

BS ISO 22900-3:2012



BSI Standards Publication

Road vehicles — Modular vehicle communication interface (MVCI)

Part 3: Diagnostic server application programming interface (D-Server API)

bsi.

...making excellence a habit.
AVA Documents™
Document Sharing Hub

National foreword

This British Standard is the UK implementation of ISO 22900-3:2012. It supersedes BS ISO 22900-3:2009, which is withdrawn.

The UK participation in its preparation was entrusted to Technical Committee AUE/16, Electrical and electronic equipment.

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

This publication does not purport to include all the necessary provisions of a contract. Users are responsible for its correct application.

© The British Standards Institution 2012.

Published by BSI Standards Limited 2012.

ISBN 978 0 580 71528 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 December 2012.

Amendments issued since publication

Amd. No.	Date	Text affected
----------	------	---------------

INTERNATIONAL STANDARD

ISO
22900-3

Second edition
2012-12-01

Road vehicles — Modular vehicle communication interface (MVCI) —

Part 3: Diagnostic server application programming interface (D-Server API)

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

*Partie 3: Interface pour la programmation des applications du serveur
de diagnostic (D-Server API)*



Reference number
ISO 22900-3:2012(E)

AVA **Documents**
Document Sharing Hub
© ISO 2012



COPYRIGHT PROTECTED DOCUMENT

© ISO 2012

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

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

Published in Switzerland

Contents

Page

Foreword	v
Introduction.....	vi
1 Scope	1
2 Normative references	1
3 Terms, definitions, symbols and abbreviated terms	1
3.1 Terms and definitions	1
3.2 Symbols.....	3
3.3 Abbreviated terms	4
4 Conventions	5
4.1 General	5
4.2 Typographical conventions and mnemonics	5
4.3 Sequence diagrams.....	6
4.4 Stereotypes	6
5 Specification release version information	6
6 Structure of a MVCI diagnostic server	6
7 Diagnostic server	10
7.1 MCD system object	10
7.2 Description of terms.....	11
7.3 Version information retrieval	16
7.4 States of the MCD system	16
7.5 State changes	19
7.6 Project configuration	19
7.7 Interface structure of server API.....	21
7.8 Collections	46
7.9 Registering/deregistering of the EventHandler.....	50
7.10 MCD value	51
7.11 Use cases	54
8 Function block Diagnostic in detail	60
8.1 Constraints.....	60
8.2 System Properties	70
8.3 Diagnostic DiagComPrimitives and Services.....	71
8.4 Suppress positive response	101
8.5 eEND_OF_PDU as RequestParameter	102
8.6 Variable length parameters	104
8.7 Variant identification	106
8.8 Use cases	117
8.9 Read DTC	135
8.10 Logical Link.....	144
8.11 Functional addressing	156
8.12 Tables	158
8.13 Dynamically Defined Identifiers (DynId).....	168
8.14 Internationalization.....	179
8.15 Special Data Groups	179
8.16 ECU (re-) programming.....	181
8.17 Handling binary flash data	188
8.18 Library.....	190
8.19 Jobs	191
8.20 ECU configuration	212

8.21	Audiences and additional audiences	229
8.22	ECU states	231
8.23	Function dictionary	234
8.24	Sub-Component data model description	242
8.25	Monitoring vehicle bus traffic.....	244
8.26	Support of VCI module selection and other VCI module features according to ISO 22900-2 ..	246
8.27	Handling DoIP entities.....	255
8.28	Mapping of D-PDU API methods	258
9	Error Codes	263
9.1	Principle.....	263
9.2	Description of the errors.....	265
Annex A	(normative) Value reading and setting by string.....	267
A.1	Datatype conversion into Unicode2 string	267
A.2	Representation floating numbers	267
A.3	Normalized floating-point numbers	268
Annex B	(normative) System parameter.....	269
B.1	Overview	269
B.2	Description of the system parameters	270
Annex C	(normative) Overview optional functionalities	272
Annex D	(informative) Monitoring message format.....	278
D.1	General.....	278
D.2	CAN format.....	278
D.3	K-Line Format.....	279
D.4	DoIP Format.....	280
Bibliography		281

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.

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

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

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

ISO 22900-3 was prepared by Technical Committee ISO/TC 22, *Road vehicles*, Subcommittee SC 3, *Electrical and electronic equipment*.

This second edition cancels and replaces the first edition (ISO 22900-3:2009), which has been technically revised.

ISO 22900 consists of the following parts, under the general title *Road vehicles — Modular vehicle communication interface (MVCi)*:

- *Part 1: Hardware design requirements*
- *Part 2: Diagnostic protocol data unit application programming interface (D-PDU API)*
- *Part 3: Diagnostic server application programming interface (D-Server API)*

Introduction

0.1 Overview

This part of ISO 22900 has been established in order to define a universal application programmer interface of a vehicle communication server application. Today's situation in the automotive market requires different vehicle communication interfaces for different vehicle OEMs supporting multiple communication protocols. However, until today, many vehicle communication interfaces are incompatible with regard to interoperability with multiple communication applications and vehicle communication protocols.

Implementation of the MVCI diagnostic server concept supports overall cost reduction to the end user because, for example, a single diagnostic or programming application will support many vehicle communication interfaces supporting different communication protocols and different vehicle communication modules of different vendors at one time.

A vehicle communication application compliant with this part of ISO 22900 supports a protocol independent D-PDU API (Protocol Data Unit Application Programming Interface) as specified in ISO 22900-2. The server application will need to be configured with vehicle- and ECU-specific information. This is accomplished by supporting the ODX data format (Open Diagnostic Exchange format) as specified in ISO 22901-1.

A server compliant with this part of ISO 22900 supports the function block Diagnostics (D). A compliant server also supports Job-Language (Java) and may support optional features like ECU (re)programming. The defined object-oriented API provides for a simple, time saving and efficient interchangeability of different servers.

The client application and the communication server do not necessarily need to run on the same computer. A remote use via an interface may also be envisaged and is supported by the design of the server API. This interface is provided for ASAM GDI, COM/DCOM ^[10] [Technology Reference COM-IDL], for C++ ^[11] [Technology Reference C++] and for Java ^[12] [Technology Reference Java].

0.2 ASAM e.V. implementation reference documents

This part of ISO 22900 references several ASAM e.V. documents which contain the Technology Reference Mapping Rules for COM-IDL, C++ and Java.

The following ASAM documents are relevant for the implementation of this part of ISO 22900:

- ASAM Technology Reference COM-IDL, *COM-IDL Technology Reference Mapping Rules* ^[10]:
this document describes the platform, programming language and linking mechanisms for the implementation of the generic object model in COM-IDL.
- ASAM Technology Reference C++, *C++ Technology Reference Mapping Rules* ^[11]:
this document describes the platform, programming language and linking mechanisms for the implementation of the generic object model in C++.
- ASAM Technology Reference Java, *Java Technology Reference Mapping Rules* ^[12]:
this document describes the platform, programming language and linking mechanisms for the implementation of the generic object model in Java.

Road vehicles — Modular vehicle communication interface (MVCI) —

Part 3: Diagnostic server application programming interface (D-Server API)

1 Scope

This part of ISO 22900 focuses on the description of an object-oriented programming interface. The objective is the ability to implement server applications, used during the design, production and maintenance phase of a vehicle communication system, compatible to each other and thus exchangeable. From a user's perspective, access and integration of on-board control units is provided by a corresponding application, the communication server and a VCI module for diagnostics. The user is granted access for the handling of control units (ECUs) in vehicles for the diagnostic services.

2 Normative references

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

ISO 14229 (all parts), *Road vehicles — Unified diagnostic services (UDS)*

ISO 14230-3, *Road vehicles — Diagnostic systems — Keyword protocol 2000*

ISO 15765 (all parts), *Road vehicles — Diagnostic communication over Controller Area Network (DoCAN)*

ISO 22901-1, *Road vehicles — Open diagnostic data exchange (ODX) — Part 1: Data model specification*

ISO 22900-2, *Road vehicles — Modular vehicle communication interface (MVCI) — Part 2: Diagnostic protocol data unit application programming interface (D-PDU API)*

3 Terms, definitions, symbols and abbreviated terms

3.1 Terms and definitions

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

3.1.1

AccessKey

path identifier through the inheritance hierarchy as defined in ISO 22901-1 ODX to a diagnostic data element

3.1.2

ancestor object **parent object**

located above in the object hierarchy with respect to a given object

3.1.3

descendant object **child object**

object, located below in the object hierarchy with respect to a given object

3.1.4

FlashJob

new class derived from MCDJob which is used to start FlashSessions within the MVCI diagnostic server

NOTE This information is provided by the databases. At the runtime object it is possible to set the FlashSession that has to be flashed by this service. Only one session can be set for one job. The application can access the priority defined in the database for every FlashSession and sort the sessions according to this priority.

The job interface of flash jobs (MCDFlashJob) extends the job interface of normal diagnostic jobs (MCDSingleECUJob) by a session object, i.e. its method prototype is extended as follows:

```
JobName(..., MCDDbFlashSession session)
```

3.1.5

FlashKey

unique identification for a flash session

3.1.6

FlashSessionClass

user-defined collection of FlashSessions, which can be used to separate FlashSessions for different tasks (e.g. sessions for data, sessions for boot, or sessions for code and data)

3.1.7

FlashSession

smallest unit that can be flashed separately by the MVCI diagnostic server, and which may consist of several data blocks

3.1.8

functional class

set of diagnostic services

3.1.9

function dictionary

hierarchical function catalog to organize external test equipment user interfaces (available at MCDDbProject):

- references to one or several ECUs and their diagnostic data content relevant for that function;
- references to services/jobs to make functions “executable”;
- definition of function input and output parameters with optional references to parameters of related services

3.1.10

interface connector

connector at the vehicle's end of the interface cable between the vehicle and the communication device

3.1.11

job

sequence of diagnostic services and other jobs with a control flow

3.1.12

location

set of diagnostic data valid on a given hierarchical level of inheritance according to ISO 22901-1 ODX

NOTE The following locations exist:

- Multiple ECU Job,
- Protocol,
- Functional Group,
- ECU Base Variant,
- ECU Variant.

3.1.13

Logical Link

set of data, identifying the physical line, the interface and protocol used for an ECU

3.1.14

physical interface link

physical connection between the VCI connector of a VCI and the interface connector

3.1.15

physical link

physical vehicle link connected to a physical interface link, so it is the connection from the interface of the diagnostic server to the ECU in the vehicle

3.1.16

physical vehicle link

unique bus system in a vehicle, so it is the connection between the vehicle connector and the ECU

3.1.17

priority

term used by test systems to decide in which order the sessions have to be flashed

3.1.18

project

pool of diagnostic data

NOTE References between such data are resolvable inside this same project.

3.1.19

sub component

ECU sub functionality or components

EXAMPLE LIN-slaves (available at `MCDDbLocation`).

3.1.20

vehicle connector

connector on a vehicle providing access to the bus systems in the vehicle

3.2 Symbols

Figure 1 shows the legend of hierarchical models.

	color print	black/white print
 Interface not derived from MCDOObject	blue	black
 Interface not directly used	white	white
 D data base interface	yellow	grey
 D run time interface	green	dark grey
 MCD data base interface	yellow	grey
 MCD run time interface	green	dark grey

Figure 1 — Legend of hierarchical models

3.3 Abbreviated terms

API	Application Programmers Interface
ASAM	Association for Standardisation of Automation and Measuring Systems
ASCII	American Standard for Character Information Interchange
AUSY	AUtomation SYstem
CAN	Controller Area Network
COM/DCOM	Distributed Component Object Model
CORBA	Common Object Request Broker Architecture
CRC	Cyclic Redundancy Check
D	Diagnostics
Diag	Diagnostic
DLL	Dynamic Link Library
DoCAN	Diagnostic communication over CAN
DOP	diagnostic Data Object Property
DoIP	Diagnostic Over Internet Protocol
DTC	Diagnostic Trouble Code
DTD	Document Type Definition
DynID	Dynamically Defined Identifiers
ECU	Electronic Control Unit
ECU MEM	Electronic Control Unit MEMory
ERD	Entity Relationship Diagram
IDL	Interface Description Language

JAVA RMI	JAVA Remote Method Invocation
KWP	KeyWord Protocol
LIN	Local Interconnect Network
MCD	Measurement Calibration Diagnostic
MDF	Module Description File
MVCI	Modular Vehicle Communication Interface
ODX	Open Diagnostic data eXchange
OEM	Original Equipment Manufacturer
PC	Personal Computer
PDU	Protocol Data Unit
SDG	Special Data Groups
SI	Système International d'unités
UDS	Unified Diagnostic Services
UTC	Coordinated Universal Time
VI	Variant Identification
VIS	Variant Identification and Selection
VIT	VehicleInformationTable
XML	eXtended Markup Language

4 Conventions

4.1 General

This part of ISO 22900 is based on the conventions discussed in the OSI Service Conventions (ISO/IEC 10731:1994) as they apply for diagnostic services.

4.2 Typographical conventions and mnemonics

Normal text of the specification is presented like this.

Source code and technical artifacts within the text are presented like `this`.

Diagrams that denote interaction sequences, relationships or dependencies between interfaces are presented using the Unified Modeling Language's (UML) convention.

The name of each interface and each class defined by this part of ISO 22900 shall use the prefix of the stereotype, e.g. "D".

The leading letter of each method and each parameter is small.

