INTERNATIONAL STANDARD

ISO 22900-3

First edition
2009-06-15

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 22900-3:2009(E)

PDF disclaimer

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.

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

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*.

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

The purpose of this part of ISO 22900 is to define a universal application programmer interface of a vehicle communication server application. At present, the automotive market requires different vehicle communication interfaces for different vehicle original equipment manufacturers (OEMs) supporting multiple communication protocols. However, up until now, many vehicle communication interfaces have been incompatible with regard to interoperability with multiple communication applications and vehicle communication protocols.

Implementation of the measurement calibration diagnostic (MCD) server concept supports overall cost reduction to the end user, e.g. because 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 protocol data unit application programming interface (D-PDU API) as specified in ISO 22900-2. The server application needs to be configured with vehicle and electronic control unit (ECU) specific information. This is accomplished by supporting the open diagnostic exchange (ODX) data format, as specified in ISO 22901-1[1)].

1) Equivalent to ASAM MCD 2 D ODX[11].

Road vehicles — Modular vehicle communication interface (MVCI) —

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

1 Scope

This part of ISO 22900 defines an object-oriented programming interface, the objective of which is to be able to implement server applications used during the design, production and maintenance phase of a vehicle communication system which are compatible with each other and therefore exchangeable.

A server compliant with this part of ISO 22900 has three function blocks:

- M: measurement,
- C: calibration, and
- D: diagnostics.

A more detailed description of the server API is given in the Programmers Reference Guide[14].

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 9141-2, *Road vehicles — Diagnostic systems — Part 2: CARB requirements for interchange of digital information*

ISO 11898-2, *Road vehicles — Controller area network (CAN) — Part 2: High-speed medium access unit*

ISO 11898-3, *Road vehicles — Controller area network (CAN) — Part 3: Low-speed, fault-tolerant, medium-dependent interface*

ISO 11992-1, *Road vehicles — Interchange of digital information on electrical connections between towing and towed vehicles — Part 1: Physical and data-link layers*

ISO 14229-1, *Road vehicles — Unified diagnostic services (UDS) — Part 1: Specification and requirements*

ISO 14230-1, *Road vehicles — Diagnostic systems — Keyword Protocol 2000 — Part 1: Physical layer*

ISO 15765 (all parts), *Road vehicles — Diagnostics on Controller Area Networks (CAN)*

ISO 22900-2:2009, *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 open diagnostic data exchange (ODX) to a diagnostic data element

3.1.2
ancestor object
parent object
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
functional class
set of diagnostic services

3.1.5
interface connector
connector at the vehicle end of the interface cable between the vehicle and the communication device

3.1.6
job
sequence of diagnostic services and other jobs with a control flow

3.1.7
location
representation of a set of diagnostic data valid on a given hierarchical level of inheritance as defined in open diagnostic data exchange (ODX)

EXAMPLE Multiple ECU Job; Protocol; Functional Group; ECU Base Variant; ECU Variant; Module.

3.1.8
logical link
set of data, identifying the physical line, the interface and protocol used for an electronic control unit (ECU)

3.1.9
physical interface link
physical connection between the vehicle communication interface (VCI) connector of a VCI and the interface connector

3.1.10
physical link
connection from the interface of the D-server to the electronic control unit (ECU) in the vehicle

NOTE A physical link is a **physical vehicle link** (3.1.11) connected to a **physical interface link** (3.1.9).

3.1.11
physical vehicle link
connection between the vehicle connector and the electronic control unit (ECU), which describes the unique bus system in a vehicle

3.1.12
project
pool of diagnostic, measurement and/or calibration data

NOTE References between such data are all resolvable within the same project.

3.1.13
vehicle connector
connector on a vehicle providing access to the bus systems in the vehicle

3.2 Abbreviated terms

A2L	ASAM 2 MC language (see ASAM MCD 2 MC[12])
API	Application Programming Interface
ASAM	Association for Standardisation of Automation and Measuring Systems
ASCII	American Standard for Character Information Interchange
AUSY	AUtomation SYstem
BVI	Base Variant Identification
BVIS	Base Variant Identification and Selection
C	Calibration
CAN	Controller Area Network
COM/DCOM	Distributed Component Object Model
CORBA	Common Object Request Broker Architecture
CRC	Cyclic Redundancy Check
D	Diagnostics
DDLID	Dynamically Defined Local ID
Diag	Diagnostic
DLL	Dynamic Link Library
DOP	diagnostic Data Object Property
D-server	Function block of MCD-server for Diagnostic
DTC	Diagnostic Trouble Code
DTD	Document Type Definition
DynId	Dynamic Identifiers
ECU	Electronic Control Unit
ECU MEM	Electronic Control Unit MEMory
ERD	Entity Relationship Diagram
IDL	Interface Definition Language
JAVA RMI	JAVA Remote Method Invocation
KWP	KeyWord Protocol
M	Measurement
MCD	Measurement Calibration Diagnostic
MC-server	Function block of MCD-server for Measurement and Calibration
MVCI	Modular Vehicle Communication Interface

ODX	Open Diagnostic data eXchange
OEM	Original Equipment Manufacturer
PC	Personal Computer
PDU	Protocol Data Unit (referred to as D-PDU API in this part of ISO 22900)
SDG	Special Data Groups
SI	Système International d'unités
UDS	Unified Diagnostic Services
UML	Unified Modeling Language
UTC	Coordinated Universal Time
VCI	Vehicle Communication Interface
VI	Variant Identification
VIS	Variant Identification and Selection

3.3 Typographical conventions and mnemonics used in this part of ISO 22900

Source code and technical artefacts within the text are presented in `Courier new` type.

Diagrams that denote interaction sequences, relationships or dependencies between interfaces are presented using the 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. “MCD”.

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 permitted in interface names, method names and parameter names, but it is permitted for constants.

The leading letter of each constant is “e” followed by the name in block capitals.

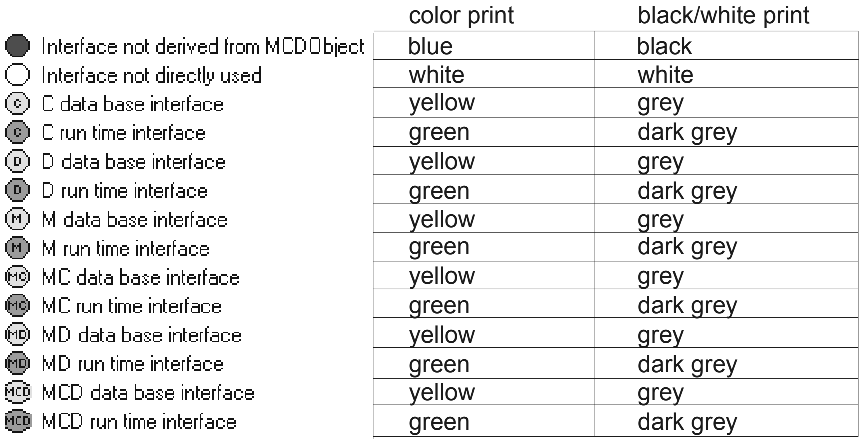
ODX element names are written in upper case, e.g. SHORT-NAME.

Names of classes, objects and interfaces in ISO 22900-3 (MCD-3D) are written in mixed fixed, e.g. MCDDbProject.

3.4 Legends for used graphics

3.4.1 Hierarchical diagrams

Figure 1 illustrates the legends for hierarchical models.



		color print	black/white print
●	Interface not derived from MCDObject	blue	black
○	Interface not directly used	white	white
Ⓒ	C data base interface	yellow	grey
Ⓒ	C run time interface	green	dark grey
Ⓓ	D data base interface	yellow	grey
Ⓓ	D run time interface	green	dark grey
Ⓜ	M data base interface	yellow	grey
Ⓜ	M run time interface	green	dark grey
MC	MC data base interface	yellow	grey
MC	MC run time interface	green	dark grey
MD	MD data base interface	yellow	grey
MD	MD run time interface	green	dark grey
MCD	MCD data base interface	yellow	grey
MCD	MCD run time interface	green	dark grey

Figure 1 — Legend for hierarchical models

3.4.2 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 in accordance with the presentation in UML and are structured as follows. The chronological sequence arises while reading from the top downwards. The commentary column, in which single activities are commented on, is placed in the left margin. Within the sequence diagram, the Client application is shown left; if necessary for the respective case, the EventHandler is shown there as well. Right of the Client (with or without EventHandler) are located the API objects necessary for the respective case. On the far right the D-server is presented if necessary.

Not all API objects possible for the respective instant of time are shown, but only those of relevance for the respective case. 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 D-server, represent the actions necessary for the respective case. The object at which the arrow points will execute the action. The grey horizontal arrows represent the return of objects.

3.5 Stereotypes

Stereotypes are abbreviation characters used in this part of ISO 22900 to mark the affiliation of statements, interfaces and methods to one of the possible function blocks. The following stereotypes are used:

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

4 General considerations

From the perspective of the user, access and integration of on-board control units is provided by a corresponding application, the communication server and a VCI module for diagnostics, as well as measurement and calibration modules.

The user is granted access to the measurement and calibration objects and to the diagnostic services. The server provides access to the shared use of several function blocks, via identical methods using only one server (the MCD-server) for all of these tasks.

The function blocks are used for the handling of ECUs in vehicles

- for taking measurements (M),
- for making adjustments of calibration parameters (C), and
- for diagnostics (D).

Not all of the three function blocks need to be supported in an implementation of a server. A vendor can choose only to implement the D function block in a D-server; implementation of the M and C function blocks would be termed an MC-server.

A compliant D-server also supports Job-Language (Java) and may support optional features such as 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 need to run on the same computer. Remote use via an interface is also possible, supported by the design of the server API. This interface is provided for COM/DCOM[17], for Java[19], and for C++[18].

5 Specification release version information

The specification release version of this part of ISO 22900 is: 2.2.0.

6 Structure of MCD systems

Each MCD-server is divided into the functional blocks of M, C, D and the database.

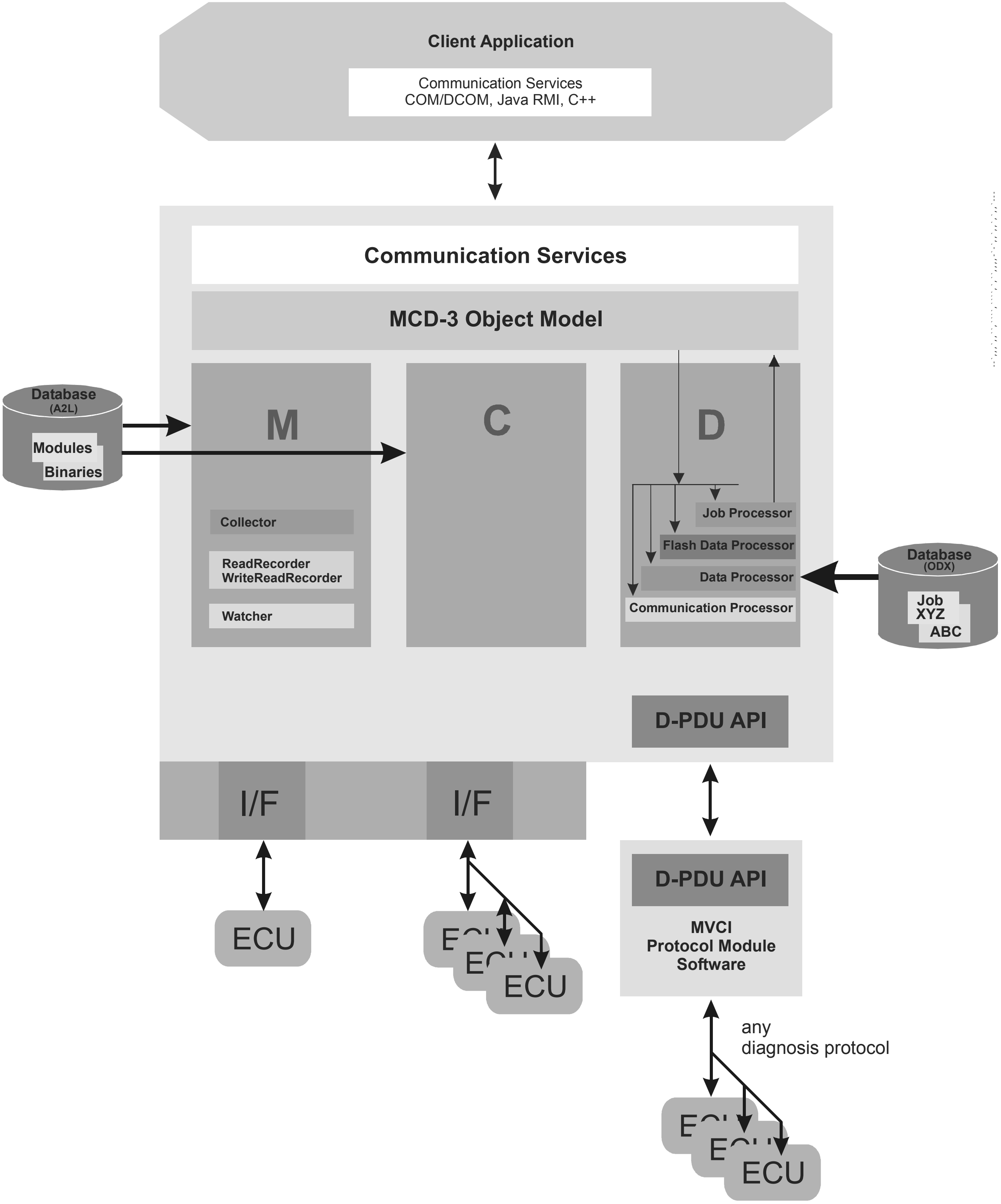


Figure 2 — Architecture of a MCD System

MCD-servers have three function blocks:

- Measurement (measurement sequences),
- Calibration (adjustment tasks) and
- Diagnostics (diagnostic services).

To realize the separate block functionalities, they shall always be accessed via the respective function block.

Within a real server, not all function blocks have to be supported. The function blocks are used for the handling of control units (ECU s) within vehicles for the realization of measurements, adjustments and for diagnostics. With the help of MCD-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 MCD-3 Object Model shall enable the implementation of M-, C-, D-, MC-, CD-, MD- and MCD-server:

- A future MC-server shall implement M-functionality following the collector principle defined in the MCD-3 MC Object Model.
- If there exists a Diagnostic service which allows reading and writing variables on ECU this is not seen as the standard MC-server functionality but as a special form of a service.
- If there exists a Diagnostic service which allows free-running data exchange between ECU and D-server this is not seen as the standard M- functionality but as a special form of a service.
- A future D-server shall implement D-server functionality following the communication primitive principle defined in the D object model.

MC-server, which meet the requirements of this specification do not necessarily need a user interface (GUI), but still may have one.

All interfaces and methods inside a function block are mandatory! Exception: MCDDbProjectconfiguration, ECU Configuration, ECU Flash Programming, DDLID, support of Diagnostic Variables and monitoring in diagnostic area is optional. Optional means that the runtime- as well as the database part of the model do not have to be implemented by a D-server that is omitting the feature in question. In case a client application calls a method that is part of an optional feature, the D-server should return an empty collection in case the method’s return type is a inheriting from MCDCollection. Otherwise, such a method call should throw a `MCDSystemException` of type `eSYSTEM_METHOD_NOT_SUPPORTED`.

The number of control units applied in vehicles is continuously increasing. The capabilities of the single control unit concerning measuring, adjusting and diagnostics become available for the MCD-server by means of control unit description files (ASAM MCD 2 database[11],[12]). The control unit description files represent a manufacturer independent data exchange format, that means any MCD-server may handle the data out of a control unit description file. All configuration data of the MCD-server, the internal data of ECU s or ECU nets and the communication methods for the ECU access are stored in the ASAM MCD 2 database. This database is MCD-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 MCD-server, that means only the MCD-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 MCD-server is guaranteed.

Also, a decoupling from the used data description exchange format (a2l or XML) takes place.

The MCD-server shall manage the database and to provide for the required and necessary information for the single Objects. The database does not belong to one specific Object, but is available within the whole MCD-server. The organization of the object or reference allocation is solved implementation specifically by the MCD-server.

Standardized drivers are used to access to the most different control units, to enable the physical and logical linkage of control units to MCD-server.

The object model supports Multi Client Systems. Although, the design has been oriented in accordance with a Single Client System, to provide for a simple use for this most typical and most frequently occurring application case. That means, that no client references are included within the single objects. The administration of the client references is done by the MCD-server and shall 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 enables the linking of MCD-server to automation systems (MCD-server API). Objective of this linking is the remote control of the MCD-server in test stands. By means of the object model the functionality is standardized, that means the interfaces for the function blocks with accompanying methods are standardized. The communication shall be realized via the particular implementation of the object model for the used platform, programming language and linking mechanisms.

Among others realized are:

- 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 of Request and Response telegrams to ECU s. 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 like KWP2000 and "Diagnostics on CAN". Nevertheless, the communication processor shall be parameterised via ODX using 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 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 data and offers an library to the Job Processor. The Data processor also handles Jobs, as they are stored in ASAM MCD 2 D ODX.

The Flash Data Processor

Is responsible for the loading of programs and data in ECU s. The flash data is part of the database. The Flash Data Processor provides access to the 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 MCD-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 D-server API to insure consistency between D-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 files.

7 Function block common MCD

7.1 MCD system object

The server is only accessible by a certain number of clients at the same time. The number of allowed clients can be requested via the method `getMaxNoOfClients`.

Every Client gets its own `MCDSystem` object (implements the MCDSystem interface) from the MCD-server. But all Clients work on the same project and database. The runtime MCDProject has the connection to a certain MCDDbProject which along with its sub items will be available after selecting the project. The selection of another project at the same time is not allowed and will throw an exception.

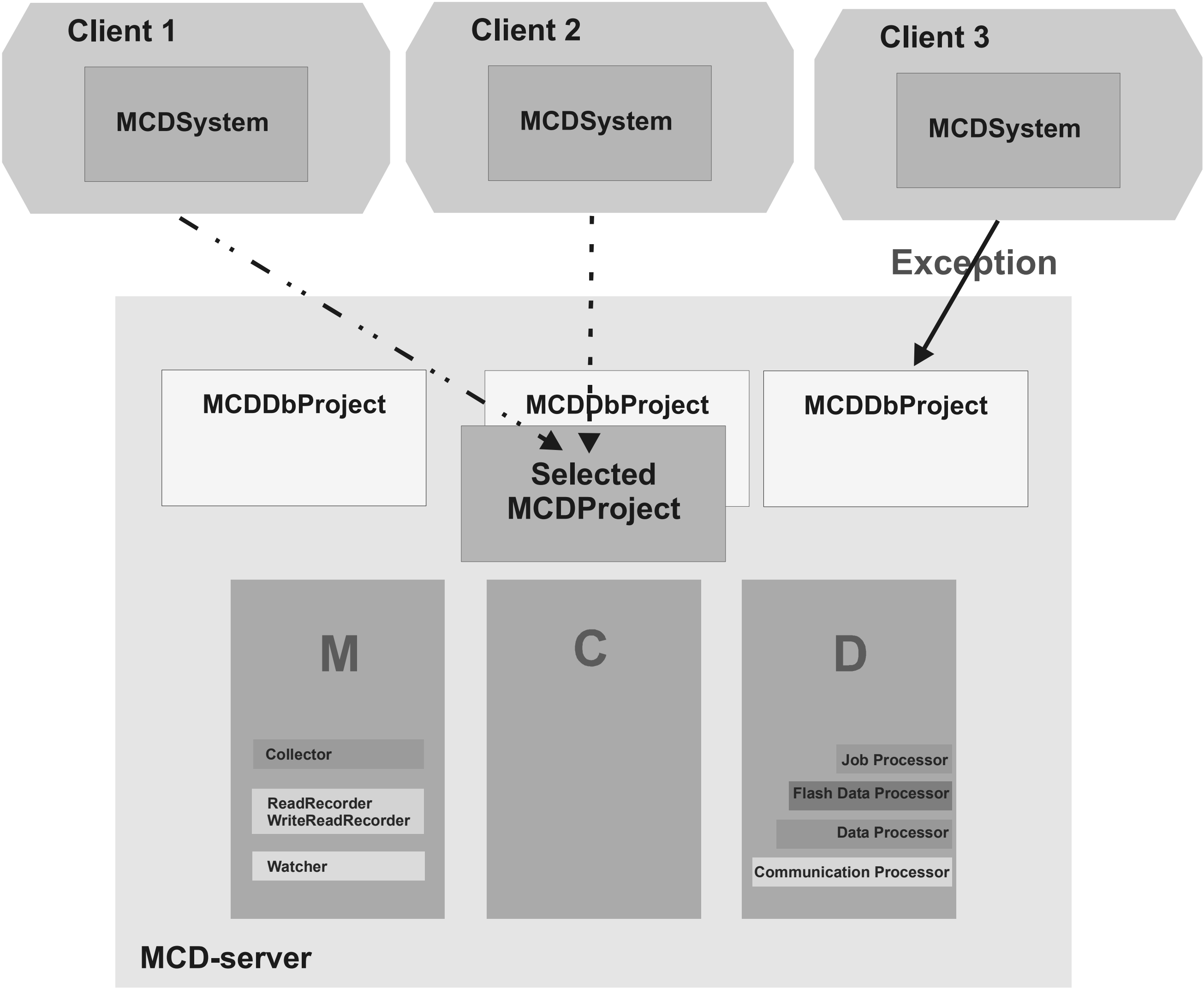


Figure 3 — System scheduling

At the moment, the other diagrams within the specification always refer to the representation within the client and are not designed for MultiClient scenarios, except where noted otherwise.

7.2 Version information retrieval

The method `MCDSystem::getASAMMCDVersion():MCDVersion` returns the ASAM release number of the interface. For this specification it is the major value 2 and the minor value 1 defined.

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

All function blocks implemented in a tool use the same ASAM 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 interface definition are kept synchronous. The so-called “interface number (IFN)” is used as a build number for the interface definition. Each interface definition maintains an own interface number. This number will be incremented continuously within a minor version. Increments are applied whenever a new interface definition 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 “interface number” is incremented separately for each technology.

7.3 Description of Terms

7.3.1 General

The key terms and definitions which apply in this part of ISO 22900 are given in Clause 3. This subclause describes the most important terms in more detail.

7.3.2 Client-controlled object

A client-controlled object is an object that is passed to the client and modification of these object does not result in a direct modification of objects within the server. All values in a MCD-server, which have an ASAM data type, are considered to be client-controlled. The client is responsible for the deletion of client-controlled objects.

7.3.3 Location

A `MCDDbLocation` is an entry point for available Services, DiagComPrimitives, Measurements, Characteristics and other data of an ECU. Please note, that for the location no AccessKey is used.

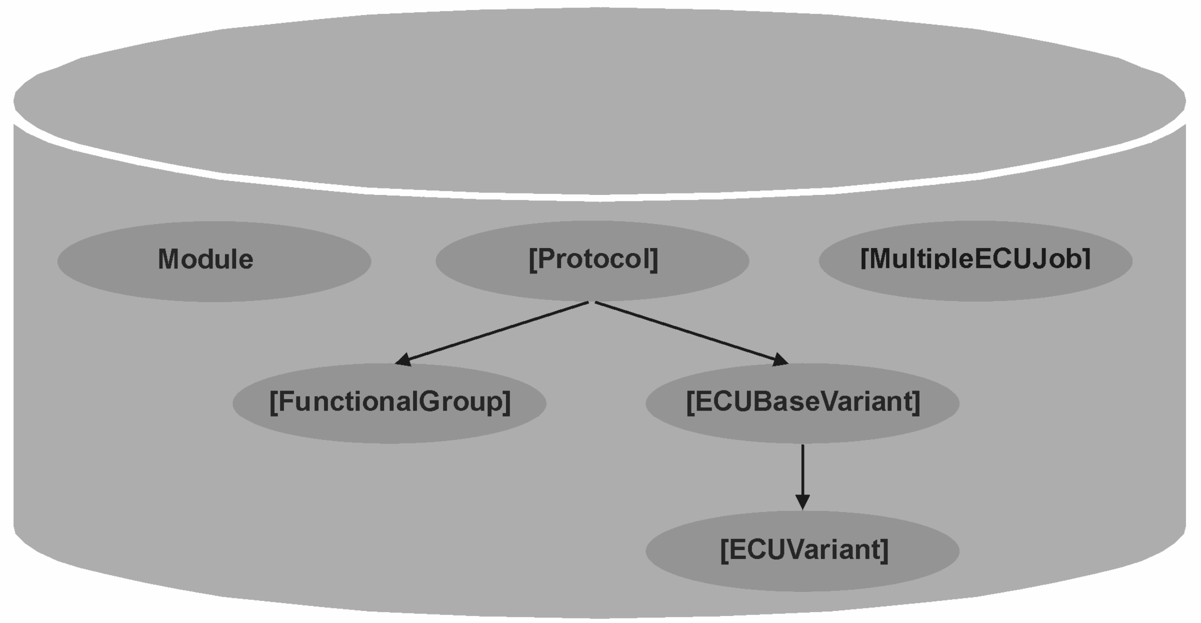


Figure 4 — Location hierarchy of ASAM MCD database

The MCDDbLocation represents the contents of the corresponding description file, e.g. ODX or A2L file.

7.3.4 Logical Link (LOGICAL-LINK)

A Logical Link represents the physical line, the used interface and protocol of an ECU as one object in an MCD-server. Typically, the Logical Link represents the access to only one ECU, but in case of a Functional Group it may also describe more than one ECU. The Logical Link is identified 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 Physical Interface. Logical Links may also be used to access the same ECU on different ways, or access more than one ECU on different links. For more details see the chapters Function block Diagnostics. Logical Links can be locked for exclusive use by one client (for more information see Locking).

7.3.5 Project

A project is a logical grouping for defined specific installations selected by the user. Within a project, all information necessary for a test installation shall be contained. It is only permitted to work within one project, this shall be considered for the logical grouping (e.g. two model series within one project). That is why 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 (Measurements, Characteristics, Diagnostic Services, ...),
- jobs,
- flash containers and
- configuration files.

Within the MCD-server, first the selection of a project out of the list of the existing projects takes place, before any database information can be accessed.

7.3.6 Server-controlled mutable object

A server-controlled mutable object is a server-controlled object that provides methods for clients for locking, access restriction, change notification and change of the object state, properties, or the content of owned sub-objects. The server-controlled mutable objects can be divided in primary and secondary server-controlled mutable objects. A primary server-controlled mutable object is a root node of a set of objects which belong together because they build a complex semantic object. The server-controlled mutable objects that are contained in a primary server-controlled mutable object are called secondary server-controlled mutable objects. The set of objects are only accessible through the object hierarchy under the root node and contain all objects until no child object is available or the child object is a primary server-controlled mutable object.

7.3.7 Server-controlled object (shared object)

A server-controlled object is an object that is not a client-controlled object in the object model. The MCD-server is responsible for the deletion of server-controlled objects.

7.4 States of the MCD system object

Within the state eINITIALIZED the MCDSystem object is transmitted to the client by the MCD-server. Within this state, the database project descriptions can be listed and general system initializations 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.

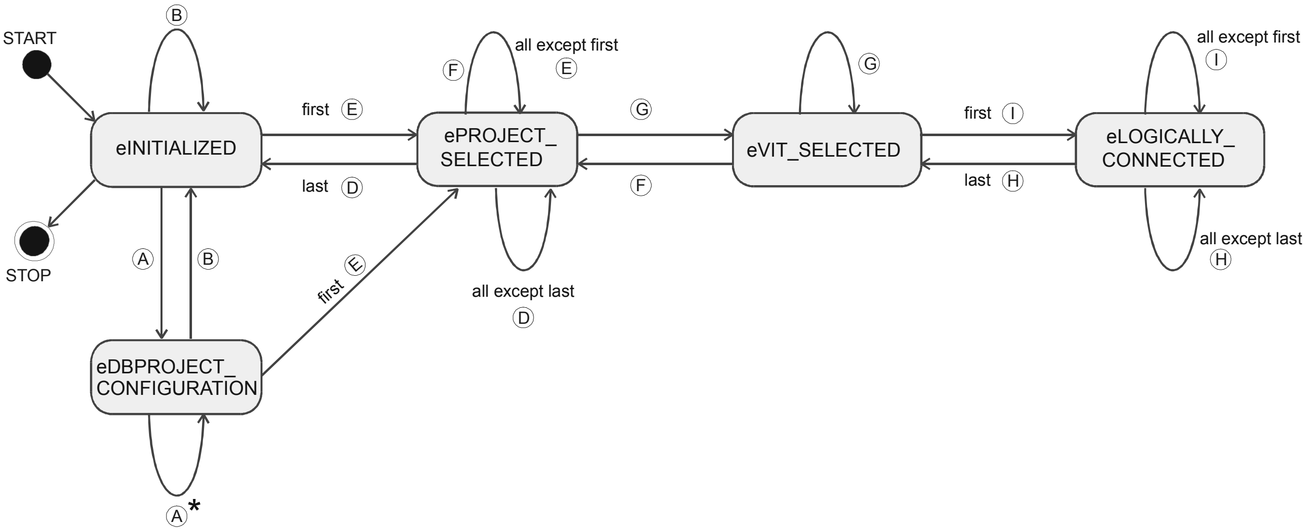
With the selection of a VehicleInformationTable the state is changed to eVIT_SELECTED.

As soon as the first Logical Link has been created in an MCD-server, the MCDSystem object's state changes to eLOGICALLY_CONNECTED. In state eLOGICALLY_CONNECTED, communication to an ECU can be established by using the available Logical Links. If the last Logical Link has been removed from the runtime project, the system automatically takes the state eVIT_SELECTED again. With deselection of the VehicleInformation Table the state ePROJECT_SELECTED is taken. By means of deselectProject() the MCDSystem Object is set to the initial state eINITIALIZED again.

The MCD-server changes the state to eDBPROJECT_CONFIGURATION by loading a database project at object MCDDbProjectConfiguration. In this state, searching in the database is possible. In MC-servers, MCBinaries and Modules can be added. In D-servers, ECU-MEMs or CONFIGURATION-DATAs can be added (removed only in state eDBPROJECT_CONFIGURATION) at any system-state except eINITIALIZED. The corresponding MCDDLogicalLink could be in any state. If additional data is added, a system event will be thrown (onSystemDbEcuMemsModified/onSystemDbEcuConfigurationDatasModified), so the clients are informed about these changes. The collection MCDDbEcuMems/MCDDbConfigurationDatas shall be reloaded by the client at the MCDDbLocation or at the MCDDbProject. ECU-MEMs or CONFIURATION-DATAs should be loaded temporarily to a MCDDbProject or permanently added to a project configuration.

ECU-MEMs are necessary for (re-) programming of an ECU. CONFIGURATION-DATAs are necessary for configure an ECU.

In case of MC-server it is also possible to add new MCDDbProject objects at the interface MCDDbProjectConfigurations as long as the system is in state eDBPROJECT_CONFIGURATION or eINITIALIZED.



Ⓐ MCD MCDDbProjectConfiguration: load

* load or add of DbProject includes an implicit close of actual opened DbProject

Ⓑ MCD MCDDbProjectConfiguration: close

Ⓓ MCD MCDSystem : deselectProject

Ⓔ MCD MCDSystem : selectProject

Ⓕ MCD MCDProject : deselectVehicleInformation

Ⓖ MCD MCDProject : selectDbVehicleInformation
MCD MCDProject : selectDbVehicleInformationByName

Ⓗ D MCDLogicalLinks : remove
MCD MCDLogicalLinks : removeByName
MCD MCDLogicalLinks : removeByIndex
MCD MCDLogicalLinks : removeAll

Ⓘ D MCDLogicalLinks : addByAccessKeyAndInterfaceResourceLink
D MCDLogicalLinks : addByAccessKeyAndPhysicalInterfaceLink
D MCDLogicalLinks : addByAccessKeyAndVehicleLink
D MCDLogicalLinks : addByDbObject
D MCDLogicalLinks : addByName
D MCDLogicalLinks : addByVariant

Figure 5 — System state transitions

The states of clients and the central state of the server (internal MCDSystem object) shall be distinguished.

