best employ them in the PLT. Annex D presents that study.

5.2.4 Heuristics for defining an API

During the development of the API, lessons were learned that could be applied to the development of other APIs.

The type of testing affects the API. The API for conformance testing may have more information than that for interoperability testing. For example, signal strengths may not be needed in interoperability testing but they may be necessary for conformance testing.

Another way of saying the same thing is that the testing configurations for interoperability testing are different from those for conformance testing. For interoperability testing, two or more implementations are connected. In conformance testing, test equipment is connected to one or more IUTs. This is a fundamental architectural difference that could require different testing information requirements in their respective API headers.

These heuristics apply only to conformance testing architectures.

The manufacturer of an IUT must define his own information requirements because he is the best situated to know what the implementation's information requirements are.

ETSI is best situated to determine what the test system's information requirements are. Thus, the following heuristics apply only to the information requirements for the TS side of the test architecture.

The manufacturer can significantly reduce the burden of developing its API requirements by using the ETSI Test System information requirements as a basis for its requirements.

The basic source for API elements is the ATS. An ETSI ATS is written in either TTCN-2 or TTCN-3. The source ATS for the API is the [HL2 ATS] written in TTCN-2. TTCN-2 concepts and components such as Abstract Service Primitives (ASPs) do not map into equivalent TTCN-3 concepts. For example, if one says ASP to a pure TTCN-3 writer, that test writer would have no idea what is meant. The equivalent code structures would very possibly be in the TTCN-3 tests but there would is no formal concept with a name to identify this code structure. Fortunately, all TTCN-2 code structures can be mapped/converted into equivalent TTCN-3 code.

Because the TTCN-2 test suite was used as one source for the API, the TTCN-2 concepts are used in the heuristics given below. The TTCN-3 reader must perform the exercise to convert these heuristics into her/his equivalent TTCN-3 code constructs.

The simplest heuristic is that the API body contains the set of all possible protocol message units; e.g. PDUs, multi-messages, packed PDUs, etc.

A simple heuristic independent of the testing language is to include in the API header all test suite data declarations that are not used in the body part of the API; i.e. all other data declarations not in the set of all possible messages. However this set of declarations is only a subset of the API header. There may be data types in the messages that are also needed in the API header data. The heuristic is simple, its application is time-consuming.

The different PCO types and their values are another source of API header information. (The PCO concept is valid for both TTCN-3 and TTCN-2.) If there is more than one PCO used in a test, the API header must contain the PCO value.

All data types TTCN-2 ASPs and their TTCN-3 equivalent, except for the protocol message units, are very good API header data type candidates.

Clocking information, like a frame number, shared between the TS and the IUT must be in the API header.

Returned TTCN-2 Test Suite Operation or Procedure types and their TTCN-3 equivalent are good API header data type candidates. Signal strength and transmitted/received frequencies often fall into this category.

MAC-IDs, connection numbers and/or identifiers, and their equivalents will always be in the API header.

Concurrent testing requires either multiple subsets of the API header data or successive sending of UDP datagrams in the same direction each of which has different API header data for distribution to the instances being concurrently tested. In either case, the adaptation layer must have additional logic to direct the successive datagrams to the proper instance or to direct the same datagram to the different instances given in the subsets.

A complete API can be large. It can dominate the size of the UDP datagram. If it is coupled with a large sized message unit, it is quite possible to exceed system MTU limits. One way of limiting size is to send only that data that is necessary for the message units being transported. That is, make all API data types optional and send only that which is necessary. This will require additional logic in the adaptation layers to determine which data elements are sent. Two factors that decide the sent data elements are the message in the body and the test case number.

The test case number is a useful, but not mandatory, API header field. Adaptation layer logic can key upon the test case number to determine adaptation layer behavior and the use of optional data elements.

The API data encoding can be in any coding scheme shared by the adaptation layers. The encoding should consider the largest integer transported. API data types can include large integers greater than 2^{32}. Some encoding schemes cannot handle integers larger than 2^{32}. In this case, a different encoding scheme must be used.

Always check to see if the IUT has its own elements to add to the API header.

Additional text strings can be added to the API header for instructing the tester, for information purposes, or for adaptation layer control logic.

Annex A: HiperaLAN2 Wire Datagram Specification

This annex specifies the wire datagram for the HiperLAN2 PLT. It is specified in ASN.1 to show the datagram's structure. PLT development and the testing conducted with it used the ASN.1 specification. However, implementors can use equivalent data types in other languages such as C, C++, Java, et cetera. The only condition is that the PLT and PUT share the same coding/decoding schemes for the wire datagram.

A.1 Wire datagram ASN.1 module

The following lists the HIPERLAN2 API necessary for running the all Abstract Test Suites for HIPERLAN2 contained in the various applicable ETSI TS (e.g. TS 101 823-2-3 [3]). Although there is an ETSI TS for each ATS, all test cases have been combined into the electronic TTCN-2 format in hip2_v014.MP. The tabular format of these test cases is in hip2_v014.MP. Both files are electronic annexes to TS 101 823-2-3 [3].

The module in clause A.1 specifies the structure of the data exchanged between the Protocol Layer Tester and the Protocol Under Test over their common wire interface. This structure is expressed in ASN.1. However, the data itself does not have to be encoded and decoded in ASN.1. Any other encoding/decoding scheme is acceptable providing that that Protocol Layer Tester and Protocol Under Test share the same scheme. Examples included C, C++, and Java coding of information.