The leading word of each method shall be a verb.

The letter “_” is not allowed in interface names, method names and parameter names, but it is allowed for constants.

The leading letter of each constant is “e” and behind this the name is written in capital letters.

ODX element names are written in upper cases, e.g. SHORT-NAME. MVCI diagnostic server Names are written in mixed fixed, e.g. MCDDbProject.

4.3 Sequence diagrams

With the help of Sequence Diagrams the interactive use of the API and the sequences for certain general cases are presented in chronological order.

The sequence diagrams are oriented according to the presentation in UML and are structured as follows. The chronological sequence arises while reading from top downwards. The commentary column, in which single activities are commented, is placed at the left margin. Within the sequence diagram the Client application is shown on the left; if necessary for the respective case, the EventHandler is shown there as well. The API objects necessary for the respective case are located to the right of the Client (with or without EventHandler). If necessary, the MVCI diagnostic server is presented at the right, outside.

Not all API objects possible for the respective instant of time are shown; instead, only those of relevance for the respective case are shown. The thin line leading down vertically from the objects represents the life line, the wider sections on it represent activities of the object.

The black horizontal arrows between the single objects, Client and MVCI diagnostic server represent the actions necessary for the respective case. The object to which the arrow points at will execute the action. The grey horizontal arrows represent the return of objects.

4.4 Stereotypes

Stereotypes are abbreviation characters which are used in MVCI diagnostic servers to mark the affiliation of statements, interfaces and methods to one of the possible parts.

Table 1 defines the stereotypes which are used in MVCI diagnostic servers.

Table 1 — Stereotypes

Stereotype	Usage of method and class is in following Function Blocks allowed
<<MCD>>	Measurement, Calibration and Diagnostic
<<D>>	Diagnosis
<<JD>>	Methods with this stereotype can only be used inside of Diagnostic Job. These methods are not available for use at the API.

5 Specification release version information

The specification release version of this part of ISO 22900 is: 3.0.0.

6 Structure of a MVCI diagnostic server

Each server is divided into the functional block "D" (Diagnostic) and the database.

Figure 2 shows the architecture of an MVCI diagnostic server.

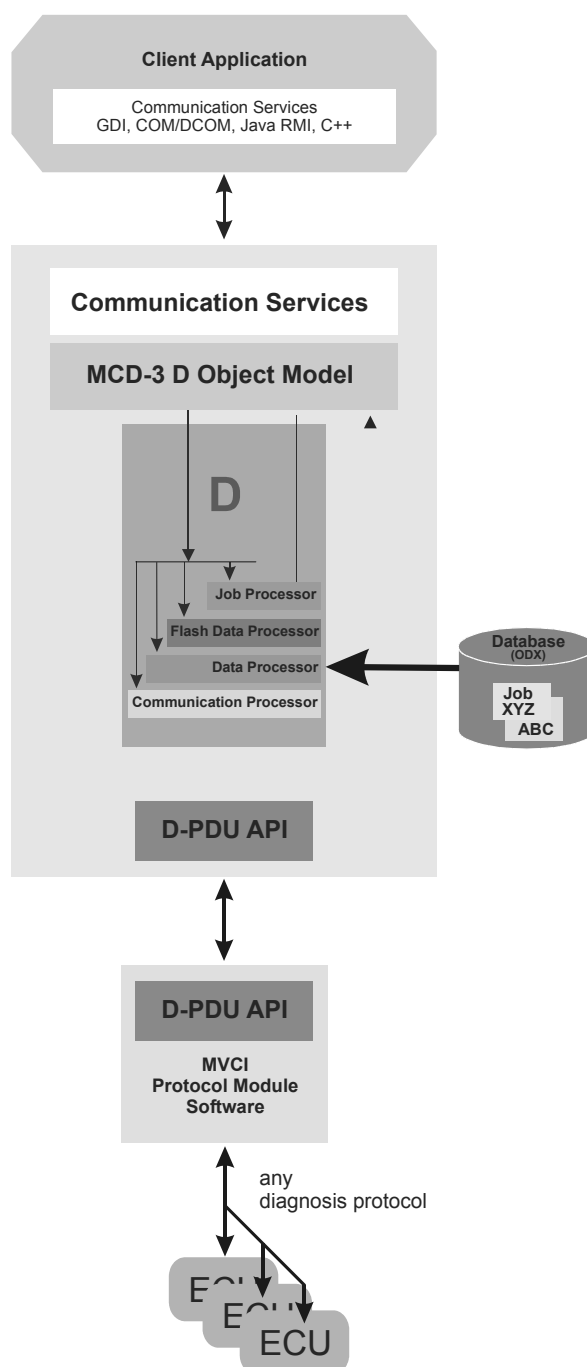


Figure 2 — Architecture of an MVCI diagnostic server

With the help of a server the control units are optimally adapted to the relevant requirements for their use in vehicles. This procedure is often referred to as “Applying”.

The following features (interfaces and methods) are optional:

- MCDDbProjectConfiguration,
- ECU Configuration,
- ECU Reprogramming (Flash),
- DynID,
- Monitoring,
- System properties,
- Function dictionary,
- SubComponents,
- Audiences,
- ECU state,
- Multiple ECU Jobs,
- PDU Time stamps,
- Library concept,
- DoIP,
- PDU API support,
- the concept of System generated Vehicle Information table.

Optional means that the runtime, as well as the database part of the model, do not have to be implemented by a diagnostic server that is omitting the feature in question. When a client application calls a method that is part of an optional feature, the diagnostic server should return an empty collection if the return type of the method is inheriting from `MCDCollection`. Otherwise, such a method call should throw an `MCDSystemException` of type `eSYSTEM_METHOD_NOT_SUPPORTED`. In the case of support of optional features these have to be implemented completely. An overview of methods which belong to optional functionalities can be found in Annex C.

The number of control units applied in vehicles is continuously increasing. The capabilities of the single control unit concerning diagnosis become available for the server by means of control unit description files (Data Description Interface). The control unit description files represent a manufacturer independent data exchange format, which means that any server may handle the data out of a control unit description file. All configuration data of the diagnostic server, the internal data of ECUs or ECU nets and the communication methods for the ECU access are stored in the ODX database. This database is server and operating system independent and therefore allows data to be exchanged between vehicle manufacturers and ECU suppliers.

An application can read out all data from the database that is necessary to drive the MVCI diagnostic server; this means only the MVCI diagnostic server can access the information of the separate control unit description files comprised within one database. With this, at the same time the consistency of the information between AUSY and MVCI diagnostic server is guaranteed.

Also, a decoupling from the used data description exchange format (XML) takes place.

The MVCI diagnostic server has to manage the database and to provide the required and necessary information for the single MVCI diagnostic server objects. The database does not belong to one specific MVCI diagnostic server object, but is available within the whole diagnostic server. The organization of the object or reference allocation is solved implementation specifically by the diagnostic server.

The object model supports Single Client Systems, to provide for a simple use for this most typical and most frequently occurring application case. This means that no client references are included within the single objects. The administration of the client references is done by the diagnostic server and has to be solved implementation specifically.

The object model has been designed in a technology and programming language independent way. It may be used remotely as well as locally.

The object model of MVCI diagnostic server enables the linking of MVCI diagnostic servers to automation systems. The objective of this linking is the remote control of the MVCI diagnostic server in test stands. By means of the object model, the functionality, which means the interfaces with accompanying methods, are standardized. The communication has to be realized via the particular implementation of the object model for the used platform, programming language and linking mechanisms.

Among others the following are realized:

- ASAM GDI,
- JAVA,
- COM/DCOM and
- C++.

The necessary specifications for this will be described and published in separate documents. For this process design patterns and mapping rules are defined and published.

All other specifications will be set up and realized implementation specifically by the respective system provider.

Within the function block diagnostic a breaking down into characteristic sub tasks will take place, which are shown in the following:

The Communication Processor is responsible for generating and analysing request and response telegrams to ECUs. This processor handles all protocol-specific tasks like timings, creation of protocol headers and checksums, etc. Diagnostic protocol specification is (at the moment) not a task of ASAM, because this is covered by ISO activities according to ISO 14230-3 KWP 2000, ISO 14229 UDS and ISO 15765 DoCAN. Nevertheless, the communication processor shall be parameterized via Communication Parameters. The Communication Processor is an interface (the only one) to the ECU.

The Data Processor is responsible for the supply of parameters and results on a physical level. By means of the Data Processor all necessary information is fetched from the database. Additionally, the Data Processor converts ECU answers from hexadecimal representation into a physical or any text representation and vice versa. The Data Processor is an interface (the only one) to the ODX database and offers an ODX library to the Job Processor. The Data processor also handles Jobs, as they are stored in the ODX database.

The Flash Data Processor is responsible for the loading of programs and data in ECUs. The flash data is part of the database. The Flash Data Processor provides access to the ASAM MCD2-ECU-MEM which contains all information about physical/logical data-/code-layout and possible combinations of data and code segments and more. The Flash Data Processor is an interface (the only one) to the flash data and offers a flash library (flash object) to the Job Processor.

The Job Processor is responsible for the execution of service sequences and only uses objects of the ASAM MVCI diagnostic server API. Via the Job Processor all processors may be accessed. The Jobs are part of the database. The Job Processor is based on the ODX format. The Job Processor provides several libraries for standardized access to the Communication Processor, Data Processor, Flash Data Processor and to the Job Processor itself. The Job Processor uses the same objects to interact with the different Processors like the ASAM MVCI diagnostic server API to insure consistency between ASAM MVCI diagnostic server API and Job Processor. The Job Processor gets its code to be executed from the Data Processor. The Data Processor itself reads the ODX database file.

7 Diagnostic server

7.1 MCD system object

The server interface is a client's first access point to the MVCI diagnostic server. From this every interface is reachable. Each client gets its own `MCDSystem` object (implements the `MCDSystem` interface) from the MVCI diagnostic server. But all clients work on the same project and database. The project has the connection to a special part of the whole database and this part will be made available after selecting the project. The selection of another project at the same time is not allowed and will throw an exception.

Figure 3 shows the system scheduling.

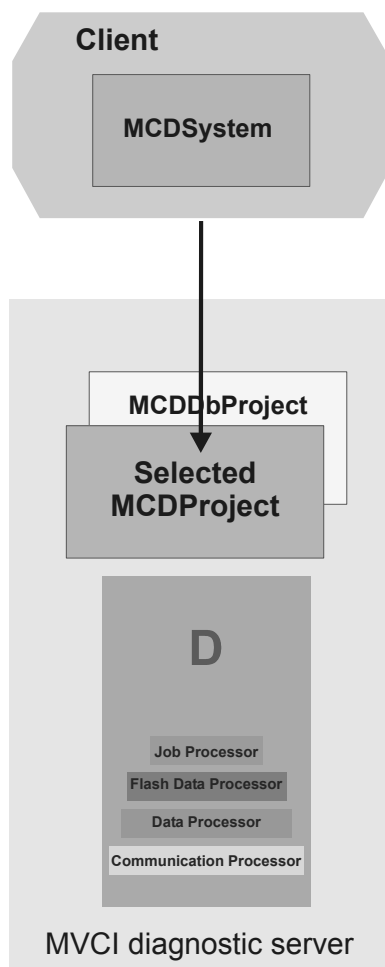


Figure 3 — System scheduling

The diagrams specified in this part of ISO 22900 always refer to the representation within the client and are not designed for MultiClient scenarios.

7.2 Description of terms

7.2.1 General

This section describes the most important terms in more detail. A brief definition is included in 3.1. For each term which is directly related to an ISO 22901-1 ODX element, the corresponding ODX element name is given in parentheses.

7.2.2 Access key (AccessKey)

By means of the AccessKey, the access position within the inheritance hierarchy of the ODX diag layers is identified.

One AccessKey element is composed of the type information [ElementIdentifier] embedded in square brackets followed by the short name of the element instance. That is, the AccessKey is a sequence of tuples of ElementIdentifier and short name. The allowed combinations of ElementIdentifiers are defined by the Locations. AccessKeys are unique within one database.

Table 2 defines the ElementIdentifiers.

Table 2 — ElementIdentifiers

ElementIdentifiers
[MultipleEcuJob]
[Protocol]
[FunctionalGroup]
[EcuBaseVariant]
[EcuVariant]

For every element accessed via an AccessKey there is a LongName and a Description. LongName and Description shall be in UNICODE.

7.2.3 Functional Class (FUNCTIONAL-CLASS)

Functional classes are groups of services (freely definable). A service can be part of multiple functional classes but can have only one semantics. A functional class is an arbitrary, user definable group of services.

7.2.4 Job (SINGLE-ECU-JOB, MULTIPLE-ECU-JOB)

Sequence of diagnostic services and other jobs with control flow inside job, based on received results. Use cases for jobs are ECU (re)programming, Encryption of seed key algorithm and gateway tests.

7.2.5 Location

A Location represents a hierarchical level for diagnostic Services.

The following locations are permitted:

- [D] Multiple ECU Job,
- [D] Protocol,
- [D] Functional Group,
- [D] ECU Base Variant,
- [D] ECU Variant.

The location is the access point to data base specific definitions (meta information) , e.g. available Services, DiagComPrimitives, CompuMethods.

Figure 4 shows the location hierarchy of ASAM MCD database.

