BS ISO 22900-2:2022

BSI Standards Publication

Road vehicles — Modular vehicle communication interface (MVCI)

Part 2: Diagnostic protocol data unit (D-PDU API)

Documents
Document Sharing Hub

Full document details: BS ISO 22900-2:2022

National foreword

This British Standard is the UK implementation of ISO 22900-2:2022. It supersedes BS ISO 22900-2:2017, which is withdrawn.

The UK participation in its preparation was entrusted to Technical Committee AUE/16, Data Communication (Road Vehicles).

A list of organizations represented on this committee can be obtained on request to its committee manager.

Contractual and legal considerations

This publication has been prepared in good faith, however no representation, warranty, assurance or undertaking (express or implied) is or will be made, and no responsibility or liability is or will be accepted by BSI in relation to the adequacy, accuracy, completeness or reasonableness of this publication. All and any such responsibility and liability is expressly disclaimed to the full extent permitted by the law.

This publication is provided as is, and is to be used at the recipient's own risk.

The recipient is advised to consider seeking professional guidance with respect to its use of this publication.

This publication is not intended to constitute a contract. Users are responsible for its correct application.

© The British Standards Institution 2022
Published by BSI Standards Limited 2022

ISBN 978 0 539 15953 0

ICS 43.040.15

Compliance with a British Standard cannot confer immunity from legal obligations.

This British Standard was published under the authority of the Standards Policy and Strategy Committee on 31 August 2022.

Amendments/corrigenda issued since publication

Date	Text affected

Documents
Document Sharing Hub

Full document details: BS ISO 22900-2:2022

INTERNATIONAL STANDARD

ISO 22900-2

Third edition
2022-06

Road vehicles — Modular vehicle communication interface (MVCI) —

Part 2: Diagnostic protocol data unit (D-PDU API)

Véhicules routiers — Interface de communication modulaire du véhicule (MVCI) —

Partie 2: Interface de programmation d'application d'unité de données du protocole de diagnostic (D-PDU API)

Reference number
ISO 22900-2:2022(E)

Documents
Document Sharing Hub
© ISO 2022

Full document details: BS ISO 22900-2:2022

COPYRIGHT PROTECTED DOCUMENT

© ISO 2022

All rights reserved. Unless otherwise specified, or required in the context of its implementation, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

© ISO 2022 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO 22900-2:2022

Contents

Page

Foreword ... vi

Introduction ... vii

1 Scope ... 1

2 Normative references ... 1

3 Terms, definitions and abbreviated terms ... 1
3.1 Terms and definitions ... 2
3.2 Abbreviated terms ... 3

4 Specification release version information ... 6
4.1 Specification release version location ... 6
4.2 Specification release version ... 6

5 Modular VCI use cases ... 6
5.1 OEM merger ... 6
5.2 OEM cross vehicle platform ECU(s) ... 6
5.3 Central source diagnostic data and exchange during ECU development ... 7
5.4 OEM franchised dealer and aftermarket service outlet diagnostic tool support ... 7

6 Modular VCI software architecture ... 7
6.1 Overview ... 7
6.2 Modular VCI D-Server software ... 8
6.3 Runtime format based on ODX ... 9
6.4 MVCI protocol module software ... 9
6.5 MVCI protocol module configurations ... 9

7 D-PDU API use cases ... 10
7.1 Overview ... 10
7.2 Use case 1: Single MVCI protocol module ... 11
7.3 Use case 2: Multiple MVCI protocol modules supported by same D-PDU API implementation ... 12
7.4 Use case 3: Multiple MVCI protocol modules supported by different D-PDU API implementations ... 13

8 Diagnostic protocol data unit (D-PDU) API ... 14
8.1 Software requirements ... 14
8.1.1 General requirements ... 14
8.1.2 Vehicle protocol requirements ... 15
8.1.3 Timing requirements for protocol handler messages ... 16
8.1.4 Serialization requirements for protocol handler messages ... 17
8.1.5 Compatibility requirements ... 19
8.1.6 Timestamp requirements ... 19
8.2 API function overview and communication principles ... 20
8.2.1 Terms used within the D-PDU API ... 20
8.2.2 Function overview ... 20
8.2.3 General usage ... 21
8.2.4 Asynchronous and synchronous communication ... 24
8.2.5 Usage of resource locking and resource unlocking ... 25
8.2.6 Usage of ComPrimitives ... 25
8.3 Tool integration ... 42
8.3.1 Requirement for generic configuration ... 42
8.3.2 Tool integrator — Use case ... 42
8.4 API functions — Interface description ... 44

Documents
Document Sharing Hub

© ISO 2022 – All rights reserved

Full document details: BS ISO 22900-2:2022

8.4.1 Overview 44
8.4.2 PDUConstruct 44
8.4.3 PDUDestruct 45
8.4.4 PDUIoCtl 46
8.4.5 PDUGetVersion 48
8.4.6 PDUGetStatus 49
8.4.7 PDUGetLastError 50
8.4.8 PDUGetResourceStatus 51
8.4.9 PDUCreateComLogicalLink 52
8.4.10 PDUDestroyComLogicalLink 55
8.4.11 PDUConnect 57
8.4.12 PDUDisconnect 59
8.4.13 PDULockResource 60
8.4.14 PDUUnlockResource 61
8.4.15 PDUGetComParam 62
8.4.16 PDUSetComParam 69
8.4.17 PDUStartComPrimitive 72
8.4.18 PDUCancelComPrimitive 76
8.4.19 PDUGetEventItem 77
8.4.20 PDUDestroyItem 78
8.4.21 PDURegisterEventCallback 79
8.4.22 EventCallback prototype 81
8.4.23 PDUGetObjectId 83
8.4.24 PDUGetModuleIds 84
8.4.25 PDUGetResourceIds 86
8.4.26 PDUGetConflictingResources 87
8.4.27 PDUGetUniqueRespIdTable 88
8.4.28 PDUSetUniqueRespIdTable 90
8.4.29 PDUModuleConnect 95
8.4.30 PDUModuleDisconnect 98
8.4.31 PDUGetTimestamp 99
8.5 I/O control section 99
8.5.1 IOCTL API command overview 99
8.5.2 PDU_IOCTL_RESET 102
8.5.3 PDU_IOCTL_CLEAR_TX_QUEUE 102
8.5.4 PDU_IOCTL_SUSPEND_TX_QUEUE 103
8.5.5 PDU_IOCTL_RESUME_TX_QUEUE 103
8.5.6 PDU_IOCTL_CLEAR_RX_QUEUE 104
8.5.7 PDU_IOCTL_CLEAR_TX_QUEUE_PENDING 104
8.5.8 PDU_IOCTL_READ_VBATT 105
8.5.9 PDU_IOCTL_SET_PROG_VOLTAGE 105
8.5.10 PDU_IOCTL_READ_PROG_VOLTAGE 106
8.5.11 PDU_IOCTL_GENERIC 107
8.5.12 PDU_IOCTL_SET_BUFFER_SIZE 108
8.5.13 PDU_IOCTL_GET_CABLE_ID 108
8.5.14 PDU_IOCTL_START_MSG_FILTER 109
8.5.15 PDU_IOCTL_STOP_MSG_FILTER 111
8.5.16 PDU_IOCTL_CLEAR_MSG_FILTER 111
8.5.17 PDU_IOCTL_SET_EVENT_QUEUE_PROPERTIES 112
8.5.18 PDU_IOCTL_SEND_BREAK 113
8.5.19 PDU_IOCTL_READ_IGNITION_SENSE_STATE 113
8.5.20 PDU_IOCTL_VEHICLE_ID_REQUEST 114
8.5.21 PDU_IOCTL_SET_ETH_SWITCH_STATE 117
8.5.22 PDU_IOCTL_GET_ENTITY_STATUS 118

Documents
Document Sharing Hub

© ISO 2022 – All rights reserved

Full document details: BS ISO 22900-2:2022

8.5.23 PDU_IOCTL_GET_DIAGNOSTIC_POWER_MODE ... 119
8.5.24 PDU_IOCTL_GET_ETH_PIN_OPTION ... 120
8.5.25 PDU_IOCTL_TLS_SET_CERTIFICATE ... 121
8.5.26 PDU_IOCTL_DOIP_GET_CURRENT_SESSION_MODE ... 122
8.5.27 PDU_IOCTL_ISOBUS_GET_DETECTED_CFS ... 123
8.6 API functions — Error handling ... 123
8.6.1 Synchronous error handling ... 123
8.6.2 Asynchronous error handling ... 123
8.7 Installation ... 124
8.7.1 Generic description ... 124
8.7.2 Windows installation process ... 124
8.7.3 Linux installation process ... 125
8.7.4 Selecting MVCI protocol modules ... 126
8.8 Application notes ... 126
8.8.1 Interaction with the MDF ... 126
8.8.2 Accessing additional hardware features for MVCI protocol modules ... 126
8.8.3 Documentation and information provided by MVCI protocol module vendors ... 126

9 Using the D-PDU API with existing applications ... 127
9.1 SAE J2534-1 and RP1210a existing standards ... 127

10 Data structures ... 128
10.1 API functions — Data structure definitions ... 128
10.1.1 Abstract basic data types ... 128
10.1.2 Definitions ... 129
10.1.3 Bit encoding for UNUM32 ... 129
10.1.4 API data structures ... 129

Annex A (normative) D-PDU API compatibility mappings ... 145

Annex B (normative) D-PDU API standard ComParams and protocols ... 146

Annex C (informative) D-PDU API manufacturer-specific ComParams and protocols ... 235

Annex D (informative) D-PDU API constants ... 237

Annex E (informative) Application defined tags ... 253

Annex F (informative) RDF and MDF description files ... 254

Annex G (informative) Resource handling scenarios ... 325

Annex H (informative) D-PDU API partitioning ... 331

Annex I (informative) Use case scenarios ... 335

Annex J (informative) OBD protocol initialization ... 376

Annex K (normative) DoIP implementation ... 392

Annex L (normative) ISOBUS ... 425

Bibliography ... 433

© ISO 2022 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of ISO documents should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT), see www.iso.org/iso/foreword.html.

This document was prepared by Technical Committee ISO/TC 22, *Road vehicles*, Subcommittee SC 31, *Data communication*.

This third edition cancels and replaces the second edition (ISO 22900-2:2017), which has been technically revised.

The main changes are as follows:

— introduction of reference to ISO 15765-5 for CAN-FD;

— introduction of secured communication for the DoIP protocol based on TLS (as defined in ISO 13400-2:2019);

— introduction of automatic DoIP reconnect handling;

— introduction of ISOBUS protocol.

A list of all parts in the ISO 22900 series can be found on the ISO website.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html.

© ISO 2022 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

Introduction

The ISO 22900 series is applicable to vehicle electronic control module diagnostics and programming.

This document was established in order to more easily exchange software and hardware of vehicle communication interfaces (VCIs) among diagnostic applications. It defines a generic and protocol independent software interface towards the modular vehicle communication interface (MVCI) protocol module, such that a diagnostic application based on this software interface can exchange the MVCI protocol module or add a new MVCI protocol module with minimal effort. Today, the automotive aftermarket requires flexible usage of different protocol modules for vehicles of different brands. Many of today's protocol modules are incompatible with regard to their hardware and software interface, such that, depending on the brand, a different protocol module is required.

The objective of this document is to specify the diagnostic protocol data unit application programming interface (D-PDU API) as a generic software interface and to provide a "plug and play" concept for access onto different MVCI protocol modules from different tool manufacturers. The D-PDU API will address the generic software interface, the protocol abstraction, its exchangeability, as well as the compatibility towards existing standards such as SAE J2534-1 and RP1210a.

The implementation of the modular VCI concept facilitates co-existence and re-use of MVCI protocol modules, especially in the aftermarket. As a result, diagnostic or programming applications can be adapted for different vehicle communication interfaces and different vehicles with minimal effort, thus helping to reduce overall costs for the tool manufacturer and end user.

Vehicle communication interfaces compliant with ISO 22900 series support a protocol-independent D-PDU API as specified in this document.

© ISO 2022 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

Documents
Document Sharing Hub

Road vehicles — Modular vehicle communication interface (MVCI) —

Part 2:
Diagnostic protocol data unit (D-PDU API)

1 Scope

This document specifies the diagnostic protocol data unit application programming interface (D-PDU API) as a modular vehicle communication interface (MVCI) protocol module software interface and common basis for diagnostic and reprogramming software applications.

This document covers the descriptions of the application programming interface (API) functions and the abstraction of diagnostic protocols, as well as the handling and description of MVCI protocol module features. Sample MVCI module description files accompany this document.

The purpose of this document is to ensure that diagnostic and reprogramming applications from any vehicle or tool manufacturer can operate on a common software interface and can easily exchange MVCI protocol module implementations.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

SAE J2411, *Single Wire CAN Network for Vehicle Applications*

3 Terms, definitions and abbreviated terms

For the purposes of this document, the following terms and definitions apply.

ISO and IEC maintain terminology databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at https://www.electropedia.org/

© ISO 2022 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

3.1 Terms and definitions

3.1.1
application
way of accessing the diagnostic protocol data unit application programming interface (D-PDU API)

Note 1 to entry: From the perspective of the D-PDU API, it does not make any difference whether an application accesses the software interface directly or through an MVCI D-Server. Consequently, in this document, the term "application" represents both ways of accessing the D-PDU API.

3.1.2
ComLogicalLink
logical communication channel towards a single electronic control unit (ECU) or towards multiple electronic control units

3.1.3
COMPARAM-SPEC
protocol-specific set of predefined communication parameters (ComParams), the value of which can be changed in the context of a layer or specific diagnostic service

Note 1 to entry: This part of the model can also contain OEM-specific ComParams.

3.1.4
ComPrimitive
smallest aggregation of a communication service or function

EXAMPLE A request message to be sent to an ECU.

3.1.5
Ethernet
physical network media type

3.1.6
local network
part of the network connected directly to the tester

Note 1 to entry: Also called "primary network" in some subclauses.

3.1.7
remote network
part of the network located behind a gateway connected to the tester

Note 1 to entry: Also called "secondary network" in some subclauses.

3.1.8
DoIP MVCI module
diagnostic communication over Internet Protocol modular vehicle communication interface module
MVCI module able to handle connections to one or multiple *DoIP entities* (3.1.11)

3.1.9
DoIP gateway
diagnostic communication over Internet Protocol gateway
gateway connected to the tester via DoIP protocol

© ISO 2022 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

3.1.10
DoIP node
diagnostic communication over Internet Protocol node
ECU connected to the tester directly via DoIP protocol

Note 1 to entry: ECU has no gateway capabilities.

3.1.11
DoIP entity
diagnostic communication over Internet Protocol entity
general term for either a *DoIP gateway* (3.1.9) or a *DoIP node* (3.1.10)

3.1.12
DoIP entity certificate
certificate issued by an *intermediate CA* (3.1.13) to *DoIP entity* (3.1.11)

Note 1 to entry: It is presented during the TLS handshake to the external test equipment to verify the authenticity of this DoIP entity.

3.1.13
intermediate CA
certificate authority which issues subordinal certificates

Note 1 to entry: A reason to issue subordinal certificates can be, for example to further to another intermediate CA or *DoIP entities* (3.1.11).

3.1.14
intermediate certificate
certificate used by an *intermediate CA* (3.1.13)

Note 1 to entry: The intermediate certificate(s) is/are either stored in the external test equipment or are presented during authentication together with the end node certificate to complete the chain of trust.

3.1.15
root CA
certificate authority (CA) acting as the root of trust

Note 1 to entry: Root CA typically issues *intermediate certificates* (3.1.14) to allow an *intermediate CA* (3.1.13) to further submit certificates.

3.1.16
root certificate
certificate of the r*oot CA* (3.1.15) used as the trust anchor

Note 1 to entry: It is securely stored and used by all entities that wants to validate end node certificates [e.g. from the *DoIP entity* (3.1.11)] together with all necessary *intermediate certificates* (3.1.14) in the chain of trust.