The data structure does not take into account specific machine limitations such as maximum integer size (MAX_INTEGER) or Maximum Transmission Unit (MTU) over IPv4. Specifically, two problems may occur.

- MAX_INTEGER on some machines is 32 bits long. `csrOffset` in `WriteReqArg` is 42 bits. The frame number is greater than 32 bits as well.
- Some PDUs such as RLC-CONNECT or RLC-DATA can have a large size. Also, the API header is greater than 500 bytes. The test equipment does not include SAR. If UDP is used, the maximum allowable datagram size (MTU) may be too small to carry the entire PDU. In this case, API behavior is unspecified.

```
--The API is presented in ASN.1. This is meant only to show the API elements and structure.
--Encoding and decoding of the API is at the user's discretion. For example, the interoperability
--and prototype API used 32 bit integers for both integer and enumerated types.
--NOTE: csrOffset is 48 bits long!!

HL2api
DEFINITIONS

    AUTOMATIC TAGS ::=

BEGIN


HLapi ::= SEQUENCE {
  hdr   HEADER,
  aPDU  HIPERLAN2pdu
  }

HIPERLAN2pdu ::= OCTET STRING

HEADER ::= SEQUENCE {
  configurationPart1          RlcConfigurationASP,
  configurationPart2          RlcConfigurationASP,
  cmd                         INTEGER,              --Range is (0..n) where n is very low, <8
  macId                       INTEGER (0..255),
  type                        INTEGER,              --Range is (0..n) where n is very low, <8
  dlccId                      INTEGER (0..63),
  ducDescr                    DucDescr,
  number                      INTEGER,              --SDU number/identifier
  delay                       INTEGER,              --Waiting time in ms between two bursts
  duration                    INTEGER,              --duration of data generation in s
  burstNumber                 INTEGER,
  offset                      INTEGER,              --Offset to start the error generation in
                                                    --the following data transmission
```

```
  length                        INTEGER (0..1024),
  correctIndication             CorrectIndication,
  snNumber                      INTEGER,
  availableSlot                 INTEGER,
  allocSomeRsp                  AllocSomeRsp,
  allocSomeReq                  AllocSomeReq,
  modifyBandwidthRsp            ModifyBandwidthStatus,
  modifyBandwidthReq            ModifyBandwidthReq,
  lockReqArg                    LockReqArg,
  lockRspArg                    LockRspArg,
  reclaimThisReq                ReclaimThisReq,
  reclaimThisRsp                ReclaimThisRsp,
  channel                       Channel,
  releaseThisRsp                ReleaseThisRsp,
  writeReqArg                   WriteReqArg,
  designatedCC                  DesignatedCC,
  channelsTxCharacteristics     ChannelsTxCharacteristics,
  bandwidthUse                  BandwidthUse
  }

RlcConfigurationASP ::= SEQUENCE {
  role                  INTEGER (0..3),
  netId                 INTEGER (0 .. 1023),
  apId                  INTEGER (0 .. 1023),
  sectorId              INTEGER (0 .. 7),
  tspxSectorId          INTEGER (0 .. 7),
  apTxLevel             INTEGER (0..15),
  apRxULlevel           INTEGER (0..7),
  version               INTEGER (0..7),
  apTrafficLoad         INTEGER (0..7),
  maximumPower          MaximumPower
  }

MaximumPower ::= ENUMERATED {
  notMaximumPower (0),
  maximumPower  (1)
  }

DucDescr ::= SEQUENCE {
  direction             Direction,
  dlccId                INTEGER (0..63),
  clConnAttr            OCTET STRING (SIZE(0..31)),
  ducDirectionDescrFw   DucDirectionDescrFw OPTIONAL,
  ducDirectionDescrBw   DucDirectionDescrBw OPTIONAL
  }

Direction ::= ENUMERATED {
  simplexForward (0),
  simplexBackward (1),
  duplex (2),
  duplexSymetric (3)
  }

DucDirectionDescrFw ::= DucDirectionDescr

DucDirectionDescrBw ::= DucDirectionDescr

DucDirectionDescr ::= SEQUENCE {
  allocationType        AllocationType,
  cyclicPrefix          CyclicPrefix,
  fecUsed               FecUsed,
  ecMode                EcMode,
  arqData               ArqData OPTIONAL,
  fecDescr              FecDescr OPTIONAL,
  fcaDescr              FcaDescr OPTIONAL,
  fsaDescr              FsaDescr OPTIONAL
  }

AllocationType ::= ENUMERATED {
  basic (0),
  fca (1),
  fsa (2)
  }

CyclicPrefix ::= ENUMERATED {
  t400ns (0),
  t800ns (1)
  }
```

```
FecUsed ::= ENUMERATED {
  fecNotUsed (0),
  fecUsed (1)
  }

EcMode ::= ENUMERATED {
  arqNotUsed (0),
  arqUsed (1),
  repetitionMode (2)
  }

ArqData ::= SEQUENCE {
  arqNrOfRetr        INTEGER (0..15),
  windowSize         WindowSize
  }

WindowSize ::= ENUMERATED {
  arqNotUsed  (0),
  wSize32  (1),
  wSize64  (2),
  wSize128  (3),
  wSize256  (4),
  wSize512  (5)
  }

FecDescr ::= SEQUENCE {
  coderType             CoderType,
  interleaverType       InterleaverType
  }

CoderType ::= ENUMERATED {
  reedSolomon216200  (0)
  }

InterleaverType ::= ENUMERATED {
  noInterleaver  (0),
  threeBranchCov (1)
  }

FcaDescr ::= SEQUENCE {
  schPerNbFrames        INTEGER (1..15),
  lchPerNbFrames        INTEGER (1..15),