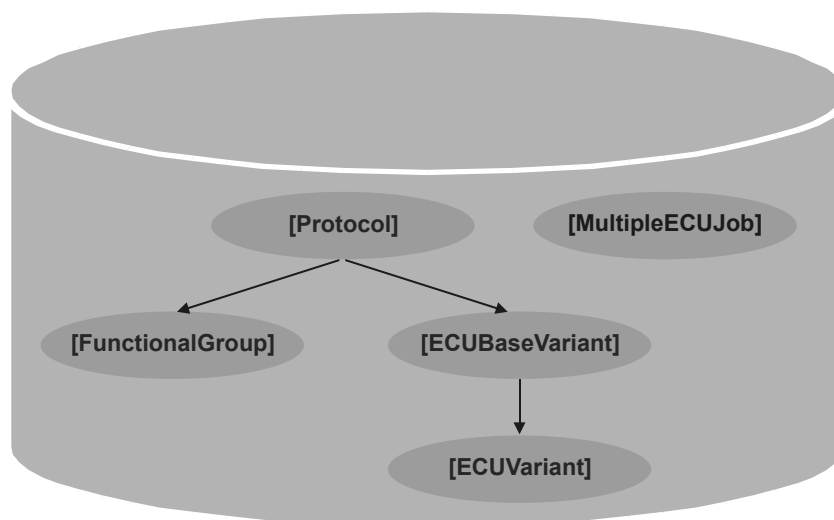


Figure 4 — Location hierarchy of ASAM MCD database

Each Location is addressed by means of an AccesKey. The following AccessKeys of possible Locations in the hierarchical system are allowed:

- [Protocol]Instancename
- [Protocol]Instancename.[FunctionalGroup]Instancename
- [Protocol]Instancename.[EcuBaseVariant]Instancename
- [Protocol]Instancename.[EcuBaseVariant]Instancename.[EcuVariant]Instancename
- [MultipleEcuJob]MultipleECUJob

Figure 5 shows the AccessKey example.

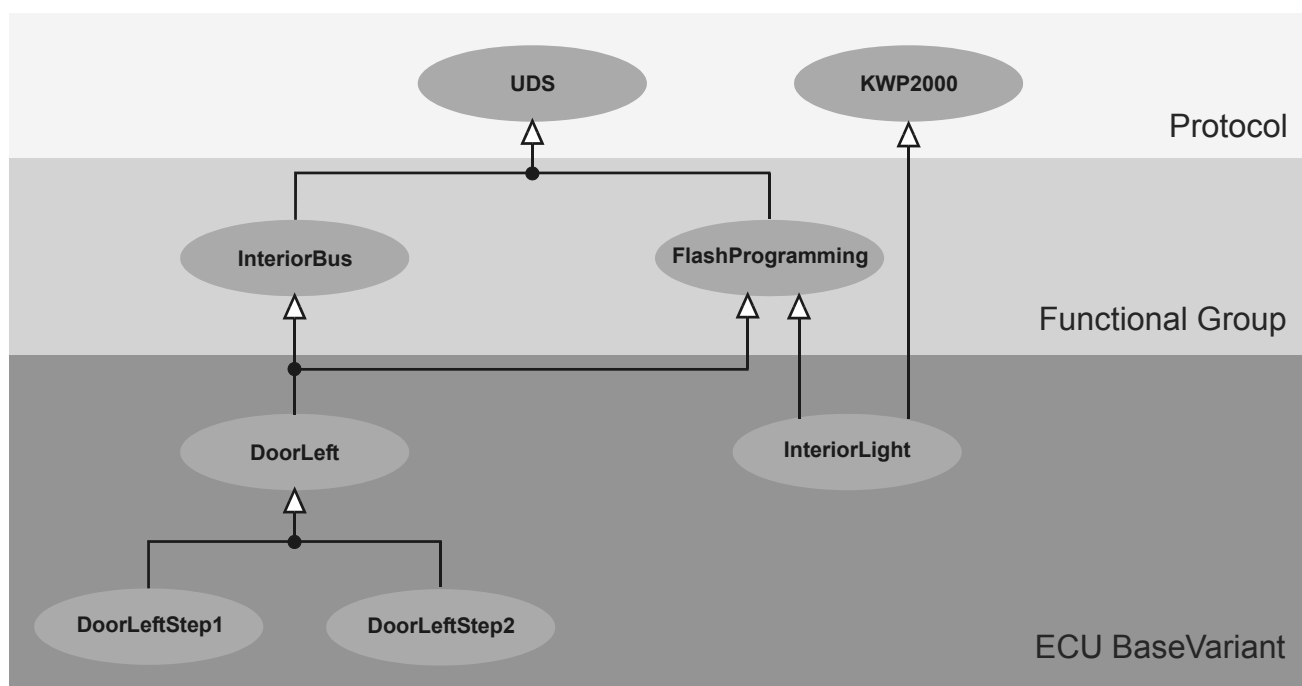


Figure 5 — AccessKey example

Resulting AccessKeys:

- [Protocol]UDS,
- [Protocol]UDS.[FunctionalGroup]InteriorBus,
- [Protocol]UDS.[FunctionalGroup]FlashProgramming,
- [Protocol]UDS.[EcuBaseVariant]DoorLeft,
- [Protocol]UDS.[EcuBaseVariant]DoorLeft.[EcuVariant]DoorLeftStep1,
- [Protocol]UDS.[EcuBaseVariant]DoorLeft.[EcuVariant]DoorLeftStep2,
- [Protocol]UDS.[EcuBaseVariant]InteriorLight,
- [Protocol]KWP2000.[EcuBaseVariant]InteriorLight,
- [Protocol]KWP2000;

7.2.6 Logical Link (LOGICAL-LINK)

A Logical Link is a logical connection to ECUs. Normally this is only one ECU, but in cases of a Functional Group it can be more than one ECU. The Logical Link is represented by a short name. Information about a Logical Link is contained in the Logical Link Table. Elements of this table are the AccessKey and the Physical Vehicle Link or Interface. Logical Links are used to access the same ECU on different ways, or access more than one ECU instance on different links. For more details see Clause 8.

7.2.7 Physical Interface Link

A Physical Interface Link is a logical connection between MVCI diagnostic server and Interface Connector. The Physical Interface Link is represented by a short name. Information about a Physical Interface Link is contained in the Interface Connector Information Table. In this table the description of the Interface connector is included. The available Physical interface links are defined by the available interfaces of a MVCI diagnostic server.

The Interface Connector Information Table has an entry for each Physical Interface Link and one connector for this Link.

The Interface Connector Information Table uses the standardized short name of a Physical Link.

7.2.8 Physical Link

A physical Link is a Physical Vehicle Link connected to a Physical Interface Link, so it is the connection from the interface of the MVCI diagnostic server to the ECU in the vehicle.

Figure 6 shows the overall scheme between different links and tables in D.

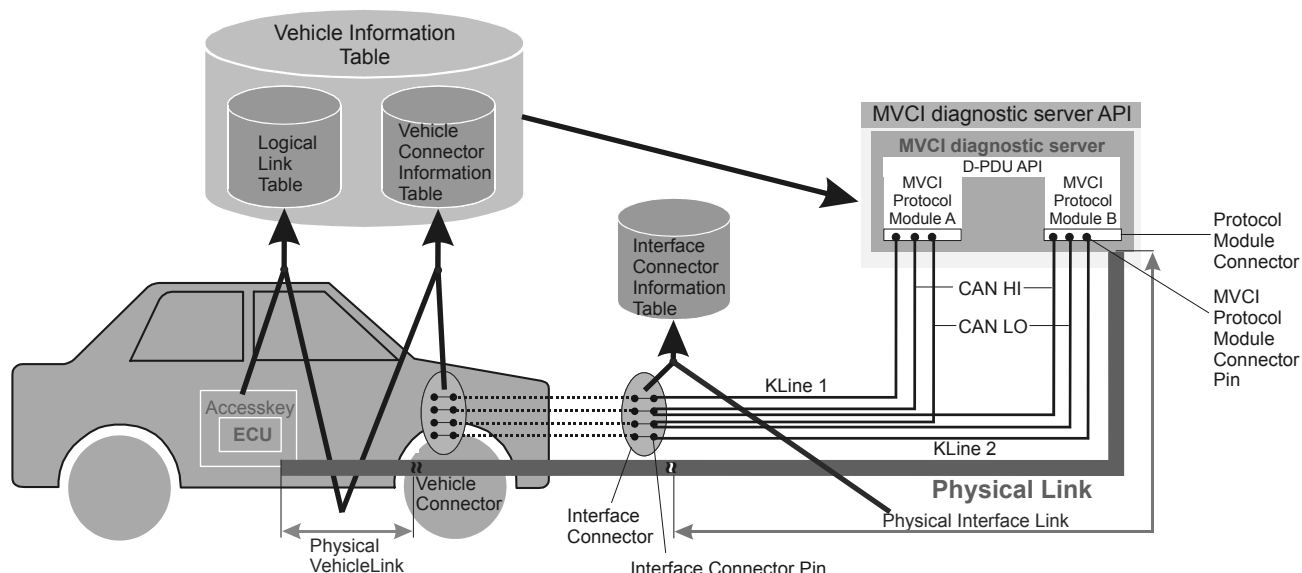


Figure 6 — Overall scheme between different links and tables in D

The pins of the Vehicle Connector are connected to identical pins of the Interface connector.

7.2.9 Physical Vehicle Link (PHYSICAL-VEHICLE-LINK)

The physical vehicle link describes the unique bus system in a vehicle, so it is the connection between the vehicle connector and the ECU. The physical vehicle link is represented by a short name. Information about a vehicle link is contained in the Vehicle Connector Information Table. In this table the Vehicle Connector Information is included.

The Vehicle Connector Information Table has an entry for each Physical Vehicle Link and one or more Vehicle Connectors (Pins) for this Link.

The available Physical vehicle links are defined by the vehicle.

7.2.10 Project

A project is a logical grouping for defined test installations selected by the user. Within a project, all information necessary for a test installation has to be contained. It is only permitted to work within one project, which has to be considered for the logical grouping (e.g. two model series within one project). It is for this reason that the project tuple is not part of the AccessKey.

At project level, the forming of manufacturer-specific hierarchies is possible, as for this level no standardization takes place.

Typically, a project contains:

- all static database information (Diagnosis Services, ...),
- jobs,
- flash containers, and
- configuration files.

Within the MVCI diagnostic server, first the selection of a project out of the list of the existing projects takes place, before any database information can be accessed. Because of this mechanism only one project can be active.

The database of a project is not restricted to a physical file.

7.3 Version information retrieval

The method `MCDSystem::getASAMMCDVersion():MCDVersion` returns the ASAM release number of the interface. For this specification it is the major value 3 and the minor value 0 defined.

The method `MCDSystem::getVersion():MCDVersion` returns the tool version.

The technology version number is available via the interface
`MCDSystem::getInterfaceNumber():A_UINT32`.

The version number of the model (specification) and the version number of the reference implementations are kept synchronous. The so-called “interface number (IFN)” is used as a build number for the reference implementations. Each reference implementation maintains its own interface number. This number will be incremented continuously within a minor version. Increments are applied whenever a new reference implementation is generated and shipped. The “interface number” is reset to zero for each new minor or major version. Please, note that this number is completely independent from the version numbers. The technology version number is incremented separately for each technology.

7.4 States of the MCD system

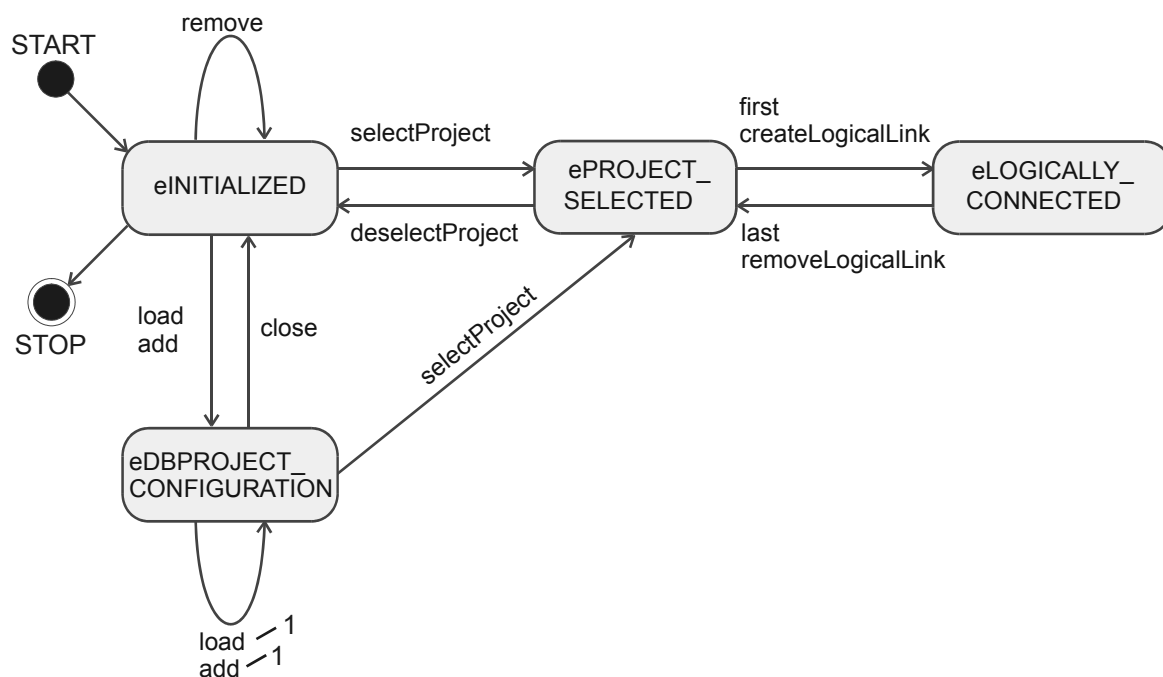
Within the state `eINITIALIZED` the `MCDSystem` object is transmitted to the client by the MVCI diagnostic server. Within this state, the database project descriptions can be listed and general system initialisations can be done.