3.2 Abbreviated terms

API	application programming interface
ASCII	American Standard for Character Information Interchange
CA	certificate authority

© ISO 2022 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

CAN	controller area network
CAN IDs	controller area network identifiers
CDF	cable description file
CLL	ComLogicalLink
ComLogicalLinks	communication logical links
ComParam	communication parameter
ComParamSpec	communication parameter specification
ComPrimitive	communication primitive
CRC	cyclic redundancy check
DLC	data link connector
DLL	dynamic link library
DoIP	diagnostic communication over Internet Protocol
D-PDU	diagnostic protocol data unit
D-Server	diagnostic server
ECU	electronic control unit
HDD	hard disk drive
HI	differential line — high
HW	Hardware
IEEE 1394	firewire serial bus
IFR	in-frame response
IGN	Ignition
IOCTL	input/output control
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
K	UART K-Line
KWP	keyword protocol
L	UART L-Line
LL	Logical Link

© ISO 2022 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

LOW	differential line — low
MDF	module description file
MOST	media oriented systems transport
MVCI	modular vehicle communication interface
ODX	open diagnostic data exchange
OEM	original equipment manufacturer
OSI	open systems interconnection
PC	personal computer
PCI	protocol control information
PGN	parameter group number
PROGV	programmable voltage
PWM	pulse width modulation
RDF	root description file
RX	UART uni-directional receive
SCI	serial communications interface
SCP	standard corporate protocol
TCP	transmission control protocol
TLS	transport layer security
TX	UART uni-directional transmit
UDS	unified diagnostic services
UDSonCAN	unified diagnostic services on CAN
UDSonIP	unified diagnostic services on IP
USB	universal serial bus
USDT	unacknowledged segmented data transfer
UUDT	unacknowledged un-segmented data transfer
VCI	vehicle communication interface
VPW	variable pulse width
XML	extensible markup language

© ISO 2022 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

4 Specification release version information

4.1 Specification release version location

Specification release version information is contained in each modular VCI release document specification under the same clause title "Specification release version information". It is important to check for feature support between modular VCI release specifications if the most recent API features shall be implemented. The D-PDU API supports the reading of version information by the API function call PDUGetVersion.

Release version information is also contained in the following files:

— root description file (RDF);

— module description file (MDF);

— cable description file (CDF);

— D-PDU API library file.

4.2 Specification release version

The specification release version of this document is 2.2.2.

5 Modular VCI use cases

5.1 OEM merger

In the past, several OEMs in the automotive industry have merged into one company.

All companies try to leverage existing (legacy) components and jointly develop new products, which are common across different vehicle types and badges.

If OEMs already had modular VCI compliant test equipment, it would be simple to connect MVCI protocol modules from merged OEMs into one chassis or device. All protocols would be supported by a single MVCI protocol module configuration without any replacement of MVCI protocol module hardware at the dealer site. The same applies for engineering and some of this concept might also work for production plants (end of line).

5.2 OEM cross vehicle platform ECU(s)

OEMs specify requirements and design electronic systems to be implemented in multiple vehicle platforms in order to avoid re-inventing a system for different vehicles. The majority of design, normal operation and diagnostic data of an electronic system are re-used if installed in various vehicles. The engineering development centres are located worldwide. A great amount of re-authoring of diagnostic data is performed to support different engineering test tools.

Providing diagnostic data in an industry standard format like ODX and XML will avoid re-authoring into various test tool specific formats at different system engineering locations. The D-PDU API supports this re-use concept by fully abstracting vehicle protocols into the industry supported ComParam descriptions.

© ISO 2022 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

5.3 Central source diagnostic data and exchange during ECU development

Single source origin of diagnostic data (as depicted in Figure 1), combined with a verification and feedback mechanism and distribution to the end users, is highly desirable in order to lower engineering expenses. Engineering, manufacturing and service specify which protocol and data shall be implemented in the ECU. This information will be documented in a structured format like XML. Furthermore, the same structured data files can be used to setup the diagnostic engineering tools to verify proper communication with the ECU and to perform functional verification and compliance testing of the ECU. Once all quality goals are met, these structured data files shall be released to the OEM database. An Open Diagnostic data eXchange (ODX) schema has been developed for the purpose of supporting these structured formatted files used for ECU diagnosis and validation.

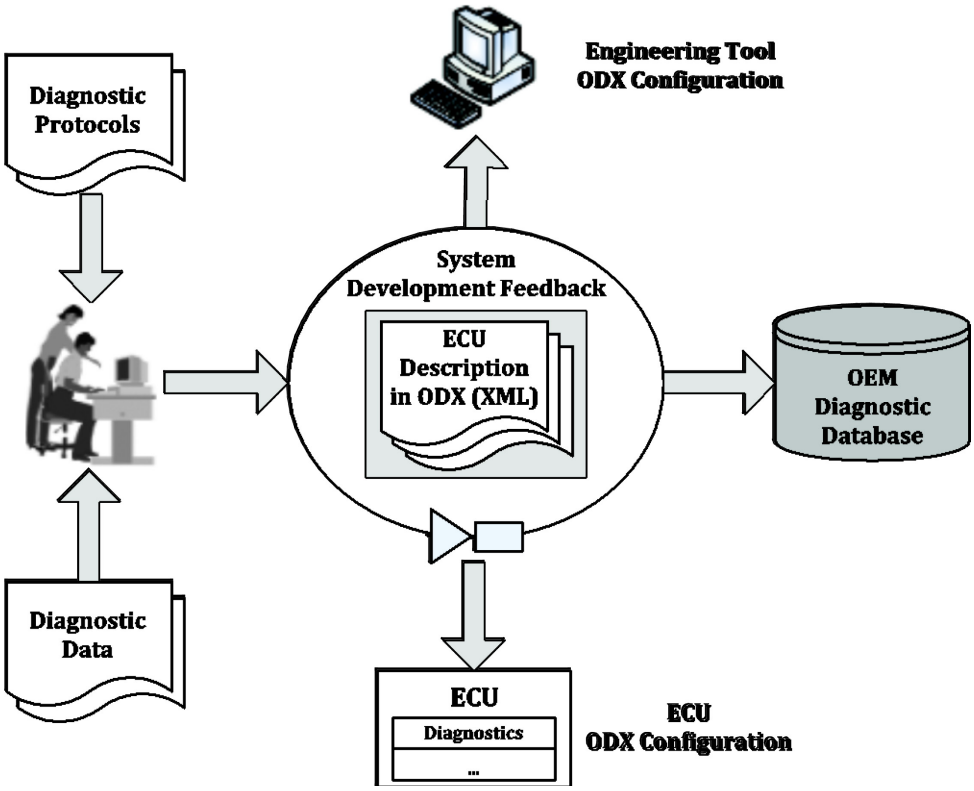


Figure 1 — Example of central source engineering diagnostic data process

5.4 OEM franchised dealer and aftermarket service outlet diagnostic tool support

The service shop uses the modular VCI hardware and software for vehicle diagnosis and enhanced procedural testing. By using the same engineering, manufacturing and service functions as those used for individual ECU testing, the reliability of the data is maintained. A modular VCI protocol module can be used with any PC (handheld or stationary) and can be utilized as an embedded device.

6 Modular VCI software architecture

6.1 Overview

The modular VCI concept is mainly based on three software components (see Figure 2):

— MVCI D-Server software;

© ISO 2022 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

— runtime data based on ODX;

— MVCI protocol module software.

The application accesses the MVCI D-Server through the MVCI D-Server API. The D-Server obtains all required information about the ECU(s) out of the ODX runtime data. Using the ODX runtime data information, the D-Server converts the application's request into a byte stream, which is called a diagnostic protocol data unit (D-PDU). The D-PDU is handed over to the MVCI protocol module through the D-PDU API. The MVCI protocol module transmits the D-PDU to the vehicle's ECU(s). The other way around, the MVCI protocol module receives the vehicle's response(s) and reports the response data to the D-Server. Again, using the ODX runtime data, the D-Server interprets the D-PDU and provides the interpreted symbolic information to the application.

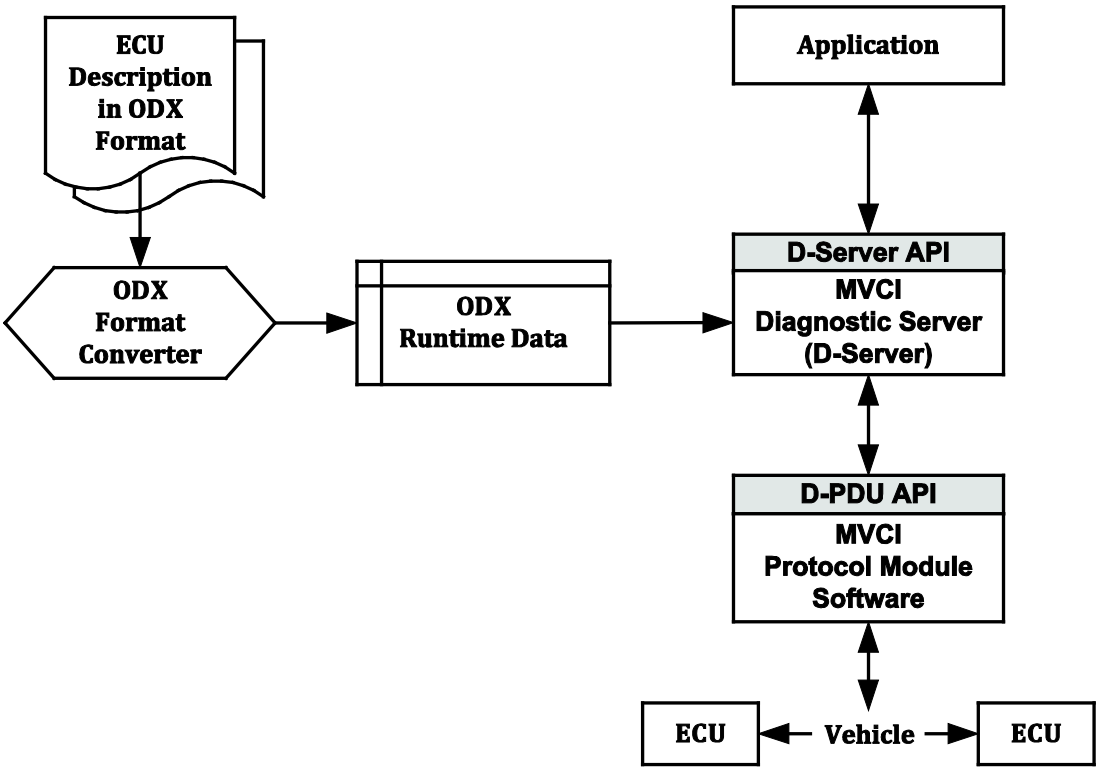


Figure 2 — MVCI software architecture

The grey shading of symbols indicates reference to the following International Standards:

— ODX: ISO 22901-1;

— D-Server API: ISO 22900-3;

— D-PDU API: ISO 22900-2;

— MVCI protocol module: ISO 22900-1.

6.2 Modular VCI D-Server software

The MVCI D-Server is accessible through the MVCI D-Server API. By accessing this API, the application may browse the available features for each ECU and initiate a request towards an ECU using simple symbolic expressions. If the request requires input parameters, they can be specified using symbolic

© ISO 2022 – All rights reserved
Documents Document Sharing Hub
Full document details: BS ISO 22900-2:2022

expressions as well. The MVCI D-server takes the symbolic request, including input parameters, and converts them into a diagnostic request message as defined at the protocol level. The diagnostic request message represents the diagnostic protocol data unit (D-PDU) as passed to the MVCI protocol module through the D-PDU API. Vice versa, the D-Server converts diagnostic response messages as retrieved from the MVCI protocol module back to symbolic information and provides it to the application.

For a detailed description and the complete MVCI D-Server API definition, see ISO 22900-3.

Mapping of D-PDU API and D-Server API is specified in Annex A.

6.3 Runtime format based on ODX

For every conversion from symbolic requests to diagnostic request messages, and vice versa for responses, the MVCI D-Server obtains the required information from the runtime database. The database defines the structure of every diagnostic request and response as supported by an ECU. The database defines byte and bit positions, width and type of every input and output parameter.

Even though the MVCI D-Server obtains its information from a runtime database, the runtime database and format are not specified by the MVCI standard. Instead, the MVCI standard defines an exchange format to import and export the ECU description across OEMs and tool suppliers. The runtime format is left up to the system designers.

The exchange format is called open diagnostic data exchange format (ODX format). For a detailed description, see ISO 22901-1.

Mapping of D-PDU API and ODX is specified in Annex A.

6.4 MVCI protocol module software

The MVCI protocol module is accessible through the D-PDU API. The application issues diagnostic requests through the D-PDU API. The MVCI protocol module takes the request D-PDU and transmits it to the vehicle's ECU(s) according to the vehicle communication protocol. Header type, checksum information and D-PDU segmentation depend on the vehicle communication protocol and shall be handled transparently by the MVCI protocol module. Also, the MVCI protocol module observes the timing between message frames and requests and responses on the physical interfaces. After completion, the MVCI protocol module simply shall deliver the response back to the application or report an error condition.

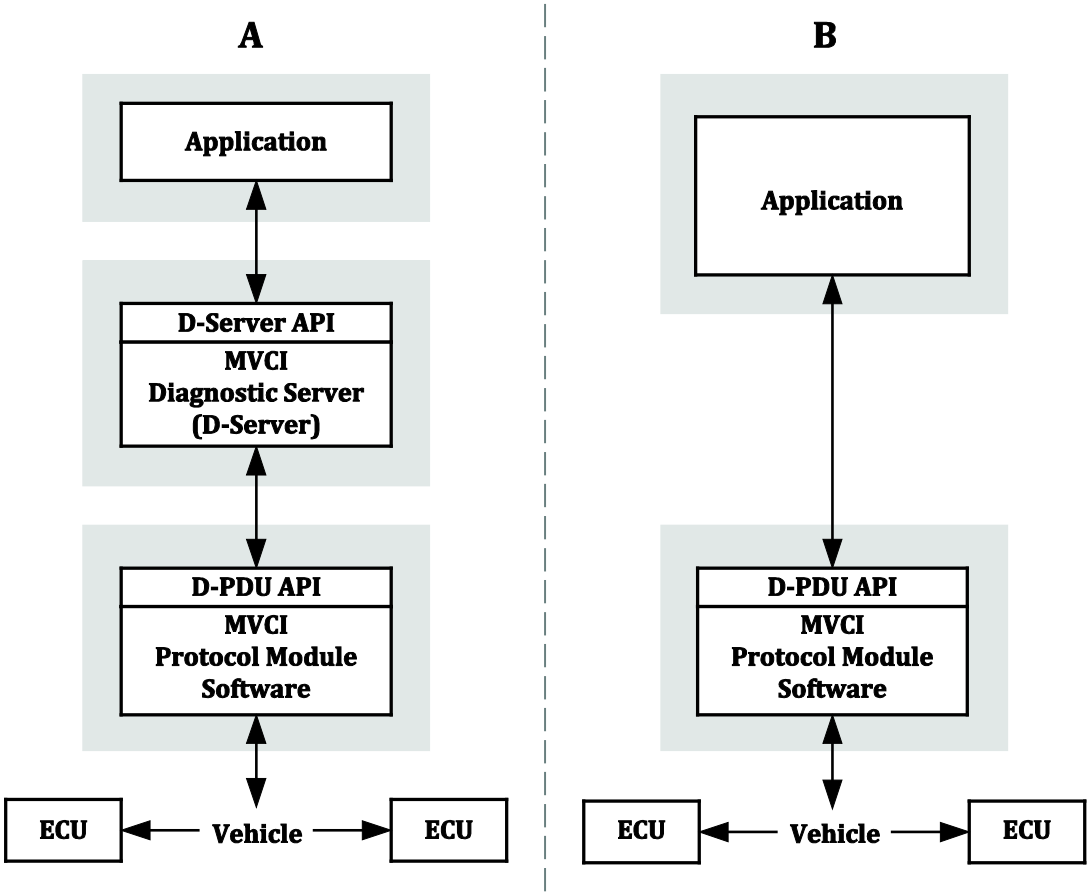
6.5 MVCI protocol module configurations

The D-PDU API and MVCI protocol modules work in many configurations. A MVCI D-Server is not required as the application interface to the D-PDU API.