  nbOfSch               INTEGER (0..1),
  phyModeSch            PhyModeSch,
  phyModeLch            PhyModeLch,
  nbOfLch               INTEGER (0..255),
  minNbOfLch            INTEGER (0..255)
  }

PhyModeSch ::= ENUMERATED {
  nophyModePropos  (0),
  cBPSK12  (1),
  cBPSK34  (2),
  cQPSK12  (3),
  cQPSK13  (4),
  c16QAM916  (5),
  c16QAM34  (6),
  c64QAM34  (7)
  }

PhyModeLch::= ENUMERATED {
  noPhyModeProposal  (0),
  cpBPSK12  (1),
  cpBPSK34  (2),
  cpQPSK12  (3),
  cpQPSK13  (4),
  cp16QAM916  (5),
  cp16QAM34  (6),
  cp64QAM34  (7)
  }


FsaDescr ::= SEQUENCE {
  phyModeLch            PhyModeLch,
  nbOfLch               INTEGER (0..255),
  minNbOfLch            INTEGER (0..255),
  startPointer          INTEGER (0..8191),
```

```
  startMacFrame         StartMacFrame
  }

StartMacFrame ::= SEQUENCE {
  repetitionCounter     INTEGER (0..4095),
  frameCount            INTEGER (0..15)
  }

CorrectIndication ::= ENUMERATED {
  receptionCorrect (0),
  receptionIncorrect (1)
  }

AllocSomeRsp ::= ENUMERATED {
  allocSomeDone (0),
  allocSomeChan (1),
  allocSomeReset(2),
  allocSomeBW (3),
  allocSomeBoth (4),
  allocSomeBadCommand (255)
  }

AllocSomeReq ::= SEQUENCE {
  msgType       Hl2LockMsgTypes,
  talkerID      INTEGER (0..63),
  listenerID    INTEGER (0..63),
  bandwidth     Bandwidth
  }

Hl2LockMsgTypes ::= ENUMERATED {
  allocateSome (0),
  modifyBandwidth (1),
  reclaimThis (2),
  releaseThis (3)
  }

ModifyBandwidthStatus ::= ENUMERATED {
  modifyBandwidthDone (0),
  modifyBandwidthChan (1),
  modifyBandwidthReset(2),
  modifyBandwidthBW (3),
  modifyBandwidthBadCommand (255)
  }

ModifyBandwidthReq ::= SEQUENCE {
  msgType       Hl2LockMsgTypes,
  streamChannel INTEGER (0..63),
  bandwidth     INTEGER (0..1023)
  }

LockReqArg ::= SEQUENCE {
  lockReqHdr    LockReqHdr,
  dataField     LockReqData
  }

LockReqHdr ::= SEQUENCE {
  destId        PhysicalId,
  tl            INTEGER (0..255),
  rt            INTEGER (0..3),
  tcode         INTEGER (0..15),
  pri           INTEGER (0..15),
  sourceId      PhysicalId,
  length        INTEGER(0..65535),
  extTcode      INTEGER (0..65535)
  }

PhysicalId ::= INTEGER (0..63)

LockReqData ::= CHOICE {
  oPCR  OPCR,
  iPCR  IPCR
  }

OPCR ::= SEQUENCE {
  onLine                INTEGER (0..1),
  broadcastCount        INTEGER (0..1),
  pointToPointCount     INTEGER (0..255),
  channel               INTEGER (0..255),
```

```
  payload               INTEGER (0..1023)
  }

IPCR ::= SEQUENCE {
  onLine                INTEGER (0..1),
  broadcastCount        INTEGER (0..1),
  pointToPointCount     INTEGER (0..255),
  channel               INTEGER (0..255)
  }

LockRspArg ::= SEQUENCE {
  aLockRspHdr   LockRspHdr,
  aRspData      LockRspData
  }

LockRspHdr ::= SEQUENCE {
  destId                PhysicalId,
  tl                    INTEGER (0..255),
  rt                    INTEGER (0..3),
  tcode                 INTEGER (0..15),
  pri                   INTEGER (0..15),
  sourceId              PhysicalId,
  rcode                 Rcode,
  length                INTEGER(0..65535),
  extTcode              INTEGER(0..65535)
  }

Rcode ::= CHOICE {
  respComplete          INTEGER(0),
  respError             LockRspError
  }

LockRspError ::= CHOICE {
  respConflictError     INTEGER(4),
  respDataError         INTEGER(5),
  respTypeError         INTEGER(6),
  respAddressError      INTEGER(7)
  }

LockRspData ::= SEQUENCE {
  oldValue              OCTET STRING
  }

ReclaimThisReq ::= SEQUENCE {
  msgType               Hl2LockMsgTypes,
  handle                GroupMacId,
  channel               INTEGER,
  payload               Bandwidth
  }

MacId ::= INTEGER (0..255)

GroupMacId ::= MacId (224..255)

Bandwidth ::= INTEGER (0..1023)

ReclaimThisRsp ::= SEQUENCE {
  status                ReclaimStatus
  }

ReclaimStatus ::= ENUMERATED {
  done (0),
  chan (1),
  reset (2),
  bandwidth (3),
  both (4),
  badCommand (255)
  }

ReleaseThisRsp ::= SEQUENCE {
  status                Hl2BusReleaseStatus
  }

Hl2BusReleaseStatus ::= ENUMERATED {
  done (0),
  chan (1),
  reset (2),
  badCommand (255)
```

```
  }

WriteReqArg ::= SEQUENCE {
  destiD                PhysicalId,
  tl                    INTEGER (0..255),
  rt                    INTEGER (0..3),
  tcode                 INTEGER (0..15),
  pri                   INTEGER (0..15),
  sourceId              PhysicalId,
  csrOffset             INTEGER,                -- 48 bits !!!
  quadletData           INTEGER
  }

DesignatedCC ::= ENUMERATED {           --Boolean enumeration
  notDesignated (0),
  designated (1)
  }

ChannelsTxCharacteristics  ::= SEQUENCE OF ChannelTxCharacteristics

ChannelTxCharacteristics ::= SEQUENCE {
  channel               Channel,
  txCharacteristics     TxCharacteristics
  }

TxCharacteristics ::= ENUMERATED {
  txCharacteristic1 (0),                --define values as required per test case
  txCharacteristic2 (1),
  txCharacteristic3 (2)
  --etc
  }

Channel ::= INTEGER (0..255)

BandwidthUse ::= ENUMERATED {
  notUsed (0),
  minimumUse (1),
  normalUse (2),
  maximumUse (3),
  saturated (4)
  }

END
```