By selecting the project to be worked on, the system takes the state `ePROJECT_SELECTED`. Within this state the database can be polled for information, but no communication to the hardware is possible.

As soon as the first Logical Link has been created, the `MCDSystem` object takes the state `eLOGICALLY_CONNECTED` and can execute the communication by means of the created Logical Links. If the last Logical Link has been removed from the runtime project, the system automatically takes the state `ePROJECT_SELECTED` again. By means of `deselectProject()` the `MCDSystem` Object is set to the initial state `eINITIALIZED` again.

The diagnostic server changes the state to `eDBPROJECT_CONFIGURATION` by loading a database project at object `MCDDbProjectConfiguration`. Searching in the database is possible in this state. Configuration (add and remove elements) of the database can also be done in this state. ECU-MEMs can be added. ECU-MEMs should be temporarily loaded to an `MCDDbProject` or permanently added to a project configuration. ECU-MEMs are necessary for flashing.

Figure 7 shows the system state transitions.



Key

1 'load' or 'add' of DbProject includes an implicit close of actual opened DbProject.

Figure 7 — System state transitions

It has to be distinguished between the states of clients and the central state of the server (internal MCDSsystem).

At the server side there may only be one state (**eINITIALIZED**, **eDBPROJECT_CONFIGURATION**, **ePROJECT_SELECTED**, **eLOGICALLY_CONNECTED**). As soon as a client has successfully called **selectProject**, the internal MCDSsystem transits to the state **ePROJECT_SELECTED**.

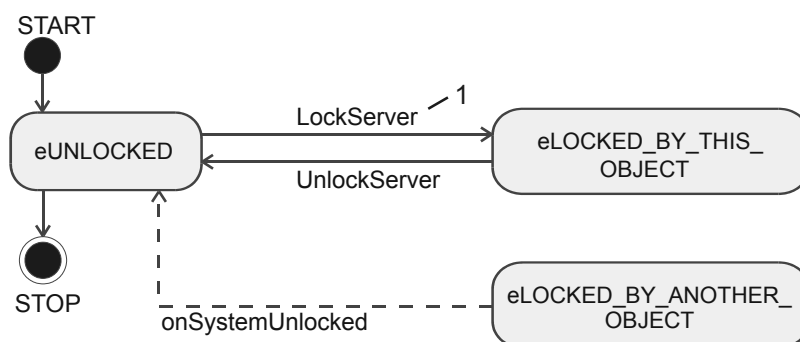
Table 3 defines the system states.

Table 3 — System states

System State	MCDDbVehicleInformation selected	MCDSystem			MCDDbProject Configuration				MCDProject				MCDDb Project
		selectProject	selectProjectByName	deselectProject	load	close	getAdditionalEcuMemNames	getActiveDbProject	selectDbVehicleInformation	selectDbVehicleInformationByName	deselectVehicleInformation	CreateLogicalLink...	
eINITIALIZED	no	X	X		X				state does not occur				
ePROJECT_SELECTED	yes							X	X ^a	X ^a	X	X	
	no			X				X	X	X			
eLOGICALLY_CONNECTED	yes							X	X ^a	X ^a		X	
	no							X	state does not occur				
eDB_PROJECT_CONFIGURATION	no	X	X		X	X	X	X	state does not occur				X

^a This is a valid action in cases where different vehicle information has not already been selected, no state transition; otherwise it is an invalid action and an exception will be thrown. See corresponding method definition.

Figure 8 shows the system lock states.



Key

- 1 this transition can only be done in system state eINITIALIZED

Figure 8 — System lock states

Each `MCDSystem` object may be locked for exclusive use of the whole MVCI diagnostic server by the client application. The locking of the `MCDSystem` may only take place within the `eINITIALIZED` state and refers to the internal server `MCDSystem` object. No other client can then communicate with the diagnostic server. Within the states `ePROJECT_SELECTED`, `eLOGICALLY_CONNECTED` and `ePROJECT_CONFIGURATION` this is not allowed. If an `MCDSystem` Object is exclusive locked, the locking client exclusively accesses the MVCI diagnostic server. All clients will be informed by system event of locking and unlocking.

`MCDProject/MCDSystem LongName` and `Description` are server-specific values and can be empty. An exception will not be used.

7.5 State changes

The behaviour for methods which should change a state should always be the same, i.e. a call for transition to a state which is available before the call throws an exception without a state change.

This behaviour is used by `MCDSystem`, `MCDLogicalLink`, `MCDDiagComPrimitive`.

If any exception occurs during a state changing operation, the state shall not be changed.

7.6 Project configuration

Within the state `ePROJECT_CONFIGURATION` of the `MCDSystem` a database project may be loaded and browsed. Runtime objects may not be created within this state. Once the transition to this state has taken place, this state can only be left by closing down the database project or selecting the corresponding runtime project. This selection implicates the saving of all changes which were made. In this state the used database project can be changed by loading or creating another database project; the so far used project will be implicitly saved if changes were made.

The Method `MCDSystem:getDbProjectConfiguration` returns a `MCDProjectConfiguration` Object, which can be used for

- browsing (without existing `RunTimeObjects`, MCD: `MCDProjectConfiguration:load`) and
- modification (D: `MCDDbProject:loadNewECUMem`) of `DbProjects`.

The state transition from `eINITIALIZED` to `ePROJECT_CONFIGURATION` takes place only after a Client opens a `DbProject` by means of `add/load`.

The Methods

- `MCDSystem::selectProject()`,
- `MCDDbProject::loadNewEcuMem()` and
- `MCDDbProjectConfiguration::getAdditionalEcuMemNames`

check the consistence of data read from the database. An error of type `eDB_INCONSISTENT_DATABASE` should be thrown when inconsistent data has been identified.

Figure 9 shows the project configuration.

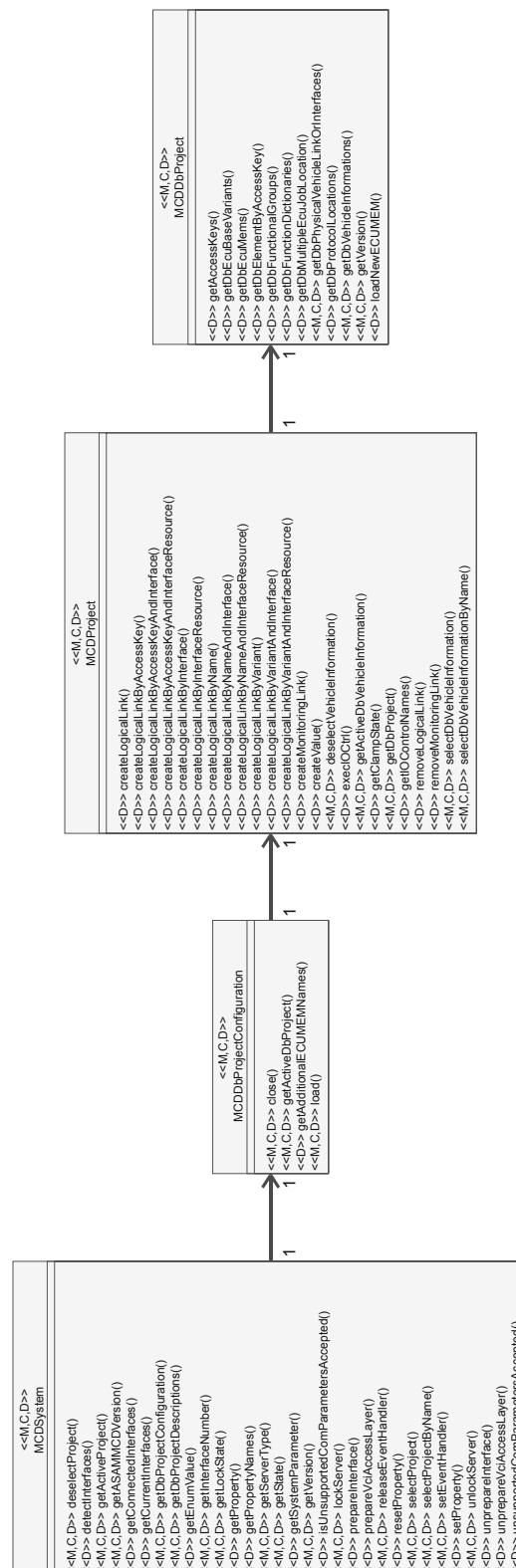


Figure 9 — Project configuration

7.7 Interface structure of server API

7.7.1 Hierarchical model overview

Figure 10 shows the hierarchical model of a diagnostic server part I.

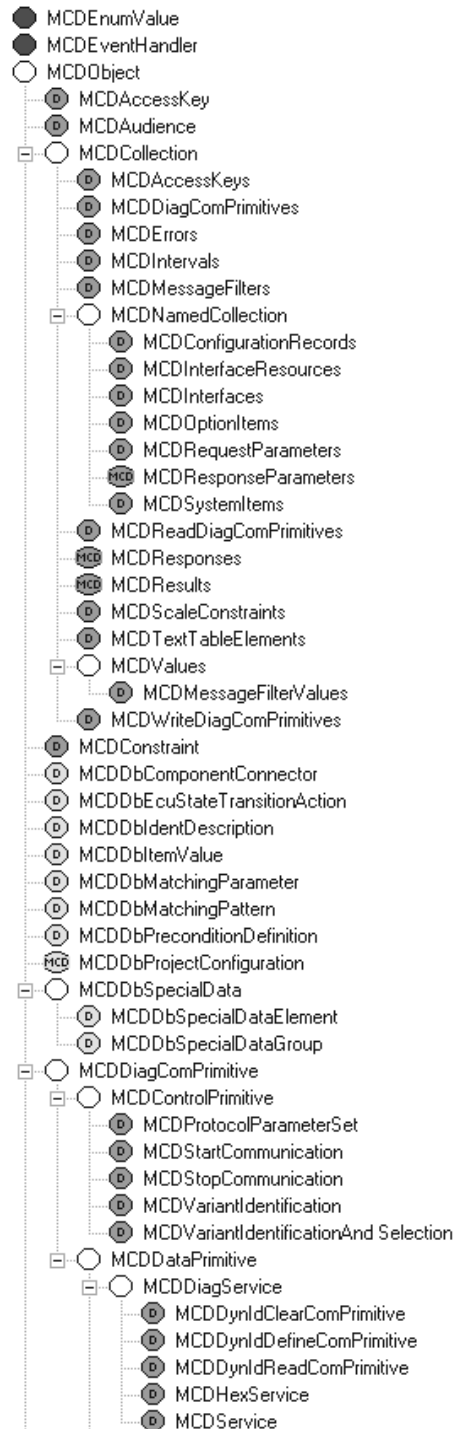


Figure 10 — Hierarchical model of a diagnostic server part I

Figure 11 shows the hierarchical model of a diagnostic server part II.

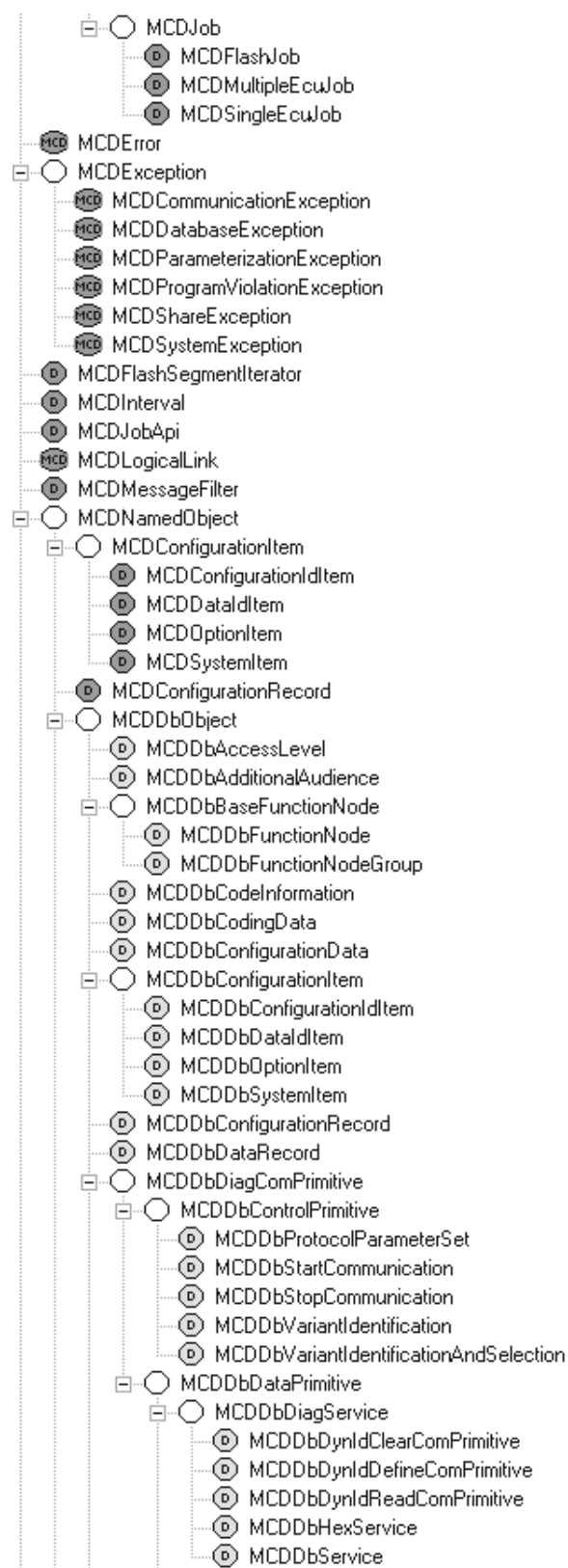


Figure 11 — Hierarchical model of a diagnostic server part II

Figure 12 shows the hierarchical model of a diagnostic server part III.