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

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

Table 1 — System states

	MCDSystem		MCDDbProjectConfiguration				MCDProject		MCDLogicalLinks		MCDDbProject	
System State	selectProject selectProjectByName	deselectProject	add load	close	getAdditionalEcuMemNames getAdditionalConfigurationDataNames	removeByIndex removeByName	selectDbVehicleInformation selectDbVehicleInformationByName	deselectVehicleInformation	addByAccessKeyAndVehicleLink addByDbObject addByName addByNames addByObjects addByVariant addByAccessKeyAndInterfaceResource	remove removeByName removeByIndex removeAll	loadNewConfigurationData loadNewConfigurationDatasByFileName loadNewEcuMem loadNewEcuMemsByFileNAme	removeConfiurationDataByName removeECUMemByName
eINITIALIZED	X	X	X	X		X						
ePROJECT_SELECTED	X[1]	X					X	X			X	
eVIT_SELECTED							X[2]	X	X	X[3]	X	
eLOGICALLY_CONNECTED									X	X[3]	X	
eDB_PROJECT_CONFIGURED	X		X	X	X						X	X

Key

X[1]	only identical project
X[2]	only identical VIT
X[3]	It is only allowed to remove a LogicalLink if it is in the state D: eCREATED

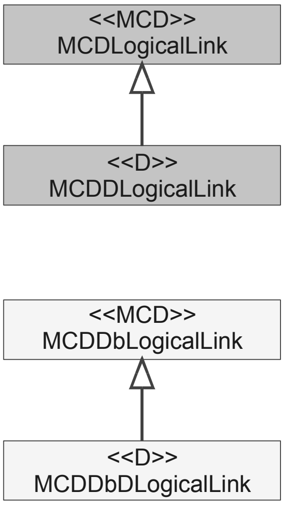
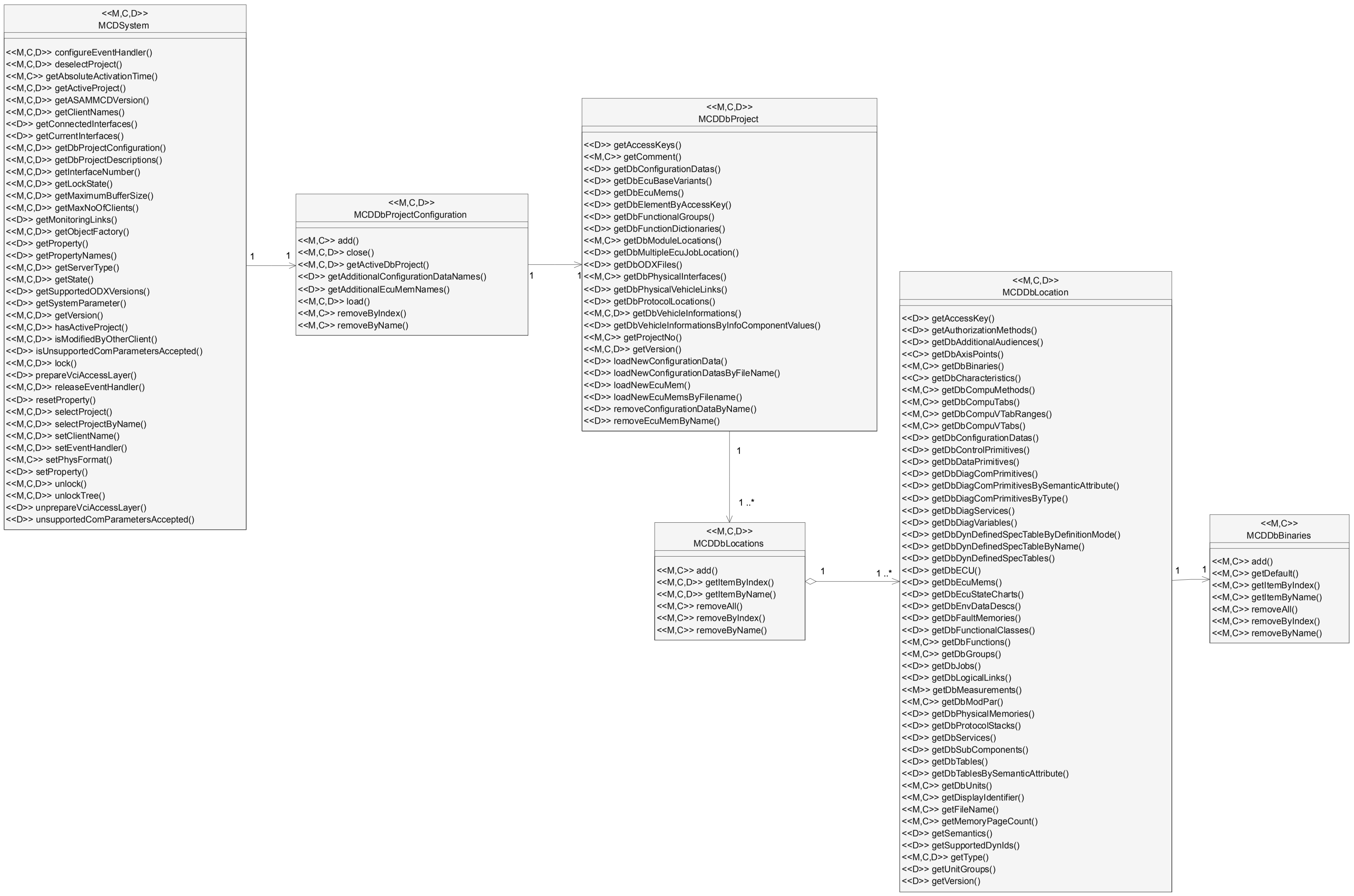


Figure 6 — Class diagram of MCDDLogicalLink

7.5 State changes

State changes in an object are triggered directly via method calls or indirectly due to internal state changes of the MCD-server. A successful state change is transported via events to the clients. These events transport the object, which is changed, and the previous state of the object. A method, which triggers a state change, will throw an exception if the state change could not be processed successfully and the state is not changed. Feedback transitions, state changes to the previous state, do not trigger any events and do not throw an exception.

7.6 Project configuration



<<M,C,D>>
MCDSystem
<<M,C,D>> configureEventHandler()
<<M,C,D>> deselectProject()
<<M,C>> getAbsoluteActivationTime()
<<M,C,D>> getActiveProject()
<<M,C,D>> getASAMMCDVersion()
<<M,C,D>> getClientNames()
<<D>> getConnectedInterfaces()
<<D>> getCurrentInterfaces()
<<M,C,D>> getDbProjectConfiguration()
<<M,C,D>> getDbProjectDescriptions()
<<M,C,D>> getInterfaceNumber()
<<M,C,D>> getLockState()
<<M,C,D>> getMaximumBufferSize()
<<M,C,D>> getMaxNoOfClients()
<<D>> getMonitoringLinks()
<<M,C,D>> getObjectFactory()
<<D>> getProperty()
<<D>> getPropertyNames()
<<M,C,D>> getServerType()
<<M,C,D>> getState()
<<D>> getSupportedODXVersions()
<<D>> getSystemParameter()
<<M,C,D>> getVersion()
<<M,C,D>> hasActiveProject()
<<M,C,D>> isModifiedByOtherClient()
<<D>> isUnsupportedComParametersAccepted()
<<M,C,D>> lock()
<<D>> prepareVciAccessLayer()
<<M,C,D>> releaseEventHandler()
<<D>> resetProperty()
<<M,C,D>> selectProject()
<<M,C,D>> selectProjectByName()
<<M,C,D>> setClientName()
<<M,C,D>> setEventHandler()
<<M,C>> setPhysFormat()
<<D>> setProperty()
<<M,C,D>> unlock()
<<M,C,D>> unlockTree()
<<D>> unprepareVciAccessLayer()
<<D>> unsupportedComParametersAccepted()
1
1
<<M,C,D>>
MCDDbProjectConfiguration
<<M,C>> add()
<<M,C,D>> close()
<<M,C,D>> getActiveDbProject()
<<D>> getAdditionalConfigurationDataNames()
<<D>> getAdditionalEcuMemNames()
<<M,C,D>> load()
<<M,C>> removeByIndex()
<<M,C>> removeByName()
1
1
<<M,C,D>>
MCDDbProject
<<D>> getAccessKeys()
<<M,C>> getComment()
<<D>> getDbConfigurationDatas()
<<D>> getDbEcuBaseVariants()
<<D>> getDbEcuMems()
<<D>> getDbElementByAccessKey()
<<D>> getDbFunctionalGroups()
<<D>> getDbFunctionDictionaries()
<<M,C>> getDbModuleLocations()
<<D>> getDbMultipleEcuJobLocation()
<<D>> getDbODXFiles()
<<M,C>> getDbPhysicalInterfaces()
<<D>> getDbPhysicalVehicleLinks()
<<D>> getDbProtocolLocations()
<<M,C,D>> getDbVehicleInformations()
<<D>> getDbVehicleInformationsByInfoComponentValues()
<<M,C>> getProjectNo()
<<M,C,D>> getVersion()
<<D>> loadNewConfigurationData()
<<D>> loadNewConfigurationDatasByFileName()
<<D>> loadNewEcuMem()
<<D>> loadNewEcuMemsByFilename()
<<D>> removeConfigurationDataByName()
<<D>> removeEcuMemByName()
1
1 ..*
<<M,C,D>>
MCDDbLocations
<<M,C>> add()
<<M,C,D>> getItemByIndex()
<<M,C,D>> getItemByName()
<<M,C>> removeAll()
<<M,C>> removeByIndex()
<<M,C>> removeByName()
1
1 ..*
<<M,C,D>>
MCDDbLocation
<<D>> getAccessKey()
<<D>> getAuthorizationMethods()
<<D>> getDbAdditionalAudiences()
<<C>> getDbAxisPoints()
<<M,C>> getDbBinaries()
<<C>> getDbCharacteristics()
<<M,C>> getDbCompuMethods()
<<M,C>> getDbCompuTabs()
<<M,C>> getDbCompuVTabRanges()
<<M,C>> getDbCompuVTabs()
<<D>> getDbConfigurationDatas()
<<D>> getDbControlPrimitives()
<<D>> getDbDataPrimitives()
<<D>> getDbDiagComPrimitives()
<<D>> getDbDiagComPrimitivesBySemanticAttribute()
<<D>> getDbDiagComPrimitivesByType()
<<D>> getDbDiagServices()
<<D>> getDbDiagVariables()
<<D>> getDbDynDefinedSpecTableByDefinitionMode()
<<D>> getDbDynDefinedSpecTableByName()
<<D>> getDbDynDefinedSpecTables()
<<D>> getDbECU()
<<D>> getDbEcuMems()
<<D>> getDbEcuStateCharts()
<<D>> getDbEnvDataDescs()
<<D>> getDbFaultMemories()
<<D>> getDbFunctionalClasses()
<<M,C>> getDbFunctions()
<<M,C>> getDbGroups()
<<D>> getDbJobs()
<<D>> getDbLogicalLinks()
<<M>> getDbMeasurements()
<<M,C>> getDbModPar()
<<D>> getDbPhysicalMemories()
<<D>> getDbProtocolStacks()
<<D>> getDbServices()
<<D>> getDbSubComponents()
<<D>> getDbTables()
<<D>> getDbTablesBySemanticAttribute()
<<M,C>> getDbUnits()
<<M,C>> getDisplayIdentifier()
<<M,C>> getFileName()
<<M,C>> getMemoryPageCount()
<<D>> getSemantics()
<<D>> getSupportedDynIds()
<<M,C,D>> getType()
<<D>> getUnitGroups()
<<D>> getVersion()
1
1
<<M,C>>
MCDDbBinaries
<<M,C>> add()
<<M,C>> getDefault()
<<M,C>> getItemByIndex()
<<M,C>> getItemByName()
<<M,C>> removeAll()
<<M,C>> removeByIndex()
<<M,C>> removeByName()

Figure 7 — Project configuration

Within the state ePROJECT_CONFIGURATION of the MCD-server 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 (MC: MCDDbProjectConfiguration:add, MC: MCDDbBinaries:add, MC: MCDDbLocations:add, D: MCDDbProject:loadNew…) of DbProjects.

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

MC-server may also delete Modules or Binaries. The deleting of database projects takes place within the state eINITIALIZED.

That means, that the MC-server may at first create a database, then use this database for measuring and/or calibration and then delete this database again. A modification of the database during measurement/calibration is not possible, as the MC-server is not within the state eDBPROJECT_CONFIGURATION.

The Methods:

- MCDSystem::selectProject(),
- MCDDbProject::loadNewEcuMem(),
- MCDDbProjectConfiguration::getAdditionalEcuMemNames and
- check the consistency of data read from the database.

An error of type eDB_INCONSISTENT_DATABASE should be thrown in case inconsistent data has been identified.

A client cannot change the server state to eDBPROJECT_CONFIGURATION if more than one client is connected to the server. In this case, the methods MCDDbProjectConfiguration::add() and MCDDbProjectConfiguration::load() throws an exception of type MCDSystemException with error code eSYSTEM_NOT_ALLOWED_IN_MULTI_CLIENT_MODE. If a client has successfully changed to the state eDBPROJECT_CONFIGURATION, no further clients can connect to the server. How the error is transported to the client is defined in the interface definition.

7.7 Interface structure of MCD-server API

7.7.1 Separation in database and runtime side

Within the whole model there is a separation into database and runtime objects.

These interfaces are colour coded in all diagrams:

- in colour prints, yellow (database) and green (runtime);
- in black/white prints, light (database) and dark (runtime).

Apart from the configuration within the MC function block the database part is invariable and is used as the basis for the creation of most of the 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, a2l, 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. These classes have the Stereotype <<enum>>.

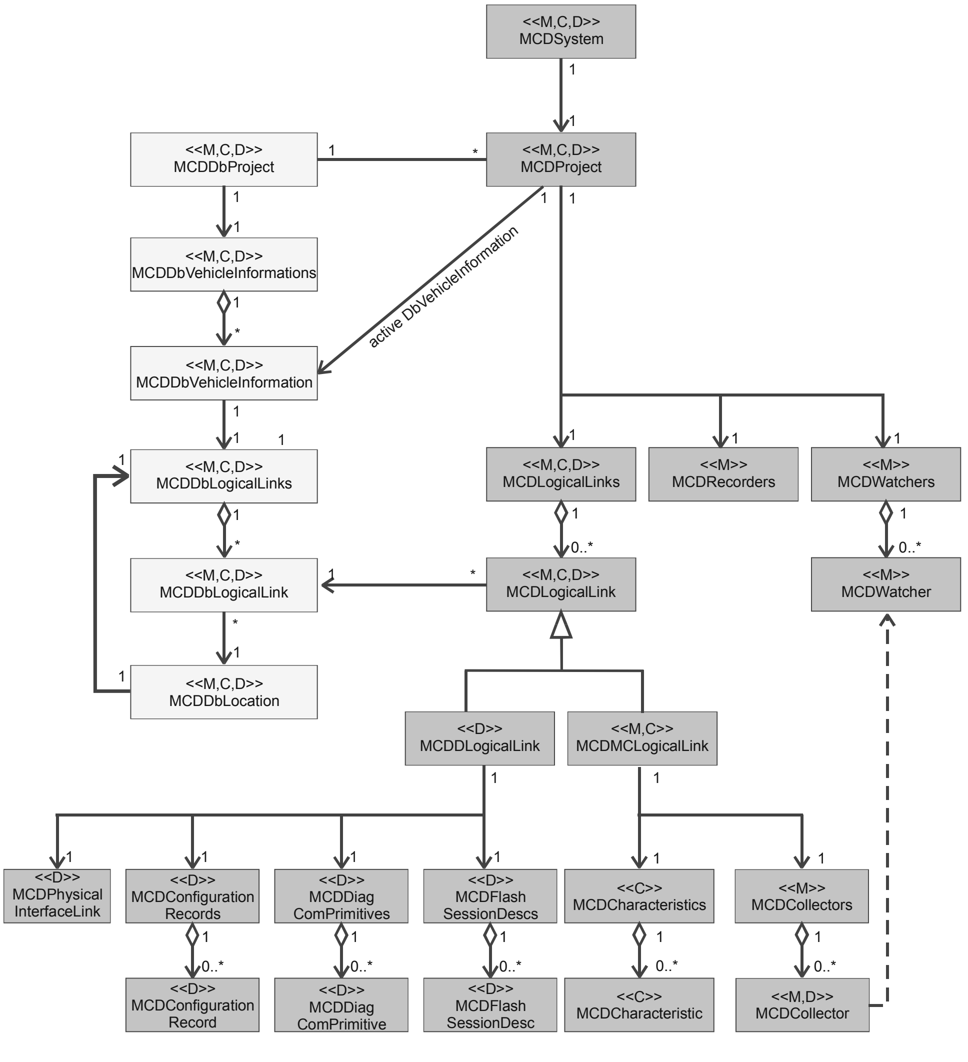

<<M,C,D>> MCDSystem
<<M,C,D>> MCDDbProject
<<M,C,D>> MCDProject
<<M,C,D>> MCDDbVehicleInformations
active DbVehicleInformation
<<M,C,D>> MCDDbVehicleInformation
<<M,C,D>> MCDDbLogicalLinks
<<M,C,D>> MCDLogicalLinks
<<M>> MCDRecorders
<<M>> MCDWatchers
<<M,C,D>> MCDDbLogicalLink
<<M,C,D>> MCDLogicalLink
<<M>> MCDWatcher
<<M,C,D>> MCDDbLocation
<<D>> MCDDLogicalLink
<<M,C>> MCDMCLogicalLink
<<D>> MCDPhysical InterfaceLink
<<D>> MCDConfiguration Records
<<D>> MCDDiag ComPrimitives
<<D>> MCDFlash SessionDescs
<<C>> MCDCharacteristics
<<M>> MCDCollectors
<<D>> MCDConfiguration Record
<<D>> MCDDiag ComPrimitive
<<D>> MCDFlash SessionDesc
<<C>> MCDCharacteristic
<<M,D>> MCDCollector
1
*
0..*

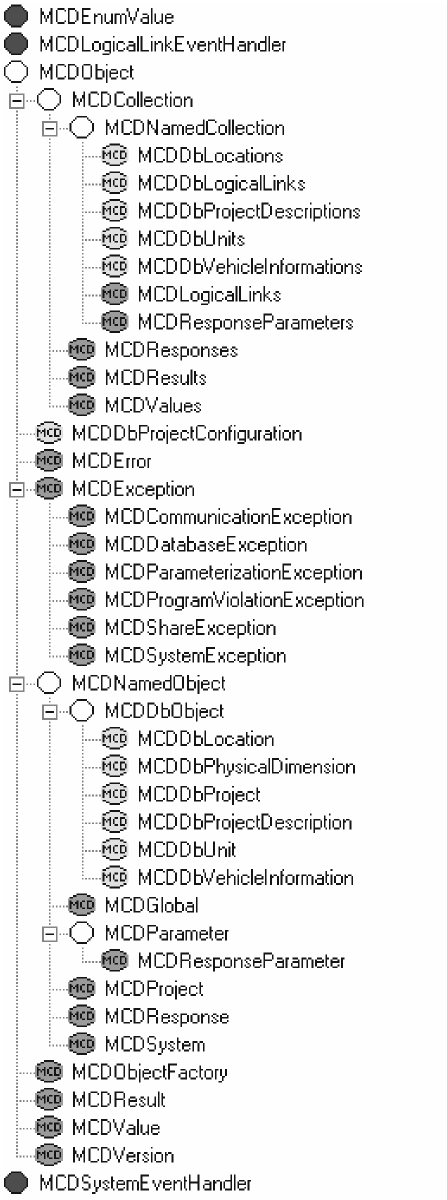
Figure 8 — Project associations

The entry object is the MCDSystem, a runtime object. From this a runtime project will be created on base of its database project. Selecting the Vehicle Information Table makes the 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.

7.7.2 Hierarchical model

The figure shows only the interfaces available for MCD. In addition to these, there exist also those for MD, MC, M, C and D. The interfaces marked with the Stereotype MCD shall be implemented by each MCD-server, no matter what type it is.

As can be seen, all database interfaces are derived from `MCDDbOject`, and all database Collections from `MCDNamedCollection`. The Runtime interfaces usually are 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.



- ● MCDEnumValue
- ● MCDLogicalLinkEventHandler
- ○ MCDObject
  - ○ MCDCollection
    - ○ MCDNamedCollection
      - MCD MCDDbLocations
      - MCD MCDDbLogicalLinks
      - MCD MCDDbProjectDescriptions
      - MCD MCDDbUnits
      - MCD MCDDbVehicleInformations
      - MCD MCDLogicalLinks
      - MCD MCDResponseParameters
    - MCD MCDResponses
    - MCD MCDResults
    - MCD MCDValues
  - MCD MCDDbProjectConfiguration
  - MCD MCDError
  - MCD MCDException
    - MCD MCDCommunicationException
    - MCD MCDDatabaseException
    - MCD MCDParameterizationException
    - MCD MCDProgramViolationException
    - MCD MCDShareException
    - MCD MCDSystemException
  - ○ MCDNamedObject
    - ○ MCDDbObject
      - MCD MCDDbLocation
      - MCD MCDDbPhysicalDimension
      - MCD MCDDbProject
      - MCD MCDDbProjectDescription
      - MCD MCDDbUnit
      - MCD MCDDbVehicleInformation
    - MCD MCDGlobal
    - ○ MCDParameter
      - MCD MCDResponseParameter
    - MCD MCDProject
    - MCD MCDResponse
    - MCD MCDSystem
  - MCD MCDObjectFactory
  - MCD MCDResult
  - MCD MCDValue
  - MCD MCDVersion
- ● MCDSystemEventHandler

Figure 9 — Hierarchical model of a MCD-server

The MCD[ObjectType]EventHandler classes are an exception concerning this classification and are not derived from MCDObject. These are used as means of communication of the MCD-server with the client and provide callback methods for the event handling.

The MCDEnumValue (realized as a singleton or single instance class) is a global converter class for all classes representing enumerations in the API. The class MCDEnumValue holds static information only and is considered to be stateless. It provides two methods to translate between the object and the string representation of an enumeration value and vice versa. The implementation of the class MCDEnumValue is specific to the interface definition. Therefore, its realization with respect to one of the interface definitions is described in the corresponding rules and regulations document.

Apart from the different EventHandler interfaces (the MCDEnumValue is a class not an interface) all interfaces of the ASAM MCD3 model are derived from MCDObject.

In the MCD-3 object model only classes without inheriting child classes can be instantiated. For example, the method MCDDbLocation::getDbEcu returns an object of type MCDDbEcu. A call to this method only may return an object of type MCDDbEcuBaseVariant, MCDDbEcuVariant or MCDDbFunctionalGroup. The MCDDbEcu class that these classes are inheriting from can be considered as abstract and no object of this class shall exist. This is valid for all applicable cases except where explicitly noted.

7.8 Structure of the database

7.8.1 Overview

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 instantiation of the runtime project.

7.8.2 Associations of DbLocation for MCD

Within this ERD the associations between a DbLocation, and the related interfaces are shown. At the DbLocation a number of collections can be accessed, which depends on the association of the DbLocation to M, C, D or MC and MD may be empty or filled with items. The methods are marked with the respective stereotypes, so that the implementation depends on the type of the server. It shall be noted, that in case of MC also the collections of M and C are accessible. The same applies for MD, where also the collections of M and D are accessible.

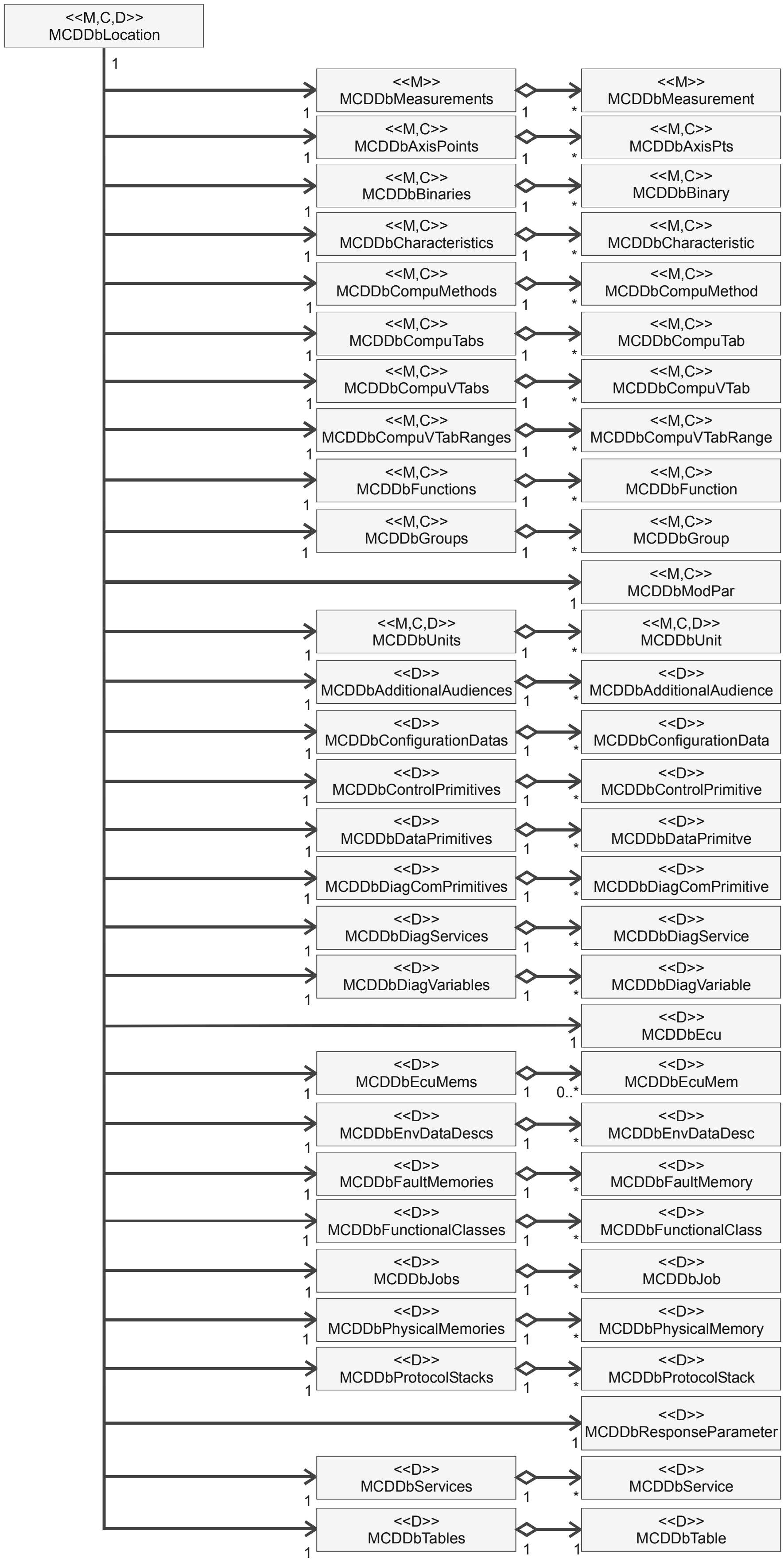


Figure 10 — DbLocation associations

7.8.3 Database within the field Measurement and Calibration

Represents the information from the underlying database description, e.g. parts of an A2L file[12].

7.8.4 Database within the field Diagnostics

The database of an D-server is based on the supported parts and elements of ODX[11] and related information as external flash data files and Java Job libraries. That is, the database part comprises, e.g. information on projects, vehicle information tables, logical links, and diagnostic services.

7.9 Collections

7.9.1 Types and methods

A collection is an object, which manages a list of objects of similar kind. Single objects within the collection may be accessed by means of an index. The index may also be used to iterate over all elements of the collection. The ordering criteria are not defined by the MCD-3 specification. Hence the index of an object could change during the lifetime of the collection. The start index in collections is always zero and the maximum usable index could be queried with the getCount method located at a collection and subtracting 1.

Access to collection items via index is considered harmful. Every time a server-controlled collection is retrieved, the index stays constant, as long as

- the same server instance is used,
- no internal change in the collection has occurred in the server, e.g. the server itself calls add/remove on the collection, and
- no add or removed method is called in the collection by any client or the project has been deselected.

The interface NamedCollection has been derived from the general collection. Named collections additionally offer the possibility to query a list with the names of all elements currently located within the collection (`getNames`) and to access an element via its name (`getItemByName`). The name inside a named collection is always unique, therefore the name could be stored during the lifetime of the collection.

If a collection provides an “add”-method, this method might throw an exception of type MCDProgramViolationException with Error Code `eRT_ELEMENT_ALREADY_EXISTS`, if the element to be added is already contained in the collection. Whether a collection throws this exception or not depends on the semantics defined for this specific collection.

NOTE Collections in MCD-3 have the semantics of a set, that is, every element can only be contained once. Here, it is sufficient that the elements in the collection can be distinguished uniquely by their shortname. Corresponding name generation rules for elements in runtime collections are defined in 7.9.2.2 and 7.9.2.3.

This semantic allows to uniquely identify if a client is the “master” of a resource. Only if the add-method returns without throwing an exception, the client that has called the add-method is considered the “master” of this particular resource.

The semantic of a <<M,C,D>>removeAll-method at a collection is “all or nothing”. That is, if at least one element cannot be removed because of insufficient rights, no element is removed but an exception of type MCDProgramViolationException with error code `eRT_INSUFFICIENT_RIGHTS` is thrown.

Some (database and runtime) collections have “removeByXXX” methods by which specific single items can be removed. After successful call of such a method the remaining items move one position up in the collection. As an consequence the index of the remaining items may change after calling such an method!

All objects in the collection MCDLogicalLinks can always be obtained by any client – independent of the cooperation level. That is, the methods getItemByName() and getItemByIndex() always return an (proxy) object. However, the cooperation level of the returned object might restrict the access to some methods at the proxy object.

A client being able to access a Logical Link (LL has cooperation level of at least eREAD_ONLY), is also able to obtain all elements in the respective MCDDiagComPrimitives collection, the respective MCDCollectors collection, and the MCDCharacteristics collection. However, the cooperation level of the returned object might restrict the access to some methods at the proxy object.

A client being able to access a Collector (Collector has cooperation level of at least eREAD_ONLY), is also able to obtain all elements in the respective MCDCollectedObjects collection. However, the cooperation level of the returned object might restrict the access to some methods at the proxy object.

This solution allows iterating through elements of a collection without causing exceptions to be thrown due to this kind of access.

NOTE The owner (master) of an object in a collection can decide to remove this object from the collection, which causes other clients proxy objects to be marked invalid.

A method returning a collection shall not throw an exception because an empty collection will be returned. All collections will be handled in exactly the same way.

As a general design rule it has been defined that get methods returning a collection shall never throw an exception, e.g. because the collection is empty. However, when accessing the database, it is always possible that inconsistencies in the database are detected by the MCD-server. This also applies to get methods for collections. As a result, the cases ‘empty collection’ because no data available in the database and ‘inconsistent database’ need to be distinguished.

To solve this, it has been proposed that all methods accessing database objects (getDbXXXX()) can throw a MCDDatabaseException with error code `eDB_INCONSISTENT_DATABASE`.

The method `getItemByIndex(A_UINT32 index)` throws an exception of type `MCDParameterizationException` with error code `ePAR_INDEX_OUT_OF_RANGE` in case the value supplied for the parameter index is greater or equal to the number of elements in the collection. This does also apply in case of an empty collection.

The numbering in collections starts at zero.

The method `getItemByName(MCDDatatypeShortname name)` throws an exception of type `MCDParameterizationException` with error code `ePAR_ITEM_NOT_FOUND`. This does also apply in case of an empty collection.