A.2 DatagramSocketAPI Java interface

The work described in the present document has been performed mainly using the Java programming language. Therefore implementations have been done using Java.

The presented interfaces have been translated into Java interface. The source code of these interfaces is presented in the following:

```
// DatagramSocketAPI.java
package org.etsi.ttcn.hl2.udp ;
public interface DatagramSocketAPI {
    public boolean connect(SocketAddress remote) ;
    public void disconnect() ;
    public boolean isConnected() ;
    public boolean send(DatagramPacket packet) ;
    public boolean receive(DatagramPacket packet, int timeout) ;
    public void close() ;
}
// DatagramPacket.java
package org.etsi.ttcn.hl2.udp ;
public interface DatagramPacket {

     public byte[] getData() ;
     public int getOffset() ;
     public int getLength() ;
     public void setData(byte[] buf, int offset, int length) ;
     public void setSocketAddress(SocketAddress address) ;
     public SocketAddress getSocketAddress() ;
     public void setData(byte[] buf) ;
```

```
    public void setLength(int length) ;
}
// SocketAddress.java
package org.etsi.ttcn.hl2.udp;
public interface SocketAddress {
    public String getHostName() ;
    public void setByHostName(String hostName) ;
    public void setPort(int port) ;
    public int getPort() ;
}
```