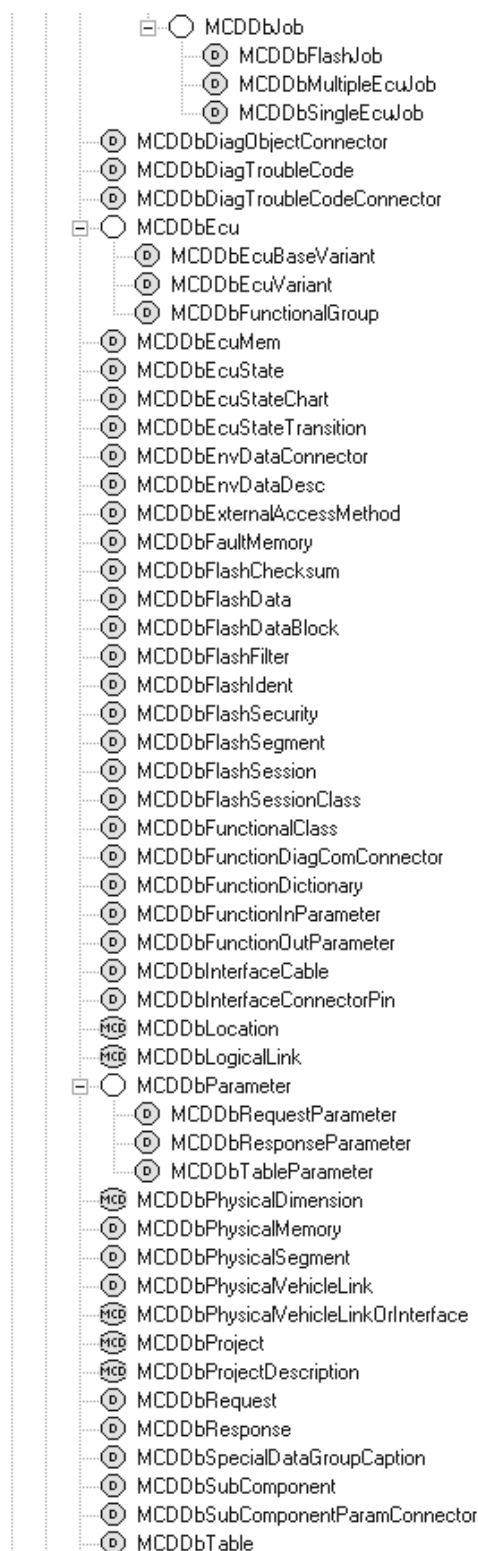


Figure 12 — Hierarchical model of a diagnostic server part III

Figure 13 shows the hierarchical model of a diagnostic server part IV.

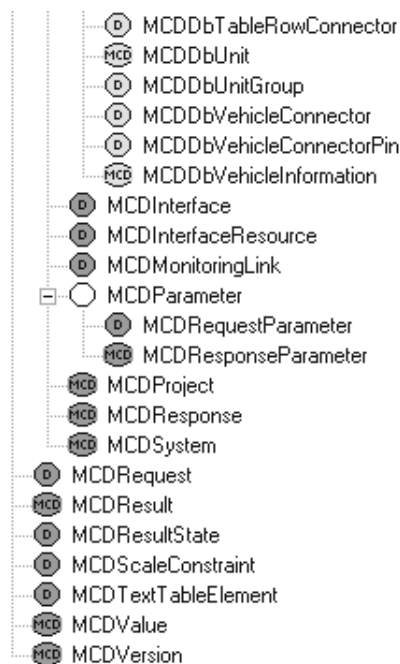


Figure 13 — Hierarchical model of a diagnostic server part IV

As can be seen, all database interfaces are derived from `MCDDbObject`, and all database Collections from `MCDNamedCollection`. The Runtime interfaces are usually derived directly from `MCDObject`. Collections, Exceptions and NamedObjects each form basic classes for a further detailed structuring. Apart from `MCDSystem` and `MCDProject` also all database interfaces belong to the NamedObjects.

The `MCDEventHandler` represents an exception concerning this classification and is not derived from `MCDObject`. It is used as means of communication of the diagnostic server with the Client and provides for Callback methods for the Event sending.

The `MCDEnumValue` is the super class of all classes representing enumerations in this API. It provides two methods to translate between the object and the string representation of the enumeration and vice versa. So string need not be delivered at the API if there is no need to, and the server can implement enumerations and their representation very efficiently with a hash map.

7.7.2 Database and runtime side

7.7.2.1 Basics

Within the whole model there is a separation into database (yellow in black/white in the light) and runtime (green in black/white in the dark). The database part is invariable and is used as the basis for the creation of most runtime objects. Database objects/classes contain the sub string "Db" in their names. This sub string signals that the corresponding class is associated with information originating from a data file (ODX, Flash Data, etc.). "Db" objects cannot be modified by the client application. A "Db" object is static, has no status and exists only once.

All access is done by methods of the interfaces. There are no attributes except in the classes which should behave like an enumeration.

The object oriented model of the API has been separated in communication (runtime) and database part, to provide for a data set with minimum redundancy and for a very slim communication layer.

The database part contains all information which could be detected from ODX, as well as environmental settings relevant for the API. Within the database part, all created objects exist only once to avoid unnecessary redundancies. That means that the respective communication objects only have a reference to its database objects, so that the information does not have to be duplicated.

The database objects are static and do not have an internal status. The once-created information content may not be modified by the Client or MVCI diagnostic server. As soon as the project has been selected, all information is available. It does not have to be existing within objects at this moment, but the option to access it has to be guaranteed within an acceptable period of time.

The access to database objects is realized optionally via the short name of the object or via iteration within the database structure.

The diagnostic server should generate suitable shortnames, as shortnames are not obtainable from ODX. These shortnames should have a prefix "#RtGen_" that uniquely classifies them as generated.

Figure 14 shows the project associations.

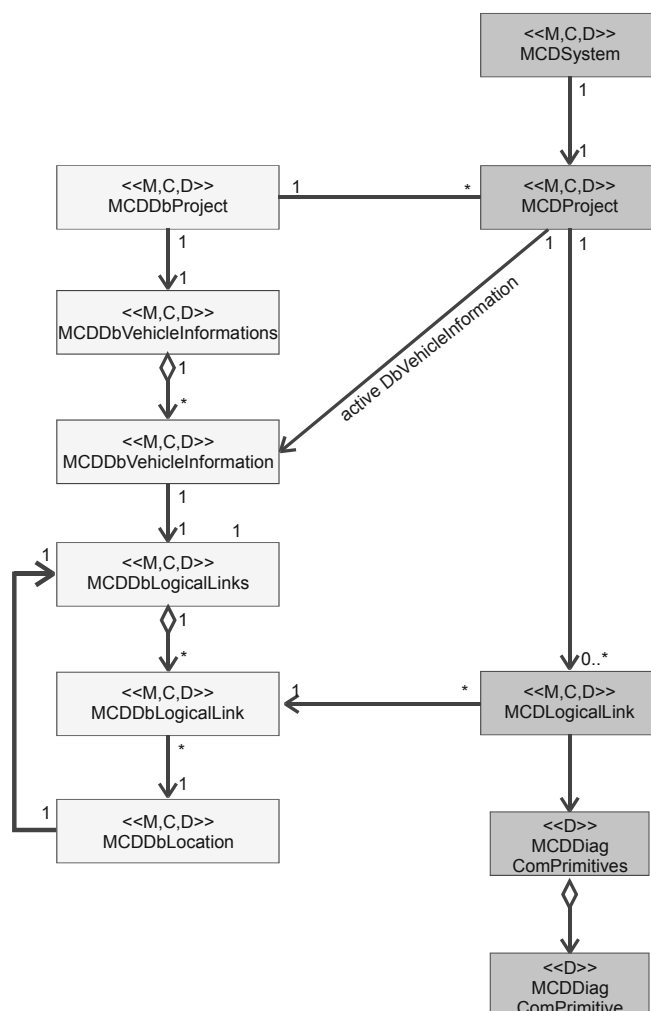


Figure 14 — Project associations

The entry object is the MCDSystem, a runtime object. From this a runtime project will be created on the basis of its database project. Selecting the Vehicle Information Table makes access to all database Logical Links possible. This access is necessary to create the runtime Logical Links from which all runtime activities can be done.

The communication part contains all interfaces via which the communication with the ECUs takes place. The majority of the runtime objects are created on the basis of database objects and thus have a direct reference to these. Some objects implementing interfaces of the communication part may be instanced only once (these are unique objects/singletons), others may be multiply instanced (multiple objects). The multiply instanced runtime objects of an identical database object may be parameterised differently.

The runtime objects may be executed (executable), have an internal status and return results.

Table 4 defines the overview RunTimeObjects.

Table 4 — Overview RunTimeObjects

Unique runtime objects /singletons	Multiple runtime objects
MCDSystem	MCDAccessKey
MCDProject	All derived from MCDDiagComPrimitive
MCDJobApi	MCDError
---	MCDException
---	All derived from MCDException
---	MCDLogicalLink
---	MCDTextTableElement
---	All derived from MCDParameter
---	MCDResponse
---	MCDResult
---	MCDResultState
---	MCDValue
---	MCDRequest
---	All derived from MCDDbObject

7.7.2.2 Structure of the database

The database which can be polled via the Object Model is structured in such a way that each database element is available only once. Because of this, redundancies are avoided. Connections between the single database elements can be queried via methods. These methods return instances or references.

All database interfaces have methods for the polling of ShortName, LongName and Description.

A more detailed description can be found with the description of the hierarchic model and the Entity Relationship diagrams.

Database information may be accessed immediately in state eDBPROJECT_CONFIGURATION or after the instancing of the RunTime Project.

If there are optional elements not filled in ODX no exception shall be used.

Within the database part templates for all DiagComPrimitives and their parameters, information for ECU (re)programming and the information concerning the ECUs and the Vehicle Connector Table are deposited.

The meta information of the different communication primitives may be polled by the interfaces derived from MCDDbDiagComPrimitive. The objects filled with information from the ODX at the moment of execution are used as a template for the runtime communication primitives.

7.7.2.3 Structure of the runtime side

On the runtime side the communication actions are carried out by means of `DiagComPrimitives`, which also include the `Services` and all three kinds of jobs (`FlashJob`, `SingleEcuJob` and `MultipleEcuJob`). A specific feature of the Diagnostic part is the use of `RequestParameters` as input for the `DiagComPrimitives` and the `ResultState` as `Snapshot` for the state of the `Result`.

`MCDDiagComPrimitive` is the superior class for all runtime communication primitives. Communication Primitives represent, for example, state transitions of the Logical Link (`MCDStartCommunication`) and real communication objects such as, for example, `Services` and `Jobs`. `Jobs` and `Services` are derived from `DataPrimitives` and thus they have to be handled alike.

7.7.3 Parent functionality

The parent functionality in objects `MCDError`, `MCDResponseParameter`, `MCDResponseParameters`, `MCDResponse`, `MCDResponses` and `MCDDiagComPrimitive` of the model should give the user the possibility to go through the model not only in top down but also in button up direction. The parent for the object which has the functionality is always the logical parent, e.g. for the `DiagComPrimitive`, the parent is the `LogicalLink`.

Table 5 defines the delivered parent.

Table 5 — Get parent functionality in diagnostic

Class	delivers
<code>MCDResponseParameter</code>	<code>MCDResponseParameters</code>
<code>MCDResponseParameters</code>	<code>MCDResponse</code> or <code>MCDResponseParameters</code>
<code>MCDResponse</code>	<code>MCDResponses</code>
<code>MCDResponses</code>	<code>MCDResult</code>
<code>MCDDiagComPrimitive</code>	<code>MCDLogicalLink</code>
<code>MCDError</code>	<code>MCDResponseParameter</code> or <code>MCDResponse</code> or <code>MCDResult</code>

7.7.4 Entity Relationship Diagrams

7.7.4.1 Objective

Within the Entity Relationship diagrams (ERD) the relations between the single interfaces or the objects to be created from these are visualized.

7.7.4.2 Relation between Vehicle Connector Information Table and Logical Link Table

Figure 15 shows the DbProject and relation to Logical Link and Vehicle Information Table.