7.9.2 RunTime collections

7.9.2.1 Overview

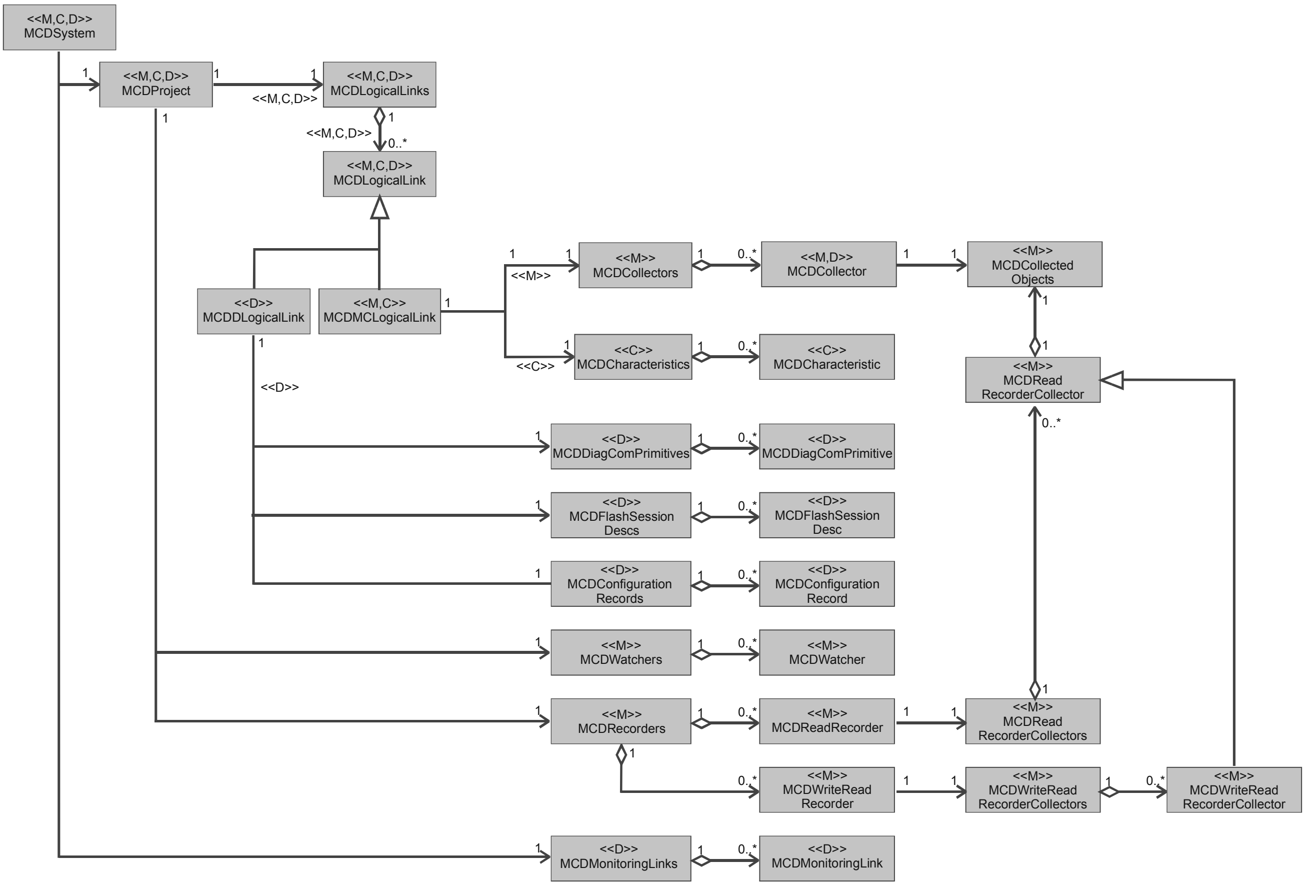


Figure 11 — RunTime collections MCD

Within this ERD the associations between `MCDProject` and its Logical Links are shown. On the one hand the relations of the runtime project to the DbProject and to the selected Vehicle Information Table are shown and on the other hand the relations of the Logical Link to the Collectors (M), Characteristics (C) and DiagComPrimitives and Services (D). Depending on the type the Logical Link is assigned to, it may create and use the respective runtime objects. The logical link type is determined by the Db object that is used for its creation.

In general, obtaining runtime collections does not cause any exception. If no corresponding information is available, an empty collection is returned.

If a database collection is obtained, an exception can be thrown in case the database is inconsistent. If the corresponding information is not available in the database, an empty collection is returned. In case an error has been detected in the database when trying to obtain the information, an exception of type MCDDatabaseException with error code eDB_INCONSISTENT_DATABASE is thrown.

If the access to a method to obtain a collection is restricted because of cooperation levels or because of locks, corresponding exceptions can be thrown:

— the called object or an object above in the object hierarchy is locked by another client (eRT_OBJECT_IS_LOCKED, MCDProgramViolationException);

— the method can't be called because of insufficient rights of the non-master client in a multi-client scenario. (eRT_INSUFFICIENT_RIGHTS, MCDProgramViolationException).

7.9.2.2 RunTime collections in MC

For all run time collections within the function block MC applies: Each Client may see and use all items of a Collection existing within the Server, independently from the fact which Client has created the Items or uses the Items. Because of this it is e.g. not possible to create several Runtime Characteristics for a database Characteristic and to parameterise them differently. That means, that existing Runtime Characteristics may be parameterised as new by each Client if no safety mechanisms are intended (locking).

There is the method `removeAll`, which deals with all objects in the collection. The MC-server will try to remove each of the objects. The objects which cannot be removed (and only these) will be left in the collection. After returning from this method the Client can call `getCount` to get the number of objects, which were not removed, and `getItemByIndex` to get the items itself. The other method depending on all objects of their collection is `MCDCollectors::deactivate():MCDCollectors`. This method will create and return a temporary collection of Collectors which could not be deactivated.

7.9.2.3 RunTime collections in D

At D-server runtime, the runtime counterpart of a DB-object can occur multiple times in the same collection, e.g. a `MCDDiagComPrimitive` in the collection `MCDDiagComPrimitives` or a `MCDResponseParameter` in the collection `MCDResponseParameters`. However, as the majority of these collections are derived from class `MCDNamedCollection`, a unique ShortName for the different objects is required.

While a collection can contain multiple runtime objects referring to the same DB-object, the following pattern is to be used to generate the ShortName of the runtime objects contained in this collection: #RtGen_<ShortnameOfCorrespondingDbObject>_<Number>.

The number in this generated ShortName is increased with every new instance of the same DB-object in the same collection. For the first instance, the number is zero. This pattern is also used if only a single instance of a DB-object is part of a collection. As this fact is nondeterministic, it cannot be predicted by the D-server whether more instances of the same DB-object will be added at a later point in time. The name of the corresponding database objects can still be obtained by using the `getDbObject()::getShortName`() methods found on the runtime objects with generated short names.

The following collections shall apply this generation rule for their runtime elements:

- `MCDResponseParameters,` if the containing `MCDResponseParameter` is of type `eFIELD`;
- `MCDRequestParameters,` if the containing `MCDRequestParameter` is of type `eEND_OF_PDU` or `eFIELD`;
- `MCDDiagComPrimitives`;
- `MCDFlashSessionDescs`;
- `MCDConfigurationRecords`.

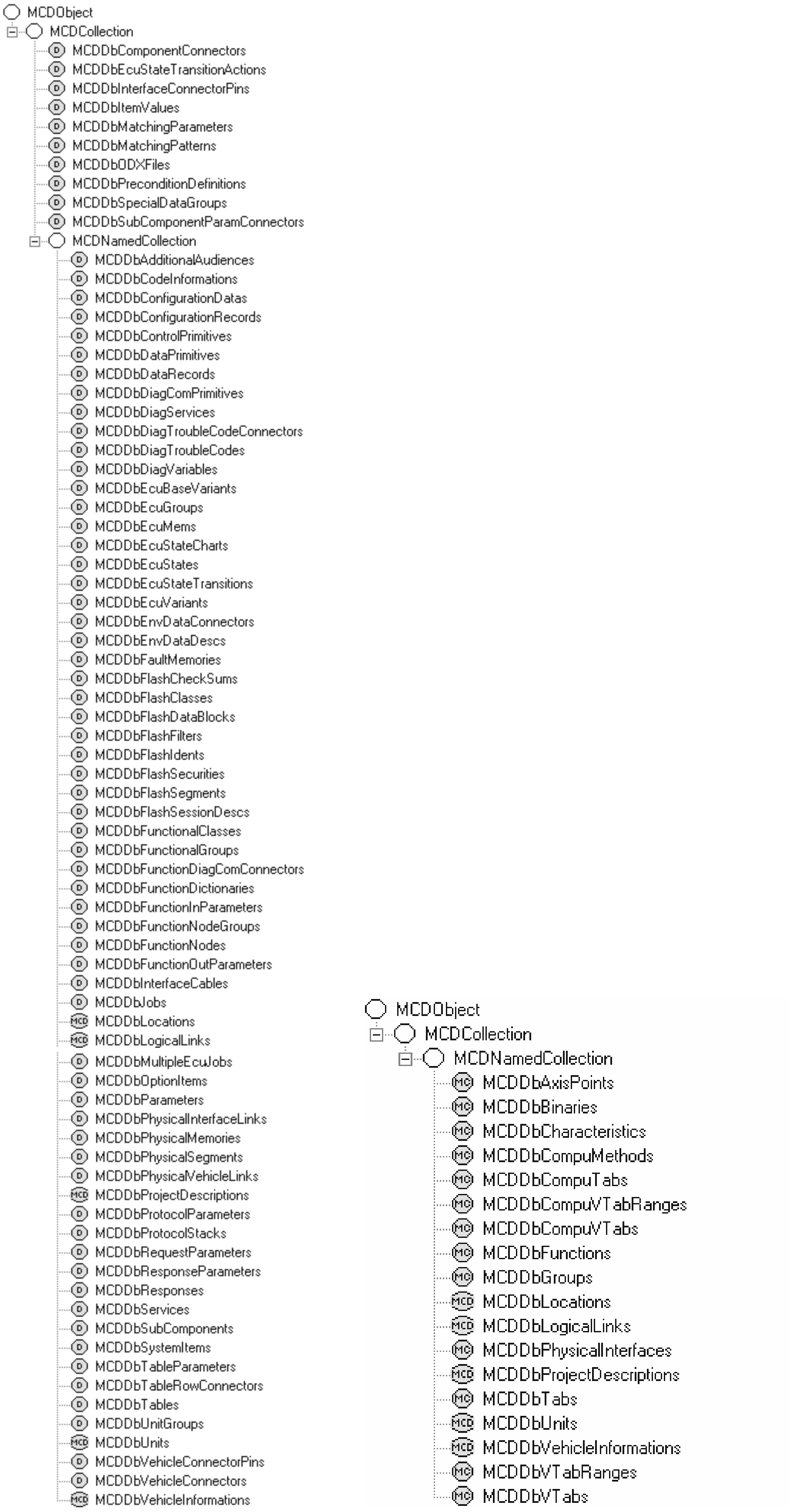
NOTE The collection `MCDLogicalLinks` is not affected, as there can only be a single instance of each logical link at runtime.

Service result structures and their parameter structures are also mapped to runtime collections. However, there is no danger that names are assigned twice in these collections, as they are always created by the D-server or a Java job.

Besides the naming rules for multiple runtime instances of Db-elements, another issue that needs consideration is the question of how to deal with indexes of elements in a runtime collection in case an element is removed. In this case, the index values of the remaining items will be recalculated to fill any gaps that are left by the removed items. For example, if an item at index/position 1 is removed from a collection which contains 3 items [at indexes 0, 1, 2], after the removal of the item the collection will contain 2 elements with indexes 0 and 1.

7.9.3 Database collections

Collections are used for the listing of identical database objects. The Collections reflect the content of the database of the used project. The database is static; that means nothing can be added (except for MC within the configuration phase and except for D for ConfigurationDatas and ECU-Mems) or modified. The collections are always derived from `MCDNamedCollection`, so that its items may be accessed via index and via name. It shall be guaranteed for the uniqueness of the names of the items within the database.



- ○ MCDObject
- ○ MCDCollection
  - (D) MCDDbComponentConnectors
  - (D) MCDDbEcuStateTransitionActions
  - (D) MCDDbInterfaceConnectorPins
  - (D) MCDDbItemValues
  - (D) MCDDbMatchingParameters
  - (D) MCDDbMatchingPatterns
  - (D) MCDDbODXFiles
  - (D) MCDDbPreconditionDefinitions
  - (D) MCDDbSpecialDataGroups
  - (D) MCDDbSubComponentParamConnectors
  - ○ MCDNamedCollection
    - (D) MCDDbAdditionalAudiences
    - (D) MCDDbCodeInformations
    - (D) MCDDbConfigurationDatas
    - (D) MCDDbConfigurationRecords
    - (D) MCDDbControlPrimitives
    - (D) MCDDbDataPrimitives
    - (D) MCDDbDataRecords
    - (D) MCDDbDiagComPrimitives
    - (D) MCDDbDiagServices
    - (D) MCDDbDiagTroubleCodeConnectors
    - (D) MCDDbDiagTroubleCodes
    - (D) MCDDbDiagVariables
    - (D) MCDDbEcuBaseVariants
    - (D) MCDDbEcuGroups
    - (D) MCDDbEcuMems
    - (D) MCDDbEcuStateCharts
    - (D) MCDDbEcuStates
    - (D) MCDDbEcuStateTransitions
    - (D) MCDDbEcuVariants
    - (D) MCDDbEnvDataConnectors
    - (D) MCDDbEnvDataDescs
    - (D) MCDDbFaultMemories
    - (D) MCDDbFlashCheckSums
    - (D) MCDDbFlashClasses
    - (D) MCDDbFlashDataBlocks
    - (D) MCDDbFlashFilters
    - (D) MCDDbFlashIdents
    - (D) MCDDbFlashSecurities
    - (D) MCDDbFlashSegments
    - (D) MCDDbFlashSessionDescs
    - (D) MCDDbFunctionalClasses
    - (D) MCDDbFunctionalGroups
    - (D) MCDDbFunctionDiagComConnectors
    - (D) MCDDbFunctionDictionaries
    - (D) MCDDbFunctionInParameters
    - (D) MCDDbFunctionNodeGroups
    - (D) MCDDbFunctionNodes
    - (D) MCDDbFunctionOutParameters
    - (D) MCDDbInterfaceCables
    - (D) MCDDbJobs
    - (MCD) MCDDbLocations
    - (MCD) MCDDbLogicalLinks
    - (D) MCDDbMultipleEcuJobs
    - (D) MCDDbOptionItems
    - (D) MCDDbParameters
    - (D) MCDDbPhysicalInterfaceLinks
    - (D) MCDDbPhysicalMemories
    - (D) MCDDbPhysicalSegments
    - (D) MCDDbPhysicalVehicleLinks
    - (MCD) MCDDbProjectDescriptions
    - (D) MCDDbProtocolParameters
    - (D) MCDDbProtocolStacks
    - (D) MCDDbRequestParameters
    - (D) MCDDbResponseParameters
    - (D) MCDDbResponses
    - (D) MCDDbServices
    - (D) MCDDbSubComponents
    - (D) MCDDbSystemItems
    - (D) MCDDbTableParameters
    - (D) MCDDbTableRowConnectors
    - (D) MCDDbTables
    - (D) MCDDbUnitGroups
    - (MCD) MCDDbUnits
    - (D) MCDDbVehicleConnectorPins
    - (D) MCDDbVehicleConnectors
    - (MCD) MCDDbVehicleInformations

- ○ MCDObject
- ○ MCDCollection
  - ○ MCDNamedCollection
    - (MC) MCDDbAxisPoints
    - (MC) MCDDbBinaries
    - (MC) MCDDbCharacteristics
    - (MC) MCDDbCompuMethods
    - (MC) MCDDbCompuTabs
    - (MC) MCDDbCompuVTabRanges
    - (MC) MCDDbCompuVTabs
    - (MC) MCDDbFunctions
    - (MC) MCDDbGroups
    - (MCD) MCDDbLocations
    - (MCD) MCDDbLogicalLinks
    - (MC) MCDDbPhysicalInterfaces
    - (MCD) MCDDbProjectDescriptions
    - (MC) MCDDbTabs
    - (MCD) MCDDbUnits
    - (MCD) MCDDbVehicleInformations
    - (MC) MCDDbVTabRanges
    - (MC) MCDDbVTabs

Figure 12 — Database collections in D and MC

7.9.4 Handling of collection of ASCIISTRING

For a better handling of the ASAM data type A_ASCIISTRING a class was introduced to the object model: `MCDDatatypeAsciiString`. It maps the A_ASCIISTRING to a more model compliant type in a one to one relation.

To allow collections of MCD data types a class `MCDDatatypeCollection` is defined. It provides the core functionality for data type collections: `getCount` and `removeAll`. The collection of objects of the type `MCDDatatypeAsciiString` is inherited from this.

7.10 EventHandler

7.10.1 Registering/deregistering of the EventHandlers

The MCD-server API provides a set of specialized event handlers. Each of these event handlers offers methods to send a set of closely related events. However, the general procedure of registering and deregistering an event handler is the same for all of these event handlers.

The current state of an object will NOT be delivered to a registered event handler by means of a synthetic event. Instead, the client registering the event handler is required to actively poll the current state by means of the corresponding methods.

The event which results from a state change or which results from an action, e.g. adding or removing elements from a collection of shared objects, is sent immediately after the new state has been reached successfully or after the corresponding action has been executed successfully.

The following methods shall be used with extreme care in the body of an event handler as their usage can result in deadlocks or cascades of events at runtime:

— setEventHandler();

— releaseEventHandler();

— execution of methods resulting in new events, especially if these events are sent to the same event handler;

— all methods, which change the data of, shared objects.

Besides above exception, it is always allowed to register, deregister, and configure event handlers at any time, provided that the cooperation level of the corresponding (shared) object is at least `READ_ONLY`. The result of these actions becomes active with the next event to be sent to the corresponding event handler.

NOTE 1 Above can result in an incomplete set of events being received at an event handler.

Because of this kind of registering several EventHandler may be connected per object at the same time. These are identified by means of the Id that is assigned at registration.

NOTE 2 These event handler IDs are server-wide unique at server runtime.

If the Server cannot register another EventHandler because of internal restrictions, e.g. resources or exceeding `MAX A_UINT32` (232) EventHandlers, a MCDProgramViolation will be thrown. It contains the error `eSYSTEM_RESSOURCE_OVERLOAD`.

However, it is recommended to register at least a MCDSystemEventHandler at the client's MCDSystem object. Otherwise, the client will not be informed about system events or unchecked system errors. Registering an event handler at any other object is optional.

NOTE 3 No events will be forwarded to a higher level. The only exception to this rule is that DiagComPrimitive events are forwarded to the MCDDiagComPrimitives collection the corresponding DiagComPrimitive is contained in. Analogously events from FlashSessionDesc are forwarded to FlashSessionDescs collection.

For a client implementing an event handler is optional (scripted client). For a MCD-server, the implementation of event handling is mandatory.

State changes (attribute changes) of shared objects, e.g. LogicalLinks, DiagComPrimitives, and Collectors should only be delivered to those clients that have attached a EventListener at the corresponding object (e.g., System, LogicalLink, Collector, or DiagComPrimitive).

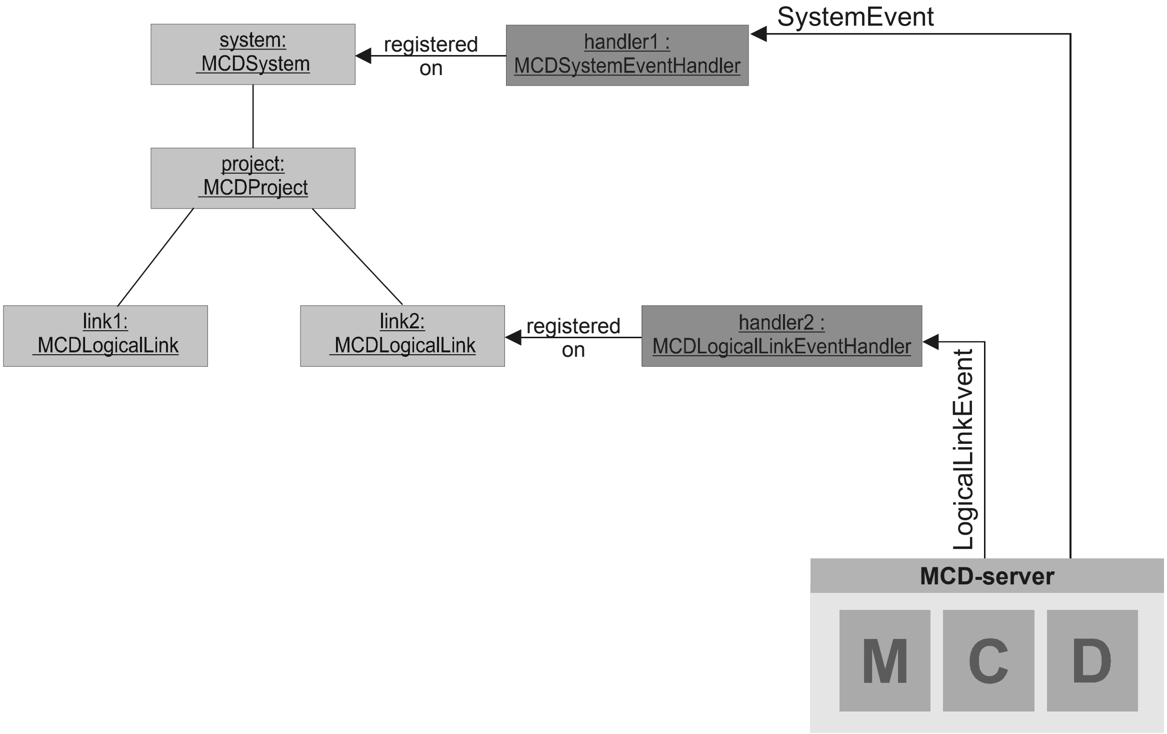


Figure 13 — Using of EventHandler

The `releaseEventHandler` method can be called in any state without throwing an exception with respect to the state – provided that the cooperation level of the corresponding (shared) object is at least `READ_ONLY` (Implementation hint to the client programmer: The `releaseEventHandler` method should be called in the same states where the corresponding `setEventHandler` had been called).

To conveniently be able to listen to all events of any DiagComPrimitive in the collection of DiagComPrimitives at a LogicalLink, it is sufficient to set an event handler at the corresponding collection. The same applies to FlashSessionDescs and MCDConfigurationRecords. To provide additional convenience, EventHandlers registered at an object are automatically deregistered by the MCD-server in case this object is removed.

EventHandlers are removed automatically in case an observed object is removed.

7.10.2 Methods of the EventHandlers

The MCDGlobalEventHandler could not be set directly. Only the derived classes (MCDCollectorEventHandler and MCDWriteReadRecorderEventHandler) are set at MCDCollector and MCDWriteReadRecorder. DiagComPrimitives Eventhandler can be set at the collection of DiagComPrimitives.

Table 2 — Methods of the MCDSystemEventHandler

Stereotype	Event	Semantic
<<D>>	`onBaseVariantIdentified`	Invoked by the MCD-server when a Base Variant within the current project is identified.
<<D>>	`onBaseVariantSelected`	Invoked by the MCD-server when a Base Variant within the current project is selected.
<<D>>	`onInterfaceError`	Invoked by the D-server whenever an interface error event is received by the D-server, i.e. the PDU-API sends one of the events PDU_ERR_EVT_VCI_HARDWARE_FAULT or PDU_ERR_EVT_LOST_COMM_TO_VCI.
<<D>>	`onMonitoringLinksModified`	Invoked by the MCD-server when a client modifies the collection MCDMonitoringLinks.
<<D>>	`onSystemDbConfigurationDatasModified`	Invoked by the MCD-server when a client modifies the collection MCDDbConfigurationDatas.
<<D>>	`onSystemDbEcuMemsModified`	Invoked by the MCD-server when a client modifies the collection MCDDbEcuMems.
<<D>>	`onSystemInterfacesModified`	A new VCI module has become available, or an available VCI module is not available any more. The application shall call getCurrentInterfaces() to retrieve the currently available modules.
<<D>>	`onSystemInterfaceStatusChanged`	The status of the given VCI module has changed.
<<M,C,D>>	`onSystemClientConnected`	Invoked by the MCD-server whenever a client has been connected to the MCDSystem.
<<M,C,D>>	`onSystemClientDisconnected`	Invoked by the MCD-server whenever a client has been disconnected to the MCDSystem.
<<M,C,D>>	`onSystemClientNameChanged`	Invoked by the MCD-server whenever a client named has been modified.
<<M,C,D>>	`onSystemDbProjectConfiguration`	Invoked by the MCD-server when the state of the MCDSystem is changed to eDBPROJECT_CONFIGURATION.
<<M,C,D>>	`onSystemError`	Invoked by the MCD-server when an error in the System occurs. This is not a special error for a Collector, Logical Link or DiagComPrimitive.
<<M,C,D>>	`onSystemInitialized`	Invoked by the MCD-server when the state of the MCDSystem is changed to eINITIALIZED.
<<M,C,D>>	`onSystemLocked`	Invoked by the MCD-server when the MCDSystem object is locked.
<<M,C,D>>	`onSystemLogicalLinksModified`	Invoked by the MCD-server when a client modifies the collection MCDLogicalLinks.
<<M,C,D>>	`onSystemLogicallyConnected`	Invoked by the MCD-server when the state of the MCDSystem is changed to eLOGICALLY_CONNECTED.
<<M,C,D>>	`onSystemProjectSelected`	Invoked by the MCD-server when the state of the MCDSystem is changed to ePROJECT_SELECTED.

Table 2 (*continued*)

Stereotype	Event	Semantic
<<M,C,D>>	`onSystemUnlocked`	Invoked by the MCD-server when the MCDSystem object is unlocked.
<<M,C,D>>	`onSystemVehicleInformationSelected`	Invoked by the MCD-server when the state of the MCDSystem is changed to eVIT_SELECTED.
<<M,C>>	`onSystemRecordersModified`	Invoked by the MCD-server when a client modifies the collection MCDRecorders.
<<M>>	`onSystemWatchersModified`	Invoked by the MCD-server when a client modifies the collection MCDWatchers, e.g. adds a MCDWatcher to the collection or removes MCDWatcher from the collection.

Table 3 — Methods of the MCDConfigurationRecordEventHandler

Stereotype	Event	Semantic
<<D>>	`onConfigurationRecordLoaded`	Invoked by the MCD-server when a configuration record is loaded.
<<D>>	`onConfigurationRecordLocked`	Invoked by the MCD-server when the MCDConfigurationRecord object is locked.
<<D>>	`onConfigurationRecordUnlocked`	Invoked by the MCD-server when the MCDConfigurationRecord object is unlocked.

Table 4 — Methods of the MCDWriteReadRecorderEventHandler

Stereotype	Event	Semantic
<<M>>	`onGlobalObjectActivated`	Invoked by the MCD-server if the respective state was entered.
<<M>>	`onGlobalObjectConfigured`	Invoked by the MCD-server if the respective state was entered.
<<M,D>>	`onGlobalObjectCreated`	Invoked by the MCD-server if the respective state was entered.
<<M>>	`onGlobalObjectLocked`	Invoked by the MCD-server if the respective state was entered.
<<M,D>>	`onGlobalObjectStarted`	Invoked by the MCD-server if the respective state was entered.
<<M>>	`onGlobalObjectUnlocked`	Invoked by the MCD-server if the respective state was entered.
<<M>>	`onWriteReadRecorderError`	Invoked by the MCD-server if an error occurred.
<<M>>	`onWriteReadRecorderModified`	Invoked by the MCD-server when a client modifies the collection MCDWriteReadRecorderCollectors or any contained MCDCollectedObjects collection, e.g. adds a collector to the collection or removes a collector from the collection.
<<M>>	`onWriteReadRecorderPaused`	Invoked by the MCD-server if the respective state was entered.
<<M>>	`onWriteReadRecorderStopped Acquisition`	This event transports the information about a MCDWriteReadRecorder if a recorder collector has stopped acquiring data due to an error state in the associated LogicalLink.

Table 5 — Methods of the MCDFlashSessionDescEventHandler

Stereotype	Event	Semantic
<<D>>	`onFlashSegmentsCleared`	Invoked by the MCD-server when a flash segment is cleared.
<<D>>	`onFlashSegmentsLoaded`	Invoked by the MCD-server when a flash segment is loaded.
<<D>>	`onFlashSessionDescLocked`	Invoked by the MCD-server when the MCDFlashSessionDesc object is locked.
<<D>>	`onFlashSessionDescUnlocked`	Invoked by the MCD-server when the MCDFlashSessionDesc object is unlocked.

Table 6 — Methods of the MCDCollectorEventHandler

Stereotype	Event	Semantic
<<D>>	`onCollectorMonitoringFramesReady`	Invoked by the MCD-server when new monitoring frames are available at a collector.
<<M>>	`onCollectorCollectedObjectsModified`	Invoked by the MCD-server when a client modifies the collection MCDCollectedObjects.
<<M,D>>	`onCollectorError`	Invoked by the MCD-server when an error occurs while using the Collector.
<<M>>	`onCollectorResultReady`	Invoked by the MCD-server when a result of a Collector is ready.
<<M>>	`onGlobalObjectActivated`	Invoked by the MCD-server if the respective state was entered.
<<M>>	`onGlobalObjectConfigured`	Invoked by the MCD-server if the respective state was entered.
<<M,D>>	`onGlobalObjectCreated`	Invoked by the MCD-server if the respective state was entered.
<<M>>	`onGlobalObjectLocked`	Invoked by the MCD-server if the respective state was entered.
<<M,D>>	`onGlobalObjectStarted`	Invoked by the MCD-server if the respective state was entered.
<<M>>	`onGlobalObjectUnlocked`	Invoked by the MCD-server if the respective state was entered.

Table 7 — Methods of the MCDGlobalEventHandler

Stereotype	Event	Semantic
<<M>>	`onGlobalObjectActivated`	Invoked by the MCD-server if the respective state was entered.
<<M>>	`onGlobalObjectConfigured`	Invoked by the MCD-server if the respective state was entered.
<<M,D>>	`onGlobalObjectCreated`	Invoked by the MCD-server if the respective state was entered.
<<M>>	`onGlobalObjectLocked`	Invoked by the MCD-server if the respective state was entered.
<<M,D>>	`onGlobalObjectStarted`	Invoked by the MCD-server if the respective state was entered.
<<M>>	`onGlobalObjectUnlocked`	Invoked by the MCD-server if the respective state was entered.

Table 8 — Methods of the MCDDiagComPrimitiveEventHandler

Stereotype	Event	Semantic
<<D>>	`onPrimitiveBufferOverflow`	Invoked by the MCD-server when the result buffer of a DiagComPrimitive has been overflown.
<<D>>	`onPrimitiveHasIntermediateResult`	Invoked by the MCD-server when the DiagComPrimitive (Job) has been delivered an intermediate result.
<<D>>	`onPrimitiveHasResult`	Invoked by the MCD-server when the DiagComPrimitive has been delivered an result. This event is sent if a DiagComPrimitive or a Job have finished asynchronous execution completely or if a repetition cycle has finished.
<<D>>	`onPrimitiveIdle`	This event is sent if the state eIDLE is reached in the state diagram of an MCDDiagComPrimitive.
<<D>>	`onPrimitiveJobInfo`	Invoked by the D-server when a Java job has sent a job info by calling MCDJobApi::setJobInfo(...).
<<D>>	`onPrimitiveLocked`	Invoked by the MCD-server when the MCDDiagComPrimitive object is locked.
<<D>>	`onPrimitiveNotExecutable`	This event is sent if the state eNOT_EXECUTABLE is reached in the state diagram of an MCDDiagComPrimitive. This event is only send if the MCDDiagComPrimitive become invalid if a variant identification and selection is processed.
<<D>>	`onPrimitivePending`	This event is sent if the state ePENDING is reached in the state diagram of an MCDDiagComPrimitive.
<<D>>	`onPrimitiveProgressInfo`	Invoked by the MCD-server when the DiagComPrimitive has been sent an progress info.
<<D>>	`onPrimitiveRepeating`	This event is sent if the state eREPEATING is reached in the state diagram of an MCDDiagComPrimitive.
<<D>>	`onPrimitiveUnlocked`	Invoked by the MCD-server when the MCDDiagComPrimitive object is unlocked.
<<D>>	`onStartMessageIndication`	Called only if PDU API fires the corresponding events (The corresponding ComParams shall be enabled)
<<D>>	`onTransmitIndication`	Called only if PDU API fires the corresponding events (The corresponding ComParams shall be enabled)

Table 9 — Methods of the MCDWatcherEventHandler

Stereotype	Event	Semantic
<<M>>	`onWatcherActivated`	This event transports the information about an MCDWatcher if the respective state was entered.
<<M>>	`onWatcherCreated`	Invoked by the MCD-server if the respective state was entered.
<<M>>	`onWatcherDeactivated`	This event transports the information about an MCDWatcher if the respective state was entered.
<<M>>	`onWatcherError`	This method is called if the watcher has an asynchronous error. Following errors could occur: the watcher could not be activated eRT_WATCHER_ACTIVATION_FAILED (MCDProgramViolationException)

Table 9 (*continued*)

Stereotype	Event	Semantic
<<M>>	`onWatcherFired`	This event transports the information about an MCDWatcher if the respective state was entered.
<<M>>	`onWatcherLocked`	This method is called if the watcher reaches the locked state.
<<M>>	`onWatcherUnlocked`	This method is called if the watcher reaches the unlocked state.