A.3 Specialized Hiperlan2 Datagrams as Java interface

The following clause displays the complete Java interface descriptions for the specialized Hiperlan2 Datagrams as used for the presented work.

```
// HL2Datagram.java
package org.etsi.ttcn.hl2.udp;

public interface HL2Datagram extends DatagramPacket {

    public static final int MAX_HL2_DATAGRAM_LENGTH = 60 ;
    public static final int HEADER_LENGTH = 12 ;
    public static final int POS_TAG = 0 ;
    public static final int POS_MAC_ID = 4 ;
    public static final int POS_PEER_MAC_ID = 5 ;
    public static final int POS_DATA_LENGTH = 8 ;

    /**
     * Returns the TAG of this HiperLan2 datagram
     *
     * @return the TAG field as integer
     */
    public int getTag()  ;

    /**
     * Sets the TAG field of this HiperLan2 datagram.
     * In addition it set the data length of this datagram
     * to the valid value, i.e. if the tag indicates a short channel
     * pdu the length will be set to 5, else to 48.
     *
     * A short channel pdu has either the tag values: 0x80, 0x82, 0x88,
     * 0x8A, 0x8E
     *
     * It has the same effect as calling setLength() with the respective
     * values.
     *
     * @param tag as defined in the UDP API
     */
    public void setTag(int tag) ;

    /**
     * Returns the mac id of this HiperLan2 datagram
     *
     * @return the mac id as integer
     */
    public int getMacID() ;

    /**
     * Sets the mac id of this HiperLan2 datagram
     *
     * @param macID as defined in the UDP API
     */
    public void setMacID(int macID) ;

    /**
     * Returns the peer mac id of this HiperLan2 datagram
     *
     * @return the peer mac id as integer
     */
    public int getPeerMacID() ;

    /**
     * Sets the peer mac id of this HiperLan2 datagram
     *
     * @param peerMacID as defined in the UDP API
```

```
     */
    public void setPeerMacID(int peerMacID) ;

    /**
     * Returns the data length of the HiperLan2 datagram
     * @return the length of the PDU in bytes
     */
    public long getDataLength() ;

    /**
     * Sets the data length of the HiperLan2 datagram.
     * If the datagram contains a long channel PDU then length should
     * be 48. In case the datagram contains a short channel PDU then length
     * should contain 5.
     *
     * If called after setTag() this would overide the length calculation
     * done in setTag()
     *
     * @param dataLength the length of the PDU in bytes
     */
    public void setDataLength(long dataLength) ;

    /**
     * Validates the format of the HL2 header of this datagram.
     * Only the format, i.e. value ranges, etc.
     *
     * @return true if the datagram has the correct format, false else.
     */
    public boolean validate() ;

}


//HL2UserPlaneDatagram.java
package org.etsi.ttcn.hl2.udp;

public interface HL2UserPlaneDatagram extends HL2Datagram {

    /**
     * Returns the RLC user data as byte array.
     *
     * @return the RLC user plane PDU
     */
    public byte[] getUserData() ;

    /**
     * Sets the RLC user data of this user plane datagram
     *
     * @param userData the RLC user data
     */
    public void setUserData(byte[] userData) ;

    /**
     * Returns the flags of this HiperLan2 User Plane datagram
     *
     * @return the flags encoded as integer
     */
    public int getFlags() ;

    /**
     * Sets the flag of this HiperLan2 User Plane datagram
     *
     * @param flags the flags encoded as integer
     */
    public void setFlags(int flags) ;

    /**
     * Returns the DLCC ID of this HiperLan2 User Plane datagram
     *
     * @return the DLCC ID encoded as integer
     */
    public int getDlccID() ;

    /**
     * Sets the DLCC ID of this HiperLan2 User Plane datagram
     *
     * @param dlccID the DLCC ID encoded as integer
     */
```

```
    public void setDlccID(int dlccID) ;

}

// HL2ControlPlaneDatagram.java
package org.etsi.ttcn.hl2.udp;

public interface HL2ControlPlaneDatagram extends HL2Datagram {

    public static final int POS_RLC_PDU_TYPE = 1 ;
    public static final int POS_EXT = 2 ;
    public static final int POS_RLC_PDU = 12 ;

    /**
     * Returns the RLC PDU Types of this control plane datagram.
     *
     * @return the RLC PDU types as defined in the UDP API
     */
    public int getRlcPDUType() ;

    /**
     * Sets the RLC PDU types of this control plane datagram.
     *
     * @param rlcPDUType the RLC PDU type as defined in the UDP API
     */
    public void setRlcPDUType(int rlcPDUType) ;

    /**
     * Returns the extension type of this control plane datagram
     *
     * @return the extension type as defined in the UDP API
     */
    public int getExtensionType() ;

    /**
     * Sets the extensions type of this control plane datagram
     *
     * @param extensionType the extension types as defined in the UDP API
     */
    public void setExtensionType(int extensionType) ;

    /**
     * Returns the RLC Control Plane PDU as a byte array.
     * The byte array can be two octets long in the case of a
     * short channel RLC PDU in downling, or 5 octets long in the
     * case of a short channel RLC PDU in Uplink, or 48 octet long
     * in the case of a long channel RLC PDU (Up/Downlink)
     *
     * @return the RLC control plane PDU
     */
    public byte[] getRlcData() ;

    /**
     * Sets the RLC control plane PDU of this control plane datagram.
     *
     * The PDU size might differ for different kind of PDUs, like short channel
     * PDUs in Up- or Downlink or Long channel PDU (Up- and Downlink).
     * <p>
     * In case the MAC ID must be provided as last octet in a short
     * channel RLC PDU in Uplink the octet must be present in the PDU.
     * <p>
     * Setting the RLC data, adjust also the dataLength of the
     * HL2ControlPlane datagram. This is the same as calling first
     * <code>setRlcData(buf)</code> immediately followed by
     * <code>setDataLength(buf.length)</code>.
     *
     * @param rlcData the RLC control plane PDU
     */
    public void setRlcData(byte[] rlcData) ;

}
```

Annex B: HiperLAN2 API Specifications

B.1 DatagramSocketAPI Specification

connect

Signature	`connect(in remote: SocketAddress): boolean`
In Parameters	`remote` the remote peer entity described as SocketAddress
Return Value	`true` if the connection could have been established, `false` otherwise
Effect	Connects this socket. The socket is configured so that it only receives datagrams from, and sends datagrams to, the given `remote` peer address. Once connected, datagrams may not be received from or sent to any other address. A datagram socket remains connected until it is explicitly disconnected.

disconnect

Signature	`disconnect(): void`
In Parameters	`none`
Return Value	`void`
Effect	The socket can receive datagrams from, and sends datagrams to, any remote address.

isConnected

Signature	`isConnect(): boolean`
In Parameters	`None`
Return Value	`true` if this socket is connected, `false` otherwise
Effect	The operation returns `true` if the socket is connected, `false` otherwise.

send

Signature	`send(in packet: DatagramPacket): boolean`
In Parameters	`packet` the packet to be send
Return Value	`true` if the send operation was successful, `false` otherwise.
Effect	Sends a datagram packet from this socket. The `DatagramPacket` includes information indicating the data to be sent, its length, the IP address of the remote host, and the port number on the remote host. On a `send` operation, if the packet's address is set and the packet's address and the socket's address (in case the socket is connected) do not match, `false` will be returned and the packet will not be send.