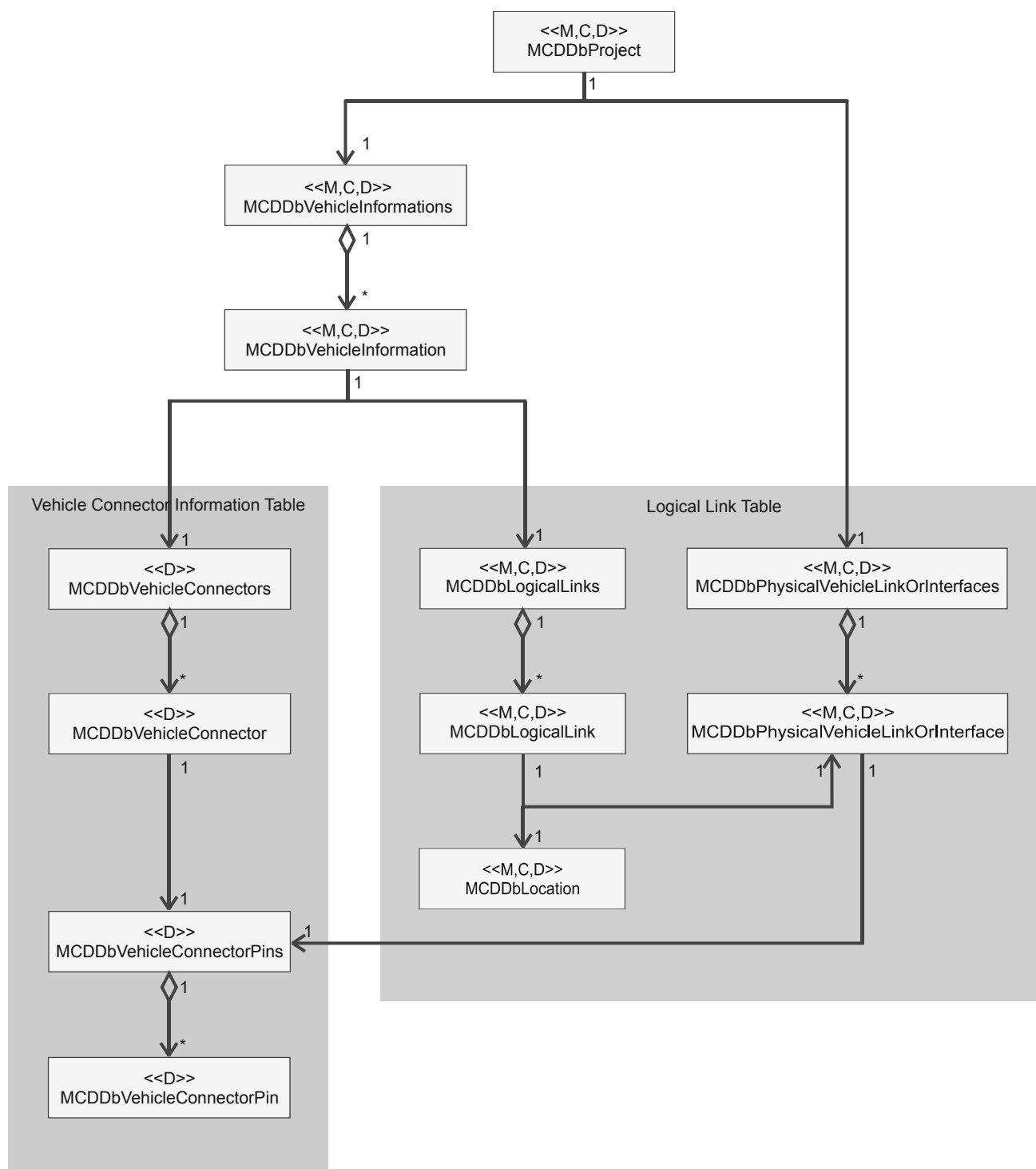


Figure 15 — DbProject and relation to Logical Link and Vehicle Information Table

Within this ERD the relations between the database interface of the project and the ECUs and Vehicle Information contained in the project are shown. The following is shown: which information may be listed in collections and in which number the respective objects should be existing within the database.

In the Logical Link Table each ECU will occur in several Locations if there are different paths to this ECU.

Exactly one Location and one Physical Vehicle Link belongs to each Logical Link, according to the entry in the Logical Link Table. The Vehicle Connectors and the ConnectorPins according to the Physical Vehicle Link can be listed. Within the Vehicle Information Table the VehicleConnectors with their pin-assignments and the respective Logical Links are listed. As far as the Logical Link is concerned, it references the Physical Vehicle Link which describes the connection between pins and ECU.

Besides the VehicleInformationTables that are defined in ODX, an MCD System may optionally support the concept of runtime generated Vehicle Information Tables.

A runtime generated Vehicle Information Table shall be named with prefix “#RtGen_” and is generated internally by the server. This VIT can be used by a client application like any other VIT. Which DbLogicalLinks will be contained in the VIT is server specific. It could be, for example, one of the following:

- The VIT contains the collection of all DbLogicalLink which are defined in single VITs (e.g. #RtGen_SmartVIT).
- The VIT contains the collection of all DbLogicalLinks which can be generated out of the DbLocations of the project (e.g. #RtGen_DefaultVIT).

7.7.4.3 ERD DbLocation

Within this ERD the associations between a DbLocation, which represents an ECU, and the related interfaces are shown. At the DbLocation a number of Collections may be polled, which are filled with items.

Figure 16 shows the location association for D.

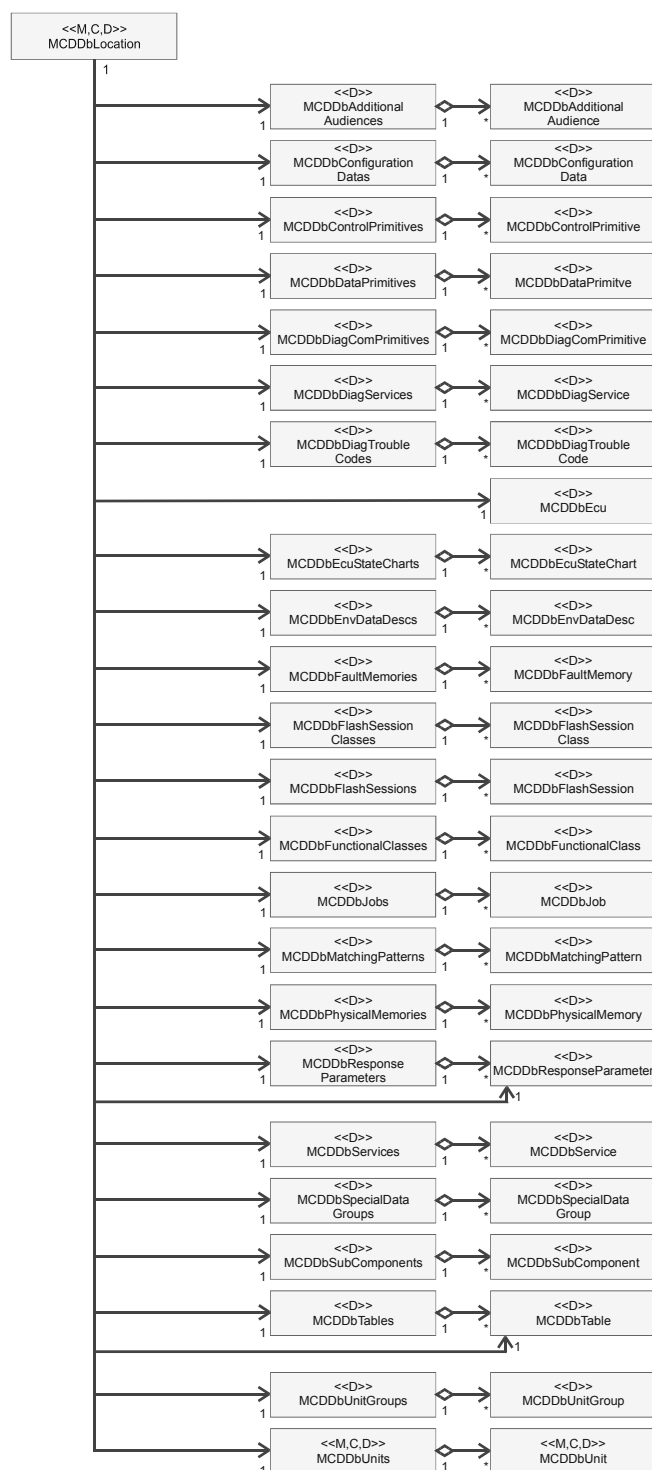


Figure 16 — Location association for D

These associations are created from the information out of ODX description files, whereas `MCDDbLocation` corresponds with ECU.

At this `MCDDbLocation` there is a link to the DbECU Interface and Collections for the different database objects that can be assigned to an ECU. Every Collection contains 0..n objects of the same class or children of a common super class.

7.7.4.4 ERD Logical Link and DiagComPrimitives

Figure 17 shows the Logical Link associations in function block D.



Figure 17 — Logical Link associations in function block D

Within this Entity Relationship Diagram, the relation between a Logical Link and the Communication Primitives (called DiagComPrimitive) which can be created within it, as well as the relations between the runtime and the database part of the communication primitives, are shown.

For each Logical Link 0 to n different DiagComPrimitives can be created at the same time.

The following runtime ControlPrimitives may be created only once per Logical Link:

- MCDStartCommunication,
- MCDStopCommunication,
- MCDVariantIdentification,
- MCDVariantIdentificationAndSelection.

All other DiagComPrimitives may be created as often as one likes and exist in parallel. They may be parameterised differently.

Each Runtime DiagComPrimitive of a type uses exactly the database DiagComPrimitive of the same type of this Location. It is not necessary to create a runtime DiagComPrimitive within the Logical Link for each database DiagComPrimitive.

7.7.4.5 ERD Request and Response Parameter associations

Figure 18 shows the request parameter associations.

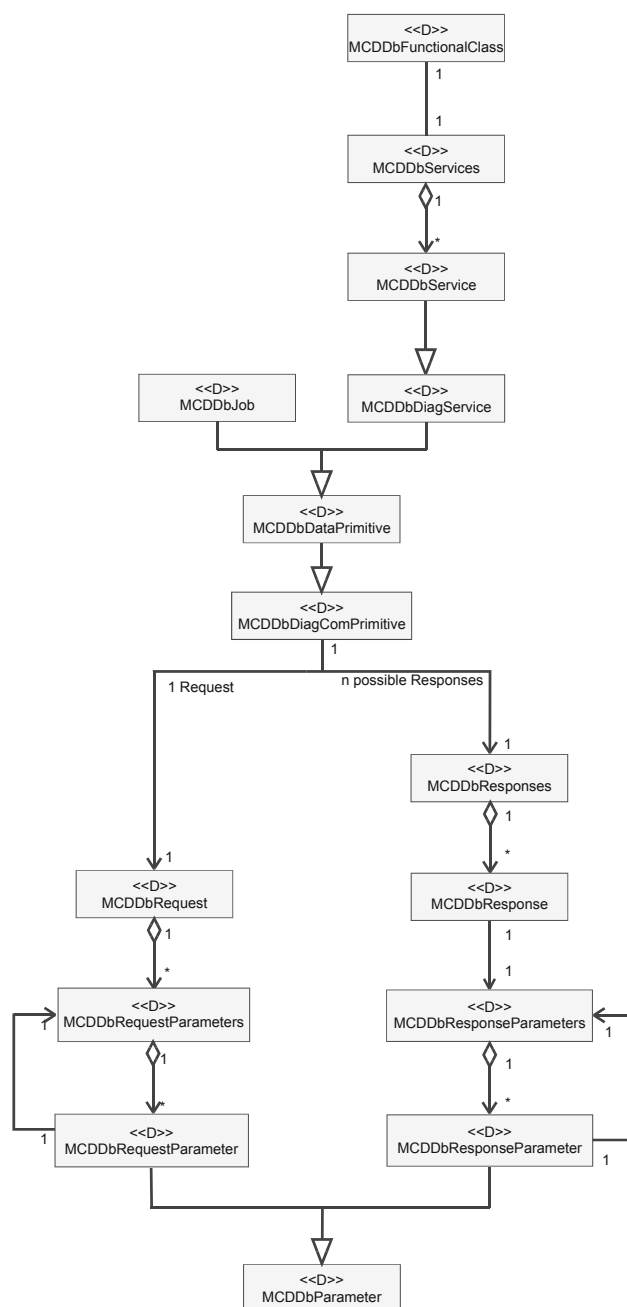


Figure 18 — Request parameter associations

Within this ERD the relations concerning the kinds of parameters of the object model are shown.

Within this ERD the database side is considered in detail. Within the upper part it is shown that `MCDDbFunctionalClass` contains a collection of Services and that Services belong to the `MCDDbDataComPrimitives`. Request Parameters always belong to each `DataComPrimitive` and possibly (that means from 0 to as many as one likes) predefined Responses together with the respective Response Parameters, which describe the general structure of the Response. This is illustrated in Figure 81.

Each Request Parameter may contain a Simple DOP or as Complex DOP a STRUCTURE or EndOfPDU, so a nested structure can be built. Response Parameter, however, may build any nested structure out of Complex and Simple DOP.

7.7.4.6 ERD result access

Figure 19 shows the response parameters associations.