Table 10 — Methods of the MCDLogicalLinkEventHandler

Stereotype	Event	Semantic
<<C>>	`onLinkCharacteristicsModified`	Invoked by the MCD-server when a client modifies the collection MCDCharacteristics.
<<D>>	`onLinkActivityIdle`	Invoked by the MCD-server when the instruction queue of a LogicalLink has performed a state transition to eACTIVITY_IDLE either through creation or from eACTIVITY_RUNNING.
<<D>>	`onLinkActivityRunning`	Invoked by the MCD-server when the instruction queue of a LogicalLink has performed a state transition to eACTIVITY_RUNNING either from eACTIVITY_IDLE or from eACTIVITY_SUSPENDED.
<<D>>	`onLinkActivitySuspended`	Invoked by the MCD-server when the instruction queue of a LogicalLink has performed a state transition to eACTIVITY_SUSPENDED from eACTIVITY_RUNNING.
<<D>>	`onLinkConfigurationRecordsModified`	Invoked by the MCD-server client modifies the collection MCDConfigurationRecords.
<<D>>	`onLinkCreated`	Invoked by the MCD-server when the state of the MCDLogicalLink is changed to eCREATED.
<<D>>	`onLinkDefinableDynIdListChanged`	Invoked by the MCD-server when a DynID list has changed.
<<D>>	`onLinkDiagComPrimitivesModified`	Invoked by the MCD-server when a client modifies the collection MCDDiagComPrimitives.
<<D>>	`onLinkFlashSessionDescsModified`	Invoked by the MCD-server client modifies the collection MCDFlashSessionDescs.
<<D>>	`onLinkQueueCleared`	Invoked by the MCD-server when the instruction queue of a LogicalLink has been cleared by invoking the corresponding method.
<<D>>	`onLinkVariantIdentified`	Invoked by the D-server in case a variant has been identified for a logical link with location type eFUNCTIONAL_GROUP.
<<D>>	`onLinkVariantSelected`	Invoked by the MCD-server when the Logical Link has been identified and selected for a logical link with location type eFUNCTIONAL_GROUP.
<<D>>	`onLinkVariantSet`	Invoked by the MCD-server when the Variant of a Logical Link is set.
<<M, C>>	`onLinkTargetOff`	This event is invoked by the MCD-server when the state of the connected target changes to off.
<<M, C>>	`onLinkTargetOn`	This event is invoked by the MCD-server when the state of the connected target changes to on.

Table 10 (*continued*)

Stereotype	Event	Semantic
<<M, C>>	`onLinkTargetUnknown`	This event is invoked by the MCD-server when the state of the connected target could not be determined.
<<M,C,D>>	`onLinkCommunication`	Invoked by the MCD-server when the state of the MCDLogicalLink is changed to eCOMMUNICATION.
<<M,C,D>>	`onLinkError`	Invoked by the MCD-server when an error on a Logical Link occurs.
<<M,C,D>>	`onLinkLocked`	Invoked by the MCD-server when a Logical link has been locked.
<<M,C,D>>	`onLinkOffline`	Invoked by the MCD-server when the state of the MCDLogicalLink is changed to eOFFLINE.
<<M,C,D>>	`onLinkOnline`	Invoked by the MCD-server when the state of the MCDLogicalLink is changed to eONLINE.
<<M,C,D>>	`onLinkUnlocked`	Invoked by the MCD-server when a Logical Link has been unlocked.
<<M>>	`onLinkCollectorsModified`	Invoked by the MCD-server when a client modifies the collection MCDCollectors.

7.10.3 Eventfilter

A Client has now the possibility to define which events should be reported, that means, if a client creates an event handler, the client is also able to select which events to be delivered by the server.

The method `configureEventHandler(eventHandlerId : A_UINT32, filterSettings : A_UINT32) : void` is used at all objects, where `setEventHandler` and `releaseEventHandler` is allowed. The method is only valid if the accompanying EventHandler exists, otherwise the error of class Run Time / Program Violation Errors `eRT_NO_ACTIVE_OBJECT` is used. If an Event Handler exists `configureEventHandler` can be called at any time.

The filter settings are defined by enumerations, which define the bit position inside the filter setting. These enumerations are eventhandler specific.

Table 11 — Enumeration MCDSystemEventTypes

Type	Name	InitValue	Documentation
<<D>>	`eFSS_ONBASEVARIANT IDENTIFIED`	0x00080000	This bit can be used to temporarily deactivate the event onBaseVariantIdentified. For this purpose, the eFSS_ONBASEVARIANTIDENTIFIED bit shall be set to zero.
<<D>>	`eFSS_ONBASEVARIANT SELECTED`	0x00100000	This bit can be used to temporarily deactivate the event onBaseVariantSelected. For this purpose, the eFSS_ONBASEVARIANTSELECTED bit shall be set to zero.
<<D>>	`eFSS_ONMONITORINGLINKS MODIFIED`	0x00200000	This bit can be used to temporarily deactivate the event onBaseVariantSelected. For this purpose, the eFSS_ONMONITORINGLINKSMODIFIED bit shall be set to zero.
<<D>>	`eFSS_ONSYSTEMDB CONFIGURATIONDATAS MODIFIED`	0x00040000	This bit can be used to temporarily deactivate the event onSystemDbConfigurationDatasModified. For this purpose, the eFSS_ONSYSTEMDBCONFIGURATIONDATASMODIFIED bit shall be set to zero.

Table 11 (*continued*)

Type	Name	InitValue	Documentation
<<D>>	eFSS_ONSYSTEMDBECUMEMS MODIFIED	0x00020000	This bit can be used to temporarily deactivate the event onSystemDbEcuMemsModified. For this purpose, the eFSS_ONSYSTEMDBECUMEMSMODIFIED bit shall be set to zero.
<<D>>	eFSS_ONSYSTEMINTERFACES MODIFIED	0x00008000	This bit can be used to temporarily deactivate the event onSystemInterfacesChanged. For this purpose, the eFSS_ONSYSTEMINTERFACESMODIFIED bit shall be set to zero.
<<D>>	eFSS_ONSYSTEMINTERFACE STATUSCHANGED	0x00010000	This bit can be used to temporarily deactivate the event onSystemInterfaceStatusChanged. For this purpose, the eFSS_ONSYSTEMINTERFACESTATUSCHANGED bit shall be set to zero.
<<M,C,D>>	eFSS_ONSYSTEMCLIENT CONNECTED	0x00000002	This bit can be used to temporarily deactivate the event onSystemClientConnected. For this purpose, the eFSS_ONSYSTEMCLIENTCONNECTED bit shall be set to zero.
<<M,C,D>>	eFSS_ONSYSTEMCLIENT DISCONNECTED	0x00000004	This bit can be used to temporarily deactivate the event onSystemClientDisconnected. For this purpose, the eFSS_ONSYSTEMCLIENTDISCONNECTED bit shall be set to zero.
<<M,C,D>>	eFSS_ONSYSTEMCLIENTNAME CHANGED	0x00000008	This bit can be used to temporarily deactivate the event onSystemClientNameChanged. For this purpose, the eFSS_ONSYSTEMCLIENTNAMECHANGED bit shall be set to zero.
<<M,C,D>>	eFSS_ONSYSTEMDBPROJECT CONFIGURATION	0x00000010	This bit can be used to temporarily deactivate the event onSystemDbProjectConfiguration. For this purpose, the eFSS_ONSYSTEMDBPROJECTCONFIGURATION bit shall be set to zero.
<<M,C,D>>	eFSS_ONSYSTEMERROR	0x00000020	This bit can be used to temporarily deactivate the event onSystemError. For this purpose, the eFSS_ONSYSTEMERROR bit shall be set to zero.
<<M,C,D>>	eFSS_ONSYSTEMINITIALIZED	0x00000040	This bit can be used to temporarily deactivate the event onSystemInitialized. For this purpose, the eFSS_ONSYSTEMINITIALIZED bit shall be set to zero.
<<M,C,D>>	eFSS_ONSYSTEMLOCKED	0x00000800	This bit can be used to temporarily deactivate the event onSystemLocked. For this purpose, the eFSS_ONSYSTEMLOCKED bit shall be set to zero.
<<M,C,D>>	eFSS_ONSYSTEMLOGICAL LINKSMODIFIED	0x00000080	This bit can be used to temporarily deactivate the event onSystemLogicalLinksModified. For this purpose, the eFSS_ONSYSTEMLOGICALLINKSMODIFIED bit shall be set to zero.
<<M,C,D>>	eFSS_ONSYSTEMLOGICALLY CONNECTED	0x00000100	This bit can be used to temporarily deactivate the event onSystemLogicallyConnected. For this purpose, the eFSS_ONSYSTEMLOGICALLYCONNECTED bit shall be set to zero.
<<M,C,D>>	eFSS_ONSYSTEMPROJECT SELECTED	0x00000200	This bit can be used to temporarily deactivate the event onSystemProjectSelected. For this purpose, the eFSS_ONSYSTEMPROJECTSELECTED bit shall be set to zero.

Table 11 (*continued*)

Type	Name	InitValue	Documentation
<<M,C,D>>	eFSS_ONSYSTEMUNLOCKED	0x00001000	This bit can be used to temporarily deactivate the event onSystemUnlocked. For this purpose, the eFSS_ONSYSTEMUNLOCKED bit shall be set to zero.
<<M,C,D>>	eFSS_ONSYSTEMVEHICLEINFORMATIONSELECTED	0x00000400	This bit can be used to temporarily deactivate the event onSystemVehicleInformationSelected. For this purpose, the eFSS_ONSYSTEMVEHICLEINFORMATIONSELECTED bit shall be set to zero.
<<M,C>>	eFSS_ONSYSTEMRECORDERSMODIFIED	0x00002000	This bit can be used to temporarily deactivate the event onSystemRecordersModified. For this purpose, the eFSS_ONSYSTEMRECORDERSMODIFIED bit shall be set to zero.
<<M>>	eFSS_ONSYSTEMWATCHERSMODIFIED	0x00004000	This bit can be used to temporarily deactivate the event onSystemWatchersModified. For this purpose, the eFSS_ONSYSTEMWATCHERSMODIFIED bit shall be set to zero.

Table 12 — Enumeration MCDConfigurationRecordEventTypes

Type	Name	InitValue	Documentation
<<D>>	eFSCR_ONCONFIGURATIONRECORDLOADED	0x00000004	This bit can be used to temporarily deactivate the event onConfigurationRecordLoaded. For this purpose, the eFSCR_ONCONFIGURATIONRECORDLOADED bit shall be set to zero.
<<D>>	eFSCR_ONCONFIGURATIONRECORDLOCKED	0x00000001	This bit can be used to temporarily deactivate the event onConfigurationRecordLocked. For this purpose, the eFSCR_CONFIGURATIONRECORDLOCKED bit shall be set to zero.
<<D>>	eFSCR_ONCONFIGURATIONRECORDUNLOCKED	0x00000002	This bit can be used to temporarily deactivate the event onConfigurationRecordUnlocked. For this purpose, the eFSCR_ONCONFIGURATIONRECORDUNLOCKED bit shall be set to zero.

Table 13 — Enumeration MCDWriteReadRecorderEventTypes

Type	Name	InitValue	Documentation
<<M>>	eFSWRR_ONGLOBALOBJECTACTIVATED	0x00000001	This bit can be used to temporarily deactivate the event onCollectorActivated. For this purpose, the eFSG_ONGLOBALOBJECTACTIVATED bit shall be set to zero.
<<M>>	eFSWRR_ONGLOBALOBJECTCONFIGURED	0x00000002	This bit can be used to temporarily deactivate the event onCollectorConfigured. For this purpose, the eFSG_ONGLOBALOBJECTCONFIGURED bit shall be set to zero.
<<M>>	eFSWRR_ONGLOBALOBJECTCREATED	0x00000004	This bit can be used to temporarily deactivate the event onCollectorCreated. For this purpose, the eFSG_ONGLOBALOBJECTCREATED bit shall be set to zero.

Table 13 (*continued*)

Type	Name	InitValue	Documentation
<<M>>	eFSWRR_ONGLOBALOBJECT LOCKED	0x00000020	This bit can be used to temporarily deactivate the event onCollectorLocked. For this purpose, the eFSG_ONGLOBALOBJECTLOCKED bit shall be set to zero.
<<M>>	eFSWRR_ONGLOBALOBJECT STARTED	0x00000008	This bit can be used to temporarily deactivate the event onCollectorStarted. For this purpose, the eFSG_ONGLOBALOBJECTSTARTED bit shall be set to zero.
<<M>>	eFSWRR_ONGLOBALOBJECT UNLOCKED	0x00000010	This bit can be used to temporarily deactivate the event onCollectorUnlocked. For this purpose, the eFSG_ONGLOBALOBJECTUNLOCKED bit shall be set to zero.
<<M>>	eFSWRR_ONWRITEREAD RECORDERERROR	0x00000080	This bit can be used to temporarily deactivate the event onWriteReadRecorderError. For this purpose, the eFSWRR_ONWRITEREADRECORDERERROR bit shall be set to zero.
<<M>>	eFSWRR_ONWRITEREAD RECORDERMODIFIED	0x00000200	This bit can be used to temporarily deactivate the event onWriteReadRecorderModified. For this purpose, the eFSWRR_ONWRITEREADRECORDERMODIFIED bit shall be set to zero.
<<M>>	eFSWRR_ONWRITEREAD RECORDERPAUSED	0x00000040	This bit can be used to temporarily deactivate the event onWriteReadRecorderPaused. For this purpose, the eFSWRR_ONWRITEREADRECORDERPAUSED bit shall be set to zero.
<<M>>	eFSWRR_ONWRITEREAD RECORDERSTOPPED ACQUISITION	0x00000100	This bit can be used to temporarily deactivate the event onWriteReadRecorderStoppedAcquisition. For this purpose, the eFSWRR_ONWRITEREADRECORDERSTOPPEDACQUISITION bit shall be set to zero.

Table 14 — Enumeration MCDFlashSessionDescEventTypes

Type	Name	InitValue	Documentation
<<D>>	eFSSD_ONFLASHSEGMENTS CLEARED	0X00000001	This bit can be used to temporarily deactivate the event onFlashSegmentsCleared. For this purpose, the eFSSD_ONFLASHSEGMENTSCLEARED bit shall be set to zero.
<<D>>	eFSSD_ONFLASHSEGMENTS LOADED	0X00000002	This bit can be used to temporarily deactivate the event onFlashSegmentsLoaded. For this purpose, the eFSSD_ONFLASHSEGMENTSLOADED bit shall be set to zero.
<<D>>	eFSSD_ONFLASHSESSIONDESC LOCKED	0X00000004	This bit can be used to temporarily deactivate the event onFlashSegmentsLocked. For this purpose, the eFSSD_ONFLASHSESSIONDESCLOCKED bit shall be set to zero.
<<D>>	eFSSD_ONFLASHSESSIONDESC UNLOCKED	0X00000008	This bit can be used to temporarily deactivate the event onFlashSegmentsUnlocked. For this purpose, the eFSSD_ONFLASHSESSIONDESCUNLOCKED bit shall be set to zero.

Table 15 — Enumeration MCDCollectorEventTypes

Type	Name	InitValue	Documentation
<<D>>	eFSC_ONCOLLECTORMONITORINGFRAMESREADY	0x00000400	This bit can be used to temporarily deactivate the event onCollectorMonitoringFramesReady. For this purpose, the eFSC_ONCOLLECTORMONITORINGFRAMESREADY bit shall be set to zero.
<<M,D>>	eFSC_ONCOLLECTORERROR	0x00000080	This bit can be used to temporarily deactivate the event onCollectorError. For this purpose, the eFSC_ONCOLLECTORERROR bit shall be set to zero.
<<M,D>>	eFSC_ONCOLLECTORRESULTREADY	0x00000200	This bit can be used to temporarily deactivate the event onCollectorResultReady. For this purpose, the eFSC_ONCOLLECTORRESULTREADY bit shall be set to zero.
<<M,D>>	eFSC_ONGLOBALOBJECTACTIVATED	0x00000001	This bit can be used to temporarily deactivate the event onCollectorActivated. For this purpose, the eFSG_ONGLOBALOBJECTACTIVATED bit shall be set to zero.
<<M,D>>	eFSC_ONGLOBALOBJECTCONFIGURED	0x00000002	This bit can be used to temporarily deactivate the event onCollectorConfigured. For this purpose, the eFSG_ONGLOBALOBJECTCONFIGURED bit shall be set to zero.
<<M,D>>	eFSC_ONGLOBALOBJECTCREATED	0x00000004	This bit can be used to temporarily deactivate the event onCollectorCreated. For this purpose, the eFSG_ONGLOBALOBJECTCREATED bit shall be set to zero.
<<M,D>>	eFSC_ONGLOBALOBJECTLOCKED	0x00000020	This bit can be used to temporarily deactivate the event onCollectorLocked. For this purpose, the eFSG_ONGLOBALOBJECTLOCKED bit shall be set to zero.
<<M,D>>	eFSC_ONGLOBALOBJECTSTARTED	0x00000008	This bit can be used to temporarily deactivate the event onCollectorStarted. For this purpose, the eFSG_ONGLOBALOBJECTSTARTED bit shall be set to zero.
<<M,D>>	eFSC_ONGLOBALOBJECTUNLOCKED	0x00000010	This bit can be used to temporarily deactivate the event onCollectorUnlocked. For this purpose, the eFSG_ONGLOBALOBJECTUNLOCKED bit shall be set to zero.
<<M>>	eFSC_ONCOLLECTORCOLLECTEDOBJECTSMODIFIED	0x00000040	This bit can be used to temporarily deactivate the event onCollectorCollectedObjectsModified. For this purpose, the eFSC_ONCOLLECTORCOLLECTEDOBJECTSMODIFIED bit shall be set to zero.

Table 16 — Enumeration MCDDiagComPrimitiveEventTypes

Type	Name	InitValue	Documentation
<<D>>	eFSDCP_ONPRIMITIVEBUFFER OVERFLOW	0x00000800	This bit can be used to temporarily deactivate the event onPrimitiveBufferOverflow. For this purpose, the eFSDCP_ONPRIMITIVEBUFFEROVERFLOW bit shall be set to zero.
<<D>>	eFSDCP_ONPRIMITIVEHAS INTERMEDIATERESULT	0x00000010	This bit can be used to temporarily deactivate the event onPrimitiveHasIntermediateResult. For this purpose, the eFSDCP_ONPRIMITIVEHASINTERMEDIATERESULT bit shall be set to zero.
<<D>>	eFSDCP_ONPRIMITIVEHAS RESULT	0x00000020	This bit can be used to temporarily deactivate the event onPrimitiveHasResult. For this purpose, the eFSDCP_ONPRIMITIVEHASRESULT bit shall be set to zero.
<<D>>	eFSDCP_ONPRIMITIVEIDLE	0x00000040	This bit can be used to temporarily deactivate the event onPrimitiveIdle. For this purpose, the eFSDCP_ONPRIMITIVEIDLE bit shall be set to zero.
<<D>>	eFSDCP_ONPRIMITIVEJOB INFO	0x00000080	This bit can be used to temporarily deactivate the event onPrimitiveJobInfo. For this purpose, the eFSDCP_ONPRIMITIVEJOBINFO bit shall be set to zero.
<<D>>	eFSDCP_ONPRIMITIVELOCKED	0x00000001	This bit can be used to temporarily deactivate the event onPrimitiveLocked. For this purpose, the eFSDCP_ONPRIMITIVELOCKED bit shall be set to zero.
<<D>>	eFSDCP_ONPRIMITIVENOT EXECUTABLE	0x00001000	This bit can be used to temporarily deactivate the event onPrimitiveNotExecutable. For this purpose, the eFSDCP_ONPRIMITIVENOTEXECUTABLE bit shall be set to zero.
<<D>>	eFSDCP_ONPRIMITIVE PENDING	0x00000100	This bit can be used to temporarily deactivate the event onPrimitivePending. For this purpose, the eFSDCP_ONPRIMITIVEPENDING bit shall be set to zero.
<<D>>	eFSDCP_ONPRIMITIVE PROGRESSINFO	0x00000200	This bit can be used to temporarily deactivate the event onPrimitiveProgressInfo. For this purpose, the eFSDCP_ONPRIMITIVEPROGRESSINFO bit shall be set to zero.
<<D>>	eFSDCP_ONPRIMITIVE REPEATING	0x00000400	This bit can be used to temporarily deactivate the event onPrimitiveRepeating. For this purpose, the eFSDCP_ONPRIMITIVEREPEATING bit shall be set to zero.
<<D>>	eFSDCP_ONPRIMITIVE UNLOCKED	0x00000002	This bit can be used to temporarily deactivate the event onPrimitiveUnlocked. For this purpose, the eFSDCP_ONPRIMITIVEUNLOCKED bit shall be set to zero.
<<D>>	eFSDCP_ONSTARTMESSAGE INDICATION	0x00000008	This bit can be used to temporarily deactivate the event onStartMessageIndication. For this purpose, the eFSDCP_ONSTARTMESSAGEINDICATION bit shall be set to zero.

Table 16 (*continued*)

Type	Name	InitValue	Documentation
<<D>>	eFSDCP_ONTRANSMIT INDICATION	0x00000004	This bit can be used to temporarily deactivate the event onTransmitIndication. For this purpose, the eFSDCP_ONTRANSMITINDICATION bit shall be set to zero.

Table 17 — Enumeration MCDWatcherEventTypes

Type	Name	InitValue	Documentation
<<M>>	eFSW_ONWATCHERACTIVATED	0x00000001	This bit can be used to temporarily deactivate the event onWatcherActivated. For this purpose, the eFSW_ONWATCHERACTIVATED bit shall be set to zero.
<<M>>	eFSW_ONWATCHERCREATED	0x00000040	This bit can be used to temporarily deactivate the event onWatcherCreated. For this purpose, the eFSW_ONWATCHERCREATED bit shall be set to zero.
<<M>>	eFSW_ONWATCHERDEACTIVATE D	0x00000002	This bit can be used to temporarily deactivate the event onWatcherDeactivated. For this purpose, the eFSW_ONWATCHERDEACTIVATED bit shall be set to zero.
<<M>>	eFSW_ONWATCHERERROR	0x00000020	This bit can be used to temporarily deactivate the event onWatcherError. For this purpose, the eFSW_ONWATCHERERROR bit shall be set to zero.
<<M>>	eFSW_ONWATCHERFIRED	0x00000004	This bit can be used to temporarily deactivate the event onWatcherFired. For this purpose, the eFSW_ONWATCHERFIRED bit shall be set to zero.
<<M>>	eFSW_ONWATCHERLOCKED	0x00000008	This bit can be used to temporarily deactivate the event onWatcherLocked. For this purpose, the eFSW_ONWATCHERLOCKED bit shall be set to zero.
<<M>>	eFSW_ONWATCHERUNLOCKED	0x00000010	This bit can be used to temporarily deactivate the event onWatcherUnlocked. For this purpose, the eFSW_ONWATCHERUNLOCKED bit shall be set to zero.

Table 18 — Enumeration MCDLogicalLinkEventTypes

Type	Name	InitValue	Documentation
<<D>>	eFSLL_ONLINK CONFIGURATIONRECORDS MODIFIED	0x00200000	This bit can be used to temporarily deactivate the event onLinkConfigurationRecordsModified. For this purpose, the eFSLL_ONLINKCONFIGURATIONRECORDSMODIFIE D bit shall be set to zero.
<<D>>	eFSLL_ONLINKACTIVITYIDLE	0x00000001	This bit can be used to temporarily deactivate the event onLinkActivityIdle. For this purpose, the eFSLL_ONLINKACTIVITYIDLE bit shall be set to zero.

Table 18 (*continued*)

Type	Name	InitValue	Documentation
<<D>>	eFSLL_ONLINKACTIVITY RUNNING	0x00000002	This bit can be used to temporarily deactivate the event onLinkActivityRunning. For this purpose, the eFSLL_ONLINKACTIVITYRUNNING bit shall be set to zero.
<<D>>	eFSLL_ONLINKACTIVITY SUSPENDED	0x00000004	This bit can be used to temporarily deactivate the event onLinkActivitySuspended. For this purpose, the eFSLL_ONLINKACTIVITYSUSPENDED bit shall be set to zero.
<<D>>	eFSLL_ONLINKDEFINABLE DYNIDLISTCHANGED	0x00020000	This bit can be used to temporarily deactivate the event onLinkDefinableDynIdListChanged. For this purpose, the eFSLL_ONLINKDEFINABLEDYNIDLISTCHANGED bit shall be set to zero.
<<D>>	eFSLL_ONLINKDIAG COMPRIMITIVESMODIFIED	0x00000080	This bit can be used to temporarily deactivate the event onLinkDiagComPrimitivesModified. For this purpose, the eFSLL_ONLINKDIAGCOMPRIMITIVESMODIFIED bit shall be set to zero.
<<D>>	eFSLL_ONLINKFLASHSESSION DESCSMODIFIED	0x00040000	This bit can be used to temporarily deactivate the event onLinkFlashSessionDescsModified. For this purpose, the eFSLL_ONLINKFLASHSESSIONDESCSMODIFIED bit shall be set to zero.
<<D>>	eFSLL_ONLINKVARIANT IDENTIFIED	0x00000400	This bit can be used to temporarily deactivate the event onLinkOneVariantIdentified. For this purpose, the eFSLL_ONLINKVARIANTIDENTIFIED bit shall be set to zero.
<<D>>	eFSLL_ONLINKVARIANT SELECTED	0x00000800	This bit can be used to temporarily deactivate the event onLinkOneVariantSelected. For this purpose, the eFSLL_ONLINKVARIANTSELECTED bit shall be set to zero.
<<D>>	eFSLL_ONLINKQUEUECLEARED	0x00002000	This bit can be used to temporarily deactivate the event onLinkQueueCleared. For this purpose, the eFSLL_ONLINKQUEUECLEARED bit shall be set to zero.
<<D>>	eFSLL_ONLINKVARIANTIDENTIFIED	0x00004000	This bit can be used to temporarily deactivate the event onLinkVariantIdentified. For this purpose, the eFSLL_ONLINKVARIANTIDENTIFIED bit shall be set to zero.
<<D>>	eFSLL_ONLINKVARIANTSELECTED	0x00008000	This bit can be used to temporarily deactivate the event onLinkVariantSelected. For this purpose, the eFSLL_ONLINKVARIANTSELECTED bit shall be set to zero.
<<D>>	eFSLL_ONLINKVARIANTSET	0x00010000	This bit can be used to temporarily deactivate the event onLinkVariantSet. For this purpose, the eFSLL_ONLINKVARIANTSET bit shall be set to zero.
<<M,C,D>>	eFSLL_GLOBALONOFF	0x80000000	This bit can be used to temporarily deactivate an event handler while maintaining its filter settings at the same time. For this purpose, the eFSSL_GLOBALONOFF bit shall be set to zero.
<<M,C,D>>	eFSLL_ONLINK COMMUNICATION	0x00000020	This bit can be used to temporarily deactivate the event onLinkCommunication. For this purpose, the eFSLL_ONLINKCOMMUNICATION bit shall be set to zero.

Table 18 (*continued*)

Type	Name	InitValue	Documentation
<<M,C,D>>	eFSLL_ONLINKCREATED	0x00000040	This bit can be used to temporarily deactivate the event onLinkCreated. For this purpose, the eFSLL_ONLINKCREATED bit shall be set to zero.
<<M,C,D>>	eFSLL_ONLINKERROR	0x00000100	This bit can be used to temporarily deactivate the event onLinkError. For this purpose, the eFSLL_ONLINKERROR bit shall be set to zero.
<<M,C,D>>	eFSLL_ONLINKLOCKED	0x00100000	This bit can be used to temporarily deactivate the event onLinkLocked. For this purpose, the eFSLL_ONLINKLOCKED bit shall be set to zero.
<<M,C,D>>	eFSLL_ONLINKOFFLINE	0x00000200	This bit can be used to temporarily deactivate the event onLinkOffline. For this purpose, the eFSLL_ONLINKOFFLINE bit shall be set to zero.
<<M,C,D>>	eFSLL_ONLINKONLINE	0x00001000	This bit can be used to temporarily deactivate the event onLinkOnline. For this purpose, the eFSLL_ONLINKONLINE bit shall be set to zero.
<<M,C,D>>	eFSLL_ONLINKUNLOCKED	0x00080000	This bit can be used to temporarily deactivate the event onLinkUnlocked. For this purpose, the eFSLL_ONLINKUNLOCKED bit shall be set to zero.
<<M>>	eFSLL_ONLINKCOLLECTORS MODIFIED	0x00000010	This bit can be used to temporarily deactivate the event onLinkCollectorsModified. For this purpose, the eFSLL_ONLINKCOLLECTORSMODIFIED bit shall be set to zero.
<<M,C>>	eFSLL_ONLINKTARGETOFF	0x00800000	This bit can be used to temporarily deactivate the event onLinkTargetOff. For this purpose, the eFSLL_ONLINKTARGETOFF bit shall be set to zero.
<<M,C>>	eFSLL_ONLINKTARGETON	0x00400000	This bit can be used to temporarily deactivate the event onLinkTargetOn. For this purpose, the eFSLL_ONLINKTARGETON bit shall be set to zero
<<M,C>>	eFSLL_ONLINKTARGET UNKNOWN	0x01000000	This bit can be used to temporarily deactivate the event onLinkTargetUnknown. For this purpose, the eFSLL_ONLINKTARGETUNK NOWN bit shall be set to zero.

In case an invalid eventHandlerId is supplied as parameter, an exception of type MCDParameterizationException with error code ePAR_INVALID_VALUE is thrown. If no event filter is set for an event handler, no event filtering will take place. The values are not influenced by system states.

It is defined, that after the creation of an Event Handler the default value 0xFFFFF... is taken, which means that all events will be transmitted.

7.11 Multi-Client capability

7.11.1 Requirements

Assumption:

- For the multi-client scenario considered here, no communication between clients is required.
- There is only a single project selected and active in a MCD-server at the same time.

The following Use Cases are considered:

- Clients need to have the ability to "list", "listen to", and "access" resources that have been created by other clients.

- Clients have the same system state.
- Clients use the “same” objects.
- In case of separate object trees for each client in the server, other clients need to have the possibility to obtain the states of other clients’ objects.
- There is nothing like a general master client. All clients connected have equal rights and can modify shared objects.
- Temporarily, one client can serve as a master client, that is, this client is granted exclusive access/modification access to certain areas of the server.
- Exclusive access to the server as a whole is granted to a single client. No other client can connect to the server.
- All clients connected to a server are informed whenever another client enters or leaves a server. In addition, it needs to be possible to find out whether there are already other clients connected to the server.
- Client is notified about actions taken by other clients and the corresponding server reactions (events sufficient).
- Notification of all clients about state changes in the server
- Server needs to be able to identify whether a client has left the server (by accident or regularly), that is, the server needs to be able to free unused resources
- The server is the master of resource coordination
- For some events, e.g. client specific error events, it is required that these error events are send to a specific client.
- There might be very simple clients that do not care about state changing events issued by other clients.

It has been proposed to split the use cases for multi-client into two groups. The first group contains all the use cases where clients can act as if no other clients are using the same server. In this group (and its solution), clients cannot list, listen to, or modify resources created and used by other clients. The second group considers all the use cases that reflect any kind of cooperation or reuse of resources.

The discussion showed that purely concurrent clients with no shared objects will never occur in a MCD-server as, e.g., LogicalLinks, DiagComPrimitives, and Collectors are always subject to being shared by different clients. These objects represent, e.g., physical resource, which are only present once, and this fact cannot be hidden from the clients by the MCD-server. That is, this scenario could only be realized by declaring that all shared resources from LogicalLinks downwards the containment hierarchy can only be instantiated once. As a result, these resources would be private to a single client and could only be accessed by this client. However, this would

- prevent the use case of listening to other clients’ resources and
- the effort of realizing this limited multi-client capability would not be significantly easier or less work.

Therefore, the purely concurrent multi-client realization is rejected and was no longer being discussed as a basic multi-client solution.