Figure 3 shows two such configurations to point out the differences.

© ISO 2022 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022



Key

A application with MVCI D-Server

B application without MVCI D-Server

NOTE From the perspective of the D-PDU API, it does not make a difference whether an application accesses the software interface directly or through an MVCI D-Server. Consequently, in this document, the term "application" represents both ways of accessing the D-PDU API.

Figure 3 — MVCI configurations

7 D-PDU API use cases

7.1 Overview

The MVCI protocol module is the key component to exchange implementations of diagnostic protocols among OEMs and tool suppliers without re-engineering already implemented software. By relying on the D-PDU API, the application may easily access other or additional MVCI protocol module implementations. In a similar way to existing standards such as SAE J2534-1 and RP1210a, applications compliant to the MVCI standard are basically independent of the underlying device as long as the required physical interface is supported and no tool supplier specific feature is required.

Even though the D-PDU API extends the capabilities beyond the definitions of SAE J2534-1 and RP1210a, the existing standards and their related devices and applications do not become obsolete by introducing the D-PDU API. Instead, the transition and co-existence of all standards are facilitated to save development and investment costs. The definition of the D-PDU API is closely related to SAE J2534-1 and RP1210a to allow mapping of functionality in both directions. However, it extends their definitions to cover the full width of enhanced diagnostics.

© ISO 2022 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

The fulfilment of the following use cases is crucial for the inter-exchange of protocol module implementations according to MVCI, SAE J2534-1 and RP1210a.

NOTE In the use case figures (see Figures 4 to 6), the grey boxes suggest a specific software component architecture. This representation is not intended to be construed as the only possible architectural solution. Depending on the situation, there can be more software components or fewer software components.

7.2 Use case 1: Single MVCI protocol module

The single MVCI protocol module configuration (see Figure 4) is the simplest configuration where the D-PDU API implementation and the MVCI protocol module hardware are obtained from the same vendor. The application will access the single MVCI protocol module through a single D-PDU API. Parallel access onto multiple D-PDU APIs is not required. Resource handling is completely covered inside the D-PDU API implementation.

This use case applies to basically all stand-alone MVCI protocol module device configurations.

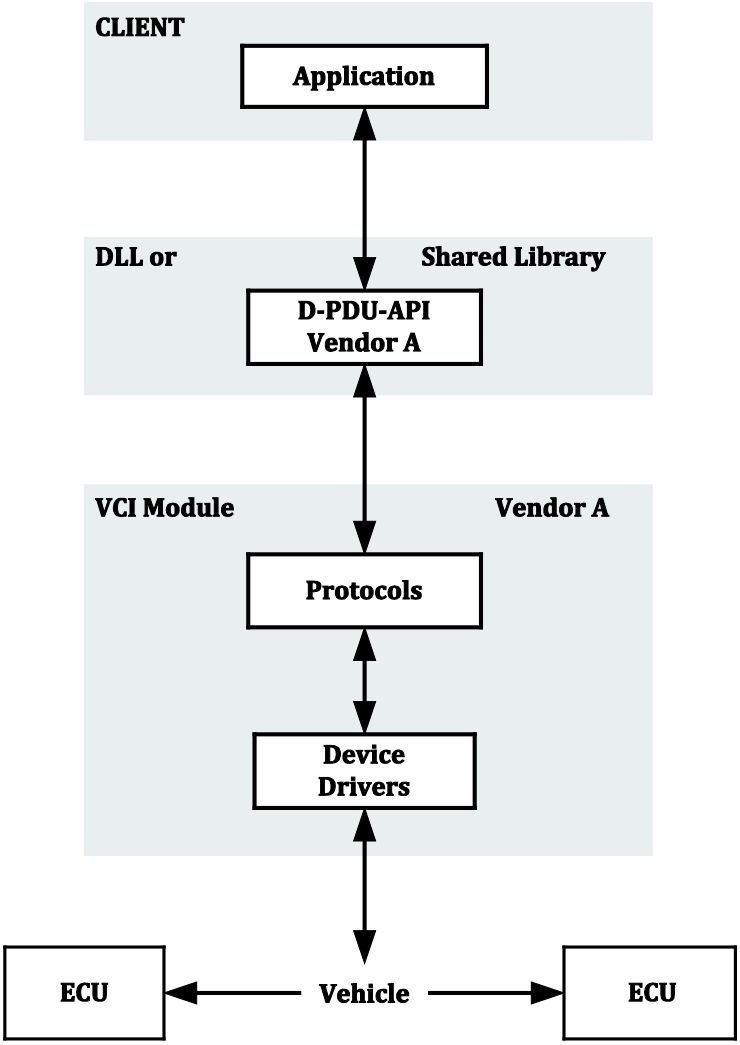


Figure 4 — MVCI configuration with single MVCI protocol module

© ISO 2022 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

7.3 Use case 2: Multiple MVCI protocol modules supported by same D-PDU API implementation

There are different configurations with multiple MVCI protocol modules. In this use case, a D-PDU API implementation may support more than one MVCI protocol module at a time, where both modules and D-PDU API implementations are from a single vendor (see Figure 5). The application will access the MVCI protocol modules through a single D-PDU API. Parallel access onto multiple D-PDU APIs is not required. However, the application may access and operate the MVCI protocol modules at the same time in parallel if the D-PDU API implementation provides the capabilities. Resource handling is completely covered inside the D-PDU API implementation.

This use case applies to MVCI protocol module device configurations where the vendor integrates support for multiple MVCI protocol modules into one software package.

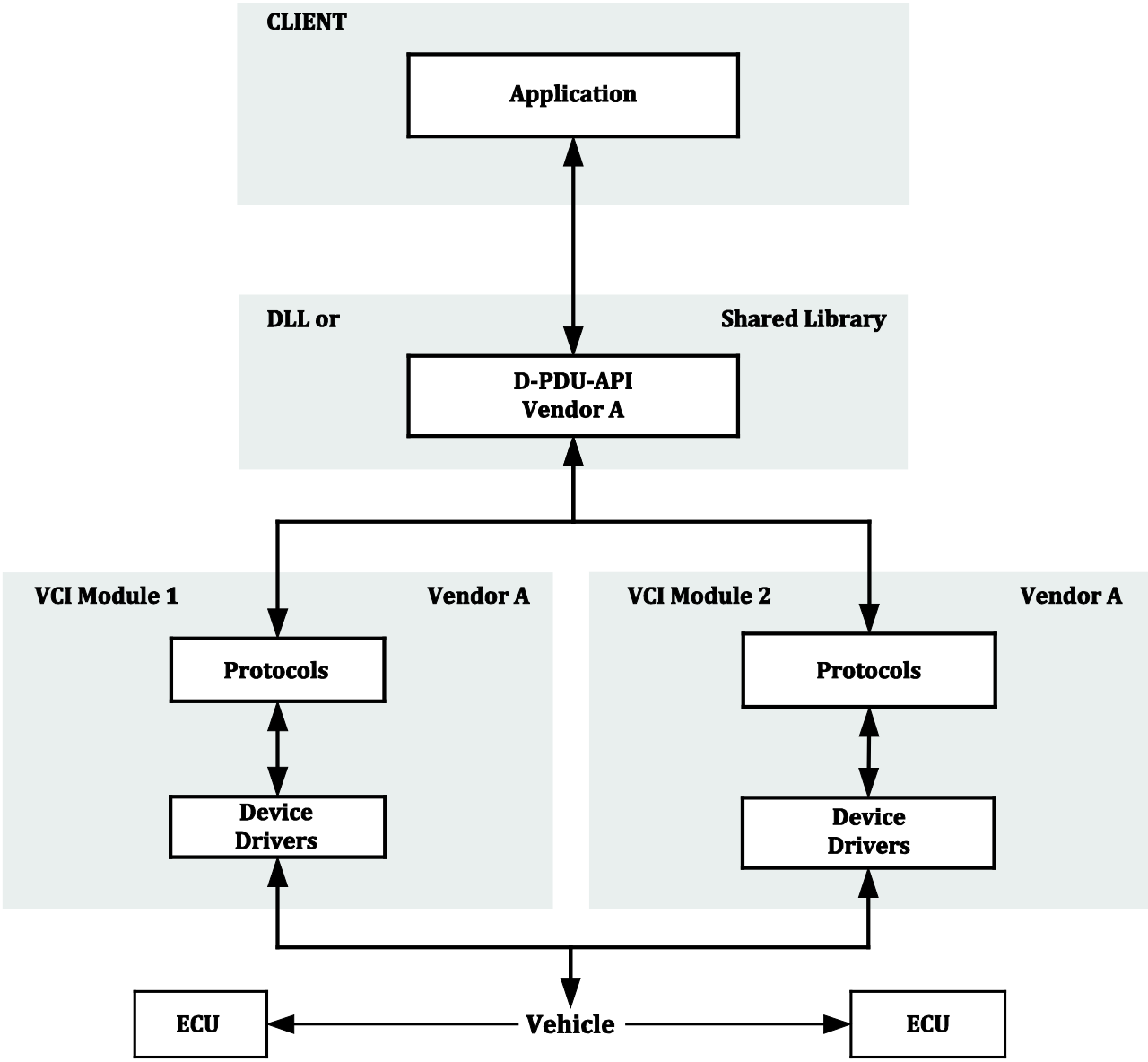


Figure 5 — Multiple MVCI protocol modules supported by same D-PDU API implementation

© ISO 2022 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

7.4 Use case 3: Multiple MVCI protocol modules supported by different D-PDU API implementations

In most cases, when combining MVCI protocol modules of different suppliers, the MVCI protocol modules are not accessed through the same D-PDU API implementation (see Figure 6). Neither of the implementations knows about the other suppliers' MVCI protocol modules. It cannot communicate with them, since the D-PDU API does not define an explicit interface hardware type, nor the communication protocol on the interface. Therefore, MVCI protocol modules of different suppliers will be addressed through separate D-PDU API implementations. Each D-PDU API implementation may support more than one MVCI protocol module at a time and more than one D-PDU API implementation may co-exist on the same runtime environment at the same time.

The application may access multiple D-PDU APIs (and their MVCI protocol modules) in parallel, if it needs to use resources of more than one D-PDU API. As a result, each D-PDU API implementation shall be able to run concurrently with other D-PDU API implementations. As in use case 2, resource handling is completely covered inside the D-PDU API implementation with respect to one implementation. As use case 3 assumes multiple D-PDU API implementations not knowing each other, the application is required to handle the resources across D-PDU API implementations.

This use case applies to MVCI protocol module device configurations where a tool supplier integrates support for multiple MVCI protocol modules of different vendors into one software package.

NOTE This use-case may not be fulfilled as, for example, for DoIP, it is impossible to get several implementations as ports are pre-defined in ISO 13400-2. It is up to the integrator to define the use-cases to be supported.

© ISO 2022 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

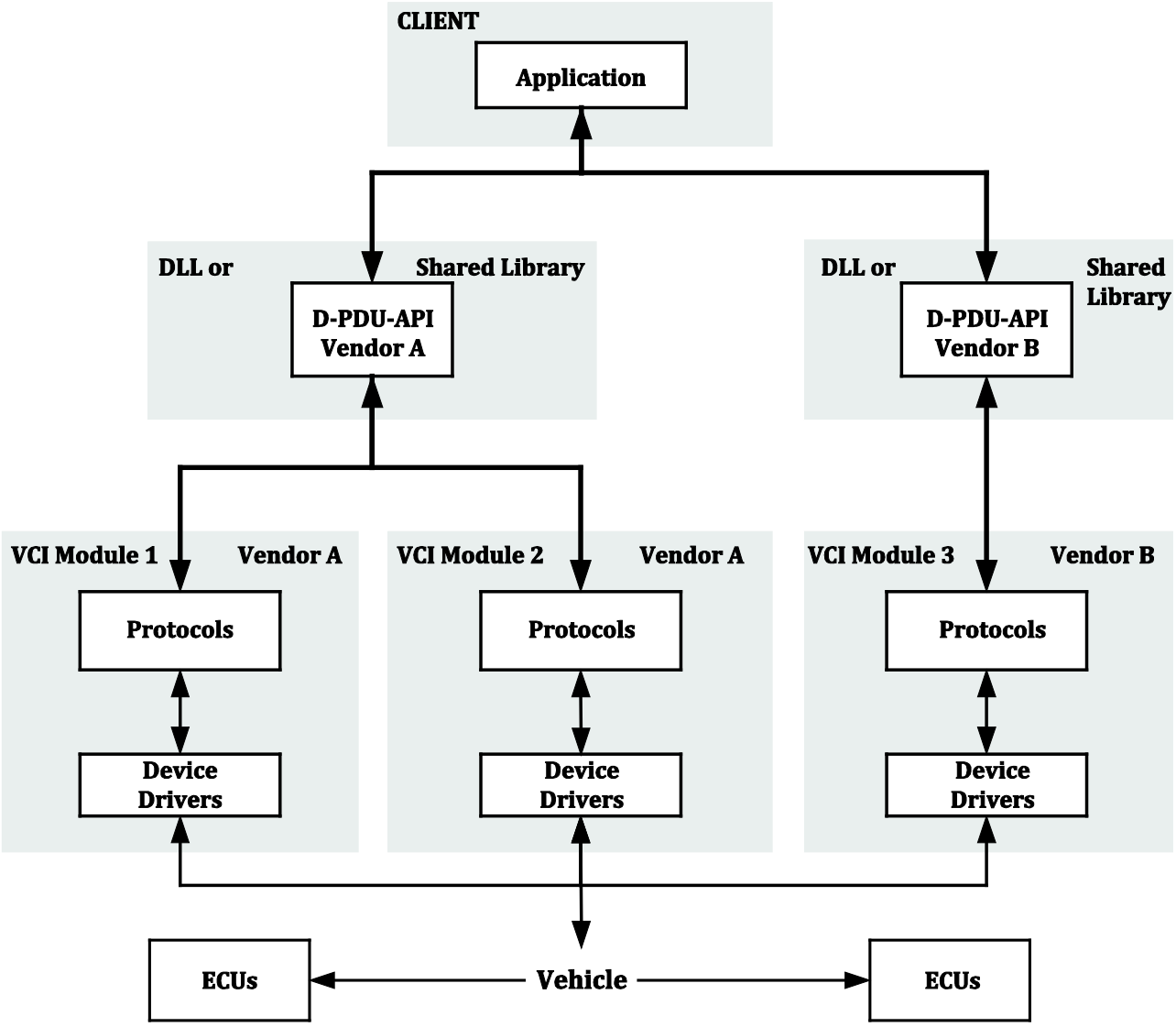


Figure 6 — Multiple MVCI protocol modules supported by different D-PDU API implementations

8 Diagnostic protocol data unit (D-PDU) API

8.1 Software requirements

8.1.1 General requirements

The MVCI devices shall be accessed through dynamically linkable software modules, i.e. dynamic link libraries for Windows systems (DLLs) and/or library modules for Linux systems. The linkable software module will be referred to as the D-PDU API implementation. It queries the available MVCI protocol modules, takes care of device identification (e.g. reading firmware version, etc.) and low-level communication with all MVCI protocol modules supported by the respective library. However, the device query and identification does not contain any functionality of the system level driver, as it is required for USB, Ethernet and wireless communication. The system low-level driver is responsible for the detection and enumeration of the device interface. The system level driver is considered a part of the delivery from the MVCI protocol module supplier, but its interface is proprietary and not part of this document.