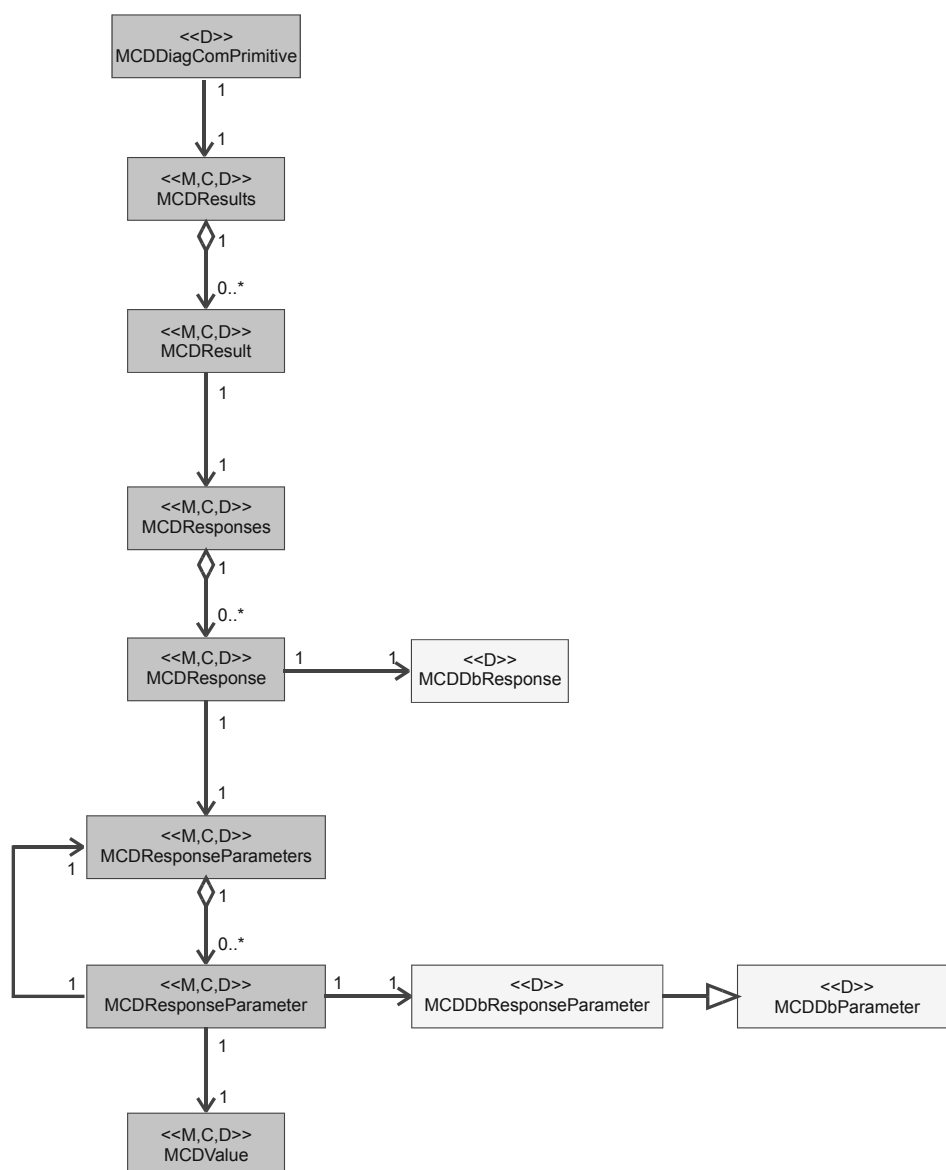


Figure 19 — Response parameters associations

Within this ERD the runtime side of the Result is considered in detail and the relation between DiagComPrimitive, Result, Response and Response Parameter up until the value of the Response Parameter is shown.

This means by calling `fetchResults()` a collection of results is returned. This collection consists of 1..n `MCDResults`. Each `MCDResult` has for each ECU used by the `DiagComPrimitive` one response. The response is described in the database with an `MCDDbResponse`. Each `MCDResponse` has a collection of Response Parameters. In the following the Response Parameters can be structured to build the structure of the return values of one ECU. The structure is given by database template.

Figure 80 illustrates the detailed result structure in the D part.

7.7.4.7 ERD Jobs

Within this Entity Relationship Diagram, the relations of the Job interfaces are shown. MCDDbJob is derived from MCDDbDataPrimitive. MCDDbMultipleEcuJob, MCDDbFlashJob and MCDDbSingleEcuJob are derived from MCDDbJob. A relation to the MCDDbVersion interface exists.

Figure 20 shows the Job associations.

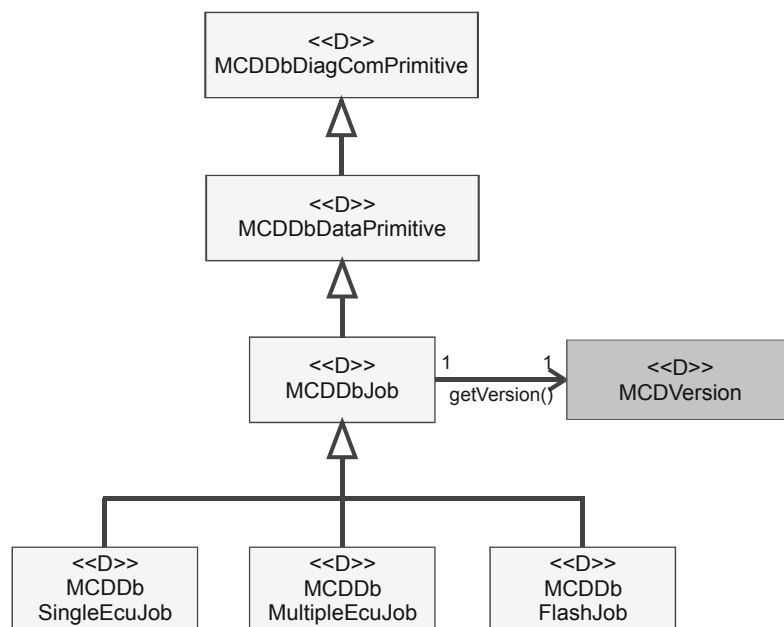


Figure 20 — Job associations

7.7.5 ODX Data Type mapping for database and runtime side

7.7.5.1 Basics

In general, runtime data has to be in conformance with the database template for the respective association. If the data type of the parameter is a complex data type, data sections are repeated.

DbObjects are static and will not be influenced by Run Time side objects. This means that the delivered dbObjects are always the same no matter which access way is used (dbSide or RTSide::getDbObject::getDbParameters).

For the special complex data type eTABLE_ROW the hints are directly included in 8.12.

In general the ODX isVisible attribute is not considered.

The following complex datatypes

- eSTRUCTURE,
- eFIELD, eEND_OF_PDU,
- eMULTIPLEXER, and
- eENVDATA, eENVDATADESC

are shown in more detail.

7.7.5.2 eSTRUCTURE

A parameter of data type eSTRUCTURE consists of 0 to n elements. Each element can be of simple or complex data type. The parameter collection below the complex eSTRUCTURE parameter contains the MCD(Db)Parameter mappings of ODX PARAM elements that are combined in the ODX STRUCTURE.

Parameters of data type eSTRUCTURE might appear in a request structure as well as in a response structure.

ODX-Data (extract) of the DB-Template

```
<DATA-OBJECT-PROP ID="SimpleDOP_ID">
  <SHORT-NAME>SimpleDOP</SHORT-NAME>
  (...)
  <DIAG-CODED-TYPE (...) >
    (...)
  </DIAG-CODED-TYPE>
  <PHYSICAL-TYPE BASE-DATA-TYPE="A_UNICODE2STRING"/>
</DATA-OBJECT-PROP>
<STRUCTURE ID="SimpleSTRUCT_ID">
  <SHORT-NAME>SimpleSTRUCT</SHORT-NAME>
  <PARAMS>
    <PARAM xsi:type="VALUE">
      <SHORT-NAME>Data_A</SHORT-NAME>
      (...)
      <DOP-REF ID-REF="SimpleDOP_ID"/>
    </PARAM>
    <PARAM xsi:type="VALUE">
      <SHORT-NAME>Data_B</SHORT-NAME>
      (...)
      <DOP-REF ID-REF="SimpleDOP_ID"/>
    </PARAM>
  </PARAMS>
</STRUCTURE>
<DIAG-SERVICE ID="ServiceXYZ_ID">
  <SHORT-NAME>ServiceXYZ</SHORT-NAME>
  (...)
  <REQUEST-REF ID-REF="Req_ID"/>
  (...)
</DIAG-SERVICE>
<REQUEST ID="Req_ID">
  <SHORT-NAME>Req</SHORT-NAME>
  (...)
  <PARAMS>
    <PARAM xsi:type="CODED-CONST">
      <SHORT-NAME>SID</SHORT-NAME>
      <DIAG-CODED-TYPE BASE-DATA-TYPE="A_UINT32" (...) >
        (...)
      </DIAG-CODED-TYPE>
```

```

</PARAM>
  <PARAM xsi:type="VALUE">
    <SHORT-NAME>DataItems</SHORT-NAME>
    (...)
    <DOP-REF ID-REF="SimpleSTRUCT_ID"/>
  </PARAM>
</PARAMS>
</REQUEST>

```

Figure 21 shows the datatype eSTRUCTURE at database and runtime side.

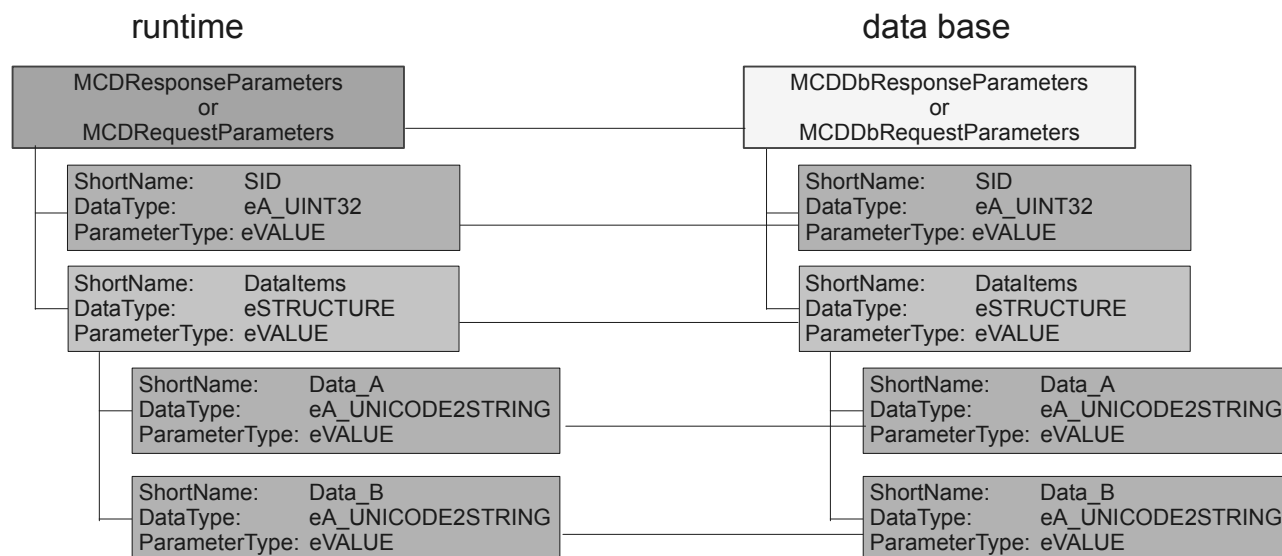


Figure 21 — Datatype eSTRUCTURE at database and runtime side

7.7.5.3 eFIELD

A parameter of data type eFIELD consists of 0 to n elements. Each element contains one element of data type eSTRUCTURE with a short name equal to the SHORT-NAME of the ODX STRUCTURE, this means the STRUCTURE element of ODX will be used independently from the isVisible data. The structure itself consists of arbitrarily simple or complex elements (the actual field content). On database side exactly one field element (eSTRUCTURE with content) is shown. On runtime side the number of included eSTRUCTURE elements depends on how often this structure is contained in the response PDU according to the termination condition of the special field type. Please note that the short names of the field elements will not be unique.

Parameters of type eFIELD appear in response structures only. They are built from ODX DOPs STATIC-FIELD, DYNAMIC-LENGTH-FIELD, DYNAMIC-ENDMARKER-FIELD, and END-OF-PDU-FIELD.

ODX-Data (Extract) of the DB-Template

```

<DATA-OBJECT-PROP ID="SimpleDOP_ID">
  <SHORT-NAME>SimpleDOP</SHORT-NAME>
  (...)
  <DIAG-CODED-TYPE (...)>
    (...)
  </DIAG-CODED-TYPE>
  <PHYSICAL-TYPE BASE-DATA-TYPE="A_UNICODE2STRING"/>
</DATA-OBJECT-PROP>
<STRUCTURE ID="SimpleSTRUCT_ID">
  <SHORT-NAME>SimpleSTRUCT</SHORT-NAME>

```

```
<PARAMS>
  <PARAM xsi:type="VALUE">
    <SHORT-NAME>Data_A</SHORT-NAME>
    (...)
    <DOP-REF ID-REF="SimpleDOP_ID"/>
  </PARAM>
  <PARAM xsi:type="VALUE">
    <SHORT-NAME>Data_B</SHORT-NAME>
    (...)
    <DOP-REF ID-REF="SimpleDOP_ID"/>
  </PARAM>
</PARAMS>
</STRUCTURE>
<DYNAMIC-ENDMARKER-FIELD ID="DYN_EM_FIELD_ID">
  <SHORT-NAME>DEMF</SHORT-NAME>
  <BASIC-STRUCTURE-REF ID-REF="SimpleSTRUCT_ID"/>
  (...)
</DYNAMIC-ENDMARKER-FIELD>
<DIAG-SERVICE ID="ServiceXYZ_ID">
  <SHORT-NAME>ServiceXYZ</SHORT-NAME>
  (...)
  <POS-RESPONSE-REF ID-REF="Rsp_ID"/>
  (...)
</DIAG-SERVICE>
<POS-RESPONSE ID="Rsp_ID">
  <SHORT-NAME>Rsp</SHORT-NAME>
  <PARAMS>
    <PARAM xsi:type="CODED-CONST">
      <SHORT-NAME>SID</SHORT-NAME>
      <DIAG-CODED-TYPE BASE-DATA-TYPE="A_UINT32" (...)>
        (...)
      </DIAG-CODED-TYPE>
    </PARAM>
    <PARAM xsi:type="VALUE">
      <SHORT-NAME>RepeatedItems</SHORT-NAME>
      (...)
      <DOP-REF ID-REF="DYN_EM_FIELD_ID">
    </PARAM>
  </PARAMS>
</POS-RESPONSE>
```

Figure 22 shows the datatype eFIELD at database and runtime side.