Regarding multi-client capabilities of a MCD-server the following interference scenarios with respect to logical links and physical links shall be considered:

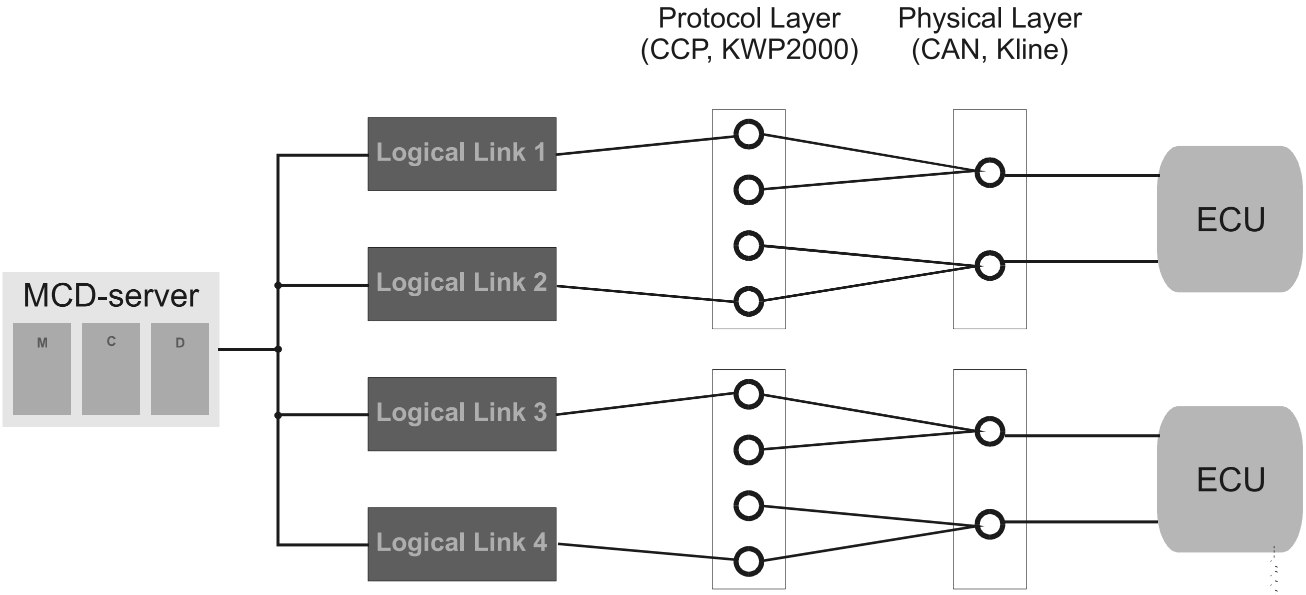


Figure 14 — Completely separated single system

Case 1: May be possible and needs to be secured by the server in a multi-client environment. However, security support is only possible if the ODX-data or A2L-files, respectively, contain the necessary information. In case of ODX, the ECU-object needs to inherit from all protocols it supports and there should only be a single ECU-object per physical ECU. Otherwise, the ODX-specification needs to be extended to contain information on the mapping of physical ECUs to the different logical ECU objects in the data.

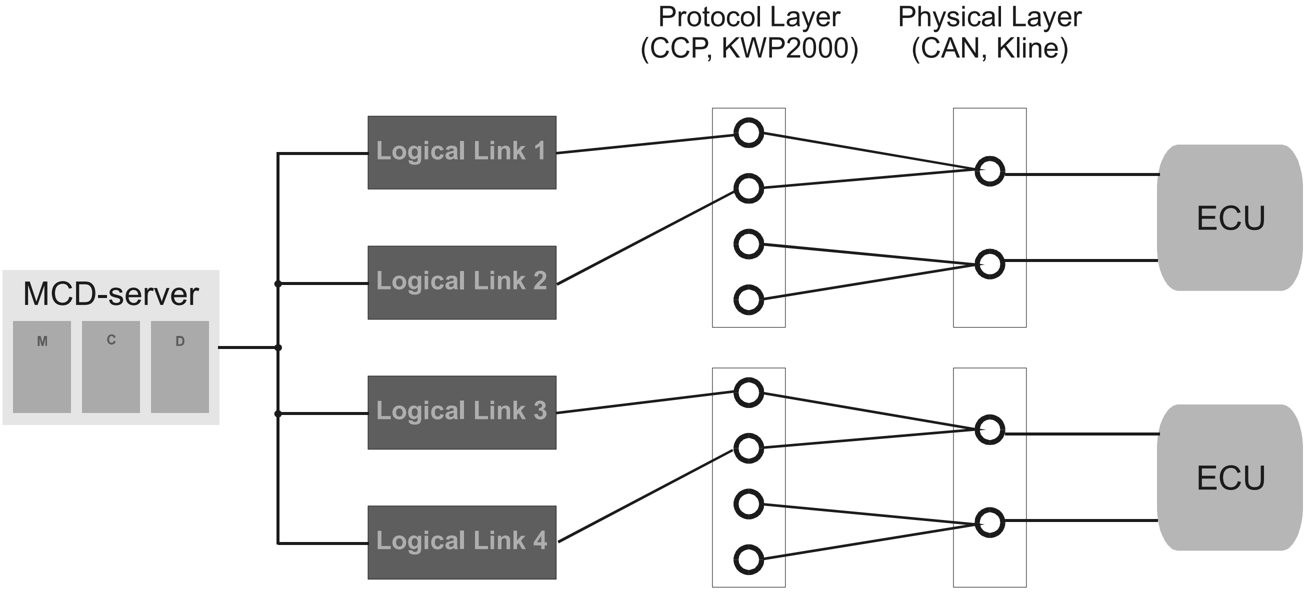


Figure 15 — Same physical layer single system

Case 2: May be possible and needs to be secured by the server in a multi-client environment. The same target ECU can be identified by considering the PhysicalLink in combination with the ECUAddress.

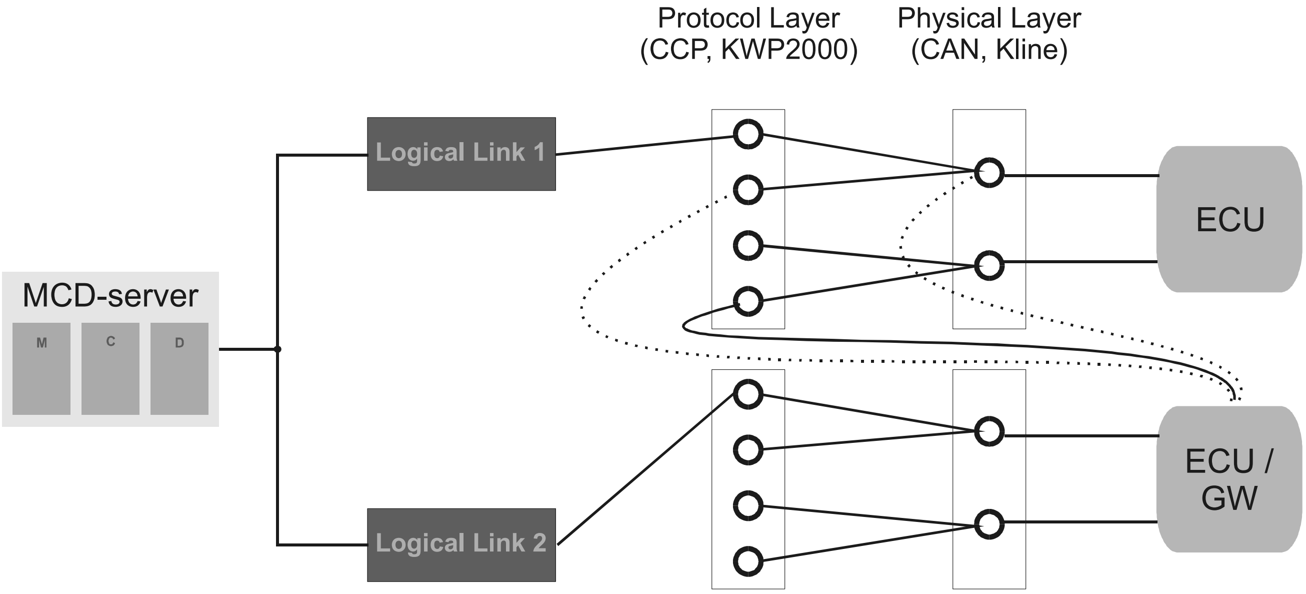


Figure 16 — Implicit dependency single system

Case 3: May be possible and needs to be secured by the server in a multi-client environment. The same target ECU can be identified by considering the containment of logical links in gateway-logical links.

7.11.2 Design

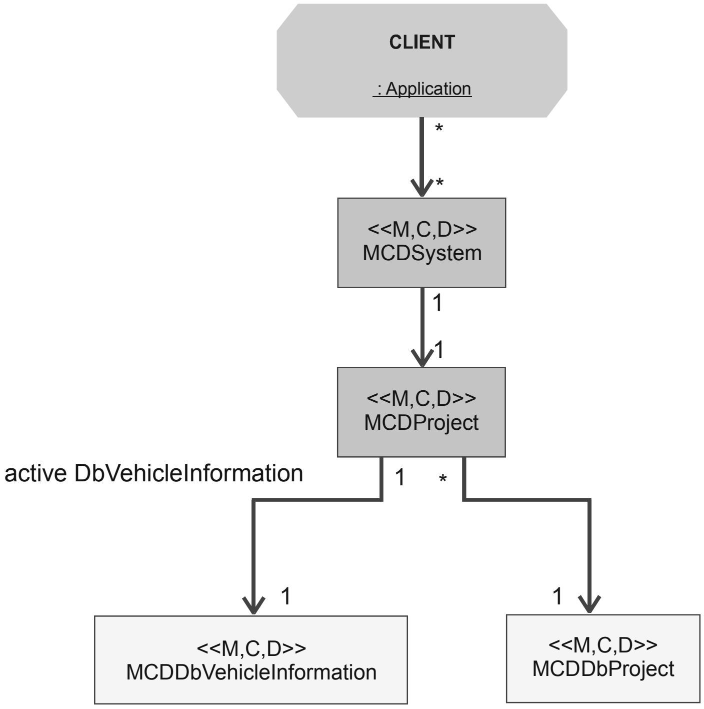


Figure 17 — UML notation for multiple clients

MCDSystem, MCDLogicaLink, MCDCollector (M), MCDWatcher (M), MCDRecorder (M), MCDFlashSessionDesc (D) and MCDDiagComPrimitive (D) are the interfaces that may be used simultaneously by several Clients and thus need rules for MultiClient behaviour.

For each of these interfaces Events for several EventHandlers may be generated in case the MCD-server supports this. The EventHandlers are identified via Id's that are generated by the MCD-server.

Within the function block MC, there is the possibility that several Clients access the same Collector. Here, the Client access to the result buffer of the Collectors is controlled by the Ids generated by the MCD-server. For the configuration that setting is valid, which had been used for the transition into the state `eOS_ACTIVATED`. It shall be noted that the buffer administration within the MC-server shall always be oriented at the “slowest” Client, that means that if for example in case of a buffer with 10 lines having 9 results in hand and Client 1 having taken 6 and Client 2 having taken 3 results only another 4 results may be taken until an overflow occurs.

For the MCDDiagComPrimitives within function block D it applies that for each Client by means of create a new DiagComPrimitive is created which can be parameterised separately. The results are sent to all Clients which registries the DiagComPrimitive Eventhandler.

7.11.3 Proxy in Multi Client Architecture

For each real object instance created by a client or by the server, there is only one such object instance in the server containing the data (server controls the real objects, client may have access via proxy objects only (transparent to the client)). The proxy is required to be able to identify which client has performed which actions.

Every new MCDSystem instance is considered a new client from a MCD-server perspective. The MCD-server needs to be able to uniquely identify each of these “client”, e.g. by creating a “proxy” object for each MCDSystem object in the server.

This means every Client has it own mirror for usage, which is internally mapped by MCD-server to the one and only real runtime object (e.g. `MCDSystem`, `MCDProject`, `MCDDLogicalLink, MCDMCLogicalLink`).

For each real object instance created by a client or by the server, there is only one such object instance in the server containing the data [server controls the real objects, client may have access via proxy objects only (transparent to the client)].

The proxy principle is to be used for all runtime objects. Each client has at maximum one proxy object per server object.

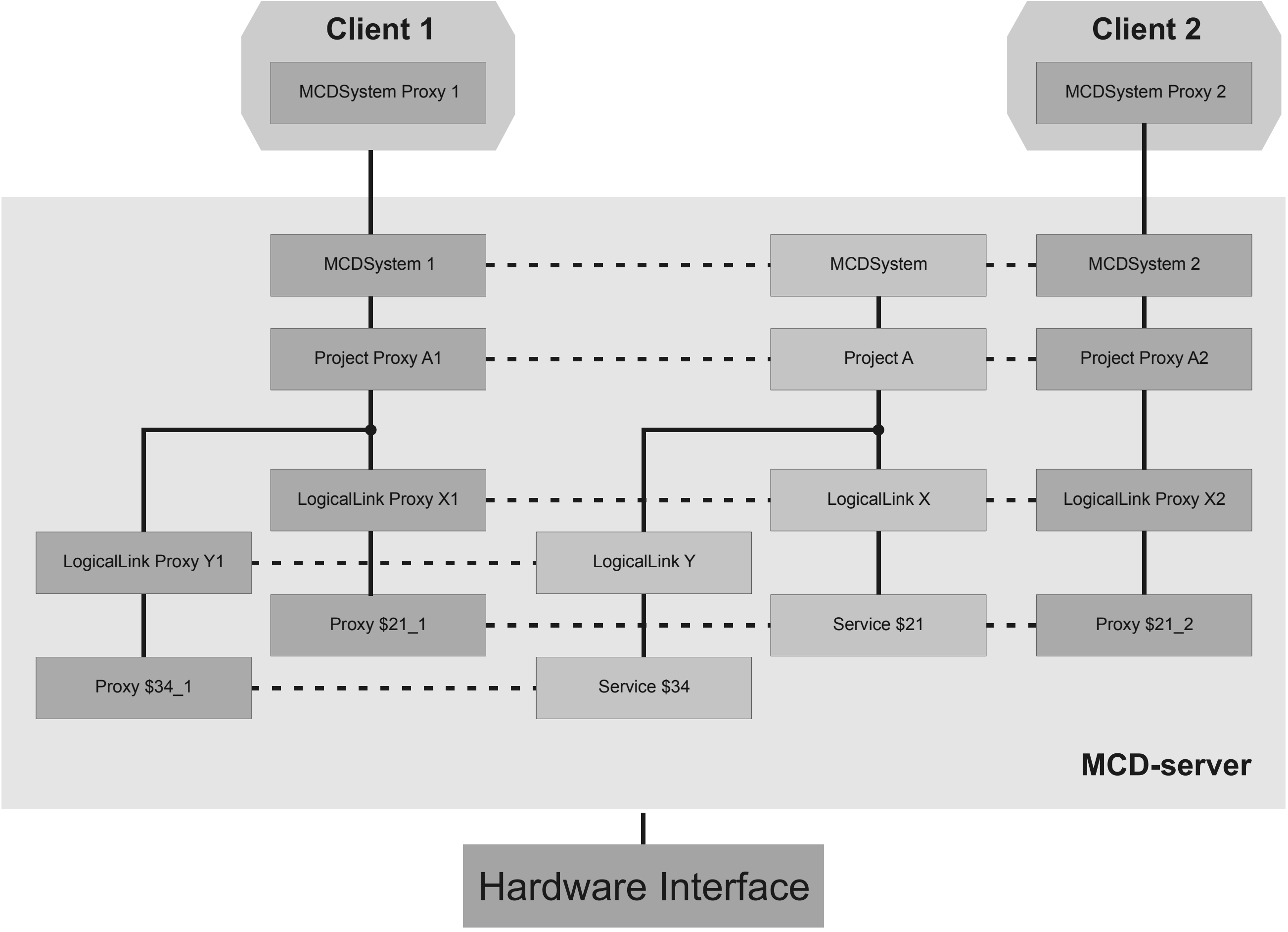


Figure 18 — Relation between Instance and Client specific proxy

Modifications to the server objects become valid directly, i.e., in the moment the new values are assigned (no delay by notification) - no matter which client performed the change through its proxy object.

All clients see the same server state (server state = union of states of all contained objects), that is, if a project is selected by one client a second client could access this project via `MCDSystem::getActiveProject()`. If one client adds an object to a (shared) collection, then this object can be obtained by any other client. However, there can only be one instance (same type, same instance, same name) of an object in the same collection.

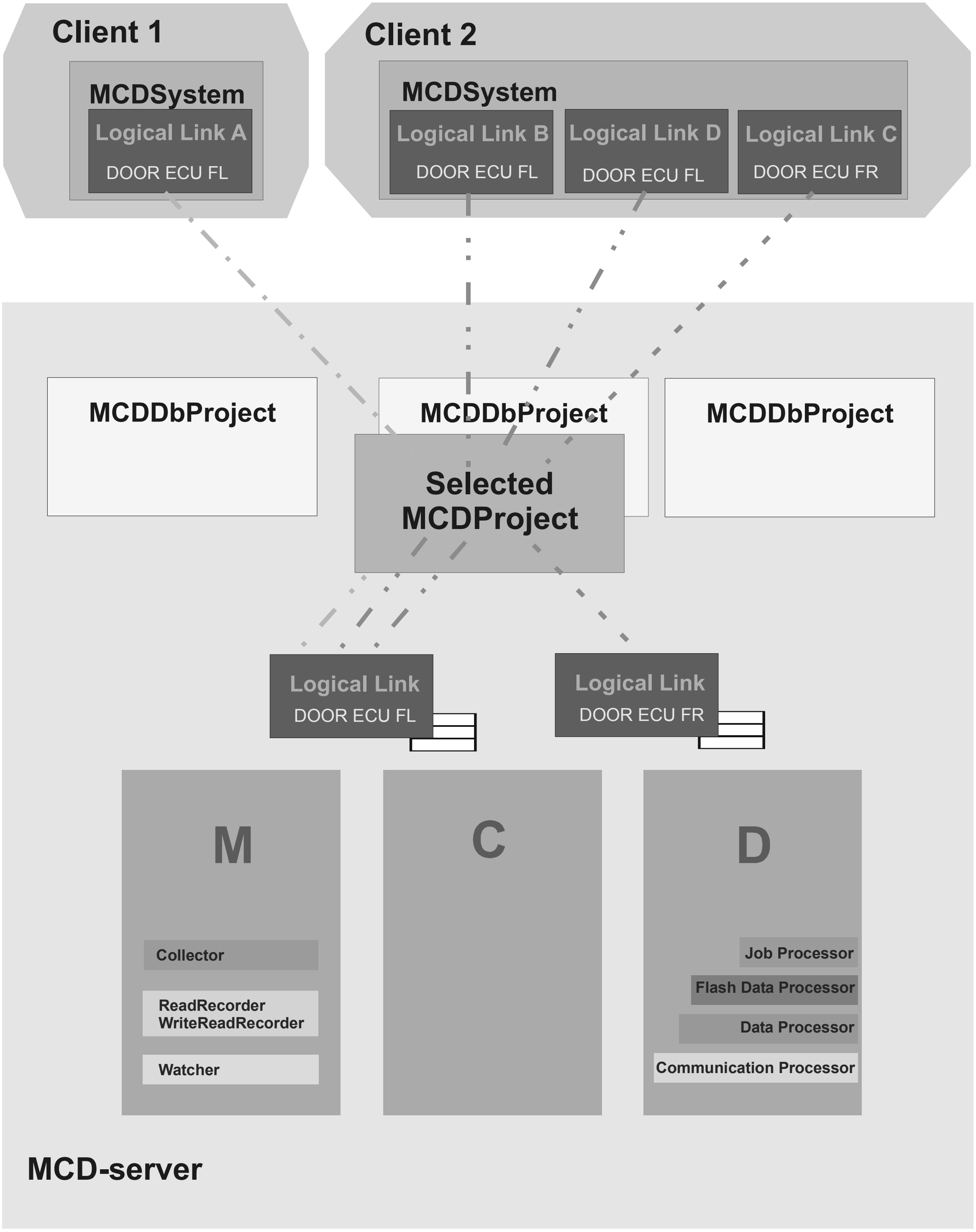


Figure 19 — Logical Link scheduling

Only one Project can be used at runtime by a server, independent from number of Clients.

Proxy Objects (runtime and DB objects) that become invalid because the server internal object has been destroyed (e.g. the current MCDProject is deselected by one client but others still have references to this project via their client-specific “proxy objects”) throw an exception of type eSYSTEM_OBJECT_DISCONNECTED when the client access these objects the next time.

7.11.4 Cooperation Level

To support a smooth cooperation of the clients in the multiclient use case, four different cooperation levels have been introduced. These describe and define which operations are possible for clients at shared resources. The basic idea behind this is that resource objects represent physical components in the server the state of which can neither be controlled nor maintained by the server. With respect to the following definitions shared resources start on the level of logical links and include all other resource objects further down the containment hierarchy.

- The level of cooperation is defined with the creation of a shared resource. The cooperation level cannot be change during the live cycle of a resource object.
- The cooperation level of a resource object further up in the containment hierarchy defines the cooperative level for all contained resource objects, except of those objects which a created in the hierarchy with an own cooperation level. In the latter case, the cooperation levels of the two shared objects are completely independent of each other.
- A client can use a resource that is owned by another client only up to the access level granted by the owning client. That is, if the owning client has created this shared object with the cooperation level `eEXTENDING_ACCESS`, another client can access this object in cooperation level `eREAD_ONLY` or `eEXTENDING_ACCESS`.
- The cooperation level “No Cooperation” creates a resource object for exclusive usage for a single client. It is not possible for other clients to access such a resource or any of the contained resources.
- The cooperation level “Read Only” creates a resource object for exclusive modification for a single client but grants read access (non-modifying access) for other clients.
- The cooperation level “Extending Modification” creates a resource object for nearly exclusive modification for a single client but grants limited modification rights to other clients. Other clients’ modifications are not to remove anything from the resource.
- The cooperation level “Full Access” creates a resource object for a certain client but grants nearly full access to this object by other clients.
- For all cooperation levels, the creating client is the master of the created resource object. Only the master is allowed to and responsible for destroying “his” resource objects. Destroying a resource object can result in “dangling” that is unusable resource objects in other clients.

For all cooperation levels it is defined for every resource class which methods can be called by other clients (non-master clients) in which cooperation level. All methods not allowed on a certain cooperation level, need to throw an exception when being called by a non-master client.

Whenever a potentially shared resource is created, the server internally denotes which client has created this resource and which access rights have been granted to other clients at creation time. If this resource is to be removed from the server, the corresponding remove method can only be called by a client with sufficient rights, that is, either by the creating client or by any client in case of full access cooperation mode.

In case a client with insufficient rights tries to remove a resource by calling a corresponding remove method, an exception of type `MCDProgramViolationException` with error code `eRT_INSUFFICIENT_RIGHTS` is thrown.

The method `isOwned()` returns true if the client calling this method has originally created the corresponding resource.

The following objects are considered encapsulated, that is, all members of these objects inherit the cooperation level from the parent:

- MCDDiagComPrimitive: All members of this object have the same cooperation level as the DiagComPrimitive. In particular, this holds for the contained MCDRequest and the associated MCDRequestParameters.

  NOTE Objects of type MCDValue are considered client-specific. Within the server MCDValue-objects are considered immutable after being set once.

- MCDCollector: The members MCDBuffer and MCDTimeDelay (StartDelay and StopDelay) inherit the cooperation level of the containing MCDCollector. The member MCDRateInfo inherits the cooperation level of the containing MCDBuffer.

- MCDWriteReadRecorder: All members of this object have the same cooperation level as the MCDWriteReadRecorder itself. The associated logical link is not considered a member and can have a different cooperation level. The members are in particular: the MCDWriteReadRecorderCollectors collection, all MCDWriteReadRecorderCollector objects, the MCDCollectedObjects collection, all MCDCollectedObject objects, the MCDRateInfo and MCDTimeDelay (StartDelay and StopDelay). The watchers set at the MCDWriteReadRecorder are not considered contained objects and therefore maintain their own cooperation level.

- MCDWatcher: There are no members in the MCDWatcher containment hierarchy, which need to inherit the cooperation level. The activation watcher has its own cooperation level and the used trigger is exclusively owned by the watcher and has no methods to change its properties. Any associated logical link inside a trigger also has its own cooperation level. The client could set any activation watcher and any associated logical link he has access to.

- MCDFlashSessionDesc: All members of this object have the same cooperation level as the FlashSessionDesc. In particular, this holds for the contained MCDFlashDataBlock and the associated MCDFlashSegments.

- MCDConfigurationRecord: All members of this object have the same cooperation level as the ConfigurationRecord. In particular, this holds for the contained MCDOptionItems.

- MCDCharacteristic: All members and derived classes of this class have the same cooperation level as the MCDCharacteristic. In particular, this is true for the members MCDMatrixCharacteristic, MCDVectorCharacteristic.

- MCDLogicalLink: All members in the LogicalLink inherit the cooperation level of the logicallink. These members are in particular: MCDCollectors, MCDCharacteristics, MCDDiagComPrimitives, MCDFlashSessionDescs, MCDInterfaceResource, MCDInterfaceResources, MCDConfigurationRecords, MCDRateInfos, MCDRateInfo.

- MCDMonitoringLink: The cooperation level of a MCDCollector objects contained in a MCDMonitoringLink is identical to the cooperation level of the monitoring link.

In order to set a member's value, the client needs to have a permissive cooperation level set at the member's parent (container). The detailed cooperation levels are shown in Annex H.

The method getCooperationLevel() returns the cooperation level of the respective shared object. In combination with the knowledge on the ownership, a client can figure out which kind of access is possible to a shared object. With this information, a clients implementation can avoid exceptions being thrown. The methods getCooperationLevel and isOwned can be called regardless of the cooperation level of the object on which they are defined.

7.11.5 Symbolic Names of Clients

The type of the clients' symbolic names will be A_UNICODE2STRING. If the symbolic name of a client is not explicitly set the pattern *#RtGen_Client<unique_number>* should be used. The server does not care when the name is set, how often the name is re-set, etc. That is, the server does not control this symbolic name. As a result, a client is neither forced to set a symbolic name nor restricted in how often or when the name is set.

NOTE Symbolic client names need not be unique.

By calling getClientNames(), a client can find out

a) the number of clients already connected (including itself) and

b) the current symbolic identifier names of all connected clients.

No unique client ID will be accessible via the MCD-server API. However, the suppliers of MCD-server are free to generate client IDs for internal usage.

Whenever a new MCDSystem object instance is requested by any client the collection of client names at the server internal MCDSystem object is extended. As a result, the number delivered by MCDSystem::getClientNames()::getCount() is increased by one for every new MCDSystem instance. Whenever a client destroys a MCDSystem instance, this instance is removed from the collection of client names and the number delivered by MCDSystem::getClientNames()::getCount() is decreased by one for every client “leaving” the server.

A server can suppress multi-client access by setting its property MaxNumberOfClients to one (1). This property of a MCD-server can be queried via MCDSystem::getMaxNoOfClients().

No Server object shall be delivered to the client if no client connection is allowed and the error eRT_MAXIMUM_NUMBER_OF_CLIENTS_REACHED will be thrown. How the error is transported is defined in the interface definitions.

7.11.6 Selection and de-selection of Project and VehicleInfo in a multi-client setting

Selection:

- The first client that calls selectProject() at its MCDSystem proxy object to select the project triggers the server to create the server internal MCDProject object and a corresponding proxy object for the client. Then the server initiates a state change from state eINITIALIZED to ePROJECT_SELECTED. All states are server states. Proxy objects do not carry a state on their own. All other clients can call either MCDSystem::getActiveProject or MCDSystem::selectProject with exactly the same MCDDbProjectDescription as the first client. Doing so, the corresponding proxy objects are created if not already present.

- If no VehicleInfo is selected, the first client calls selectDbVehicleInformation() or selectDbVehicleInformationByName() at its MCDProject proxy object. This results in the server creating the server-internal MCDDbVehicleInformation-object and a corresponding proxy object for the client. All other clients can obtain an proxy to the active VehicleInformation by either calling getActiveDbVehicleInformation() or above methods with parameters matching the active project. These calls are executed on their respective MCDProject proxy objects. There will be at maximum one proxy object per client.

De-selection:

- Every client can correctly leave the server by performing the following steps (in the order given):
  - remove all logical links, including its contained members, it is the master of,

  - deselect the VehicleInformation, which causes the corresponding proxy object to be disconnected from the server object and marked as “invalid”,
  - deselect the Project, which causes the corresponding proxy object to be disconnected from the server object and marked as “invalid”, and
  - destroy its MCDSystem proxy object.

- Whenever, the client successfully deselects a VehicleInfo or a Project, the reference counters of the corresponding server objects are decreased.
- All proxy objects can only be destroyed by the clients but they can be marked as “invalid” (see above).

General rules:

- For every proxy object being created, the server internally increases the “reference counter” at the corresponding server object.
- For every proxy object being invalidated by the client (remove or deselect-method called), the server decreases the “reference counter” at the corresponding server object. A deselectProject()-call is a feedback state transition in state `eINITIALIZED` and therefore no exception is thrown by the server.
- After a client has released a proxy object (remove- method, deselect-method, etc.), the corresponding client-specific proxy-object becomes invalid. All accesses to this object result in an exception of type `eRT_OBJECT_DISCONNECTED`.
- If the “reference counter” at a server object is zero, the server is free to destroy this server objects. In addition, all necessary state transitions are initiated by the server to keep the system state consistent.
- To prevent proxy objects for Projects and VehicleInformations from really being destroyed in the server because of the client having reset its reference to such an object, the MCDSystem proxy objects maintains references to its MCDProject-proxy and MCDDbVehicleInformation-proxy. These references are removed if and only if the client correctly deselects the VehicleInfo and the Project.

7.11.7 Notification

In general, if a state change occurs in the server, all clients are notified which have attached an event listener at the corresponding object.

Object references for server-controlled objects are still delivered in events. However, with the proxy principle defined for multi-client, the server is to deliver a reference to the client-specific proxy object with the event.

In case of a state change, a separate event is send for the target state reached. In this event, the source state is delivered as a parameter.

A flag at the class of a primary server-controlled mutable object signals that the object has been modified by another client. For this purpose, the method `MCD[Class]::isModifiedByOtherClient() : A_BOOLEAN` is used. This method returns true if the primary object itself or any of the associated secondary objects has been modified by another client. The return value of this method is specific to the client’s proxy object of the primary object. After a client has called the `isModifiedByOtherClient` method, the value of the flag is reset to false.

Primary server-controlled mutable objects:

- `MCDSystem` (signals that at least one of its own or of the selected project’s collections has been modified);
- `MCDLogicalLink` and derived classes (signals that at least one of its own collections has been modified);

- `MCDDiagComPrimitive` and derived classes (signals that at least one of its request parameters has been modified);

- `MCDCollector` (signals that at least one of its collected objects has been modified);

- `MCDFlashSessionDesc` [signals that the collection of FlashSegments has been modified (clear, load, add) or that a FlashSegment has been altered (setXxxx)];

- `MCDConfigurationRecord` [signals that the value of the `MCDConfigurationRecord` has been modified (setValue) or that the value of one of the contained `MCDOptionItem` objects has been altered (setValueXxxx)];

- `MCDWatcher` (signals that the MCDWatcher has been modified);

- `MCDWriteReadRecorder` (signals that at least one of its collected objects has been modified);

- `MCDMonitoringLink` (signals that message filters modified).

NOTE A modification to a contained collection that is server-controlled leads to a modification of the primary server-controlled mutable object. A state change of a primary server-controlled mutable object or one of its subobjects does not set the `isModifiedByOtherClient` flag to true. The current state can be retrieved by a corresponding method. The list of all methods which lead to a change in the modification flag is given in Annex M. The `isModifiedByOtherClient` flag is only set for those changes which can be identified by a client by means of getter method etc.

Events conforming to the pattern `on[CollectionName]Modified` are thrown whenever a collection of server-controlled objects is changed. Collections which report changes via events are

- MCDLogicalLinks,

- MCDDiagComPrimitives,

- MCDCollectors,

- MCDCharacteristics,

- MCDFlashSessionDescs,

- MCDRecorders,

- MCDWatchers,

- MCDCollectedObjects (MCDWriteReadRecorder or MCDCollector),

- MCDConfigurationRecords,

- MCDInterfaces,

- MCDMonitoringLinks.

No state change event is raised when an object is created.

7.11.8 Remove shared objects

Whenever a client with sufficient rights (owner or sufficient cooperation level) successfully removes an object from a collection, e.g. a Logical Link from the `MCDLogicalLinks` collection, the corresponding server object is deleted including its contained members. In addition, all proxy objects to this server object are marked invalid.

The removal of an object should always conform to the associated state model.

This means if a logical link is removed from the collection MCDLogicalLinks successfully, all elements of the contained collections MCDDiagComPrimitives, MCDFlashSessionDescs, MCDConfigurationRecord MCDCollectors, and MCDCharacteristics are also removed and all corresponding proxy objects are marked invalid.