© ISO 2022 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

Multiple MVCI protocol modules may be separately accessible by the application using the same D-PDU API implementation. This could typically apply to MVCI protocol module variants from a single supplier. However, MVCI protocol modules may also be accessible through separate D-PDU API implementations. Such cases would typically occur when combining multiple MVCI protocol modules from different suppliers.

In order to declare the capabilities of a D-PDU API implementation, a MVCI protocol module vendor shall provide a module description file (MDF) in XML. The MDF shall contain all information about supported MVCI protocol module types, bus types, protocols and communication links, as well as all related information regarding parameters and I/O controls. The application may parse the file for resources and use them dynamically. It may also make use of static information. In the latter case, the application developer could create a C header file containing and statically matching all necessary resource IDs. For a detailed description and structure of the files, see F.2.

All API functions return a SNUM32 value representing the function result.

The D-PDU API implementation shall not be restricted to a dedicated operating system or programming language and shall be portable. However, for unique and clear definition, C was chosen as the programming language to describe the API.

In general, the D-PDU API implementation shall be made available as a dynamically linkable software module independent of the target operating system. The approach of a separate software module guarantees easy exchange. However, in some cases, it may be useful or more appropriate to link the software module statically. Those cases are considered as proprietary solutions and shall not be the main target of this document.

The D-PDU API implementation shall support, at a minimum, single clients and asynchronous, multi-thread operation. Multi-client support is not a requirement, but may be offered as an additional feature by an MVCI protocol module vendor. A multi-client implementation shall support multiple sessions and links in parallel. For every communication link, the implementation shall take care of queuing communication requests.

The D-PDU API implementation shall cover full duplex and event-driven communication, enabling coverage of advanced vehicle communication principles (e.g. response on event, periodic transmission, etc.).

8.1.2 Vehicle protocol requirements

The D-PDU API functions shall be protocol independent. Since protocol standards have frequently changed in the past and new protocols will be released in the future, the D-PDU API needs to be flexible and generic enough to easily cover additional protocols not taken into account at time of definition. In order to provide the application, the capability to use any protocol and any of its tool supplier specific implementations, all protocol-related ComParams shall be documented in a standardized and generic manner. The documentation is stored in a module description file (MDF) in XML and shall be provided by the MVCI protocol module supplier.

In general, there is no minimum set of protocols with respect to the D-PDU API. However, in order to provide the required SAE J2534-1 and RP1210a compatibility, and to avoid interference of D-PDU API implementations of different suppliers, minimal protocol naming conventions are necessary.

Current standardized protocols are specified in Annex B, Annex J specifies D-PDU API usage for emission-related diagnostics, Annex K specifies the implementation for DoIP and Annex L for ISOBUS protocol.

Documents
Document Sharing Hub

© ISO 2022 – All rights reserved

Full document details: BS ISO 22900-2:2022

8.1.3 Timing requirements for protocol handler messages

There can be unexpected results if two requests are made to the same ECU "simultaneously". Most ECUs will ignore the second request, but some ECUs will ignore both requests. As a result, the protocol handler shall properly serialize requests across multiple CLLs, while still allowing valid parallel communication.

It should also be emphasized that for the cases where CLLs are sharing a physical serial bus, all timing requirements (CP_P3Min, CP_P3Phys and CP_P3Func) shall be satisfied before subsequent transmissions can occur (see I.7).

Message serialization is also required for some complex single CLL scenarios. Consider the case where TesterPresent messages, functionally addressed ComPrimitives and physically addressed ComPrimitives are all occurring on one bus. Message serialization may be required to ensure that the protocol handler adheres to proper inter-transmit timing and receive timing. All standard protocols have their timing individually defined using ComParams. This allows for minor changes in the timing behaviour of a protocol in order to satisfy the unique attributes of an installed vehicle ECU.

For certain messages, the ability to be able to send the request immediately is desirable. One such message would be a TesterPresent where it is undesirable that they are prevented from being sent by a previous request taking too long. This can happen when a request is extended by a 0x78 pending response. It is possible to send these messages so long as sending them does not cause undesirable effects. Table 1 shows the possible scenarios where requests can be sent immediately following a previous request and when the previous request shall be allowed to complete or timing applied.

Timing between message requests on the serial bus is the responsibility of the MVCI protocol handlers. The MVCI protocol module shall ensure that the inter-message timing is correctly managed (see Table 1).

Table 1 — Functional and physical address handling

Previous request type	Current request type	Handling in the tester
Physical with response	physical with or without response	Wait until the completion of the previous physical request (positive response or negative response other than 0x78) before transmitting a new physical request. There is no timing applied to this rule, the tester can transmit the physically addressed request immediately after the previous physically addressed service has been finished (final response received).
	functional with response	Wait until the completion of the previous physical request (positive response or negative response other than 0x78) before transmitting the functional request that requires a response. There is no timing applied to this rule, the tester can transmit the functionally addressed request immediately after the previous physically addressed service has been finished (final response received).
	functional without response	The tester is allowed to transmit the functional request that does not require a response immediately (e.g. any physical request followed by a functional TesterPresent). NOTE For physical cyclic responses, the tester waits for the first response before sending a functional TesterPresent.
Physical without response	physical with or without response	Wait CP_P3Phys after the completion of the previous physical request without response before transmitting a new physical request.
	functional with response	Wait CP_P3Phys after the completion of the previous physical request without response before transmitting a new functional request.
	functional without response	The tester is allowed to transmit the functional request that does not require a response immediately (e.g. any physical request followed by a functional TesterPresent).

© ISO 2022 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

Previous request type	Current request type	Handling in the tester
Functional with response	physical with or without response	Wait until the completion of the previous functional request that requires a response (positive response or negative response other than 0x78) before transmitting the physical request. There is no timing applied to this rule, the tester can transmit the physically addressed request immediately after the previous functionally addressed service has been finished (final response received).
	functional with or without response	Wait until the completion of the previous functional request that requires a response (positive response or negative response other than 0x78) before transmitting the next functional request that does not require a response. There is no timing applied to this rule, the tester can transmit the functionally addressed request immediately after the previous functionally addressed service has been finished (final response received).
Functional without response	physical with or without response	The tester is allowed to transmit the physical request immediately (e.g. functional TesterPresent followed by any physical request).
	functional with or without response	Wait CP_P3Func after the transmission of the functional request that does not require a response before transmitting the functional request that does or does not require a response.

8.1.4 Serialization requirements for protocol handler messages

Many vehicle serial bus protocols require serialization of messages sent on the bus, e.g. sending functional and physical messages on a shared physical serial bus can be accomplished if the physically addressed ECU is not in the functional group (i.e. a group of ECUs which can be addressed with the same functional address). For this case, the CLLs share the ECU being addressed and the messages/frames need to be serialized. See Figure 7.

© ISO 2022 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

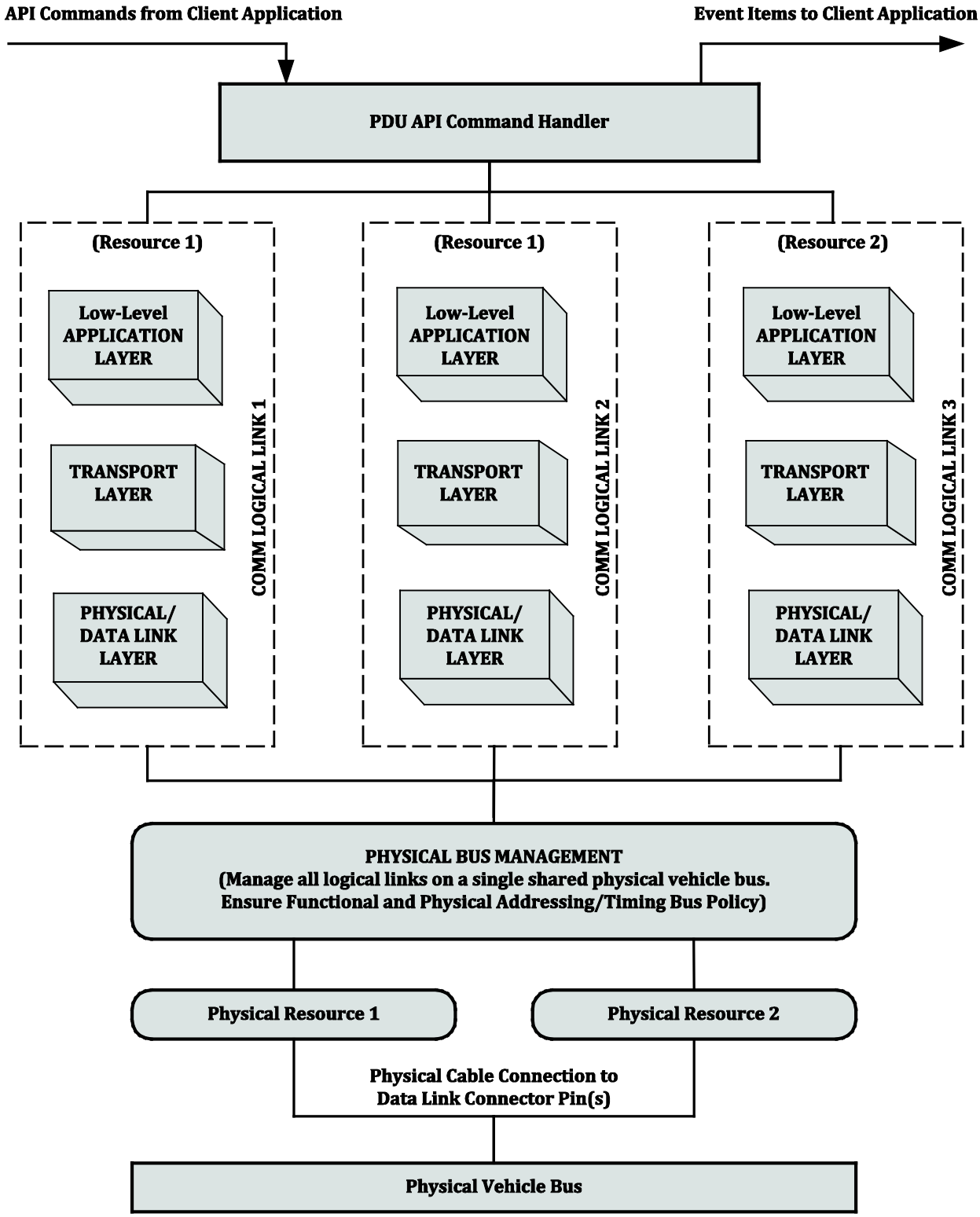


Figure 7 — CLLs sharing physical bus with message serialization example

© ISO 2022 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

8.1.5 Compatibility requirements

8.1.6 Timestamp requirements

8.1.6.1 General

This subclause describes the requirements of the timestamp mechanism that shall be used for the D-PDU API.

The unit of all timestamps is the microsecond and is defined in a 32-bit value. The granularity of the timestamp is limited by the capability of the device. The time base is reset to zero within the PDU_IOCTL_RESET function and after boot-up. All of the logical links, events and errors of one device derive their timestamps from the same time base.

The D-PDU API does not have any mechanism to detect a timestamp overflow. The application shall take care of an overflow.

8.1.6.2 Timestamp for transmitted messages

For all UART-based protocols, the timestamp will be taken at the end (last bit) of the message. In the case of a controller-based protocol, the timestamp will be taken when the controller indicates the successful transmission of the message or the last frame of a message.

8.1.6.3 Timestamp for received messages

For all UART-based protocols, the timestamp will be taken at the end (last bit) of the message. In the case of a controller-based protocol, the timestamp will be taken when the controller indicates the successful reception of the message or the last frame of a message. Indication of the start of a message or first frame is handled as an event and is described below.

8.1.6.4 Timestamp for events, errors and indications

For events and errors, the timestamp will be taken when the event or error is detected.

8.1.6.5 Timestamp for start of message indication

The indication of the start of message (see Table D.5) will be handled as outlined below.

— For UART-based protocols, the timestamp will be taken at the first bit of the received message. In this case, the timestamp shall be calculated (see example below).

EXAMPLE To calculate the timestamp for the first bit of a 9 600 baud line running 8N1, the formula would be:

$$b_{1,\mathrm{ts}} = B_{1,\mathrm{ts}} - \left[\left(\frac{1}{9\,600}\right) \times 10\right]$$

where

$b_{1,\mathrm{ts}}$ is the first bit time stamp;

$B_{1,\mathrm{ts}}$ is the first byte time stamp;

1 is the time, in s;

9 600 is the baudrate in bit/s;

10 is the number of bits of 8N1UART configuration.

© ISO 2022 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

— For controller-based protocols, the timestamp will be taken when the controller indicates the reception of the first frame of the message.

8.2 API function overview and communication principles

8.2.1 Terms used within the D-PDU API

8.2.1.1 Resource

A resource defines a communication channel towards a single ECU or towards multiple ECUs. It covers diagnostic protocols, and MVCI protocol module hardware (transceivers, UART, multiplexer, etc.), including the Diagnostic Link Connector (DLC) of the MVCI protocol module.

8.2.1.2 ComLogicalLink

A ComLogicalLink (CLL) defines a logical communication channel towards a single ECU or towards multiple ECUs. The configuration of the CLL is based on the selected bustype, pins and protocol. A CLL can be created on an existing and available resource. The protocol for the CLL is configured using ComParams. The D-PDU API is not restricted in the number of CLLs opened on a single resource, unless there are limitations for the protocol. The client application shall be aware of conflicts on resources.

8.2.1.3 ComPrimitive

A ComPrimitive (CoP) is a basic communication element holding data and controlling the exchange of data between the D-PDU API implementation and the ECU.

8.2.1.4 ComParam

A ComParam is a protocol communication parameter used to define the functionality of the vehicle communication protocol selected for a ComLogicalLink. Each protocol has a set of applicable ComParams that are set to a default value upon creation of a ComLogicalLink. A set of ComParams is used to individually define communication to a single ECU or a functional group of ECUs.

Annex B specifies ComParams and protocols for standard protocols while Annex C provides guidelines for vendor-specific ComParams and protocols.

8.2.2 Function overview

Table 2 lists all D-PDU API functions and classifies them in functional groups.

Table 2 — D-PDU API functions overview

Function	Description
General functions	
PDUConstruct	Constructor with optional, manufacturer-specific arguments. Required to call for each D-PDU API implementation prior to any other D-PDU API function call.
PDUDestruct	Destructor required as last D-PDU API function call to free all resources.
PDUModuleConnect	Connects to a specific MVCI protocol module.
PDUModuleDisconnect	Disconnects from a specific MVCI protocol module.
PDURegisterEventCallback	Registers or unregisters a callback function for event notification.
PDUIoCtl	Invokes I/O control functions of an MVCI protocol module or ComLogicalLink.