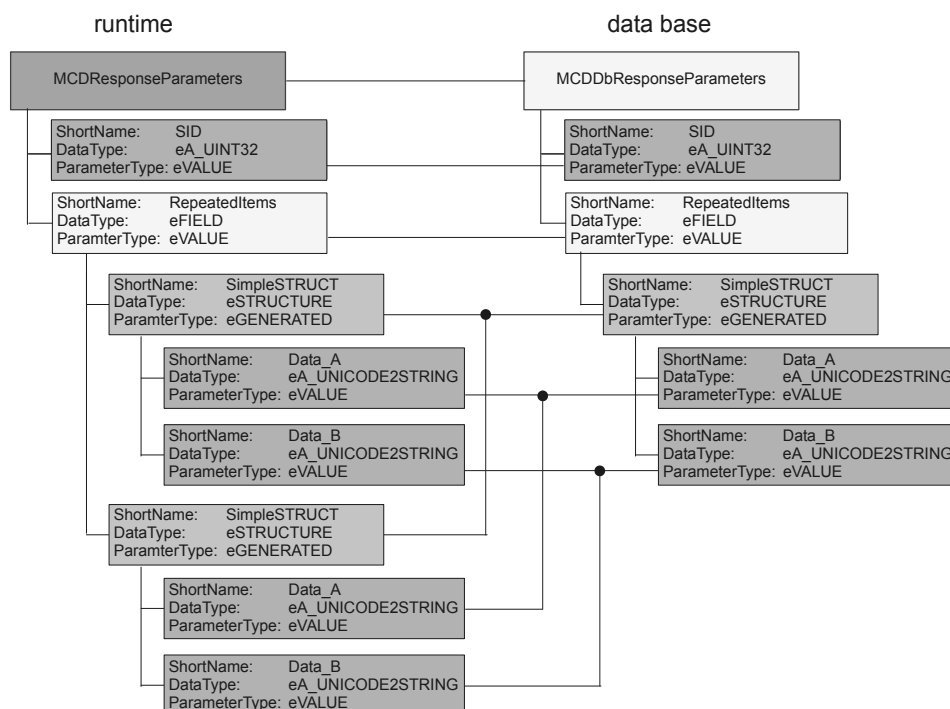


Figure 22 — Datatype eFIELD at database and runtime side

7.7.5.4 eEND_OF_PDU

A parameter of data type eEND_OF_PDU consists of 0 to n elements. Each element contains one element of data type eSTRUCTURE with a short name equal to the SHORT-NAME of the ODX STRUCTURE; this means the STRUCTURE element of ODX will be used independently from the isVisible data. The structure itself consists of arbitrarily simple or complex elements (the actual field content). On database side exactly one field element (eSTRUCTURE with content) is shown. On runtime side the number of included eSTRUCTURE elements depends on how often this structure shall be repeated in the request PDU. Initially, this number is equal to the minimum number of items defined by MIN-NUMBER-OF-ITEMS in ODX. It might be extended calling `MCDRequestParameter::addParameters` until the maximum given by MAX-NUMBER-OF-ITEMS in ODX is reached. Please note that the short names of the field elements will not be unique.

Parameters of type eEND_OF_PDU appear in request structures only. They are built from ODX END-OF-PDU-FIELD.

ODX-Data (Extract) of the DB-Template

```
<DATA-OBJECT-PROP ID="SimpleDOP_ID">
  <SHORT-NAME>SimpleDOP</SHORT-NAME>
  (...)
  <DIAG-CODED-TYPE (...) >
    (...)
  </DIAG-CODED-TYPE>
  <PHYSICAL-TYPE BASE-DATA-TYPE="A_UNICODE2STRING"/>
</DATA-OBJECT-PROP>
<STRUCTURE ID="SimpleSTRUCT_ID">
  <SHORT-NAME>SimpleSTRUCT</SHORT-NAME>
  <PARAMS>
```

```
<PARAM xsi:type="VALUE">
  <SHORT-NAME>Data_A</SHORT-NAME>
  (...)
  <DOP-REF ID-REF="SimpleDOP_ID"/>
</PARAM>
<PARAM xsi:type="VALUE">
  <SHORT-NAME>Data_B</SHORT-NAME>
  (...)
  <DOP-REF ID-REF="SimpleDOP_ID"/>
</PARAM>
</PARAMS>
</STRUCTURE>
<END-OF-PDU-FIELD ID="EOP_FIELD_ID">
  <SHORT-NAME>EOPF</SHORT-NAME>
  <BASIC-STRUCTURE-REF ID-REF="SimpleSTRUCT_ID"/>
  <MAX-NUMBER-OF-ITEMS>4</MAX-NUMBER-OF-ITEMS>
  <MIN-NUMBER-OF-ITEMS>2</MIN-NUMBER-OF-ITEMS>
</END-OF-PDU-FIELD>
<DIAG-SERVICE ID="ServiceXYZ_ID">
  <SHORT-NAME>ServiceXYZ</SHORT-NAME>
  (...)
  <REQUEST-REF ID-REF="Req_ID"/>
  (...)
</DIAG-SERVICE>
<REQUEST ID="Req_ID">
  <SHORT-NAME>Req</SHORT-NAME>
  <PARAMS>
    <PARAM xsi:type="CODED-CONST">
      <SHORT-NAME>SID</SHORT-NAME>
      <DIAG-CODED-TYPE BASE-DATA-TYPE="A_UINT32" (...)>
        (...)
      </DIAG-CODED-TYPE>
    </PARAM>
    <PARAM xsi:type="VALUE">
      <SHORT-NAME>RepeatedItems</SHORT-NAME>
      (...)
      <DOP-REF ID-REF="EOP_FIELD_ID"/>
    </PARAM>
  </PARAMS>
</REQUEST>
```

Figure 23 shows the datatype eEND_OF_PDU at database and runtime side.

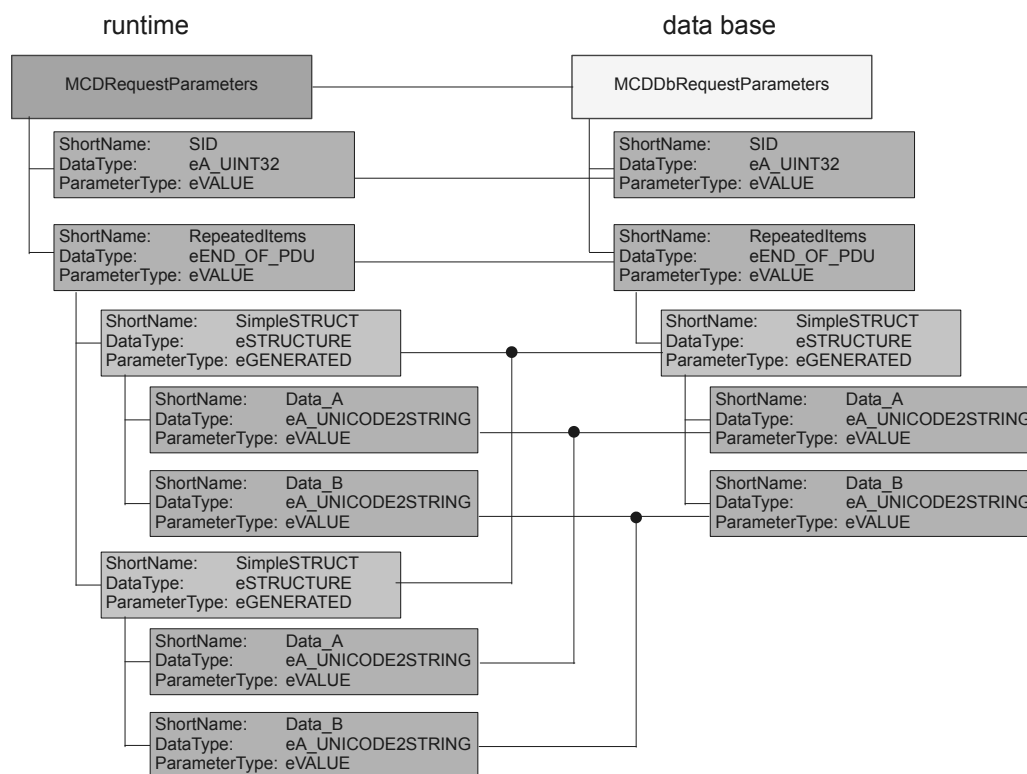


Figure 23 — Datatype eEND_OF_PDU at database and runtime side

7.7.5.5 eMULTIPLEXER

A parameter of data type eMULTIPLEXER consists of 0 to n branches. Each branch contains one element of data type eSTRUCTURE with name equal to CASE SHORT-NAME; this means the STRUCTURE element of ODX will be used independently from isVisible data. The structure itself consists of arbitrarily simple or complex elements. On database side all branches of a multiplexer are described including possibly empty cases which do not reference a STRUCTURE, and the optional default case. On runtime side only one branch is available.

ODX-Data (Extract) of the DB-Template

```
<DATA-OBJECT-PROP ID="SwitchKeyDOP_ID">
  <SHORT-NAME>SwitchKeyDOP</SHORT-NAME>
  (...)
</DATA-OBJECT-PROP>
<DATA-OBJECT-PROP ID="SimpleDOP_ID">
  <SHORT-NAME>SimpleDOP</SHORT-NAME>
  (...)
  <DIAG-CODED-TYPE (...)>
    (...)
  </DIAG-CODED-TYPE>
  <PHYSICAL-TYPE BASE-DATA-TYPE="A_UNICODE2STRING"/>
</DATA-OBJECT-PROP>
<STRUCTURE ID="DefaultCaseSTRUCT_ID">
  <SHORT-NAME>DefaultCaseSTRUCT</SHORT-NAME>
  <PARAMS>
    <PARAM xsi:type="VALUE">
```

```
<SHORT-NAME>DefaultData_A</SHORT-NAME>
(...)
<DOP-REF ID-REF="SimpleDOP_ID"/>
</PARAM>
<PARAM xsi:type="VALUE">
  <SHORT-NAME>DefaultData_B</SHORT-NAME>
  (...)
  <DOP-REF ID-REF="SimpleDOP_ID"/>
</PARAM>
</PARAMS>
</STRUCTURE>
<STRUCTURE ID="Case1STRUCT_ID">
  <SHORT-NAME>Case1STRUCT</SHORT-NAME>
  <PARAMS>
    <PARAM xsi:type="VALUE">
      <SHORT-NAME>Case1Data_A</SHORT-NAME>
      (...)
      <DOP-REF ID-REF="SimpleDOP_ID"/>
    </PARAM>
    <PARAM xsi:type="VALUE">
      <SHORT-NAME>Case1Data_B</SHORT-NAME>
      (...)
      <DOP-REF ID-REF="SimpleDOP_ID"/>
    </PARAM>
    <PARAM xsi:type="VALUE">
      <SHORT-NAME>Case1Data_C</SHORT-NAME>
      (...)
      <DOP-REF ID-REF="SimpleDOP_ID"/>
    </PARAM>
  </PARAMS>
</STRUCTURE>
<STRUCTURE ID="Case2STRUCT_ID">
  <SHORT-NAME>Case2STRUCT</SHORT-NAME>
  <PARAMS>
    <PARAM xsi:type="VALUE">
      <SHORT-NAME>Case2Data</SHORT-NAME>
      (...)
      <DOP-REF ID-REF="SimpleDOP_ID"/>
    </PARAM>
  </PARAMS>
</STRUCTURE>
<MUX ID="MUX_ID">
  <SHORT-NAME>Multiplexer</SHORT-NAME>
  (...)
  <SWITCH-KEY>
    (...)
    <DATA-OBJECT-PROP-REF ID-REF="SwitchKeyDOP_ID"/>
  </SWITCH-KEY>
  <DEFAULT-CASE>
    <SHORT-NAME>DefaultCase</SHORT-NAME>
    <STRUCTURE-REF ID-REF="DefaultCaseSTRUCT_ID"/>
  </DEFAULT-CASE>
</CASES>
  <CASE>
    <SHORT-NAME>Case1</SHORT-NAME>
    <STRUCTURE-REF ID-REF="Case1STRUCT_ID"/>
    (...)
  </CASE>
  <CASE>
    <SHORT-NAME>Case2</SHORT-NAME>
```