If a client calls a removeXXX()method at a runtime collection and if this call is successful, an event at every object removed from the collection is raised. The event is sent to every event handler registered for this object and – in case of event forwarding – the event is also sent to every event handler at the higher level.

The events are also sent if the server needs to remove objects, e.g. because a client has died.

The event to be sent is called on...Modified (Collection) where the parameter supplied the collection the object that has been removed from.

Implementation hint: The client removing objects from a collection should first remove its event handler from these objects. Otherwise, all events raised because of the removal will also be sent to the removing client.

Implementation hint: If event handlers are always associated with the client-specific proxy object of the corresponding event source, the server does not need to track which client is connected to which proxy. The object supplied as parameter to the method on...Modified needs to be the client's proxy object. Otherwise, the client cannot identify which collection was modified!

7.11.9 Locking

The following classes, including derived classes, shall support locking:

- MCDSystem: To temporarily avoid that other clients access any resource in the server. Prevent large memory imprint on limited servers. Temporarily enforce single client mode.
- MCDWatcher: To temporarily avoid that other clients change the configuration of a watcher.
- MCDWriteReadRecorder: To temporarily avoid that other clients change the configuration of a recorder.
- MCDLogicalLink: To temporarily avoid that other clients access, use or modify any resources associated with the logical link.
- MCDCollector: To temporarily avoid that other clients change the configuration of a collector.
- MCDFlashSessionDesc: To temporarily avoid that other clients change the configuration of a FlashSessionDesc.
- MCDDiagComPrimitive: To temporarily avoid that other clients change the configuration or execution of a DiagComPrimitive.
- MCDCharacteristic: To temporarily avoid that other clients adjust the Characteristic.
- MCDConfigurationRecord: To temporarily avoid that other clients modify the configuration string represented by a MCDConfigurationRecord this object can be locked.

So that the locking client could see the lock state of the object. A call to lock should also be allowed in case an object is locked. The states, which could be retrieved, are as follows:

- eLS_LOCKED : The queried object is locked.
- eLS_UNLOCKED: The queried object is unlocked.
- eLS_INHERITED_LOCK: The queried object has inherited a lock from one of the parents or referencing objects.
- eLS_CHILD_LOCKED: The queried object is unlocked but at least one child or referenced object is locked.

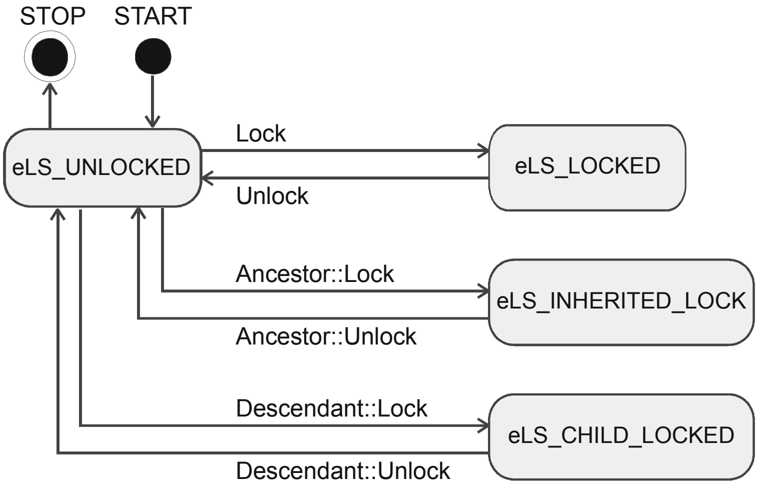


Figure 20 — State diagram Locking

The same client can call Unlock/Lock multiple times without throwing an exception. Calling Unlock/Lock by another client at a locked object will throw an exception.

A successful call to lock could be made by any client, but only if no object downwards in the object tree is locked by any client (including itself) and the locking client need to have full access to all objects accessible through the locking object. That means that the locking client is either the owner of the object or the cooperation level of this object is `eFULL_ACCESS`. In addition, the rule applies for all contained objects. In the case the lock call was not successful the exception `eRT_LOCK_FAILED` will be thrown.

It is not allowed to remove an object if any object which is accessible through the object, which should be removed, is locked by any client. It this case an error `eRT_OBJECT_IS_LOCKED`, `eRT_PARENT_OBJECT_IS_LOCKED` or `eRT_SUBOBJECT_IS_LOCKED` is thrown.

Since a client who is locked out is not able to call any method at the locked object, it might be interested in deregistering its event handler. The call of the method releaseEventHandler is therefore allowed.

Even if the client cannot access any method in a locked object, it will receive all events sent by this object.

If a client calls lock on an object, all objects, which are accessible through this object, change their internal state to locked.

An object which is locked, triggers a state change and sends a corresponding trigger event. For contained lockable objects, the change of the lock state of the parent object is registered but no trigger event is sent for the contained object.

Lock, unlock and unlockTree are allowed in any state of lockable objects and any state of child or referenced objects. Only the access rights and the lock state are checked. The methods are only allowed in cooperation level `eFULL_ACCESS`.

To guarantee that no request parameters are overridden by another client, that is, to avoid concurrent modification, `lock()` and `unlock()` methods are required at `MCDDiagComPrimitive`, `MCDBuffer`, `MCDCollector`, `MCDCollectedObjects`, and `MCDCharacteristic`. To implement secure client applications, explicit usage of these methods is required. If not used, the server does not guarantee security. If malprogrammed clients harm ECUs, no D-server supplier can be made responsible or liable.

The access to lock-methods is not restricted by any server or object state. That is, a lock-method can be called by any client at any time if

- the client has access to this object and
- the access to the lock-method is not restricted by a cooperation level and
- no other object above or below in the object hierarchy is locked by another client.

In case an object is locked by the removing client, removal is allowed: First remove all dependent inherited locks, then unlock the object, finally remove object and potentially also contained objects.

In case an object is locked by another client, removal is not allowed: The exception eRT_OBJECT_IS_LOCKED is thrown.

In case an object has an inherited lock by the removing client, removal is allowed: First remove all dependent inherited locks, then remove the inherited lock of the object, finally remove object and potentially also contained objects.

In case an object has an inherited lock by another client, removal is not allowed: The exception eRT_PARENT_OBJECT_IS_LOCKED is thrown.

In case all locked sub-objects are locked by the removing client, removal is allowed: First remove all dependent inherited locks, then unlock the sub-objects, finally remove object and also contained objects.

In case at least one locked sub-object is locked by another client, removal is not allowed: The exception eRT_SUBOBJECT_IS_LOCKED is thrown.

In case all sub-objects with inherited locks have the inherited lock by the removing client, removal is allowed: First remove all dependent inherited locks, finally remove object and also contained objects.

In case at least one sub-object has an inherited lock by another client, removal is not allowed: The exception eRT_SUBOBJECT_IS_LOCKED is thrown.

All calls to the locked object will throw an eRT_OBJECT_IS_LOCKED exception, if a client already has access to an object, which then enters a locked state, or an object above in the object hierarchy enters the locked state.

7.12 Client Controlled Objects

All values in a MCD-server, which have an ASAM data type, are considered client-controlled objects. Even more, these ASAM data types can be mapped to programming language specific data types by the different interface definition.

The following types of objects shall be considered client-controlled objects. That is, these objects are passed to the client and modification of these objects does not result in a direct modification of objects within the server.

- MCDAccessKeys
- MCDBaseVariantIdentificationResult
- MCDDatatypeAccessKey / MCDAccessKey
- MCDDatatypeAsciiString
- MCDDatatypeAsciiStrings
- MCDDatatypeDescription
- MCDDatatypeInfoStr
- MCDDatatypeLongname

- MCDDatatypeMcName
- MCDDatatypeShortname
- MCDDatatypeTime
- MCDDatatypeUnicode2String
- MCDDatatypeUnicode2Strings
- MCDError
- MCDErrors
- MCDException and derived classes
- MCDFlashSegmentIterator
- MCDResponse
- MCDResponseParameter
- MCDResponseParameters
- MCDResponses
- MCDResult
- MCDResults
- MCDValue
- MCDValueArray
- MCDValueCurve
- MCDValueMap
- MCDValueMatrix
- MCDValues
- all variables of one of the basic datatypes
- all classes derived from MCDBaseTriggerSource

In case of get-methods client-controlled objects which are return values are returned by copy. In case of set-methods or other methods having a client-controlled object as input, only the value(s) of this client-controlled object shall be considered in the D-server and not the objects themselves.

A Java-Job can be considered a client who is executed in the user-space of the server. It accesses methods at the D-server API just as any other client. As a consequence, the same rules for “client-controlled objects” need to apply as for any other client.

7.13 Resource Release

7.13.1 Use cases

- A client has died silently and has left occupied resources in the server, e.g. proxy and server objects.
- A client has left the server without correctly releasing all its resources in the server, e.g. proxy objects and server objects.
- A client is able to suspend its interaction with the server without losing its resources in the server.
- A client should be able to perform similar actions continuously in long time period without losing other resources it needs after leaving the loop. Example: A client continuously reads data from an ECU for two

hours. After this programmed time, it exits the loop and performs further actions on the same logical link. Here, the server should not terminate the logical link.

7.13.2 Requirements

- The solution should be as smart as possible, that is, it should not involve complicated mechanisms where not necessary. Furthermore, the solution should not overcomplicate the implementation of simple clients, e.g. simple diagnostic scripts.
- The server should be able to detect whether a client is still available (using its proxy objects, etc.) or not. If a client is detected as being unavailable, all its resources should be correctly removed by the server.
- It is not required to detect whether a client is going berserk, that is, that a client is stuck in an unintended endless loop.
- The server is not required to ask clients for permission before deleting resources in the server – independent of the fact whether proxy objects are marked invalid or whether server objects are removed.
- The timeout interval used by the server to detect unavailable clients should be server-specific (only one interval for all clients). Furthermore, the interval should be configurable before server runtime, i.e. in a configuration file. This makes the timeout interval a constant value at server runtime.
- A client should not be able to modify the server’s timeout interval at runtime.
- It should be able to switch off timeout-based detection of unavailable clients, e.g. by setting the timeout interval to infinite.

7.13.3 Solution

For non-event clients a timeout mechanism shall be used to detect client starvation. This means every call from a client resets the lease time for the client in the server. If during the lease time no call is made from the client all client proxies are removed and the corresponding server object references decreased. The lease time is defined by the server implementation.

If an object is released due to client starvation by the owning client and other clients still have proxies to this object or any child object the server marks all proxies of the starved client as invalid. If any of the proxies releases the last reference to a server object, the corresponding server object is released. In case the starved client was the owner of a server object, the server deletes the corresponding server object and all subobjects. Proxies from other clients to these server objects are marked invalid before. Project and VehicleInformation objects are never owned by any client. Therefore, the server deselects these objects instead of removing them. As a result, other client’s proxy objects are not invalidated. In case of Project and VehicleInformation, the server objects are deleted after the last client has deselected them.

7.14 Critical Section, Critical Groups of Methods

Methods in MCD are not atomic units of execution, that is, the control flow can switch back and forth between different threads executing different methods at any time. In particular, this also applies to the methods lock() and unlock(). As a result, the concurrent execution of different methods accessing the same resources can (and will) cause unpredictable execution states and crash the server. There will be no deterministic control flow any more.

The solutions to this problem are so-called critical sections:

- Define groups of methods for each shared resource where all methods within one group together define a critical section, that is, these methods are controlled by the same semaphore, monitor, etc.
- In particular, all methods that cause a state change at a certain resource should be contained in the same group.
- Each time a client calls a method from such a group, the corresponding call shall wait (e.g. in a queue to avoid starvation) until a client executing another method from the same group has left the critical section.

For the time being, every MCD-server vendor shall define its own critical sections. As a result, servers of different vendors may behave differently.

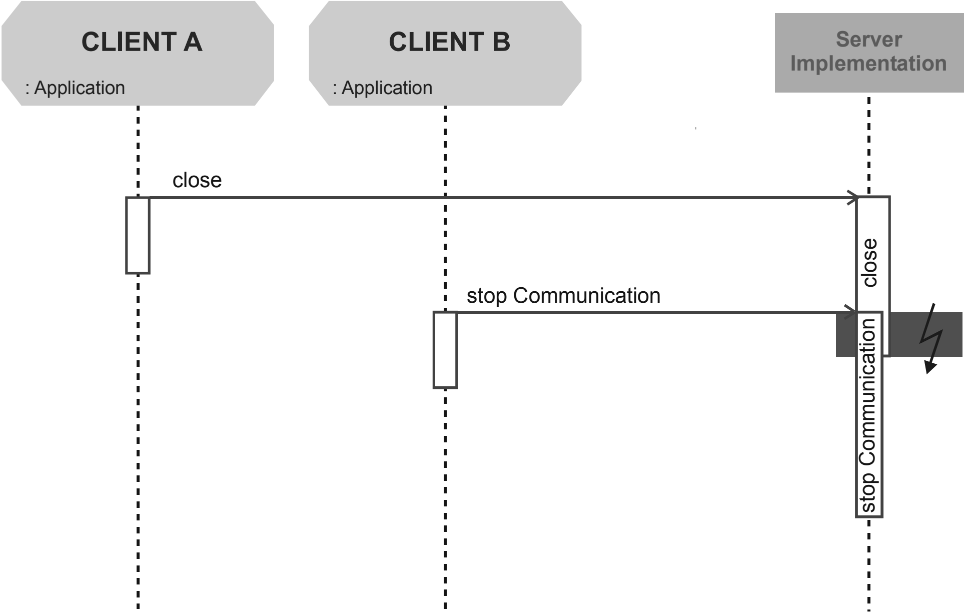


Figure 21 — Critical Section

7.15 Result access

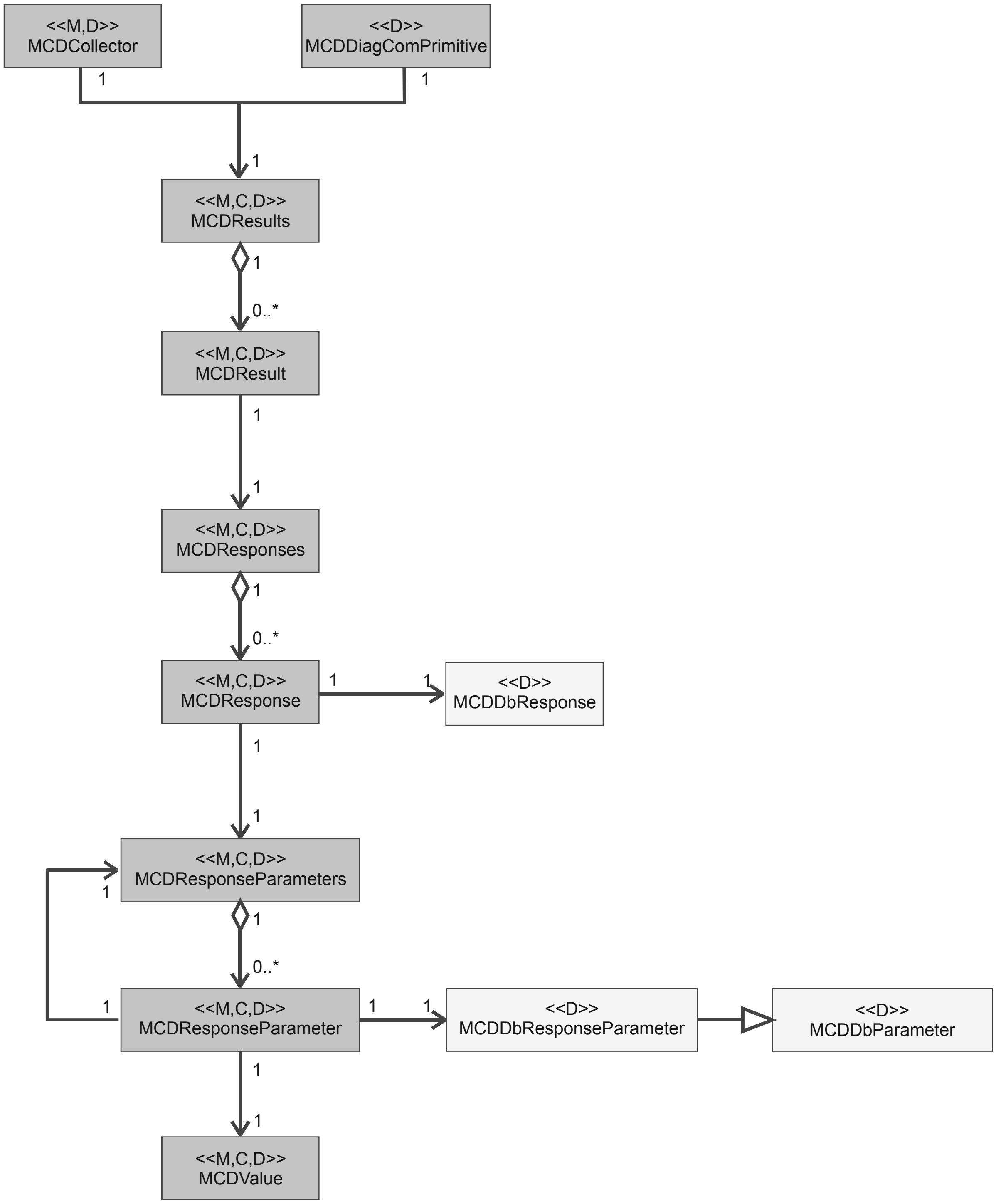


Figure 22 — ResponseParameters associations

Within this ERD the runtime side of the Result is considered in detail and the relation between Collector or 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 0..n MCDResults. Each `MCDResult` has for each ECU / Module used by the DiagComPrimitive / Collector one response. Only in D function block the response is described in the database with a `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 / Module. In D function block the structure is given by database template, in MC function block the structure is defined by Collector configuration.

Within the figures Result Structure in MD (Figure 37 — Collector result structure) and D function block (Figure 122 — Result structure DTC from example) this is illustrated in detail.

7.16 MCD value

7.16.1 Value types

A "Physical value" is a value, which is given in physical representation, e.g. "7 km/h". A physical value has an value ("7") and optionally an unit ("km/h"). In general, all values retrieved from or set at an D-server are physical values.

A "coded value" is a raw value internally used by the D-server. For example, the "7" is coded in a byte like "0x1A" where the bits 2 to 6 define the value.

The conversion between the coded and the physical value is defined by DOP in ODX. A full example of physical and coded values can be found in the ODX specification. Instead of coded values hexadecimal values are used in MC area.

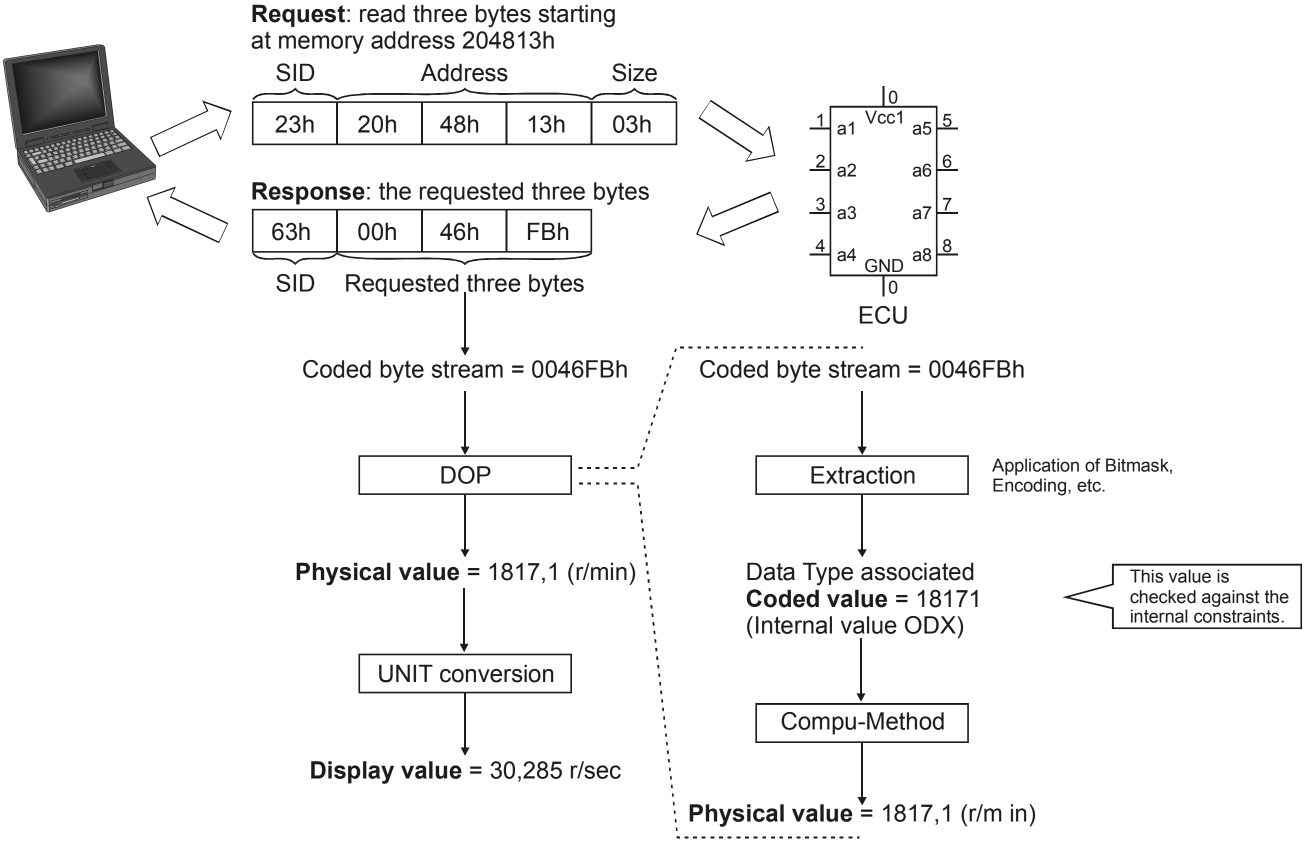

Request: read three bytes starting at memory address 204813h
SID
Address
Size
23h
20h
48h
13h
03h
Response: the requested three bytes
63h
00h
46h
FBh
SID
Requested three bytes
0
Vcc1
1
a1
2
a2
3
a3
4
a4
5
a5
6
a6
7
a7
8
a8
GND
0
ECU
Coded byte stream = 0046FBh
DOP
Physical value = 1817,1 (r/min)
UNIT conversion
Display value = 30,285 r/sec
Coded byte stream = 0046FBh
Extraction
Application of Bitmask, Encoding, etc.
Data Type associated
Coded value = 18171
(Internal value ODX)
This value is checked against the internal constraints.
Compu-Method
Physical value = 1817,1 (r/m in)

Figure 23 — Scale Constraints Internal physical

If the constraint matching fails the extraction/packaging process will stop. That means, if the value violates the internal constraints in case of a response the computational method will not be applied. In this state `MCDResponseParameter::getValue()` will throw an MCDProgramViolationException `eRT_ELEMENT_NOT_AVAILABLE` and `getCodedValue()` returns the bad value.

If the value violates the physical constraints in case of a request the computational method will not be applied. MCDRequestParameter::setValue() will throw an MCDParameterizationException ePAR_INVALID_VALUE and in this state MCDRequestParameter::getValue() and MCDRequestParameter::getCodedValue() will return a MCDProgramViolationException eRT_ELEMENT_NOT_AVAILABLE.

7.16.2 Method getValue

7.16.2.1 Behaviour of <D> MCDRequestParameter::getValue()

The method getValue() always delivers an object of type MCDValue.

- In case a value has been set at a MCDRequestParameter using the method MCDRequestParameter ::setValue before, a copy of the corresponding server-internal MCDValue object is delivered.
- In case MCDRequestParameter::setValue() has not been called before but a physical default value is defined in ODX, an object of type MCDValue initialised with this default value is returned.
- In case MCDRequestParameter::setValue() has not been called before and no physical default value is defined in ODX, the method getValue() returns a correctly typed MCDValue object with the internal status "uninitialized". The method MCDValue.isValid() returns true, if the MCDValue Object is initialized. That is, calling a getXXX-Method at this MCDValue (where XXX represents the type of this MCDValue) will result in an exception being thrown. The type of this exception is MCDProgramViolationException with error code eRT_VALUE_NOT_INITIALIZED.

7.16.2.2 Behaviour of <MCD> MCDResponseParameter::getValue()

- In case a MCDResponseParameter does not contain a valid value, MCDResonseParameter::getValue() throws an exception of type MCDProgramViolationException with error code eRT_ELEMENT_NOT_AVAILABLE.
- In case a MCDResponseParameter contains a valid value, a copy of the corresponding MCDValue object is returned.
- Java-Jobs only: In case a value has been set at a MCDResponseParameter using the method MCDResponseParameter::setValue() before, a copy of the corresponding server-internal MCDValue object is delivered.

7.16.2.3 Behaviour of <MCD> MCDParameter::getValue()

- In case a MCDParameter does not contain a valid value, MCDParameter::getValue() throws an exception of type MCDProgramViolationException with error code eRT_ELEMENT_NOT_AVAILABLE.
- In case a MCDResponseParameter contains a valid value, a copy of the corresponding MCDValue object is returned.

MCDValue objects returned to a client by the method getValue() are client-controlled objects. That is, the client is responsible for destroying the corresponding objects (user space objects the life cycle of which is bound to the client).

7.16.3 Method setValue

The method setValue(MCDValue) at objects of type MCDRequestParameter (and MCDResponseParameter in Java-Jobs) takes the MCDValue object supplied as parameter and overwrites the server-internal MCDValue object (if present) that is attached to the class/object the method setValue() has been called at. Overwriting is performed by copy to detach server-internal from client-controlled objects.

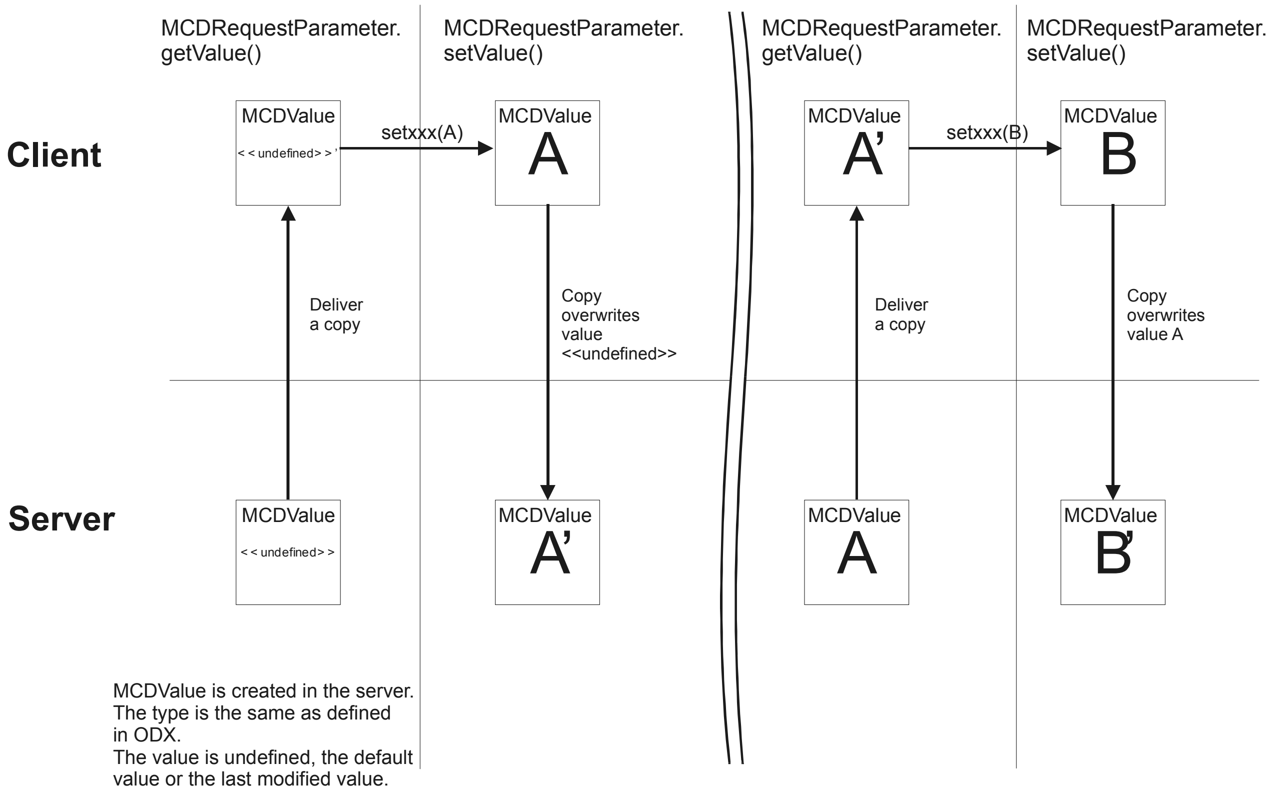
Requestparameter of type eSYSTEM cannot be overwritten by MCDRequest::setPDU(),MCDRequestParameter::setValue(), MCDRequestParameter::setValueUnchecked(). (Same as with CODED-CONST and PHYS-CONST.)

7.16.4 Method createValue

The method MCDObjectFactory::createValue() creates an empty MCDValue object in the server and returns a copy to the client. A MCDValue object created by this factory method is "uninitialized" and its data type is "eNO_TYPE". The real data type of such a MCDValue is defined with the first setXxxValue-method being called, that is, Xxx becomes the MCDValue's data type.

In contrast, the method <JD> MCDJobApi::createValue(MCDDatatype dataType) always delivers a correctly typed MCDValue object with respect to the method's parameter "dataType". This object does not contain a valid value. That is, calling a getXXX-Method at this MCDValue (where XXX represents the type of this MCDValue) will result in an exception being thrown. The type of this exception is MCDProgramViolationException with error code eRT_VALUE_NOT_INITIALIZED if the data type of the MCDValue object is equal to the value of the method's parameter "dataType". The type of this exception is MCDProgramViolationException with error code eRT_VALUE_OF_DIFFERENT_TYPE if the data type of the MCDValue object is not equal to the value of the method's parameter "dataType".

MCDValue objects returned to a client or Java-Job are client-controlled objects. That is, the client/Java-Job is responsible for destroying the corresponding objects (user space objects the life cycle of which is bound to the client/Java-Job).



MCDValue is created in the server.
The type is the same as defined
in ODX.
The value is undefined, the default
value or the last modified value.