receive

Signature	`receive(inout packet: DatagramPacket, in timeout: int): boolean`
In Parameters	`packet` a `DatagramPacket` where the received packet could be stored in `timeout` the amount of milliseconds the operation blocks when waiting to receive a packet
Return Value	`true` if a packet has been received or a timeout has occurred, `false` in any other error condition
Effect	Receives a datagram packet from this socket. When this method returns, the DatagramPacket's buffer is filled with the data received. The datagram packet also contains the sender's IP address, and the port number on the sender's machine. This method blocks until a datagram is received or the indicated time (in milliseconds) has passed. If a timeout has occurred packet will be returned unmodified. The length field of the datagram packet object contains the length of the received message. If the message is longer than the packet's length, the message is truncated.

close

Signature	`close(): void`
In Parameters	None
Return Value	Void
Effect	Closes the socket. All resources bound to this socket will get released.

getData

Signature	`getData(): byte[]`
In Parameters	`none`
Return Value	the buffer used to receive or send data
Effect	Returns the data buffer. The data received or the data to be sent starts from the `offset` in the buffer, and runs for `length` long. The values for `offset` and `length` can be retrieved with the respective operations.

getLength

Signature	`getLength(): int`
In Parameters	`None`
Return Value	The length of the data to be sent or the length of the data received.
Effect	Returns the length of the data to be sent or the length of the data received. The length of the data specifies the number of bytes in the byte buffer being relevant.

getOffset

Signature	`getOffset(): int`
In Parameters	`None`
Return Value	The offset of the data to be sent or the offset of the data received.
Effect	Returns the offset of the data to be sent or the offset of the data received.

getSocketAddress

Signature	`getSocketAddress(): SocketAddress`
In Parameters	`None`
Return Value	the associated `SocketAddress` of this `DatagramPacket` if present, or `null` else
Effect	Gets the `SocketAddress` (usually IP address + port number) of the remote host that this packet is being sent to or is coming from.

setData

Signature	`SetData (in buf: byte[]): void`
In Parameters	`buf` the data for the byte buffer
Return Value	Void
Effect	Set the data buffer for this packet, with the `offset` of this `DatagramPacket` set to 0, and the `length` set to the length of `buf`. `offset` and `length` can be retrieved with the respective operations.

setData

Signature	`setData (in buf: byte[], in offset: int, in length: int): void`
In Parameters	`buf` the buffer to set for this packet `offset` the offset into the data `length` the length of the data and/or the length of the buffer used to receive data
Return Value	void
Effect	Set the data buffer for this packet. This sets the data, `length` and `offset` of the packet.

setLength

Signature	`setLength(in length: int) void`
In Parameters	`length` the length to set for this packet.
Return Value	void
Effect	Set the length for this packet. The length of the packet is the number of bytes from the packet's data buffer that will be sent, or the number of bytes of the packet's data buffer that will be used for receiving data. The length must be lesser or equal to the offset plus the length of the packet's buffer.

setSocketAddress

Signature	`setSocketAddress(in address: SocketAddress) void`
In Parameters	`Address` the `SocketAddress`
Return Value	Void
Effect	Sets the `SocketAddress` (usually IP address + port number) of the remote host to which this datagram is being sent.

B.2 SocketAddress specification

getHostName

Signature	`getHostName(): String`
In Parameters	`None`
Return Value	the host name of this `SocketAddress`
Effect	If no `hostName` has been provided before the dotted IP-address `"0.0.0.0"` will be returned.

setByHostName

Signature	`setByHostName(in hostname: String): void`
In Parameters	`host` the specified host, or `null` for the local host.
Return Value	Void
Effect	Sets this `SocketAddress` to the IP address of a host, given the host's name. The host name can either be a machine name, such as `"portal.etsi.org"`, or a dotted ip-address of the form „`212.234.161.115`'. The port of this `SocketAddress` remains unchanged.

setPort

Signature	`SetPort(in port: int) void`
In Parameters	`port` the specified port number
Return Value	`Void`
Effect	Sets this `SocketAddress` to this port number. The host name of this `SocketAddress` remains unchanged.

getPort

Signature	getPort(): int
In Parameters	None
Return Value	the port number
Effect	Returns the port number of this SocketAddress.

B.3 Java interface

The work described in the present document has been performed mainly using the Java programming language. Therefore implementations have been done using Java.

The presented interfaces have been translated into Java interface. The source code of these interfaces is presented in the following:

```
// DatagramSocketAPI.java
package org.etsi.ttcn.udp ;
public interface DatagramSocketAPI {
    public boolean connect(SocketAddress remote) ;
    public void disconnect() ;
    public boolean isConnected() ;
    public boolean send(DatagramPacket packet) ;
    public boolean receive(DatagramPacket packet, int timeout) ;
    public void close() ;
}
// DatagramPacket.java
package org.etsi.ttcn.udp ;
public interface DatagramPacket {

     public byte[] getData() ;
     public int getOffset() ;
     public int getLength() ;
     public void setData(byte[] buf, int offset, int length) ;
     public void setSocketAddress(SocketAddress address) ;
     public SocketAddress getSocketAddress() ;
     public void setData(byte[] buf) ;
     public void setLength(int length) ;
}
// SocketAddress.java
package org.etsi.ttcn.udp;
public interface SocketAddress {
    public String getHostName() ;
    public void setByHostName(String hostName) ;
    public void setPort(int port) ;
    public int getPort() ;
}
```

Annex C: Clocks and timing

C.1 Clocks and timing

The development of the API raised a question concerning clocks. Is it advantageous for PLT and PUT to share the same clock whose value can be manipulated? In this way, the time waiting for timers to expire during testing could be reduced thereby saving time for those testing. In effect, a shared clock could allow a "time warp" to the next testable condition like the expiry of a timer or the reception of a signal. This is similar to moving our personal clock ahead one hour for Daylight Savings Time in order to wake up earlier.