© ISO 2022 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

Function	Description
Information functions	
PDUGetVersion	Obtains version information for a specified MVCI protocol module and its D-PDU API implementation.
PDUGetStatus	Obtains runtime information (e.g. status, life sign, etc.) from an MVCI protocol module, ComLogicalLink or ComPrimitive.
PDUGetLastError	Obtains the code for the last error that occurred in an MVCI protocol module or ComLogicalLink.
PDUGetTimestamp	Obtains the Timestamp information for a specific MVCI protocol module.
PDUGetObjectId	Obtains an ID for a given short name for a PDUObjectType. This is in addition to the possibility of parsing the MDF file.
Resource management	
PDUGetResourceIds	Obtains all resource IDs that match the requested resource information.
PDUGetResourceStatus	Obtains the status of the requested resource ID.
PDUGetConflictingResources	Obtains a list of resource IDs that are in conflict with the given resource ID (e.g. two resources sharing the same physical controller).
PDUGetModuleIds	Obtains the IDs of all MVCI protocol modules currently connected to the D-PDU API.
PDULockResource	Obtains a lock on the requested resource ID.
PDUUnLockResource	Releases the lock on the requested resource ID.
ComLogicalLink handling	
PDUCreateComLogicalLink	Creates a ComLogicalLink for a given D-PDUResource.
PDUDestroyComLogicalLink	Destroys a ComLogicalLink.
PDUConnect	Physically connects the previously created ComLogicalLink to the communication line.
PDUDisconnect	Physically disconnects the previously connected ComLogicalLink from the communication line.
Link and ComParam handling	
PDUGetComParam	Obtains current value of specified ComParam.
PDUSetComParam	Sets specified ComParam to given value. Overwrites previous values.
PDUGetUniqueRespIdTable	Obtains a table of unique response identifiers. Each unique response identifier is associated with a set of ComParams used to uniquely define an ECU response.
PDUSetUniqueRespIdTable	Sets a table of unique response identifiers. Each unique response identifier value is assigned by the application.
Message handling	
PDUStartComPrimitive	Starts the operation of given ComPrimitive (e.g. sending/receiving data).
PDUCancelComPrimitive	Cancels current execution of given ComPrimitive (e.g. cancel a running periodic send operation or an operation which has not yet been executed).
PDUGetEventItem	Retrieves item data (e.g. received data) for given event source (i.e. MVCI protocol module, ComLogicalLink and ComPrimitive).
PDUDestroyItem	Destroys given item.

8.2.3 General usage

The sequences of function calls differ between the D-PDU API and existing APIs like for SAE J2534-1 and RP1210a. Therefore, the following subclauses show a minimal sequence of calls in order to facilitate understanding. Also, the terms “asynchronous” and “synchronous” communication are defined in the subsequent subclauses.

Documents
Document Sharing Hub

© ISO 2022 – All rights reserved

Full document details: BS ISO 22900-2:2022

In order to initialize the D-PDU API implementation and to prepare communication for one channel, the application needs to go through the following minimal sequence of API function calls.

Table 3 lists the D-PDU API functions in sequential order to facilitate better understanding.

Table 3 — General usage of D-PDU API function calls

Action #	Function call from application	D-PDU API
Initial connection to D-PDU API library		
1	PDUConstruct	Initialize D-PDU API library. This does not necessarily connect to an MVCI protocol module. This begins the process of figuring out what MVCI protocol modules are available. Information about the available modules is retrieved via the PDUGetModuleIds function.
	PDUGetModuleIds	Retrieve the list of available MVCI protocol modules and their handles and modules types.
	PDURegisterEventCallback	(optional) Register global system callback.
Initial connection to an MVCI protocol module		
2	PDUModuleConnect	Connect the D-PDU API library to one or more MVCI protocol modules.
	PDURegisterEventCallback	(optional) Register module callback.
Setting up a ComLogicalLink		
3	PDUCreateComLogicalLink	Create a ComLogicalLink (based on protocol, pins and Bustype).
	PDURegisterEventCallback	(optional) Register ComLogicalLink callback.
	PDUGetUniqueRespIdTable	Retrieve the applicable set of ComParams for unique identification of ECUs determined by the specific protocol for the ComLogicalLink.
	PDUGetComParam	Get ComParam item structures initialized with the default ComParams for the selected protocol.
	PDUSetComParam	Set the ComParams for the ComLogicalLink.
	PDUSetUniqueRespIdTable	Configure the UniqueResponseIdentifierTable for all possible ECU Responses to be received on the ComLogicalLink.
	PDUConnect	Connect the ComLogicalLink to the communication line.
Starting vehicle communications on a ComLogicalLink		
4	PDUStartComPrimitive (PDU_COPT_STARTCOMM or PDU_COPT_SENDRECV)	A PDU_COPT_STARTCOMM ComPrimitive is required for certain protocols as the first ComPrimitive (e.g. ISO 14230). A PDU_COPT_STARTCOMM ComPrimitive is also used to control the ability to start sending TesterPresent messages (see CP_TesterPresentHandling). Once TesterPresent handling is enabled, the message is sent immediately, prior to the initial TesterPresent cyclic time (CP_TesterPresentTime). For all other vehicle bus communications, a PDU_COPT_SENDRECV ComPrimitive is used to begin vehicle communications.

© ISO 2022 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

Action #	Function call from application	D-PDU API
Using a ComLogicalLink for vehicle communications		
5	PDUStartComPrimitive (PDU_COPT_SENDRECV) PDUSetComParam PDUGetEventItem PDUDestroyItem	Set up and start ComPrimitives for vehicle bus activity: Send only, Send Receive, or Receive only. Change ComParams (temporary changes per ComPrimitive or permanent changes per PDU_COPT_UPDATEPARAM type of ComPrimitive). Retrieve event items. Destroy event item memory from D-PDU API memory.
Stopping vehicle communications on a ComLogicalLink		
6	PDUStartComPrimitive (PDU_COPT_STOPCOMM)	A PDU_COPT_STOPCOMM ComPrimitive will stop all communication on a ComLogicalLink.
Connecting to a newly available MVCI protocol module		
7	Receive callback indicating new MVCI protocol module detected by D-PDU API. PDUGetModuleIds PDUModuleConnect PDURegisterEventCallback	Receive a system event callback with an information type PDU_IT_INFO indicating the list of modules has changed: PDU_INFO_MODULE_LIST_CHG Retrieve the new list of available MVCI protocol modules and their handles and modules types. Any previously detected MVCI protocol modules will return the same hMod handles. NOTE 1 If detection of an MVCI protocol module was lost and then detection is re-established, the module handle (hMod) will not be the same as the previous handle. This ensures that any event items stored in the initial connection will still be available for reading prior to a PDUModuleDisconnect. Connect to the new MVCI protocol module. (optional) Register module callback.
Loss of communication to an MVCI protocol module and reconnection		
8	Receive callback indicating communications to an MVCI protocol module has been lost.	Receive a module event callback indicating communication is lost to an MVCI protocol module. (See PDU_ERR_EVT_LOST_COMM_TO_VCI). The hMod information is part of the callback. The client application should call PDUModuleDisconnect after all items have been retrieved from the module event queue. NOTE 2 All ComPrimitives active for all ComLogicalLinks on the module will generate a PDU_COPST_CANCELLED status event item. All ComLogicalLinks on the module will generate a PDU_CLLST_OFFLINE status event item. NOTE 3 If the MVCI protocol module still maintains power after it has lost communication with a client session, it disconnects from all vehicle buses associated with that client session and self cleans up all resources associated with the client session. NOTE 4 The module handle (hMod) is preserved until a PDUModuleDisconnect is called.

© ISO 2022 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

Action #	Function call from application	D-PDU API
	PDUModuleDisconnect	Once PDUModuleDisconnect is called, the hMod handle is no longer valid, all event items in the queues are freed and any related module memory reserved by the D-PDU API library is unreserved.
	Wait for a Callback indicating status change of MVCI protocol module.	Receive a system event callback with an information type PDU_IT_INFO indicating the list of modules has changed: PDU_INFO_MODULE_LIST_CHG.
	PDUGetModuleIds	Retrieve the new list of available MVCI protocol modules and their handles and modules types. Observe that the hMod from the MVCI Protocol module that had a communication loss is no longer listed as an available hMod. NOTE 5 See action 7 (above) for steps to reconnect to a module after a loss of communications event.
Disconnect from a MVCI protocol module		
9	PDUDisconnect	Disconnect the ComLogicalLink from the communication line.
	PDUDestroyComLogicalLink	Destroy the ComLogicalLink.
	PDURegisterEventCallback(NULL)	(optional) Unregister the module event callback. From now on no events will be signalled to the application for the module.
	PDUModuleDisconnect	Disconnect a specific MVCI protocol module from the D-PDU API library and free all reserved memory resources.
Disconnection from the D-PDU API library		
10	PDUDestruct	De-initialize the D-PDU API and destroy/free any internal resources.

8.2.4 Asynchronous and synchronous communication

8.2.4.1 General

The asynchronous communication operation of the D-PDU API implies that calls to the API are immediately returned even though the requested activity might still be running or is still waiting for execution inside the D-PDU API implementation. The D-PDU API uses asynchronous calls to support all types of vehicle communication requirements (e.g. non-cyclic, cyclic, event driven communication, etc.), as well as status changes and error detection events.

The synchronous communication operation of the D-PDU API implies that calls to the API are returned with the requested information to the application (e.g. PDUGetComParam).

The D-PDU API functions allow the application to use both event callback (asynchronous) and polled (synchronous) communication principles to exchange data with the D-PDU API.

8.2.4.2 Event callback (asynchronous mode)

In this case, the communication between application and the D-PDU API implementation is completely event driven. The application may register an application-specific event callback function by calling PDURegisterEventCallback. Any events queued into an empty ComLogicalLink's event queue or the events that are already queued at the point in time the callback function is registered will cause the callback function to be invoked. The callback function will be called on the thread of the D-PDU API. It is the responsibility of the application to minimize the time spent in this callback. This specification suggests that the application callback function post an event to wake another thread to do the reading of the event data. If the application shall make a D-PDU API function call in the callback routine, then PDUGetEventItem is the only permitted call.

© ISO 2022 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

Additional technical information concerning callback are specified in Annex E.

8.2.4.3 Polling (synchronous mode)

In this case, the application does not make use of the event callback mechanism. The application initiates the D-PDU API functions (just as in asynchronous mode) and uses the PDUGetStatus and PDUGetEventItem functions to detect status changes and to read event items from the event queues.

8.2.5 Usage of resource locking and resource unlocking

A ComLogicalLink has the ability to lock different elements of a physical resource it is connected to using the D-PDU API function PDULockResource. Through locking, a ComLogicalLink can prohibit all other ComLogicalLinks access to the resource if necessary. For example, if there are two ComLogicalLinks sharing a vehicle bus, and an ECU needs to be reprogrammed, the physical resource (the vehicle bus) can be locked during the reprogramming sequence. The API function PDUUnlockResource is used to release the lock on a physical resource.

8.2.6 Usage of ComPrimitives

8.2.6.1 ComPrimitive overview

To provide a generic data exchange mechanism for different communication principles, several ComPrimitive types are specified. The behaviour and usage of each ComPrimitive type depends on the specific communication protocol used with a ComLogicalLink. These issues shall be described for each specific communication protocol implementation.

Table 4 describes the different ComPrimitive types.

Table 4 — ComPrimitives overview

ComPrimitive type	Description
PDU_COPT_STARTCOMM	Start communication with ECU by sending an optional request. The detailed behaviour is protocol dependent. For certain protocols (e.g. ISO 14230), this ComPrimitive is required as the first ComPrimitive. This ComPrimitive is also required to put the ComLogicalLink into the state PDU_CLLST_COMM_STARTED which allows for TesterPresent messages to be enabled (see CP_TesterPresentHandling). Once TesterPresent handling is enabled, the message is sent immediately, prior to the initial TesterPresent cyclic time (CP_TesterPresentTime).
PDU_COPT_STOPCOMM	Stop communication with ECU by sending an optional request. The detailed behaviour is protocol dependent. After successful completion of this ComPrimitive type, the ComLogicalLink is placed into PDU_CLLST_ONLINE state and no further TesterPresent messages will be sent. A PDU_COPT_STARTCOMM ComPrimitive might be required by some protocols (e.g. ISO 14230) to begin communications again.
PDU_COPT_UPDATEPARAM	Copies ComParams and the UniqueRespIdTable related to a ComLogicalLink from the working buffer to the active buffer. Prior to update, the values need to be passed to the D-PDU API by calling PDUSetComParam and/or PDUSetUniqueRespIdTable, which modifies the working buffer. NOTE 1 If the CLL is in the PDU_CLLST_COMM_STARTED state and TesterPresent handling is enabled (see CP_TesterPresentHandling), any changes to one of the TesterPresent ComParams will cause the TesterPresent message to be sent immediately, prior to the initial TesterPresent cyclic time. NOTE 2 The protocol handler always waits the proper P3Min time before allowing any transmit. See CP_P3Min, CP_P3Func, CP_P3Phys.

© ISO 2022 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

ComPrimitive type	Description
PDU_COPT_SENDRECV	Send request data and/or receive corresponding response data (single or multiple responses). See 10.1.4.17 for detailed settings.
PDU_COPT_DELAY	Wait the given time span before executing the next ComPrimitive.
PDU_COPT_RESTORE_PARAM	Converse functionality of PDU_COPT_UPDATEPARAM. Copies ComParams related to a ComLogicalLink from active buffer to working buffer.

8.2.6.2 ComPrimitive send/receive cycle overview

8.2.6.2.1 General

Each ComPrimitive is controlled by a PDU_COP_CTRL_DATA structure (see 10.1.4.17). Table 5 describes how the send and receive cycles are used. For more examples, see also Figures 8 to 16.

Table 5 — ComPrimitives send/receive cycles

PDU_COPT_SENDRECV, PDU_COPT_STARTCOMM, PDU_COPT_STOPCOMM		
	NumSendCycles	**NumReceiveCycles**
SEND and RECEIVE	Number of send cycles to be performed; −1 for infinite send operation.	Number of expected responses per request; −1 for infinite receive operation, −2 for multiple responses.
SEND ONLY	Number of send cycles to be performed; −1 for infinite send operation	0
RECEIVE ONLY	0	Number of receive cycles to be performed; −1 for infinite receive operation, −2 for multiple responses.
No data to transmit or receive (e.g. PDU_COPT_STARTCOMM with no pPduData message)	0	0

8.2.6.2.2 NumReceiveCycles description

The NumReceiveCycles is the number of expected complete responses for a ComPrimitive. An infinite receive ComPrimitive (−1) will usually never finish (i.e. never generate a PDU_COPST_FINISHED status item, see PDU_STATUS_DATA) unless timing is enabled for cyclic responses (see ComParam CP_CyclicRespTimeout) and therefore shall usually be cancelled (see 8.4.18).

8.2.6.2.3 NumSendCycles description

The NumSendCycles is the number of periodic transmits to be sent on the vehicle bus. The periodic time interval is specified in milliseconds in the PDU_COP_CTRL_DATA structure (see 10.1.4.17). If NumSendCycles is equal to −1, it is considered an infinite send ComPrimitive. An infinite send ComPrimitive will never finish and shall always be cancelled (see 8.4.18). A periodic send ComPrimitive will have state transitions to and from PDU_COPST_EXECUTING to PDU_COPST_WAITING for each periodic interval.

Table 6 defines some possible combinations of ComPrimitive types and cycle types.

© ISO 2022 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

Table 6 — ComPrimitives combinations

NumSendCycles	NumReceiveCycles	Description
−1	1	Continuously send requests and look for one response per request. If the response is not received before a receive timeout occurs (e.g. P2Max), the ComPrimitive will set a receive timeout error event (PDU_ERR_EVT_RX_TIMEOUT) and proceed to the next send interval.
1	>0	Send one request and wait for the requested number of responses. The ComPrimitive is completed when the requested number of responses are received or a receive timeout occurs (e.g. P2Max).
1	−1	Send one request and look for continuous responses. This case is equivalent to ODX IS-CYCLIC = TRUE. The application shall cancel this ComPrimitive since it will never finish. A cyclic receive timeout can be used if the NumReceiveCycles is set to infinite (−1). (See CP_CyclicRespTimeout ComParam.) In the case of a cyclic receive timeout, the ComPrimitive will transition to PDU_COPST_FINISHED.
1	−2	Send one request and wait for a receive timeout (e.g. P2Max). Should be used for functional addressing when number of responses is unknown or with physical addressing with possible unknown number of responses. This case would be appropriate for ODX IS-MULTIPLE = TRUE. See Table 9 which contains the behaviour's explanation.
1	0	Send one message and ignore any responses.
−1	0	Continuously send messages and ignore any responses.
0	−1	Look continuously for received message cycle. Because there is no transmit then there is no relevant protocol specific response timeout, as such only the cyclic receive timeout (see CP_CyclicRespTimeout ComParam) will be used to indicate a completion. If CP_CyclicRespTimeout =0 and NumReceiveCycles is IS-CYCLIC, this ComPrimitive will never finish, so the application shall have to cancel it.
0	>0	Look for the specified number of received messages. The CoP will be finished on the reception of the specified number of messages or, when enabled, with an error on the expiry of CP_CyclicRespTimeout (see CP_CyclicRespTimeout ComParam). If CP_CyclicRespTimeout = 0 (not enabled) and the specified number of messages has not been received, this ComPrimitive will never finish, so the application shall have to cancel it.
The usage of NumSendCycles=0 and NumReceiveCycles=-2 is forbidden.		