Remark: setting/adjusting
the value is possible
via MCDValue methods
e.g. setUint32()

Figure 24 — Behaviour of get and set value methods for MCDRequestParameter

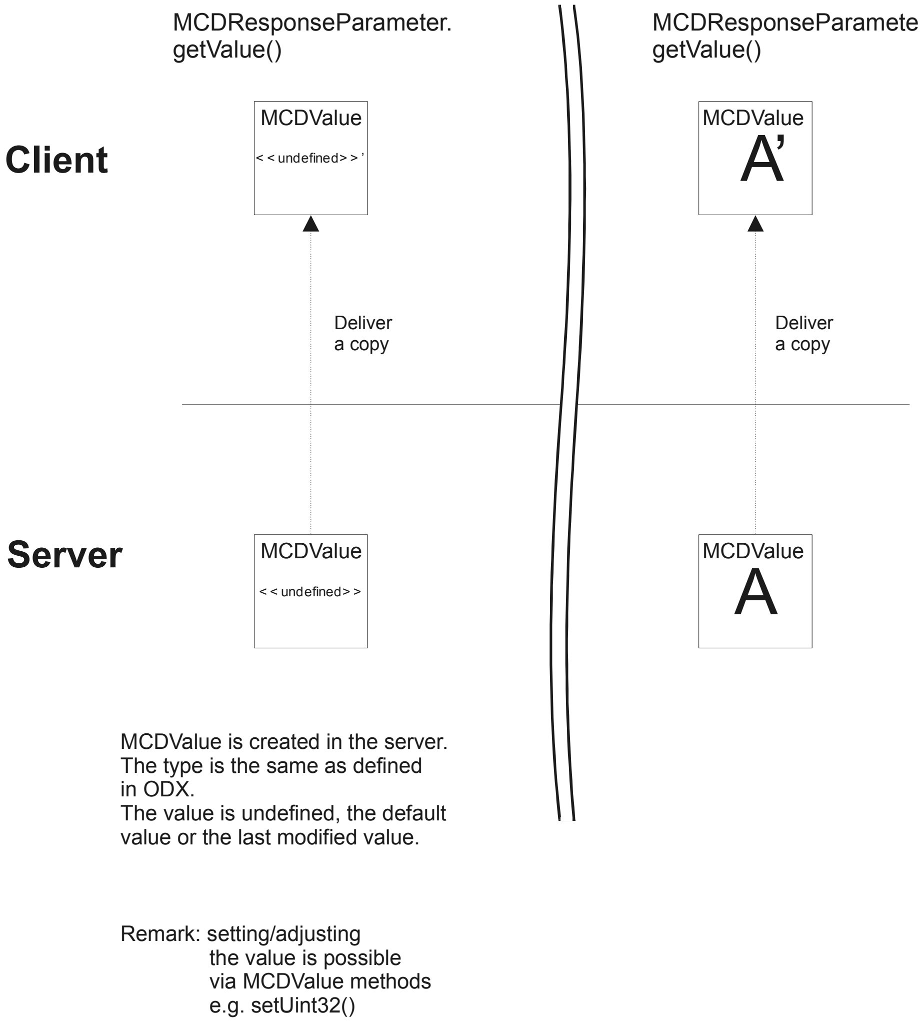
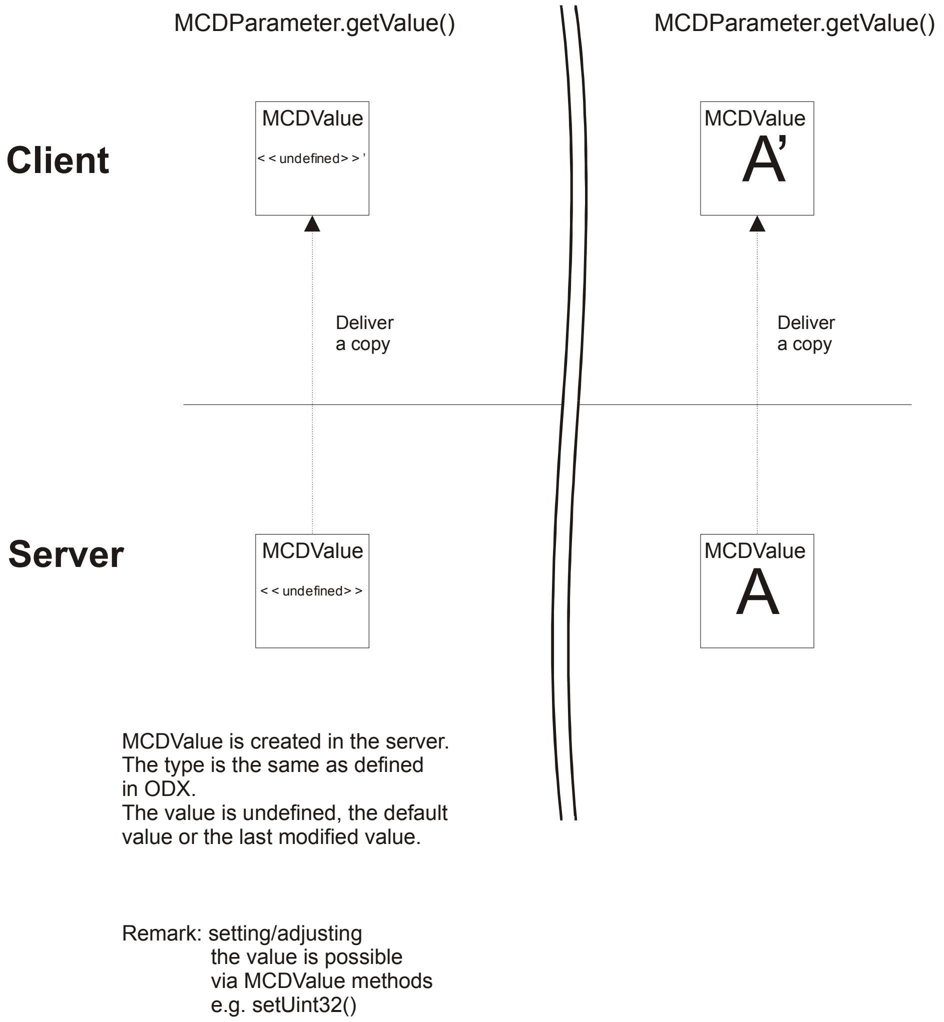


Figure 25 — Behaviour of get and set value methods for MCDResponseParameter

The method `MCDValue::isValid` indicates that the `MCDValue` object contains a valid type and is initialized with a valid value.

`MCDValue` objects will have an internal status “uninitialized” in case this object does not have a valid value.



MCDValue is created in the server.
The type is the same as defined
in ODX.
The value is undefined, the default
value or the last modified value.

Remark: setting/adjusting
the value is possible
via MCDValue methods
e.g. setUint32()

Figure 26 — Behaviour of get and set value methods for MCDParameter

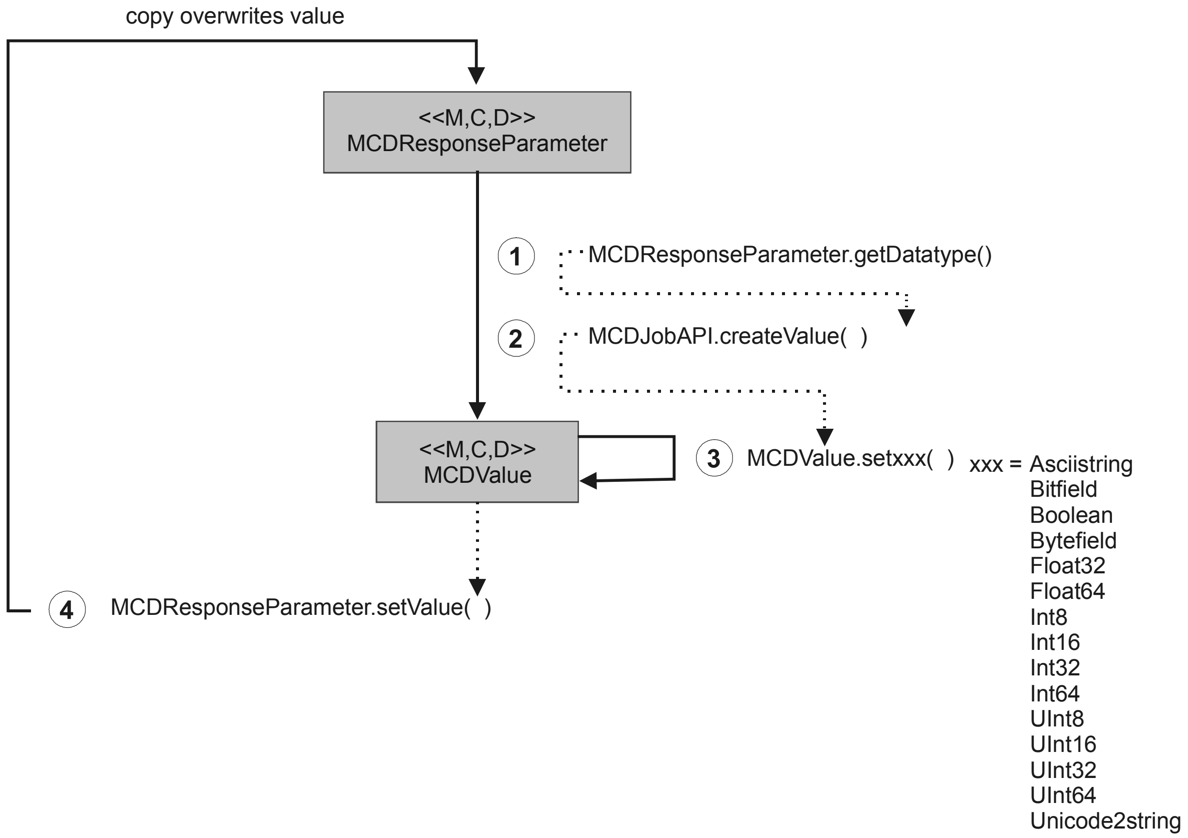


Figure 27 — MCDValue in jobs

7.17 Use cases

7.17.1 View

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

It is acted from the view of a single Client and it is assumed that each action is executed successfully.

7.17.2 Instantiation of projects

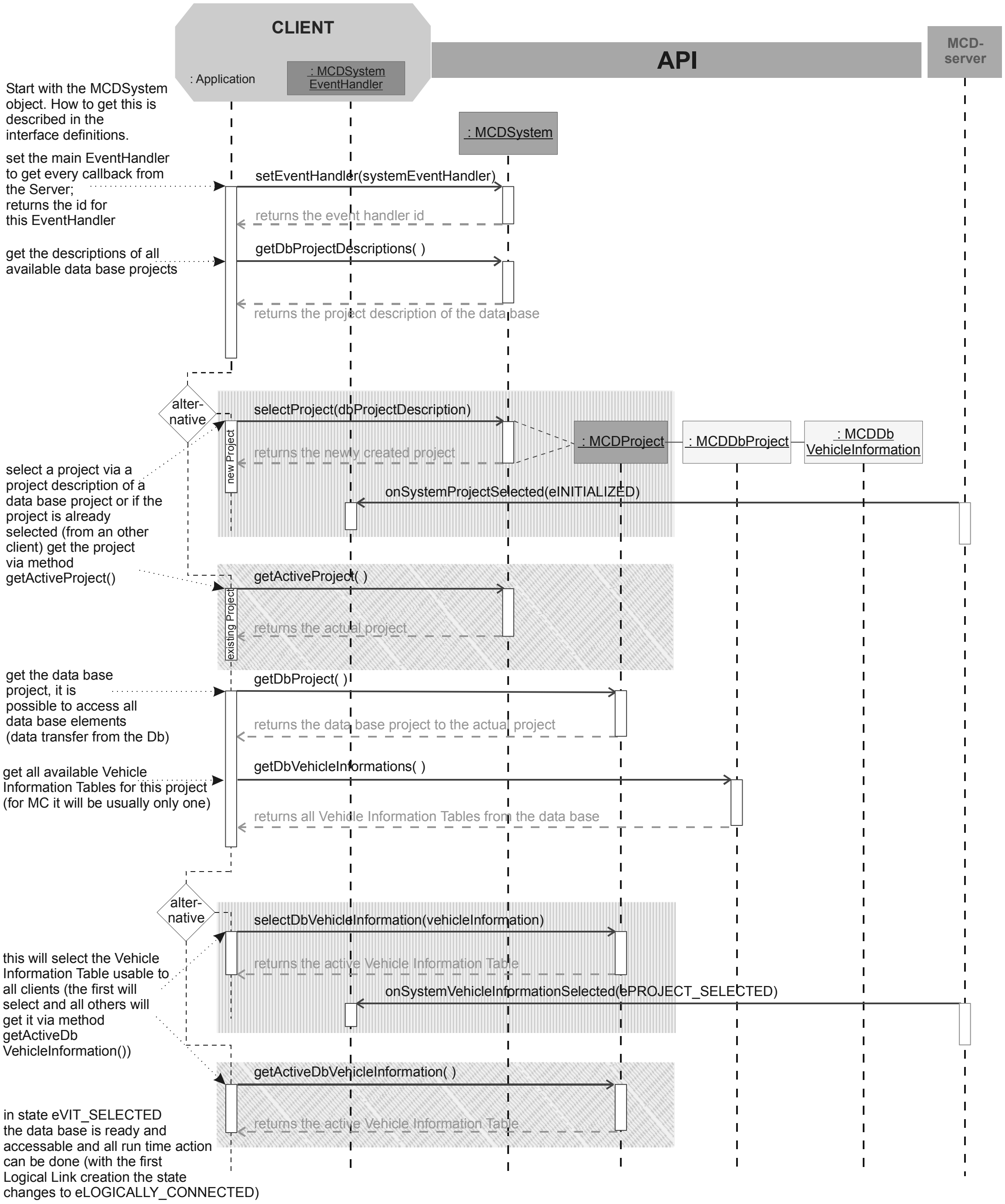

CLIENT
API
MCD-server
: Application
: MCDSystem EventHandler
: MCDSystem
Start with the MCDSystem object. How to get this is described in the interface definitions.
set the main EventHandler to get every callback from the Server; returns the id for this EventHandler
setEventHandler(systemEventHandler)
returns the event handler id
get the descriptions of all available data base projects
getDbProjectDescriptions( )
returns the project description of the data base
alter-native
selectProject(dbProjectDescription)
: MCDProject
: MCDDbProject
: MCDDb VehicleInformation
new Project
returns the newly created project
onSystemProjectSelected(eINITIALIZED)
select a project via a project description of a data base project or if the project is already selected (from an other client) get the project via method getActiveProject()
getActiveProject( )
existing Project
returns the actual project
get the data base project, it is possible to access all data base elements (data transfer from the Db)
getDbProject( )
returns the data base project to the actual project
get all available Vehicle Information Tables for this project (for MC it will be usually only one)
getDbVehicleInformations( )
returns all Vehicle Information Tables from the data base
alter-native
selectDbVehicleInformation(vehicleInformation)
this will select the Vehicle Information Table usable to all clients (the first will select and all others will get it via method getActiveDb VehicleInformation())
returns the active Vehicle Information Table
onSystemVehicleInformationSelected(ePROJECT_SELECTED)
getActiveDbVehicleInformation( )
returns the active Vehicle Information Table
in state eVIT_SELECTED the data base is ready and accessable and all run time action can be done (with the first Logical Link creation the state changes to eLOGICALLY_CONNECTED)

Figure 28 — Starting work with MCD-server

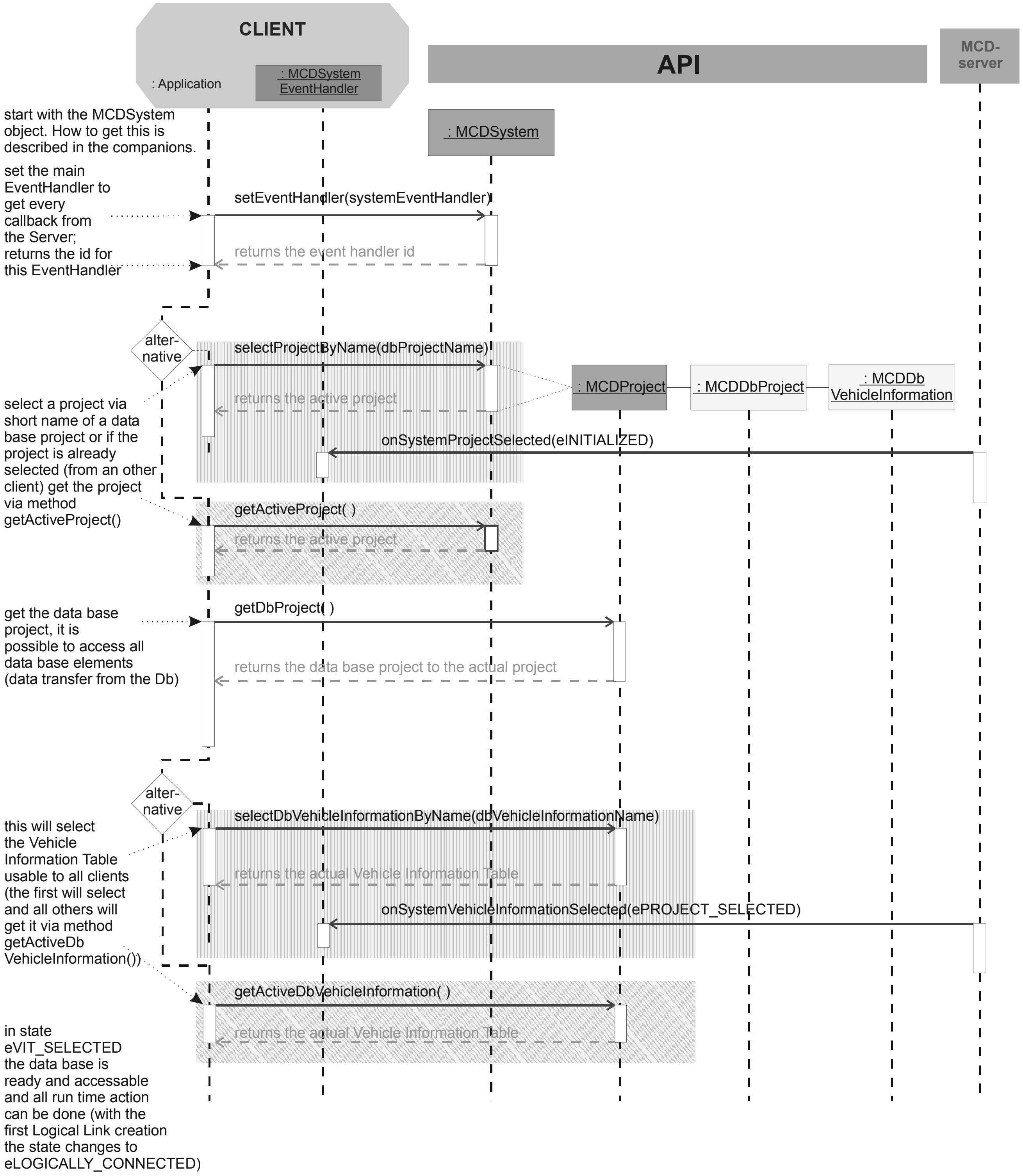
As an entry into the API the Client gets the MCDSystem object from the MCD-server. How this is done for the separate mappings of the object model for different programming languages and platforms, Java RMI, COM/DCOM, C++) shall be described within the respective documents mapping the MCD object model.

The object MCDSystem is handed over within the state eINITIALIZED. First register the main EventHandler to get all events delivered by the MCD-server. Now the Client may poll the collection of all projects descriptions available within the database. In the next step the Client will select one of these projects and thus provide for access to the database contents from the ASAM MCD 2 database (MC and D) with Vehicle Information Table, Logical Link Table and environmental variables for this project. After this, the MCDSystem object is within the state ePROJECT_SELECTED. If another Client has already selected a project this Client can ask for the active project, because only one project can be active at one time.

Now, out of the project the desired Vehicle Information Table is selected from the collection of available Vehicle Information Tables on the depending database project. Within this table all MCDDbDLogicalLink objects are located, which are necessary for the connection to the ECUs.

In the area of MC only one VehicleInformationTable exists, which is named DEFAULT.

Alternatively to the procedure mentioned above, the Client might skip the polling for information from the database, if the short names of all necessary objects are already known. This possibility is shown in the following diagram.


CLIENT
: Application
: MCDSystem EventHandler
API
MCD-server
: MCDSystem
setEventHandler(systemEventHandler)
returns the event handler id
alter-native
selectProjectByName(dbProjectName)
returns the active project
: MCDProject
: MCDDbProject
: MCDDb VehicleInformation
onSystemProjectSelected(eINITIALIZED)
getActiveProject( )
returns the active project
getDbProject( )
returns the data base project to the actual project
alter-native
selectDbVehicleInformationByName(dbVehicleInformationName)
returns the actual Vehicle Information Table
onSystemVehicleInformationSelected(ePROJECT_SELECTED)
getActiveDbVehicleInformation( )
returns the actual Vehicle Information Table

start with the MCDSystem object. How to get this is described in the companions.

set the main EventHandler to get every callback from the Server; returns the id for this EventHandler

select a project via short name of a data base project or if the project is already selected (from an other client) get the project via method getActiveProject()

get the data base project, it is possible to access all data base elements (data transfer from the Db)

this will select the Vehicle Information Table usable to all clients (the first will select and all others will get it via method getActiveDb VehicleInformation())

in state eVIT_SELECTED the data base is ready and accessable and all run time action can be done (with the first Logical Link creation the state changes to eLOGICALLY_CONNECTED)

Figure 29 — Starting work with MCD-server (via ShortName)

7.17.3 Database access

For the `MCDDbMCLogicalLink, MCDDbDLogicalLink` selected out of the Logical Link collection, the information stored in the Logical Link Table can be accessed: the ShortName, the Physical Vehicle Link or Physical Interface (and its type) and from the corresponding DbLocation the AccessKey.

Depending on the type of the DbLocation (`eMODULE` for MC and all other for D) there are several collections filled with database elements like Services and DiagComPrimitives for the D function block and Characteristics, Measurements and so on for MC function block. For the D function block the information about the Gateway property from the Logical Link Table can be accessed too.

This sequence diagram shows only the access to the top most elements of the database, which are the elements below shown in the corresponding ERD diagrams (see Figure 48 — Location association for D).

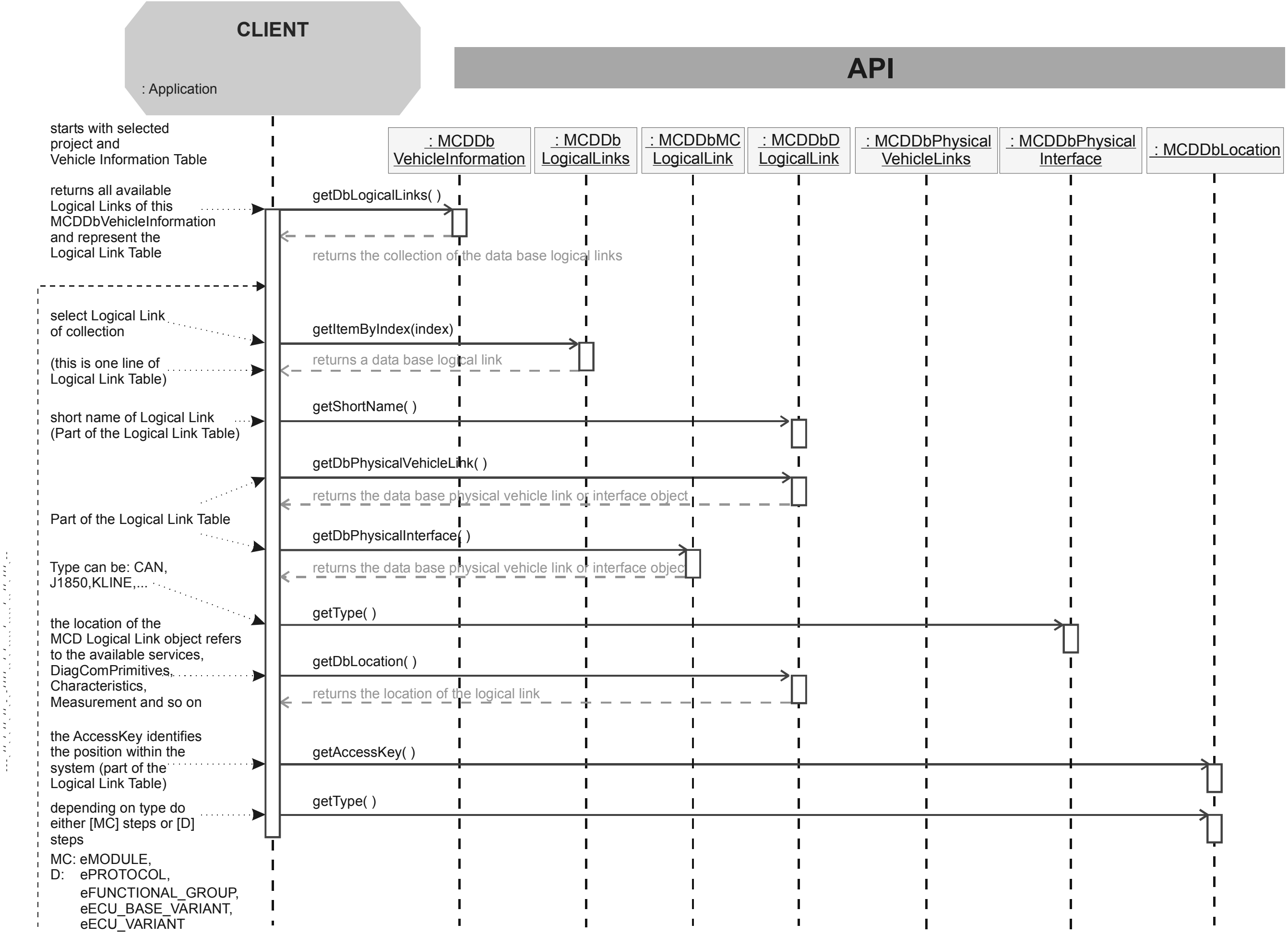

CLIENT
: Application
API
: MCDDb VehicleInformation
: MCDDb LogicalLinks
: MCDDbMC LogicalLink
: MCDDbD LogicalLink
: MCDDbPhysical VehicleLinks
: MCDDbPhysical Interface
: MCDDbLocation
starts with selected project and Vehicle Information Table
returns all available Logical Links of this MCDDbVehicleInformation and represent the Logical Link Table
getDbLogicalLinks( )
returns the collection of the data base logical links
select Logical Link of collection
getItemByIndex(index)
(this is one line of Logical Link Table)
returns a data base logical link
short name of Logical Link (Part of the Logical Link Table)
getShortName( )
getDbPhysicalVehicleLink( )
returns the data base physical vehicle link or interface object
Part of the Logical Link Table
getDbPhysicalInterface( )
returns the data base physical vehicle link or interface objec
Type can be: CAN, J1850,KLINE,...
getType( )
the location of the MCD Logical Link object refers to the available services, DiagComPrimitives, Characteristics, Measurement and so on
getDbLocation( )
returns the location of the logical link
the AccessKey identifies the position within the system (part of the Logical Link Table)
getAccessKey( )
depending on type do either [MC] steps or [D] steps
getType( )
MC: eMODULE,
D: ePROTOCOL,
eFUNCTIONAL_GROUP,
eECU_BASE_VARIANT,
eECU_VARIANT

Figure 30 — Database access (Part 1)

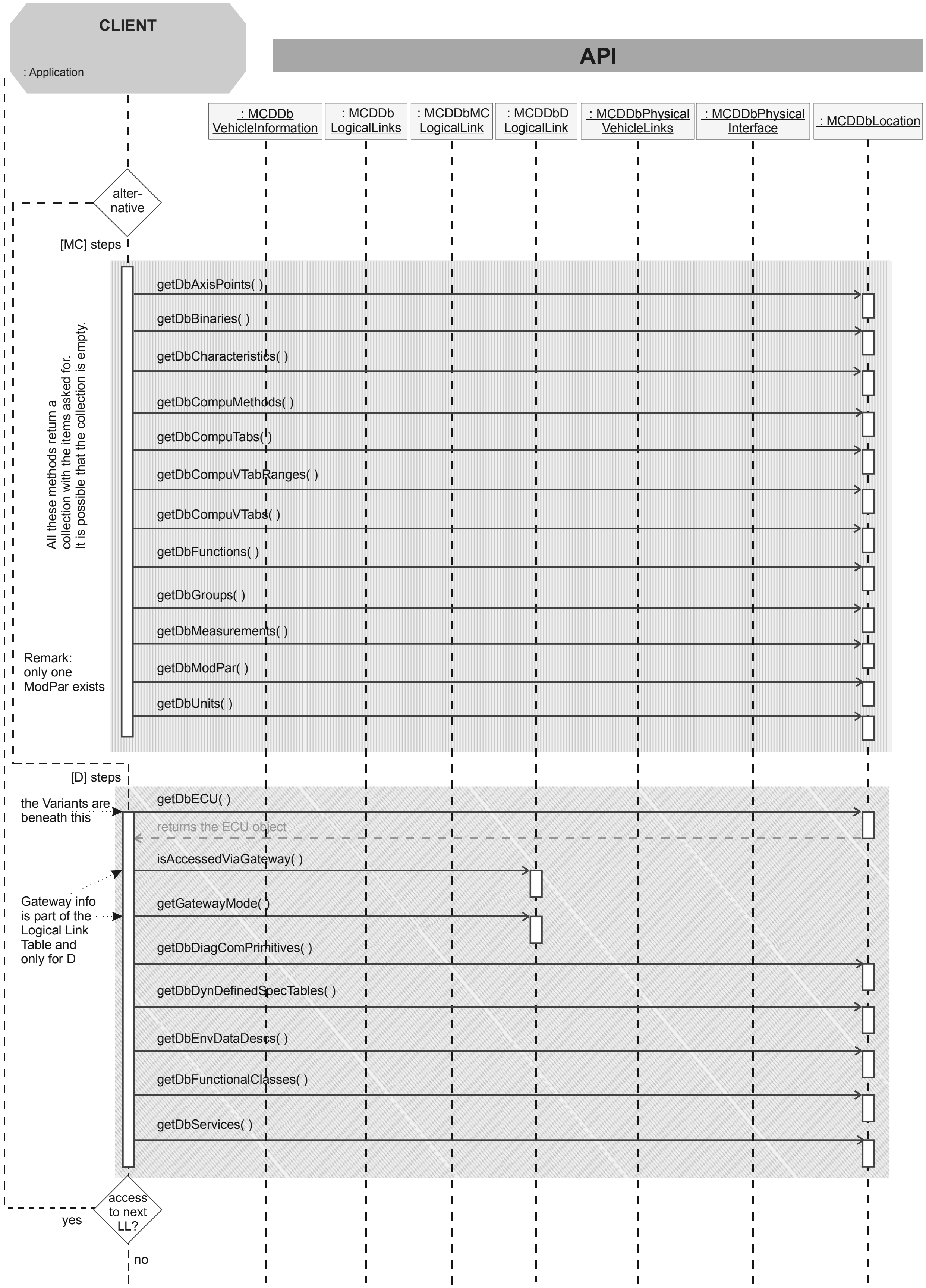


Figure 31 — Database access (Part 2)

7.17.4 Destruction

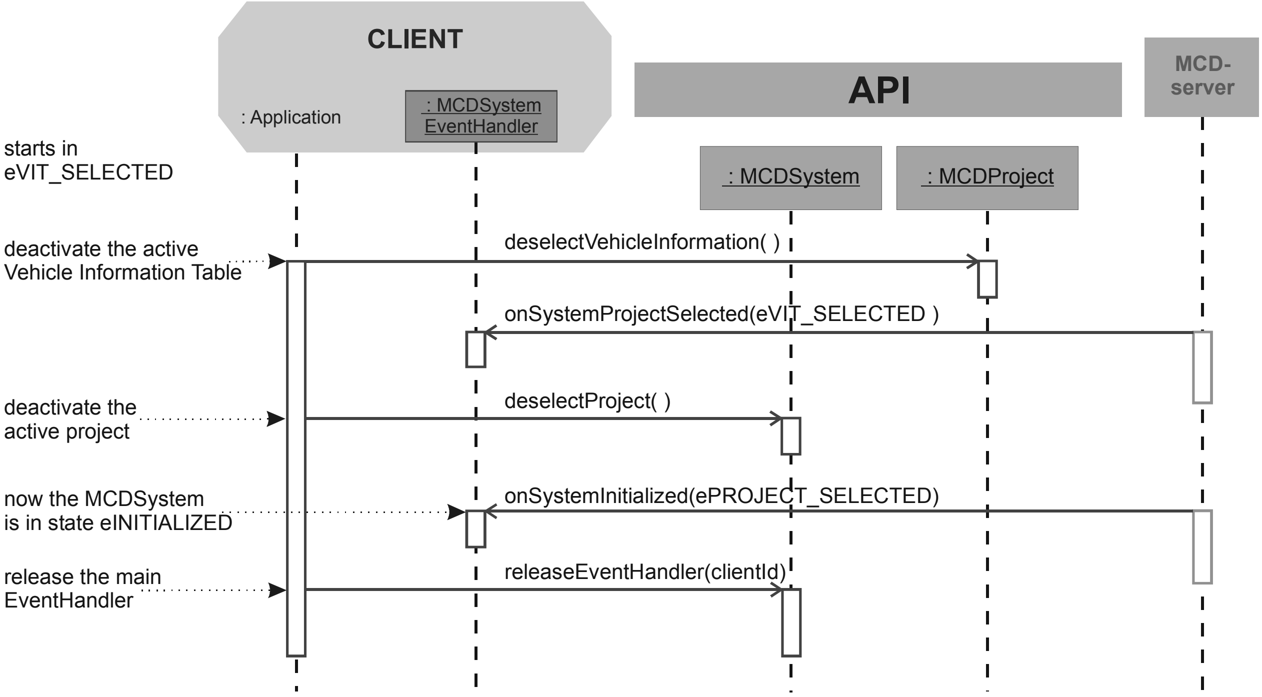

CLIENT
: Application
: MCDSystem EventHandler
API
MCD-server
: MCDSystem
: MCDProject
starts in eVIT_SELECTED
deactivate the active Vehicle Information Table
deselectVehicleInformation( )
onSystemProjectSelected(eVIT_SELECTED)
deactivate the active project
deselectProject( )
now the MCDSystem is in state eINITIALIZED
onSystemInitialized(ePROJECT_SELECTED)
release the main EventHandler
releaseEventHandler(clientId)

Figure 32 — Destruction

This Sequence diagram starts in the `MCDSystem` state `eVIT_SELECTED` and with an active Vehicle Information Table. Logical Links don't exist at this time anymore.