In some BRAN testing, time warping could be very useful because some timers have a duration of 6 000ms. These long timers coupled with the scaling required by less than real-time simulations could result in long wait times. See the clause on "Timing".

This issue resulted in the following analysis.

C.1.1 Clocks

In testing, clocks are the source for timer values and expiration. The PLT and PUT each require a clock. They can share the same clock or each can have their own independent clock.

An example of a PLT and PUT sharing the same clock is where the frame number generated by the one is used as the clocking signal by the other. For HiperLAN2, this is possible for real implementations since AP periodically sends frames including a frame number that increments by one each time sent.

Independent clocks are usually used in both implementations and test equipment. Normally, the test equipment and the SUT each have and use their own clock.

The clock can be integrated into the PLT or PUT respectively or separated from it(i.e. internal versus external clocks). An oscillator within a device is an internal clock example. Such a device usually does not have an interface that test equipment or an observer can access. Thus, internal clocks allow their time to be observed but the time value cannot be manipulated. Thus, the time is absolute.

External clocks have an interface that allow time values to be changed. An example is the system clock of a computer. If an application program accesses the system clock for its timing purposes, the application program does so through an interface. Theoretically, any clock can be connected to this interface. Thus, the time values can be manipulated via the interface. The time is relative to the clock on the interface.

Table C.1 shows the possible combinations for clocks external and internal for the TE and SUT and presents an example for each. These clocks are unshared.

Table C.1: Internal and External Clock Examples for Unshared Clocks

SUT Clock	TE Clock	Example
Internal	Internal	Interoperability testing of two HiperLAN2 devices with one used as a "golden" SUT.
Internal	External	Conformance Testing a real HiperLAN2 implementation over its air interface.
External	Internal	SUT is a TTCN executable using the system clock. TE is an SDL using internal logic for clocking an not the system clock. Admittedly a far-fetched example but SDL can be used to validate the TTCN test suite. In this case, SUT is TTCN ATS and TE is the SDL.
External	External	Testing using PLT and PUT that each derive its timing from PC system clocks.

Table C.2 shows the possible combinations for shared clocks external and internal to the TE and SUT with examples.

Table C.2: Internal and External Clock Examples for Shared Clocks

SUT Clock	TE Clock	Example
Internal	Internal	Impossible case. By definition, two clocks that are internal cannot be shared.
Internal	External	Conformance Testing a real HiperLAN2 implementation over its air interface. The TE uses the SUT's frame number. In this case the SUT must be a HiperLAN2 AP. An AT does not generate the frame number.
External	Internal	Say that a protocol specification requires a device on one side of the protocol to generate the clocking for the other side. When the TE is on the clock generator side, then the clock could be considered internal to the TE if the clocking conforms to the specification. Obviously, "time warping" via an external clock is non-conformant in this case. The SUT then relies upon the TEs timing indication. An example might be an AT relying upon the AP for timing information and the TE has the AP role.
External	External	Testing using PLT and PUT that each derive their timing information from a shared buffer/stack. This is the classic "time warping" example.

C.1.2 Timing

There are some long timers in the BRAN protocols. One of the scenarios of testing wireless protocols over wire includes a TTCN platform running tests against software simulations (e.g. SDL). Such simulations are likely not able to run implementations in real time. For example, decoding a DL frame, determining the corresponding protocol action, and encoding the UL response within the delay of less then a millisecond is unlikely with an SDL simulation. Therefore, time scaling will be necessary to allow the simulation adequate time. The scaling may be by a factor from 10 to 100. This could ultimately result in having to wait ten minutes or so real time to see if the timer under test operates correctly.

It would be interesting for the tester to reduce this amount of wait time when testing opposite slow simulations or testing-thus the "time warp" idea.

C.1.3 Time Warping

The principle of time warping is simple. The TE and SUT each jump the same amount of time thereby speeding up test execution time. Of course, behaviour must not have occurred during the time warp. The time warp must occur in "dead time" for both the SUT and the TE. Unobserved behaviour ruins any value of testing.

The principle is shown in figure C.1.

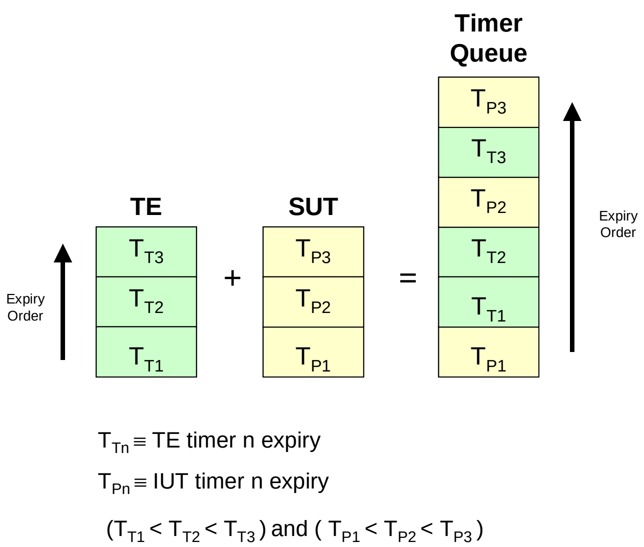


Figure C.1: Timer Queue Setup for Time Warping

In figure C.1, the timer events for the Test Equipment and the IUT in the SUT are arranged in chronological order in a timer queue. The timer events for the TE are known when the executable test case makes a call to the TRI to start or stop a timer.