NOTE If cyclic time is set to 0 in the PDU_COP_CTRL_DATA, then the ComPrimitive is put on the transmit queue after each completion cycle, but is at a lower priority than other ComPrimitives and TesterPresent Messages.

8.2.6.3 ComPrimitive principles

8.2.6.3.1 General

The following subclauses describe how to use ComPrimitives for different communication principles known from automotive communication protocols. All of the described actions assume the D-PDU API has been initialized and a ComLogicalLink has been created (for details, see Table 3). In addition to the D-PDU API function calls shown in the tables, additional API calls can be used for additional functions (e.g. status requests, etc.). Any memory allocation or de-allocation initiated by create, start and destroy calls is handled within the D-PDU API.

© ISO 2022 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

8.2.6.3.2 Starting communication

To initiate communication between a tester and an ECU, different initialization methods exist for various communication protocols. For OBD initialization handling, see Annex J. The list below describes different standard initialization use cases.

a) No initialization (using PDU_COPT_SENDRECV)

— Directly start sending a request to the ECU using a PDU_COPT_SENDRECV ComPrimitive. This method is used for protocols like CAN and SAE J1850 which do not require an initialization sequence.

b) No initialization (using PDU_COPT_STARTCOMM)

— Directly start sending a request to the ECU after a PDU_COPT_STARTCOMM ComPrimitive. This method is used for protocols like CAN and SAE J1850 which do not require an initialization sequence, but might use a ComPrimitive message to enter diagnostic mode (e.g. StartDiagnosticSession service). An optional message can be sent for the PDU_COPT_STARTCOMM ComPrimitive for these protocols.

— No initialization response: If an optional message is sent with NumReceiveCycles != 0, then the D-PDU API waits for a response message. If an optional message is not sent, the ComPrimitive finishes right away and no result data is returned. The internal state of the ComLogicalLink changes accordingly (PDU_CLLST_COMM_STARTED).

c) 5-baud initialization (using PDU_COPT_STARTCOMM)

— Initialize the communication link to the ECU by sending a 5-baud initialization sequence. This method is used for many K-Line protocols, e.g. KWP2000. During 5-baud initialization, the tester sends the ECU address on the bus and then calculates the baud rate to be used for further communication. See ComParams (CP_InitializationSettings, CP_5BaudMode, CP_5BaudAddressFunc, CP_5BaudAddressPhys).

— 5 baud initialization response The NumReceiveCycles shall be set to 1 for the ECU key bytes to be returned to the client application, in this case, a result message data structure (see 10.1.4.11.4 PDU_RESULT_DATA) will contain the key bytes in the following order:

  — PDU[0]=key byte 1;

  — PDU[1]=key byte 2.

— A PDUGetComParam can be called to read the CP_Baudrate ComParam which will contain the calculated baud rate.

— No optional message will be allowed for a 5-baud initialization start communication request. The client application should send a new SendRecv ComPrimitive for communication after a 5-baud initialization completes.

— After a 5-baud initialization sequence completes, the protocol handler will begin sending keep-alive messages (see CP_TesterPresent ComParams) if enabled.

— If the ECU key bytes indicate that extended timing is supported, the ComParam CP_ExtendedTiming can be used to override the default values of ISO 14230-2.

© ISO 2022 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

d) Fast initialization (using PDU_COPT_STARTCOMM)

— Initialize the communication link to the ECU by sending a wake-up pattern optionally followed by a service request provided in the ComPrimitive request data.

— Fast initialization response: The PDU_COPT_STARTCOMM ComPrimitive behaves like a normal SendRecv ComPrimitive if an optional request message is contained in the data. Therefore, any ECU responses that match the expected response structure will be returned to the client application.

— If the ECU key bytes indicate that extended timing is supported, the ComParam CP_ExtendedTiming can be used to override the default values of ISO 14230-2.

e) TesterPresent messages (ComLogicalLink State = PDU_CLLST_COMM_STARTED)

— TesterPresent messages will only be enabled when the ComLogicalLink is in the state PDU_CLLST_COMM_STARTED. A successful PDU_COPT_STARTCOMM ComPrimitive is required to enter this state (pCoPData for this ComPrimitive is optional). See CP_TesterPresentHandling for more information on TesterPresent message enabling. Once TesterPresent handling is enabled, the message is sent immediately, prior to the initial TesterPresent cyclic time (CP_TesterPresentTime).

See Table 7 for a generic approach to starting communications on a ComLogicalLink.

Table 7 — Starting communications

Sequence	Action	Description
1	PDUCreateComLogicalLink	Create a logical link on a physical resource. The state is PDU_CLLST_OFFLINE until connected.
2	PDUSetComParam	Set ComParams required for ECU communication, like ECU target address, initialization settings, etc.
3	PDUSetUniqueRespIdTable	Set of ComParams required for uniquely identifying different ECUs. A unique identifier that will be provided by the application during this function call is returned to the client application indicating which ECU response matched the ComPrimitive. The unique identifier is returned to the client application in a PDU_RESULT_DATA structure.
4	PDUConnect	Physically connect the resource to the vehicle bus. The state of the ComLogicalLink is now PDU_CLLST_ONLINE. The vehicle bus can now be monitored with receive only type of ComPrimitives. A transmit on the vehicle bus is possible via a PDU_COPT_STARTCOMM or PDU_COPT_SENDRECV ComPrimitive.

© ISO 2022 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

Sequence	Action	Description
5	Optional: PDUStartComPrimitive (PDU_COPT_STARTCOMM)	This ComPrimitive is placed on the ComPrimitive queue. Upon execution, the ECU communication will be initialized using the configured initialization method. An optional Start Communication Message can be sent to the ECU. ComParams are used to define the type of initialization to perform and the addressing type (physical/functional). In case of fast initialization, the request message, which is defined in the message data structure (pCopData), will be sent after the wakeup pattern. Immediately after successful initialization, the following occurs. Any ECU responses matching the CoP expected response structure are returned as result items (PDU_IT_RESULT). If TesterPresent handling is enabled (see CP_TesterPresentHandling), the message is sent immediately, prior to the initial TesterPresent cyclic time (CP_TesterPresentTime). After initial transmission, the periodic intervals are started. The ComLogicalLink is set to PDU_CLLST_COMM_STARTED. The ComPrimitive status is set to PDU_COPST_FINISHED.
6	PDUStartComPrimitive (PDU_COPT_SENDRECV)	Continue with ECU communication using ComPrimitives. NOTE TesterPresent messages can only be enabled if the ComLogicalLink is in the state PDU_CLLST_COMM_STARTED (see sequence step 5 above).

8.2.6.3.3 Stopping communication

The recommended approach for stopping communication can be found in Table 8.

Table 8 — Stopping communication

Sequence	Action	Description
1	….	Running ECU communication using ComPrimitives
2	PDUStartComPrimitive (PDU_COPT_STOPCOMM)	This ComPrimitive is placed on the ComPrimitive queue. Upon execution, all ECU communication will be terminated. An optional Stop Communication Message can be sent to the ECU. If no message is sent to the ECU, the ComPrimitive status directly changes to PDU_COPST_FINISHED. Immediately after successful stopping of ECU communications, the following occurs. Any ECU responses matching the CoP expected response structure are returned as result items (PDU_IT_RESULT). The ComLogicalLink is set to PDU_CLLST_ONLINE. All currently queued ComPrimitives and currently executing ComPrimitives are cancelled (PDU_COPST_CANCELLED). Periodic TesterPresent Messages are stopped.

8.2.6.3.4 Send and receive handling

The recommended approach for send and receive message handling can be found in Table 9.

© ISO 2022 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

Table 9 — ComPrimitive send/receive handling

Function	Description
Sending Message Handling	A ComLogicalLink shall be in the state PDU_CLLST_COMM_STARTED for TesterPresent messages to be periodically sent. A ComLogicalLink can only be placed in a Comm Started state after a successful completion of a PDU_COPT_STARTCOMM ComPrimitive (see 8.2.6.3.2). When a protocol running on a ComLogicalLink has been properly configured using the ComParams for the selected protocol type, then the D-PDU API can correctly construct a full D-PDU to be sent to the vehicle's ECU. Proper construction of a message is based on raw mode configuration of the ComLogicalLink (see D.2.3) and is protocol specific. Therefore, proper construction could consist of message header bytes, transport layer handling and checksumming features. Furthermore, messages are constructed based on the type of addressing scheme selected (physical or functional addressing defined as a ComParam). In non-raw mode, the first byte of the pCopData of the ComPrimitive received from the application would usually consist of the service ID (if applicable to the protocol).
Receive Message Handling	Every message received from the vehicle bus will first be subjected to the Pass/Block filters (if configured). Initially, the D-PDU API configures the filters based on the UniqueResponseIdentifierTable. The client application can override this auto configuration of the filters by using any of the PDUIoCtl commands PDU_IOCTL_xxx_MSG_FILTER. Next, the ECU message will be checked for correctness (checksum, PCI information, etc). The UniqueRespIdTable is then referenced to determine a match to a known ECU identifier. The UniqueRespIdentifer table can be configured to pass all ECU identification (see 8.4.28.7). The UniqueRespIdentifier will be used in the PDU_IT_RESULT event item to indicate to the application which ECU the message belongs to. Finally, the D-PDU API will determine a match of the ECU message to a ComPrimitive Expected Response Structure (see 10.1.4.19). The D-PDU API will first compare the ECU message to the current active Send/Receive ComPrimitive and if not matched it will search through the Receive Only list. The message is considered bound when it matches to the first ComPrimitive and no further ComPrimitive matching is continued after the initial match. If the response cannot be bound to any ComPrimitive's ExpectedResponseStructure, the message is discarded. NOTE 1 A transport layer uses the UniqueRespIdentifer table and the ComParams from the currently active SendRecv ComPrimitive for initial receive handling of frames/messages. If the ComLogicalLink does not have an active SendRecv ComPrimitive, then the ComLogicalLinks active ComParam buffer is used. Once the frame/message is bound to a ComPrimitive, the set of ComParams attached to the ComPrimitive is used for any further processing (e.g. receive timing).
SEND ONLY NumReceiveCycles = 0 (PDU_COPT_SENDRECV)	If the NumReceiveCycles is equal to zero and the NumSendCycles is not equal to zero, then the ComPrimitive is considered to be a send only ComPrimitive. This type of ComPrimitive can still be periodic by setting the delay time interval in the PDU_COP_CTRL_DATA structure (see 10.1.4.17). The D-PDU API will construct the proper message to be transmitted, but will not set up any receive timers for responses from an ECU(s). Any ComParams defined for "time between transmits" (i.e. CP_P3min, CP_P3Func, CP_P3Phys) will be used to ensure proper vehicle bus timing. If the protocol type allows for an immediate transmit of another message, then any pending ComPrimitives would be available for immediate execution. Once the ComPrimitive has completed all transmission send cycles (NumSendCycles), the status of the ComPrimitive is set to PDU_COPST_FINISHED and the status item (see 10.1.4.11.1) is placed on the ComLogicalLink's event queue.

© ISO 2022 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

Function	Description
RECEIVE ONLY NumSendCycles = 0 (PDU_COPT_SENDRECV)	If the NumSendCycles is equal to zero and the NumReceiveCycles is not equal to zero, then the ComPrimitive is considered to be a receive only ComPrimitive. The NumReceiveCycle count is used to monitor the vehicle bus for messages that match the ExpectedResponseStructure (see 10.1.4.19). When the receive count has been reached, the ComPrimitive transitions to PDU_COPST_FINISHED. Once all the expected responses have been received, the status of the ComPrimitive is set to PDU_COPST_FINISHED and the status item (see Structure for status) is placed on the ComLogicalLink's event queue. NOTE 2 A cyclic receive timeout can be used if the NumReceiveCycles is set to infinite (−1) or >0 (see CP_CyclicRespTimeout ComParam). In the case of a cyclic receive timeout, the ComPrimitive will transition to PDU_COPST_FINISHED. Otherwise, the application cancels the ComPrimitive via PDUCancelComPrimitive. NOTE 3 No pCopData bytes are supplied in this ComPrimitive type.
SEND AND RECEIVE NumReceiveCycles != 0 NumSendCycles != 0 (PDU_COPT_SENDRECV)	When both the NumSendCycles and the NumReceiveCycles are not equal to zero, the ComPrimitive will attempt to deliver the number of responses specified in NumReceiveCycles each time the ComPrimitive is sent. If the number of responses specified by NumReceiveCycles is not found before a receive timeout occurs (e.g. P2Max), the ComPrimitive will generate an error event, indicating that a receive timeout occurred. If the ComPrimitive is periodic, it will NOT transition to PDU_COPST_FINISHED even on a receive timeout, the cyclic transmissions will continue. The D-PDU API will construct the proper message to be transmitted. Once the message has been properly transmitted, the receive timers for the ComLogicalLink will be enabled waiting for matching ECU responses. See ExpectedResponseStructure and PDU_COP_CTRL_DATA (see 10.1.4.17). Multiple Expected Responses (IS-MULTIPLE) (see 10.1.4.17, PDU_COP_CTRL_DATA – NumReceiveCycles = -2): Each received message will reset the receive timer (e.g. P2Max) on a matching response. A receive timeout with no matching responses from any ECU will generate an error event (PDU_ERR_EVT_RX_TIMEOUT). The ComPrimitive will end without an error when a receive timeout has occurred (e.g. P2Max) and at least one valid response has been received. All functional requests with expected responses should set the NumReceiveCycles to IS-MULTIPLE (−2). Infinite Responses (IS-CYCLIC) (see 10.1.4.17, PDU_COP_CTRL_DATA – NumReceiveCycles = −1): When the ComLogicalLink completes the transmit and receives the first positive response, the ComPrimitive is placed into a receive only mode. This allows other ComPrimitives to be transmitted on the ComLogicalLink while still receiving responses from the cyclic ComPrimitive. The timing restrictions imposed by Table 1 (CP_P3Phys or CP_P3Func) will only apply for the first receive cycle, once this has occurred and the ComPrimitive transitions to receive only (see Table 9), no such timing rules shall apply to this ComPrimitive (i.e. it shall be treated as if it were a receive only ComPrimitive). A cyclic receive timeout can be used if the NumReceiveCycles is set to infinite (−1). (See CP_CyclicRespTimeout ComParam). In the case of a cyclic receive timeout, the ComPrimitive will transition to PDU_COPST_FINISHED. Otherwise, the application shall cancel the ComPrimitive via PDUCancelComPrimitive. Once all the expected responses have been received and all transmission send cycles have been completed, the status of the ComPrimitive is set to PDU_COPST_FINISHED and the status item (see 10.1.4.11.1) is placed on the ComLogicalLink's event queue.

© ISO 2022 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

Function	Description
7F Handling	This level of protocol message handling has been moved to the D-PDU API to ensure proper low-level real-time requirements. This functionality can be handled in RawMode if the Response Code Offset ComParam is correctly configured and the number of header bytes can be determined by the protocol handler (some protocols allow configuration of header byte count via ComParams). Negative response (0x7F) handling can be enabled or disabled by properly setting the correct ComParams in a ComLogicalLink. (See ComParam CP_RCxxHandling in Annex B.) Only response codes 0x21, 0x23 and 0x78 are configurable to be handled by the D-PDU API. Not all protocol implementations support all negative responses codes. The ComLogicalLink shall actively process a ComPrimitive with expected responses to a request to proceed with any 0x7F handling (i.e. negative response handling is not enabled for receive only ComPrimitives). The D-PDU API will bypass any addressing information (based on protocol) to determine if an ECU has responded with a 0x7F. The 0x7F code is typically contained in the first byte of the payload data, followed by the requested service id and the Response Code (see CP_RCByteOffset for configurable response code byte offset processing). The Response Code is extracted from the payload data and used to determine if further 0x7F handling is necessary by the D-PDU API. Based on the type of Response Code, the D-PDU API will begin the timing/retry handling. The D-PDU API is protected from infinite retries by CP_RCxx CompletionTimeout ComParam. If the ECU fails to generate a positive response in the specified Completion Timeout time, then the D-PDU API will generate an error event item indicating no responses received PDU_ERR_EVT_RX_TIMEOUT. If negative response handling is not enabled or the response code is not recognized by the D-PDU API, then the negative response message is placed in a result item and added to the ComLogicalLink's event queue.

NOTE PDU_ERR_INVALID_PARAMETERS will be returned for ComPrimitives of type PDU_COPT_SENDRECV when NumSendCycles>0 and PDU is empty. When the NumSendCycles is set =0, the PDU will be ignored.

8.2.6.3.5 ComPrimitives in non-raw mode

The idea of non-raw mode is that everything about an ECU or a functional group of ECUs is known and can be configured using ComParams for the selected protocol. The ComParam information would normally be contained in a database. A database schema, such as ODX, and a COMPARAM-SPEC have been developed to support the concept of fully abstracting ComParams to be used by the D-PDU API.

Each ComLogicalLink is directed to a single ECU and/or a functional group of ECUs. The ECU information can be configured initially and not changed during the course of ECU communication. If switching between ECUs is necessary on a single vehicle serial bus, then it is suggested that the application create another ComLogicalLink for the additional ECU communication.

It shall be noted that an additional ComLogicalLink cannot be created if it would result in the use of a non-shareable resource (e.g. DLC pins have already been reserved by another ComLogicalLink with a different protocol id). Also, if a second ComLogicalLink requires a "Start Communication Message" which causes the vehicle bus pins to be driven into a different impedance state, this situation would have an undeterminable behaviour on the vehicle bus. All other ComLogicalLinks that are sharing the resource would begin to report errors because communication has been lost to their specific ECU (e.g. ISO 14230/ISO 9141-2 initialization sequence would cause another ComLogicalLink using the same protocol and pins to lose communication to an ECU).

© ISO 2022 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

8.2.6.3.6 ComPrimitives in raw mode (PassThru)

The idea of raw mode is that everything about an ECU or a functional group of ECUs and the protocol is known by the client application. The application takes on the responsibility over the entire protocol message structure, including header bytes and checksums. The only exception would be the necessary requirements of things like the transport layer (see ISO 15765-2) and inter-byte/inter-message timing (see ISO 9141-2/ISO 14230).

The application shall handle all knowledge about protocol header byte configuration, CAN IDs, extended addressing, negative responses, receive timeout timing (e.g. P2Max), TesterPresent handling and the binding of response(s) to the request. Therefore, the application cannot be protocol-independent (abstract) in its use of the D-PDU API.

For ISO 15765, SAE J1939 and ISO 11898, the first 4 bytes of the pCopData shall be the CAN ID (11-bit or 29-bit). If extended addressing is enabled (see D.2.1), then the byte following the CAN ID contains the extended address byte.

8.2.6.4 ComPrimitive sequence diagrams

8.2.6.4.1 Single request/single response (master/slave communication) — Event notification

This communication principle is used for classic master/slave communication (e.g. ISO 14230 KWP 2000 standard services). See Figure 8.

© ISO 2022 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

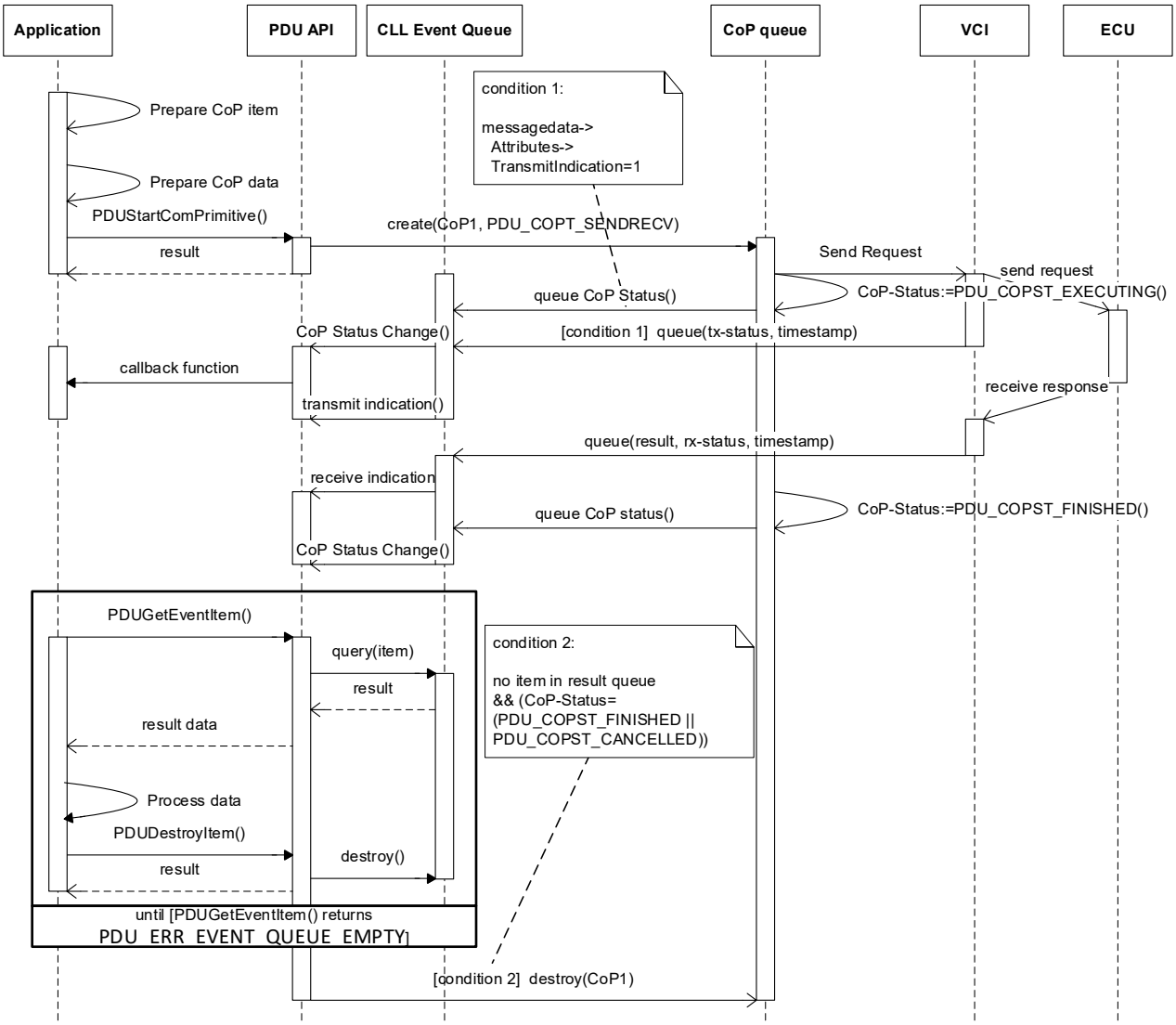


Figure 8 — Single request-single response (master, slave communication)

NOTE A callback is only initiated when an event is placed into an empty queue.

8.2.6.4.2 Single request/single response (master/slave communication) — Polling mode

Applications working without event notification can poll for results related to a specific ComPrimitive by calling PDUGetStatus. In this case, the callback function will not be called as depicted in Figure 9.

© ISO 2022 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

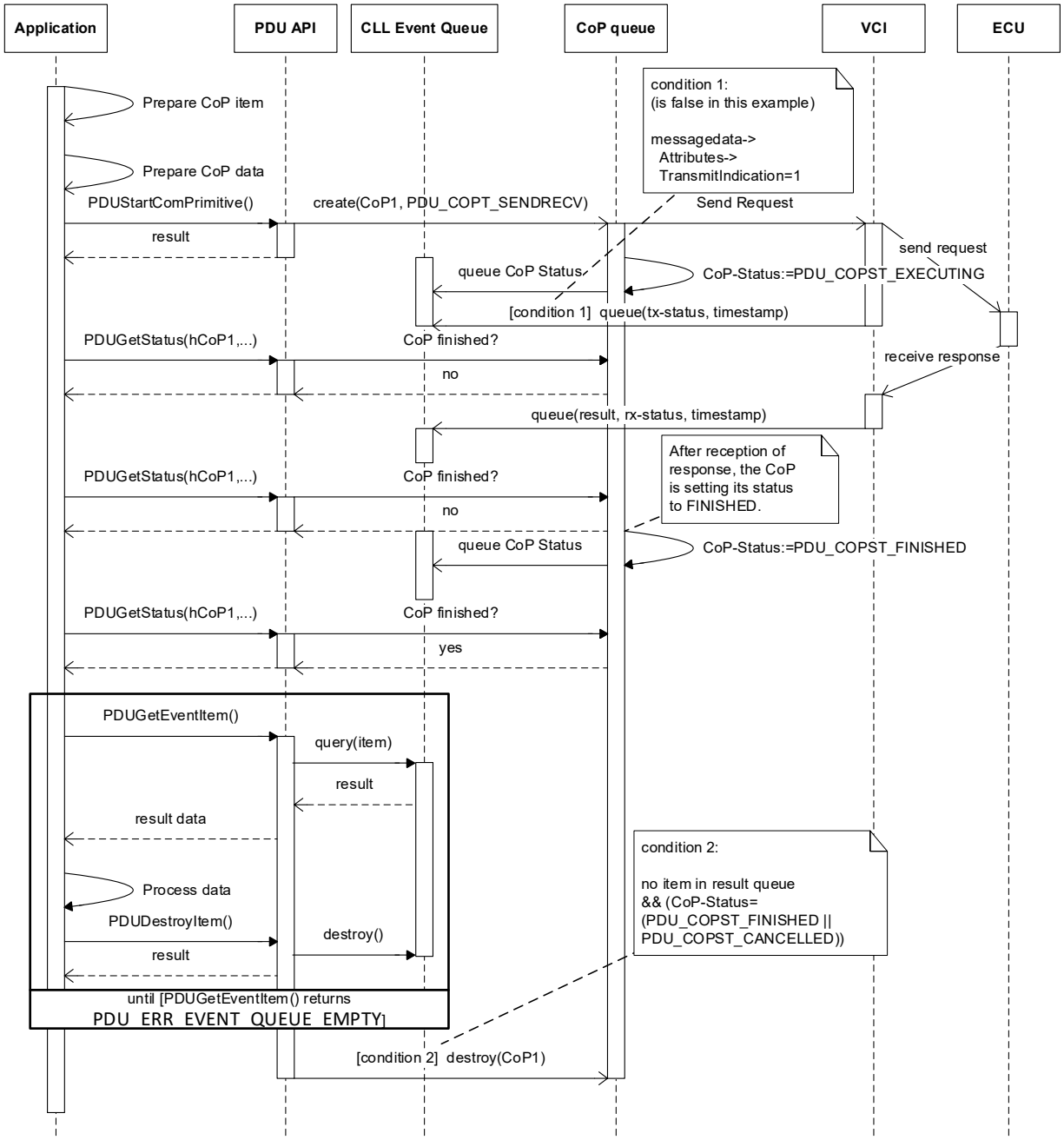


Figure 9 — Single request–single response (master, slave communication) — Polling mode

8.2.6.4.3 Single request/multiple responses

This communication principle is used in many protocols for finite and infinite communication sequences. The basic principle for this ComPrimitive type is shown in Figure 10.

© ISO 2022 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

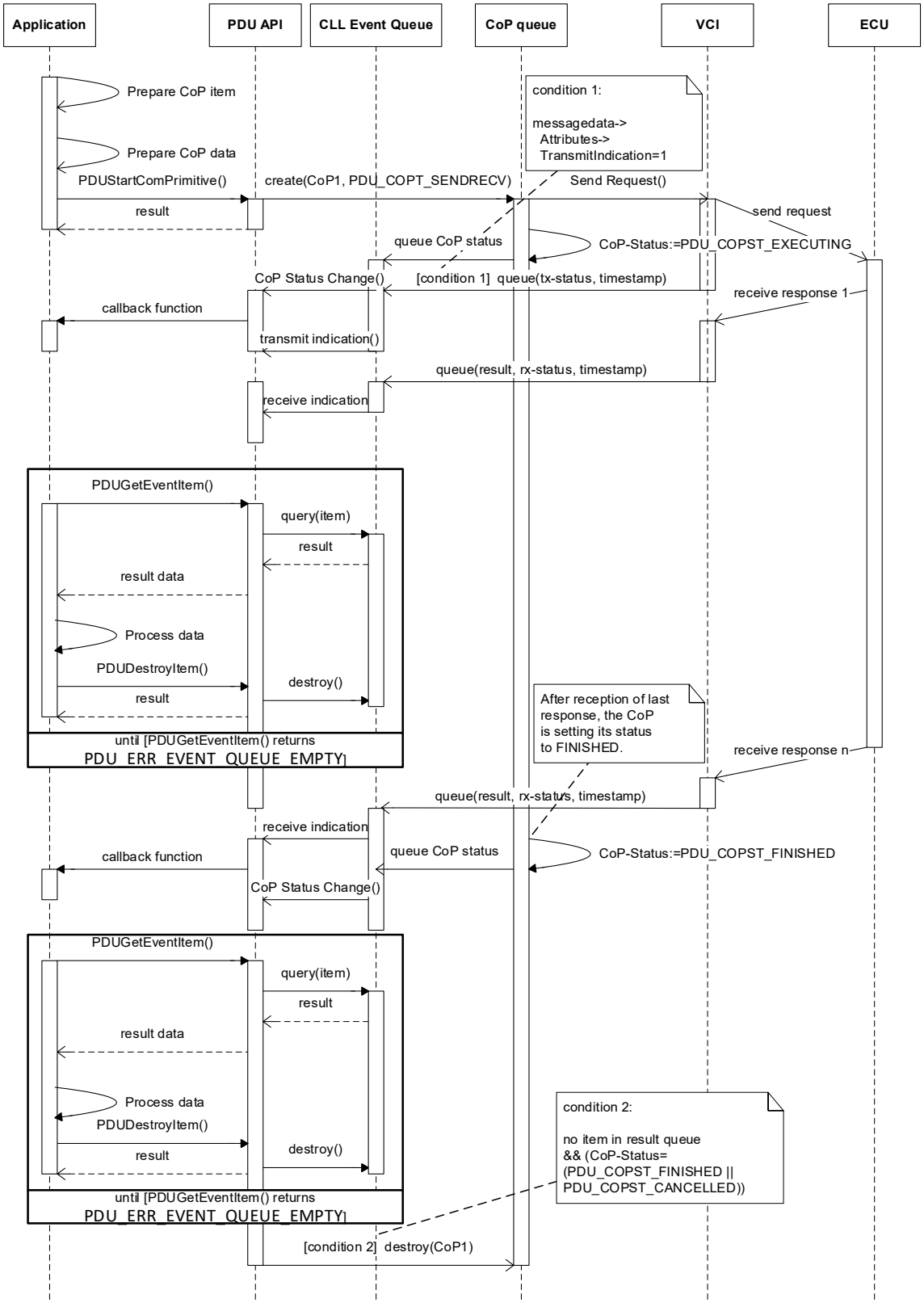


Figure 10 — Single request, multiple responses

© ISO 2022 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

NOTE The API function calls PDUGetEventItem and PDUDestroyItem, as well as the result data processing by the application, can be executed while the D-PDU API's receiving process of ECU responses runs in parallel.