Knowing the timer events in the IUT are problematical. If the IUT schedules timer events using an external clock, then the calls to schedule those timers can be monitored in order to build the SUT timer event queue. External clocks are used in protocol simulations and prototypes. Production implementations use internal or external clocks. For UDP/IP testing, it is likely that the IUT will use an external clock and, thus, it is likely the IUT timer queue can be developed.

If the SUT has an internal clock that the tester cannot manipulate, then time warping is not possible. For example, if the TE uses an external interrupt for timing, but the SUT uses an internal oscillator for its timing, then one cannot "warp" the oscillator to a time in the future by the any increment. Thus, the warping cannot be synchronized making it useless.

Once both timer queues are known, they are combined into one queue in chronological order to run the test.

In running a test, only one of two events are possible: a message is sent/received or a timer expiry. Practically speaking, two or more of these events cannot occur simultaneously since the events occur in a stack-like FIFO manner. Thus, we consider only two different events can occur.

- Sent/received messages:
  - For receiving a message, there are guard timers set awaiting the arrival of an expected message. If the received message is the expected message, these guard timers are cancelled and operation continues. This means that the guard timers are removed from the consolidated timer queue. If the received message is unexpected, then the test case stops. All timers are now useless and removed from the queue except for those needed to close out the test case and bring the IUT to a given state.
  - Timers or management entities cause message transmission. In the event of management entities transmitting a message, they usually start guard timers because a response to the message is required. These timers are added in chronological order to the queue. For timer-caused message transmission, see the point immediately below.
- Timer expiry occurs when a timer in the queue expires firing the actions necessary to send the message. When a timer in the queue expires, it is removed from the queue. The expiry of timer usually causes some response that then causes other timers to be set and placed in the queue.

The times used in the queue can be either absolute or relative. Absolute time starts at zero at the start of the first event. The setting of a timer is an event as well as transmitting or receiving a message. All times placed on the queue are based upon this first event. Relative time is the adjustment of the times in the timer queue by determining the time between the last event and the event that has just occurred and subtracting it from all the timer values in the queue. The type of time used (relative or absolute) is a matter of implementation convenience.

Note that random events can generated using a pseudo random number generator to determine the occurrence in time of the event. In this way, random events are placed into the timer queue.

Figure C.2 is an example of the queue, in relative time, after the earliest timer in figure C.2 has expired.

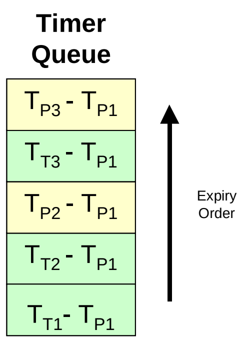


Figure C.2: Timer Queue after Timer Expiry (Relative Time)

C.1.4 Using Time Warping

The advantage of using time warping is to reduce testing time. One can skip to the next timer or message event rather than waiting for a timer event to occur in real or simulated time. In some cases, simulated time can be significantly longer than real time and lead to very long waits between events. In the event of one BRAN protocol, this wait time can be up to 6 minutes per event. With several events in a test case, a tester would have to spend a half-hour or more running one test case.

Several hand walk-throughs of test cases using time warping were conducted. Time warping was applicable to all tests and arrived at the same results as real-time testing. Scenarios were devised to "break" the concept but the concept remained intact.

Warping was not tried on a prototype implementation or simulation.

As discussed above, time warping could not be applied to IUT/SUT with internal clocks. Simply, there is no way to "warp" the IUT/SUT into the same time as the TE.

Using warping on a prototype implementation or simulation will require software to access the external clock interfaces, form the timer queues, and manipulate them. The time and resources needed to develop this software must be compared against the time and resources saved during test execution. If tests are to be run often, warping appears to be advantageous. If not, then the straightforward use of time appears to be advantageous.

Time warping was not used for UDP/IP testing for resource reasons. Since the STF is exploring the feasibility of UDP/IP testing, the actual time spent in running tests is significantly less than the total time for setting up of the TE, conversion of the transfer syntax, and the conversion of TTCN-2 to TTCN-3. The benefits obtained from warping would not outweigh the time required to develop the software necessary to implement warping.

However, warping may be viable for HiperLAN2 work if UDP/IP testing is pursued.

Annex D: The Java™ code of UDP/IP based virtual tester prototype

The Java code developed for the HiperLAN2 UDP/IP based virtual tester prototype is contained in the archive HIPERLAN_TestAdapter.zip contained in the archive tr_102365v010201p0.zip which accompanies the present document.

NOTE: Java™ is the trade name of a product supplied by Sun Microsytems™. This information is given for the convenience of users of the present document and does not constitute an endorsement by ETSI of the product named. Equivalent products may be used if they can be shown to lead to the same results.

History

Document history		
V1.1.1	June 2005	Publication
V1.2.1	June 2005	Publication