An event callback does not have to be called for each data item. It is up to the callback routine to check for further event items before returning. A callback is initiated either when an event is placed into an empty queue or when events are already queued when the PDURegisterEventCallback function is called that registers a callback function.

PDUGetEventItem actions may also be mixed with event callback function actions.

8.2.6.4.4 Single request/multiple responses — Functional addressing

Figures 11 to 14 show examples of single request/multiple response ComPrimitives for specific protocol implementations. The API calls are not shown in detail because they are already described in the figures above.

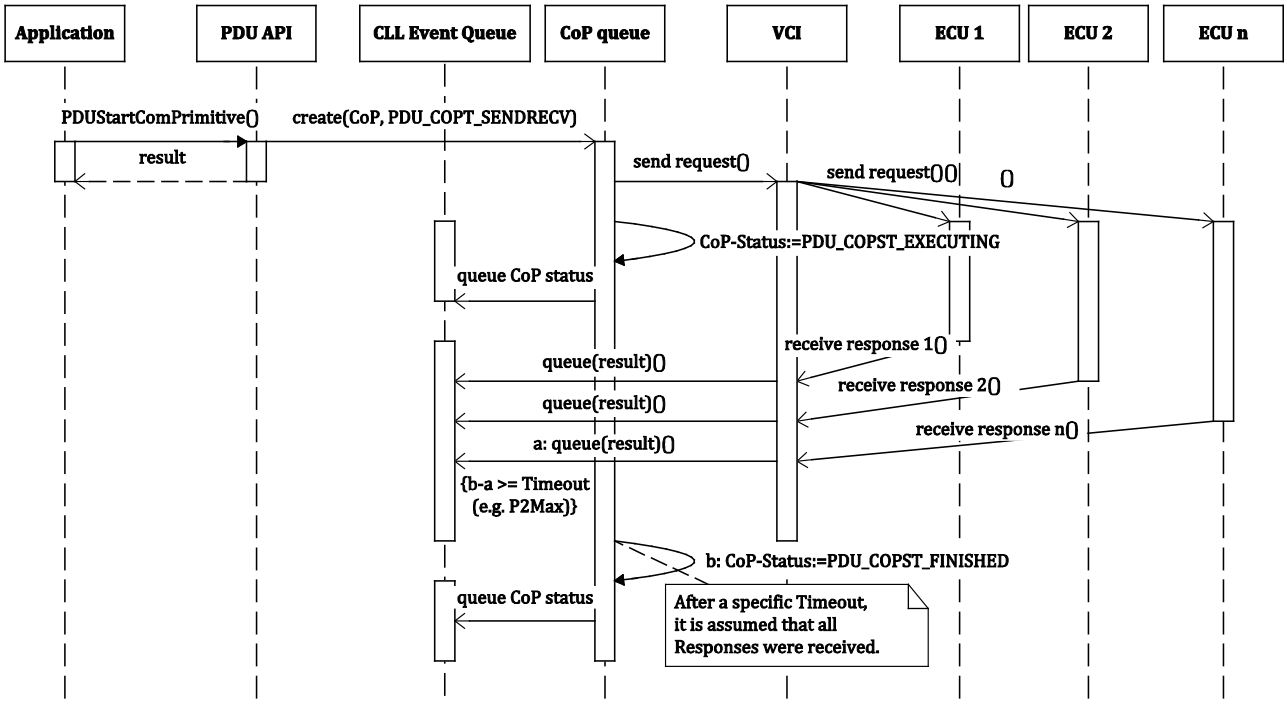


Figure 11 — Example of single request, multiple responses — ISO 14230-3 functional addressing

© ISO 2022 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

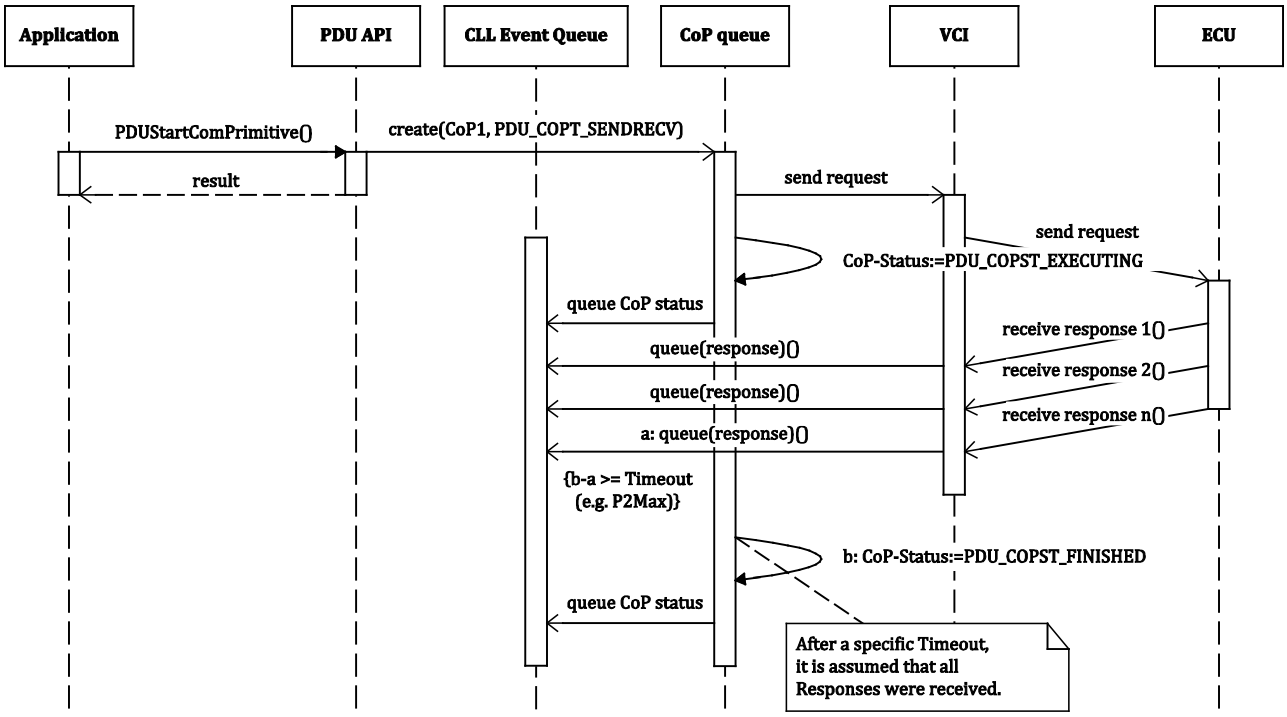


Figure 12 — Example of single request, multiple responses — ISO 14229-1 finite periodic mode (e.g. time window)

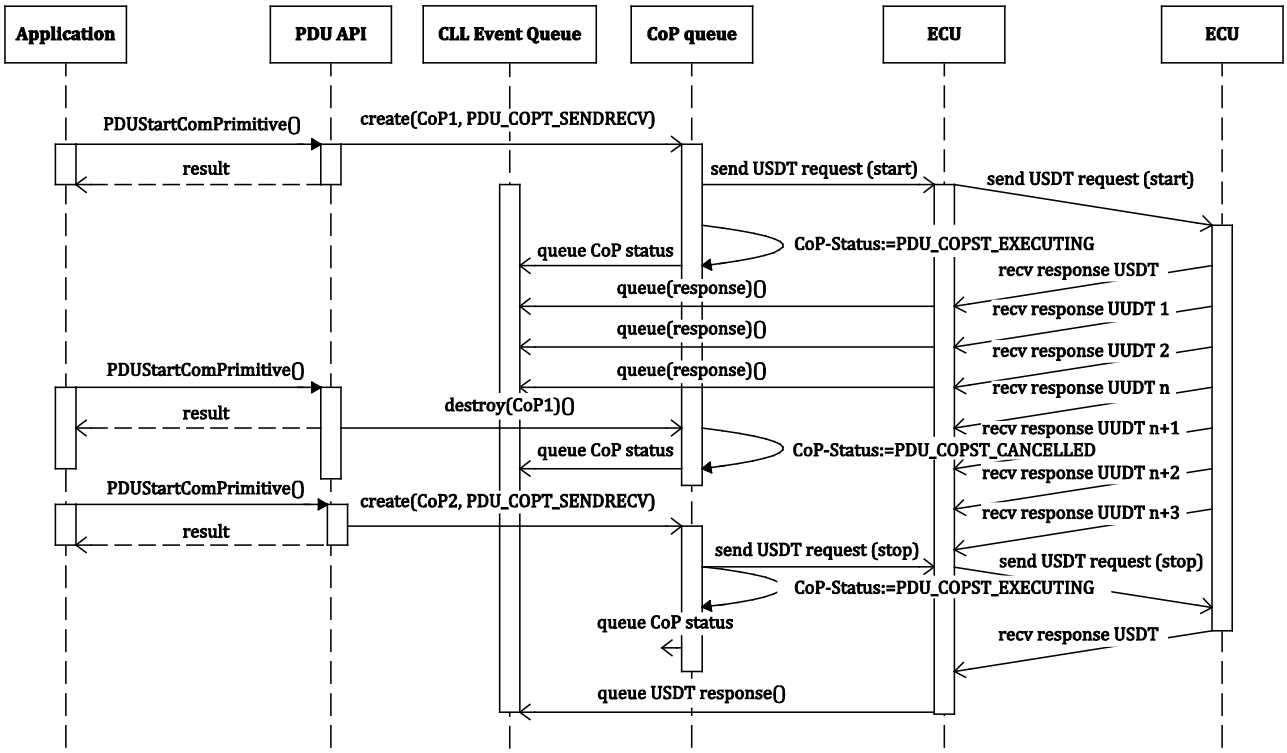


Figure 13 — Example of single request, multiple responses — ISO 14229-1 ReadDataByPeriodic-Identifier, type #2

© ISO 2022 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022

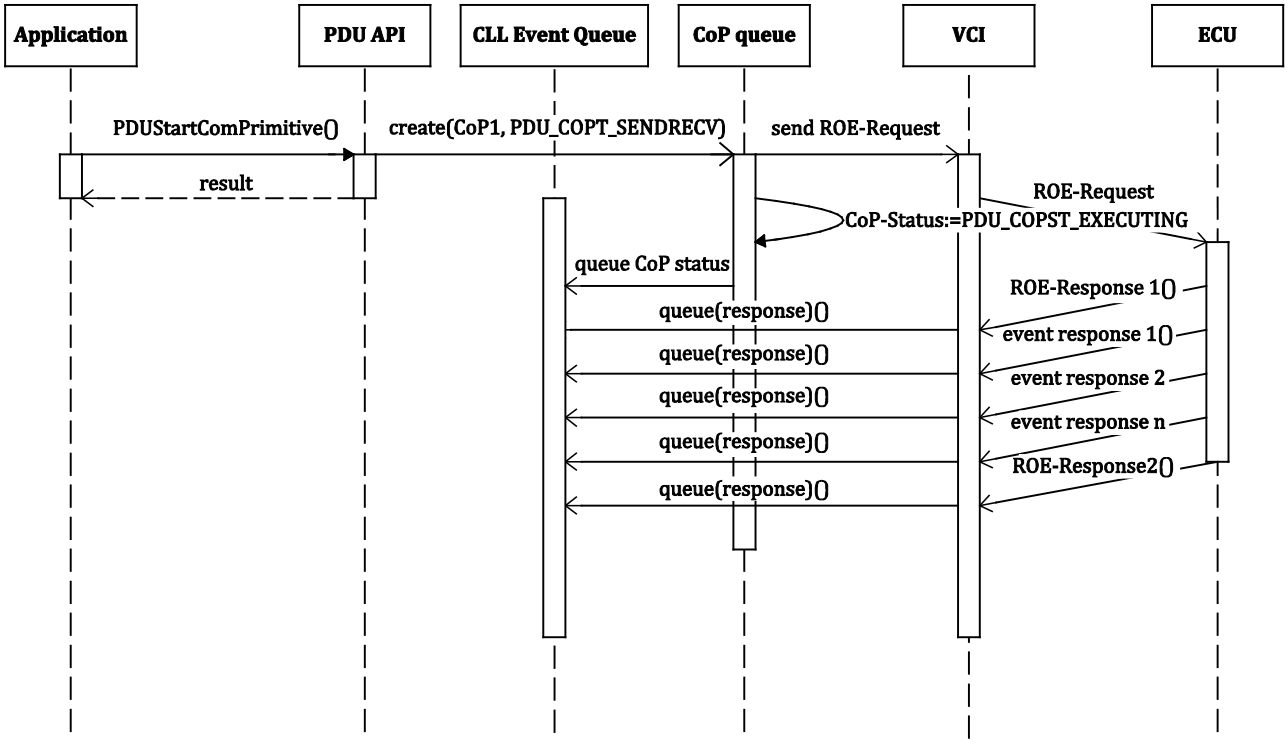


Figure 14 — Example of ISO 14229-1 ResponseOnEvent (RoE)

8.2.6.4.5 Single or multiple requests/no responses

This communication principle is used especially in on-board communication (e.g. CAN). See Figure 15.

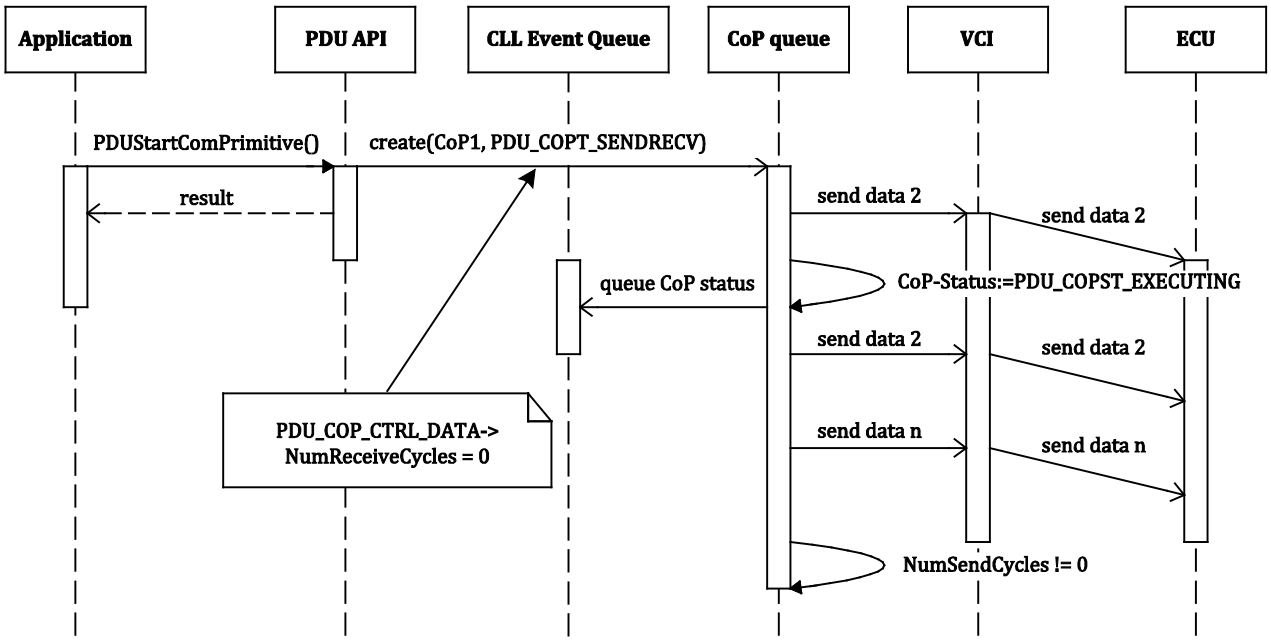


Figure 15 — Single or multiple requests, no responses

© ISO 2022 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO 22900-2:2022