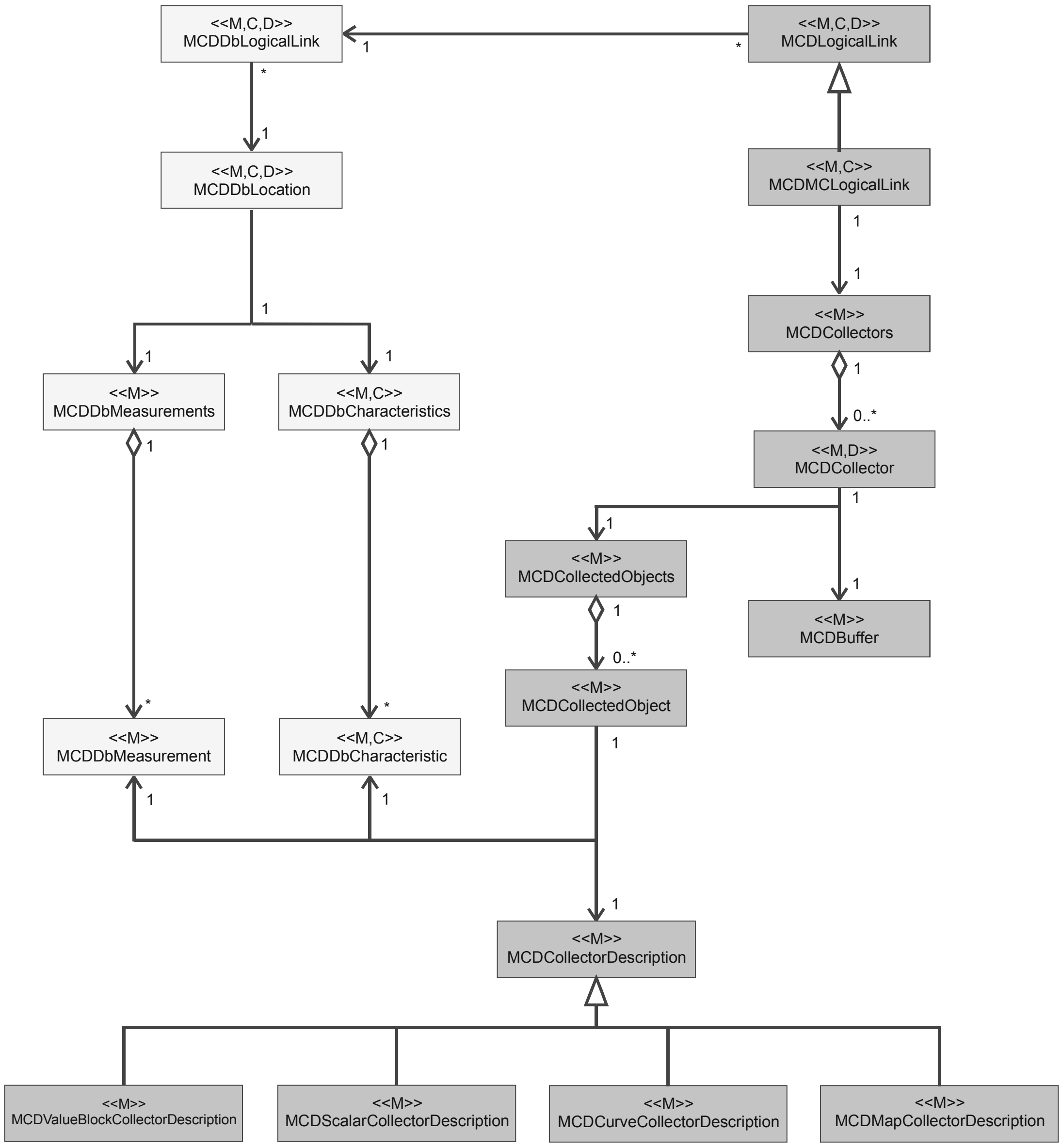
First deactivate the Vehicle Information Table and after successful deactivation an event will be delivered. Then the project can be deactivated and for this state transition of the `MCDSystem` object an event will be delivered too. The last action is to release the main EventHandler.

8 Function block Common MD

8.1 Collector

8.1.1 ERD

The following ERD shows the associations for the Collectors. On the left side the database objects are shown yellow, the runtime objects are shown on the right side (green). Each Collector is conclusively connected to exactly one Logical Link. By using the database Logical Link, the client has access to the database characteristics and database measurements which are available for this collector. At the runtime side, the collector consists of the collection for the objects that the collector should collect (`MCDCollectedObjects`), and the buffer which holds the measured data. The Collection of objects contains `MCDCollectedObjects`, which represent a concatenation of DB objects (Measurements / Characteristics) and its CollectorDescription. The Collector description is different depending on the type of the object.



<<M,C,D>> MCDDbLogicalLink
<<M,C,D>> MCDLogicalLink
<<M,C,D>> MCDDbLocation
<<M,C>> MCDMCLogicalLink
<<M>> MCDCollectors
<<M>> MCDDbMeasurements
<<M,C>> MCDDbCharacteristics
<<M,D>> MCDCollector
<<M>> MCDCollectedObjects
<<M>> MCDBuffer
<<M>> MCDCollectedObject
<<M>> MCDDbMeasurement
<<M,C>> MCDDbCharacteristic
<<M>> MCDCollectorDescription
<<M>> MCDValueBlockCollectorDescription
<<M>> MCDScalarCollectorDescription
<<M>> MCDCurveCollectorDescription
<<M>> MCDMapCollectorDescription
1
*
0..*

Figure 33 — Collector associations

8.1.2 Concept

The task of the Collectors is to acquire the measured values of different objects (Measurements or Characteristics) with a common rate over a defined period of time (continuous data acquisition).

The sequence of objects to be acquired (or sub-objects) is predetermined by the AUSY.

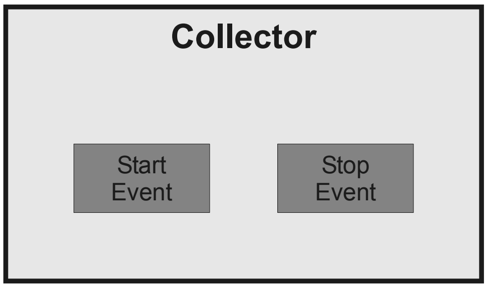


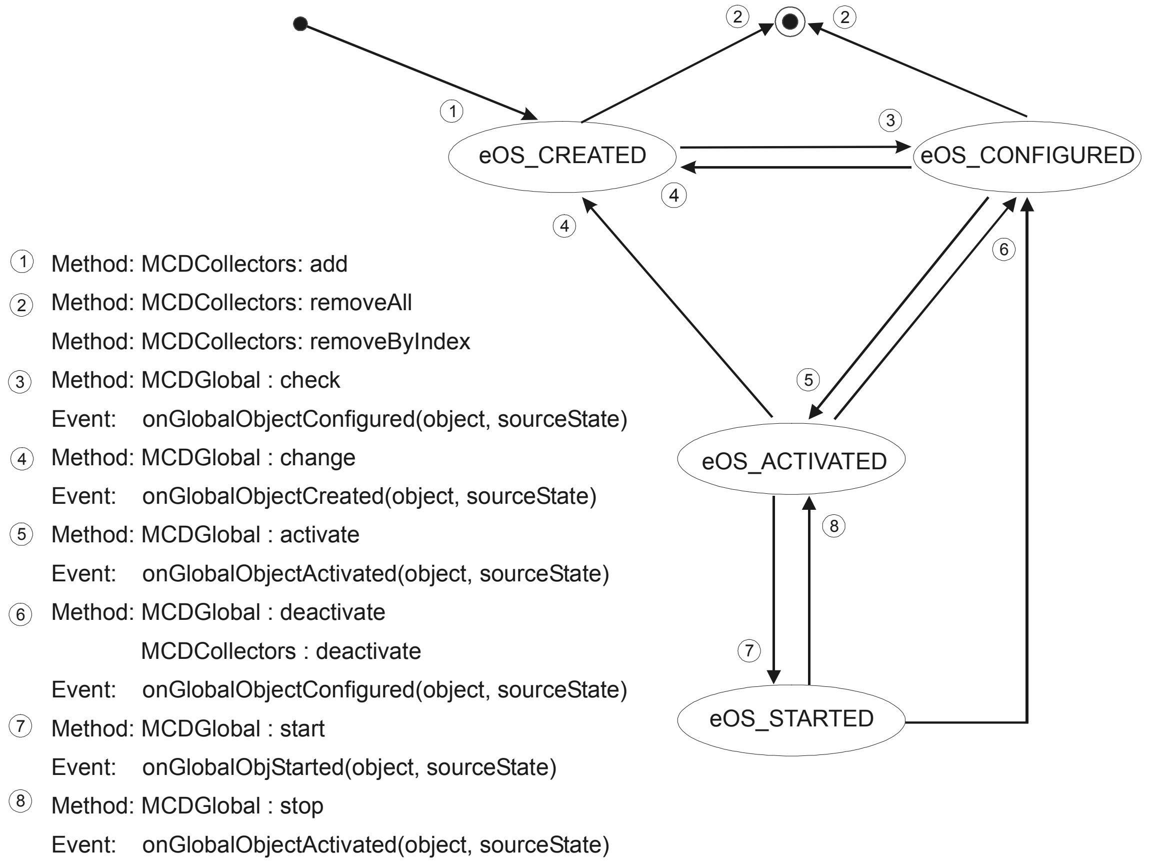
Figure 34 — Trigger conditions of collectors

The Collector contains trigger conditions for start and stop of the data acquisition.

As a condition, commands of the AUSY-system (Start / Suspend) are permitted.

Collectors may be instantiated and deleted. Before usage, the Collectors shall be activated. For a complete configuration, at minimum the list of the objects to be acquired shall be given. For all other settings exist defaults.

Collectors have the following state diagram:



1 Method: MCDCollectors: add

2 Method: MCDCollectors: removeAll
Method: MCDCollectors: removeByIndex

3 Method: MCDGlobal : check
Event: onGlobalObjectConfigured(object, sourceState)

4 Method: MCDGlobal : change
Event: onGlobalObjectCreated(object, sourceState)

5 Method: MCDGlobal : activate
Event: onGlobalObjectActivated(object, sourceState)

6 Method: MCDGlobal : deactivate
MCDCollectors : deactivate
Event: onGlobalObjectConfigured(object, sourceState)

7 Method: MCDGlobal : start
Event: onGlobalObjStarted(object, sourceState)

8 Method: MCDGlobal : stop
Event: onGlobalObjectActivated(object, sourceState)

Figure 35 — State transition diagram of a collector

The states can be explained as follows:

`eOS_CREATED`: The object has been established and may be addressed by the AUSY. A data acquisition is not possible yet. Within the state `eOS_CREATED`, the objects to be acquired may be added to an acquisition list. This list is empty after creation. Within the state `eOS_CREATED`, the configuration of the Collector is carried out.

`eOS_CONFIGURED`: The MC-server has checked, that the objects to be acquired may be acquired continuously with a common rate. No measurement takes place within the state `eOS_CONFIGURED`. For every LogicalLink exist a DEFAULT rate for all Measurements and Characteristics.

`eOS_ACTIVATED`: The Collector is completely ready for use. By means of Activate, the measurement takes place and the measured values are filled to the ring buffer. Within the state `eOS_ACTIVATED` the start trigger condition is monitored (AUSY trigger command or Watcher). No data transfer to the AUSY takes place. Within the state `eOS_ACTIVATED`, the ECU selected via the Logical Link shall be online.

`eOS_STARTED`: The transition to this state is carried out independently by the MC-server. The state transition is initiated after the Collector has been activated and the start condition has become true (AUSY trigger command or Watcher) and all other preconditions have been fulfilled (e.g. start delay) . Within the state `eOS_STARTED` a data transfer to the AUSY can take place and the stop trigger condition is monitored.

All state transitions are reported by events.

If a Collector is within the state `eOS_ACTIVATED` or `eOS_STARTED`, an automatic state transition to the state `eOS_CONFIGURED` takes place if the Logical Link becomes Offline. For this, it is of no importance if the offline status is locally detected by the MC-server or predetermined by the AUSY. The state change is reported by the event `onGlobalObjConfigured`.

A Collector can be started by Ausy or Watcher. Any state change during a collector start and stop shall be reported immediately via an event to the client. A time delay has no influence to it.

In case a client registers an event handler at a collector during a measurement, which means the collector is in state `eOS_STARTED / eOS_ACTIVATED`, the buffer pointer will be positioned at the oldest entry in the ring buffer, which results in a filled buffer for the client. The client will receive immediately an onCollectorResultReady event, if the values in the buffer exceed the event limit for the client.

Reconfiguration of MCDCollector and contained objects, e.g. MCDCollectedObjects or MCDBuffer is only allowed in state `eOS_CREATED`.

Table 19 — Collector states

			MCDCollectors		MCDGlobal						MCDCollector				MCDCollected Objects	MCDBuffer		MCDTime Delay
System State	Dependend LL State	Collector State	deactivate	removeByIndex removeAll	check	change	activate	deactivate	start stop	setStartWatcher setStopWatcher	setNoOfSamples ToFireEvent	fetchMonitoringFrames	fetchResults	createBufferIdFor Polling	addScalar addCurve addMap removeAll removeByIName removeByIndex addAscii addValueBlock	setDownSampling setTimeStamping	setSize setRate	configure
eLOGICALLY_CONNECTED	eCREATED		▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
	eOFFLINE	eOS_CREATED		X		X				X¹	X			X	X	X	X	X
		eOS_CONFIGURED	X	X	X	X		X			X	▨		X		X		
		eOS_ACTIVATED	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
		eOS_STARTED	▨	▨	▨	▨	▨	▨	▨	▨	▨		▨	▨	▨	▨	▨	▨
		eOS_PAUSED	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
	eONLINE	eOS_CREATED		X	X	X				X¹	X			X	X	X	X	X
		eOS_CONFIGURED	X	X	X	X	X	X			X	▨		X		X		
		eOS_ACTIVATED	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
		eOS_STARTED	▨	▨	▨	▨	▨	▨	▨	▨	▨		▨	▨	▨	▨	▨	▨
		eOS_PAUSED	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
	eCOMMUNI CATION	eOS_CREATED		X		X				X¹	X	X		X	X	X	X	X
		eOS_CONFIGURED	X		X		X	X			X	▨		X				
		eOS_ACTIVATED	X			X	X	X	X		X	▨	X	X				
		eOS_STARTED	X					X	X		X	X	X	X				
		eOS_PAUSED	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨

Key

▨	State could not be reached
X¹	MCDWatcher shall be in state eWS_INACTIVE or eWS_CREATED

The following properties existing within the Collector may be influenced:

Object List: List of the objects to be acquired. Sub ranges of an object may be selected as well. The list of objects is empty after instantiating a Collector. With one Collector only objects of one module may be acquired. This is realized by the direct dependency of a collector to the Logical Link. One `MCDDbObject` (`MCDDbMeasurement` or `MCDDbCharacteristic`) can only be used once in a collector.

Time Stamp: Defines, if time information shall precede each sample of acquired data. The time represents a relative time, which refers to the first activation of a Collector, Watcher or Recorder in the whole server.
Default: No

Rate: Defines the acquisition rate. The rate is given by a unit together with a value.

Depending on the respective tool, there is a server wide valid default setting for the sampling rate of Collectors. How this default setting may be polled or modified shall be looked up in the users manual of the respective tool.

DownSampling: Defines, by how many samples the data is reduced before storage. Shall be a value greater than zero. The value means that every x^{th} value is stored and all values in between are discarded.
Default: 1 that means all values are stored in the buffer

Start delay: Defines the period of time after the start condition has been met (positive sign), during which no values shall be acquired yet, or the period of time before the start condition has been met (negative sign) during which the values shall already be acquired.
Default: 0

Stop delay: Defines the period of time after the stop condition has been met (positive sign), during which values shall still be acquired, or the period of time before the stop condition has been met (negative sign), during which no values shall be acquired anymore. In case of a negative stop delay time, the depth of the buffers shall be designed accordingly.
Default: 0

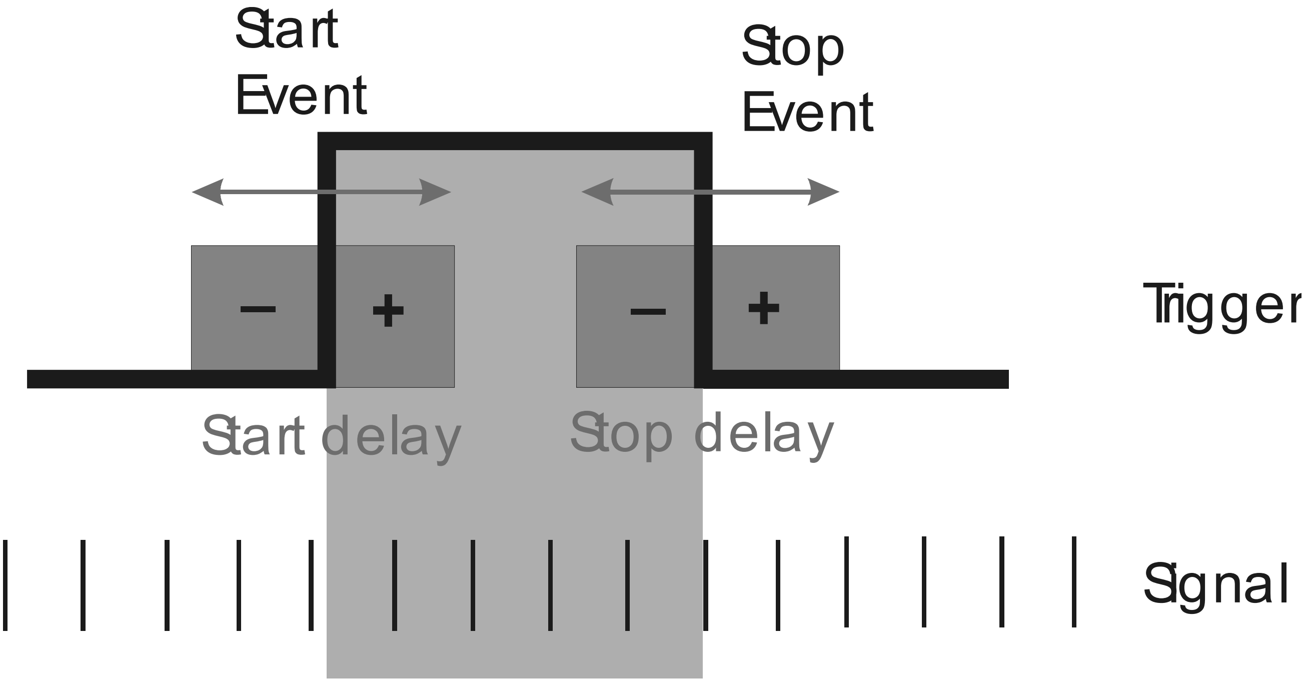


Figure 36 — Start- and stop event triggering

NoOfSamples: This number defines how many new samples shall be collected in the collector's buffer to fire again an `onCollectorResultReady` event (Default value is one).

BufferSize: Defines the size (in samples) of the ring buffer for the temporary storage of the measurements. The current filling level of the buffer may be inquired by means of event `onCollectorResultReady` or by the method `getFillingLevel`.
Default: The buffer size of a result buffer is Collector-specific. For Collectors, the default buffer size is one (server wide value) Sample line.

The physical representation of measurements is always A_FLOAT64 or A_ASCIISTRING, since the computation method which translates the hex values only support these data types. The resulting data type could be read from the type for computation method.

Collectors shall be identified in the FetchResult method, so that the client could create its own measurement management.

After instantiation of a new collector, the MCD-server automatically generates a new name for it. This name is unique. The name is composed by using the Logicallink ShortName and RateInfo ShortName. Additionally the number of the current collector used. The unique number is assigned only once to a collector during the lifetime of a server. With this name the identification of collectors is possible after the reference has been released by the client and should be again referenced via the collectors collection,

e.g.

```
<ShortName Logicallink> :: < ShortName RateInfo>  ::   <CollectorUniqueNumber>
"CCPsim_CCP_CAN1        :: 100ms                  ::   2"
```

The name of collected objects in runtime non recorders collectors is the shortname of the associated Db object, since it is assumed that only one element for each associated DbObject could exists in the `MCDCollectedObjects` collection.

8.1.3 Result access

8.1.3.1 Object list configuration

Measurements or Characteristics may be entered into the measurement acquisition list of a Collector. At configuration, the respective database object (`MCDDbMeasurement`, `MCDDbCharacteristic`) is named and provided with a CollectorDescription. This defines the RepresentationType to be used (Physical or ECU). In case of a Characteristic, depending on the kind of Characteristic (Ascii, ValueBlock, Scalar, Curve or Map), it has additionally to be defined what shall be acquired (Z-Value, X-Axis, Y-Axis or combination). In case of a ValueBlock, Curve or Map also ranges or working points can be defined additionally. The global support of characteristics or the support of curves and maps by a collector is optional.

Objects of MCDCollectedObjects are not removed by state transitions.

8.1.3.2 Structure of the collector sample list

The data acquisition within Collectors follows a ring buffer principle. For this, the following structure is applied for any line of the buffer:

Table 20 — Elements of sample list

TimeStamp	Value list
Optional	
A_FLOAT64	
Returns the relative time since the start of the first activated watcher, collector, or recorder. The value is to be interpreted as follows: The number before the decimal point is the number of seconds since the absolute activation time as can be obtained from the MCDServer. The decimal places represent the fraction of a second.	For each object of the object list one value entry is placed. The data types are determined by the collected objects (Db objects and collector description objects)

The MC-server can decide upon the specific object type, the number of objects and depending on whether a time stamp is entered or not upon the specific need of resources for a line (sequence of Samples).

The depth of the ring buffer (number of lines) is configured by the AUSY-system. The filling level of the buffer may be queried.

The maximum buffer size supported by a MCD-server can be obtained by <MCD> MCDSystem::getMaximumBufferSize()::A_UINT32. The maximum buffer size shall be at least one, that is, a MCD-server needs to support buffers for scripted clients. In case a client wants to set a buffer size greater than MaxBufferSize, an exception of type is ePAR_VALUE_OUT_OF_RANGE thrown and the buffer size is not modified.

The data is transported by means of the method fetchResults, which delivers MCD result objects. For this, it is predetermined, how many lines at maximum shall be read out with one access operation.

The filling of the buffer is carried out at the responsibility of the MC-server in accordance with the preset rate. While doing so, it shall be guaranteed for the synchronicity for the samples of one line. A buffer entry can only be made if data are available from the control units.

The MCDResult object returned by the call to fetchResults() is owned by the client. That is, copies are delivered.

Exception: The rate eCSU_FRAME_AVAILABLE should be used to deliver samples in one telegram independent from the total configuration of the collector samples. Thus, it is possible that within one collector measurements and / or characteristics from different telegrams can be acquired. If one telegram is received an incomplete result is generated, because it only contains the values that were part of the telegram. In this case, the warning eRT_RESULT_INCOMPLETE is inserted in the MCDError object of the MCDResults. The result will be transmitted as a complete line, which contains not valid elements in place of the missing values. MCDValue.isValid() signals if a value is present or not in the current result of a collector. The time stamp is generated out of the message that was received for the parameters with valid data.

The error code eRT_COMMUNICATION_ERROR is specified in the case the data acquisition was interrupted temporarily and could be restarted.

If the AUSY does not scan the measurements from the MC-server in an adequate data stream in accordance with their emergence at the ECUs, a buffer overflow may occur. For the overflow it is defined, that the oldest values are overwritten by current values; that means in case of a call always the latest current values are available. In case of an overflow, the flag hasOverflow on the MCDResults object is set to true and the MCDError object on the MCDResults object is set to eRT_COL_BUFFER_OVERFLOW. In the faultless case the status has the value eNO_ERROR.

In case of a MCDCollector, a buffer overflow is signalled by an event onCollectorError containing an error of type <MC> eRT_COL_BUFFER_OVERFLOW. In addition, the existence of a buffer overflow can be queried by using the method <MCD> MCDResults::hasOverflow():A_BOOLEAN. Moreover, the method <MC> MCDResults::getError() obtained the most severe error from all contained MCDResult objects. In case of no error is present, an empty MCDError object is returned.

If more than one result sample is available, a summary error will be reported. A Summary error is reported if at least one of the results of a result collection includes such an error or warning. Only the error with the highest severity will be reported.

In case of an internal error that leads to a not working collector, the client should stop the collector. This shall be determined by interpretation of the vendor specific error code.

In case the buffer starts overriding elements that have not been fetched by all clients, an overflow event is sent to those clients that will “lose” results, that is, to those clients that have not fetched the overridden results.

If more than one error exists in one sample (result) line, only the error with the highest severity will be transported.

Errors (e.g. eRT_COL_BUFFER_OVERFLOW, eRT_INTERNAL_ERROR, eRT_COMMUNICATION_ERROR) and warnings (eRT_RESULT_INCOMPLETE) will also be reported in the onCollectorError Event.

8.1.3.3 Result structure

MCDResult objects are static and stateless.

The results of a Collector meet the general structure of `MCDResults`. One result (single line of the Collector) consists of exactly one `MCDResponse` object, as the results come from only one ECU.

This `MCDResponse` consists of 1 up to 2 `MCDResponseParameter` objects. The first Response Parameter is the optional time stamp, which can be set at the configuration of the Collector. The second Response Parameter of this collection is the ObjectStruct as entry point for the single acquired objects.

Thus, the single acquired Measurements and Characteristics are always located on the second level of the Response Parameters. In case of Measurements, the acquired value is located directly on this second level. In case of Characteristics, this level may be branched further, as for Characteristics X-Axis, Y-Axis, Z-Values or combinations may be acquired. That means that in case of Characteristics on the second level of the Response Parameters only the name of the Characteristic as a Type Structure is located.

The further branching into a fourth Response Parameter level separate between Axis (X or Y) and Z-Value.

Generally applies, that in case of complex data types as `eFIELD` (e.g. values or axis values) or `eSTRUCTURE` further Response Parameter objects are subordinated.

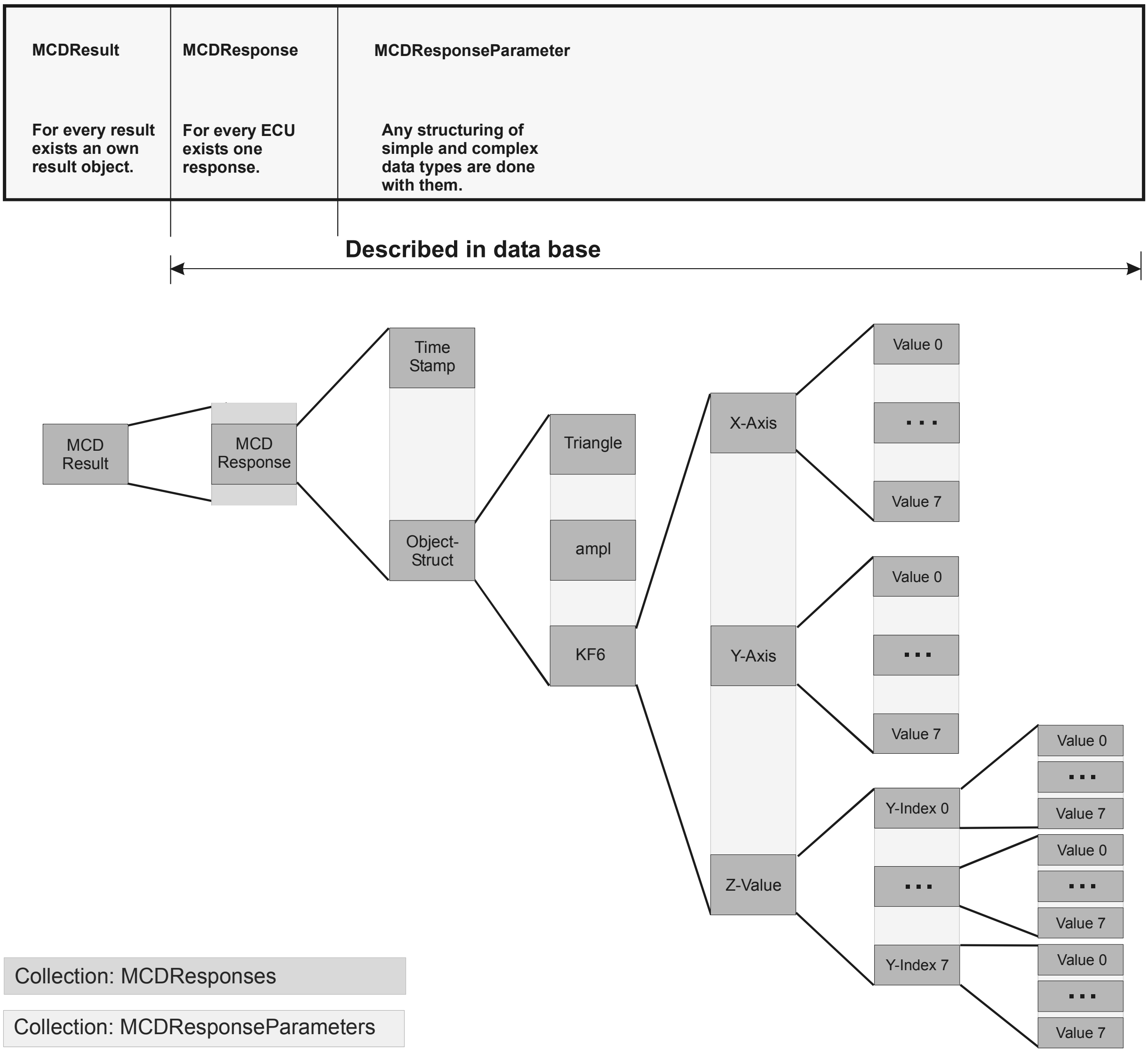


Figure 37 — Collector result structure

It is possible to access the single ResponseParameters by Index or by short name.

The following specifications apply, whereat the identifiers given in the column name shall be used normatively for the single ResponseParameters. The Measurement- and Characteristic names in italics shall be replaced by the respective database name. The values of axes and values are numbered consecutively starting with 0.

Table 21 — Names and types of ResponseParameters in collectors

Element	ShortName	DataType
Collector SampleList	ObjectStruct	eSTRUCTURE
Measurement	*Measurement Name*	Measurement dependent in case of HEX representation and dependent on the computation method in case of PHYS representation.
Curve Axis	Axis	eFIELD
Map X Axis	X-Axis	eFIELD
Map Y Axis	Y-Axis	eFIELD
Scalar (is direct the Z value)	*Characteristic Name*	Characteristic dependent for Z-Value in case of HEX representation and dependent on the computation method in case of PHYS representation.
Curve *1)	*Characteristic Name*	eSTRUCTURE
Map *1)	*Characteristic Name*	eSTRUCTURE
Value Block *1)	*Characteristic Name*	eFIELD
ASCIIString	*Characteristic Name*	A_ASCIISTRING
Curve Z Value	Z-Value	eFIELD
Map Z Value	Z-Value	eFIELD
Values of Curve, Values of Map for X-axis or Y-Axis	Value 0 .. Value n-1 *2)	Characteristic dependent in case of HEX representation and dependent on the computation method in case of PHYS representation.
Values of Map and ValueBlocks for Z-Values first dimension	YIndex 0 ... YIndex m-1 *2)	eFIELD
Values of Map and ValueBlocks for Z-Values second dimension	Value 0 .. Value n-1 *2)	Characteristic dependent in case of HEX representation and dependent on the computation method in case of PHYS representation.

*1) If in case of a Curve, ValueBlock or Map a single value is selected via the range of the CollectorDescription, this value is located also on the fourth Responseparameter level (level Values), whereat the term Value is used as object name and the characteristic depending type of the value is used as type. This is independent from the fact, whether the selection has been related to a single value (Z-Value) or an interpolation node (X-Axis, Y-Axis). This means that in case of a single value (axis or Z) always the sequence ObjectStuct (STRUCTURE)- Characteristic (STRUCTURE) - Z-Value (FIELD) - Value (Characteristic dependent) is used.

*2) The numbering of the single value elements independently from the selection that has been made via the CollectorDescription selection always starts with 0, that means that for example within the numbering there is no relation to the node position.

In case of `MCDCollectorDescriptionType` is `eCDI_VALUE_MAP_WP` or `eCDI_VALUE_CURVE_WP` the current value of the adjustable object and the respective working point is stored. This means in case of a Map you have the FIELD Elements X-Axis, Y-Axis and Z-Value with each time one element inside.

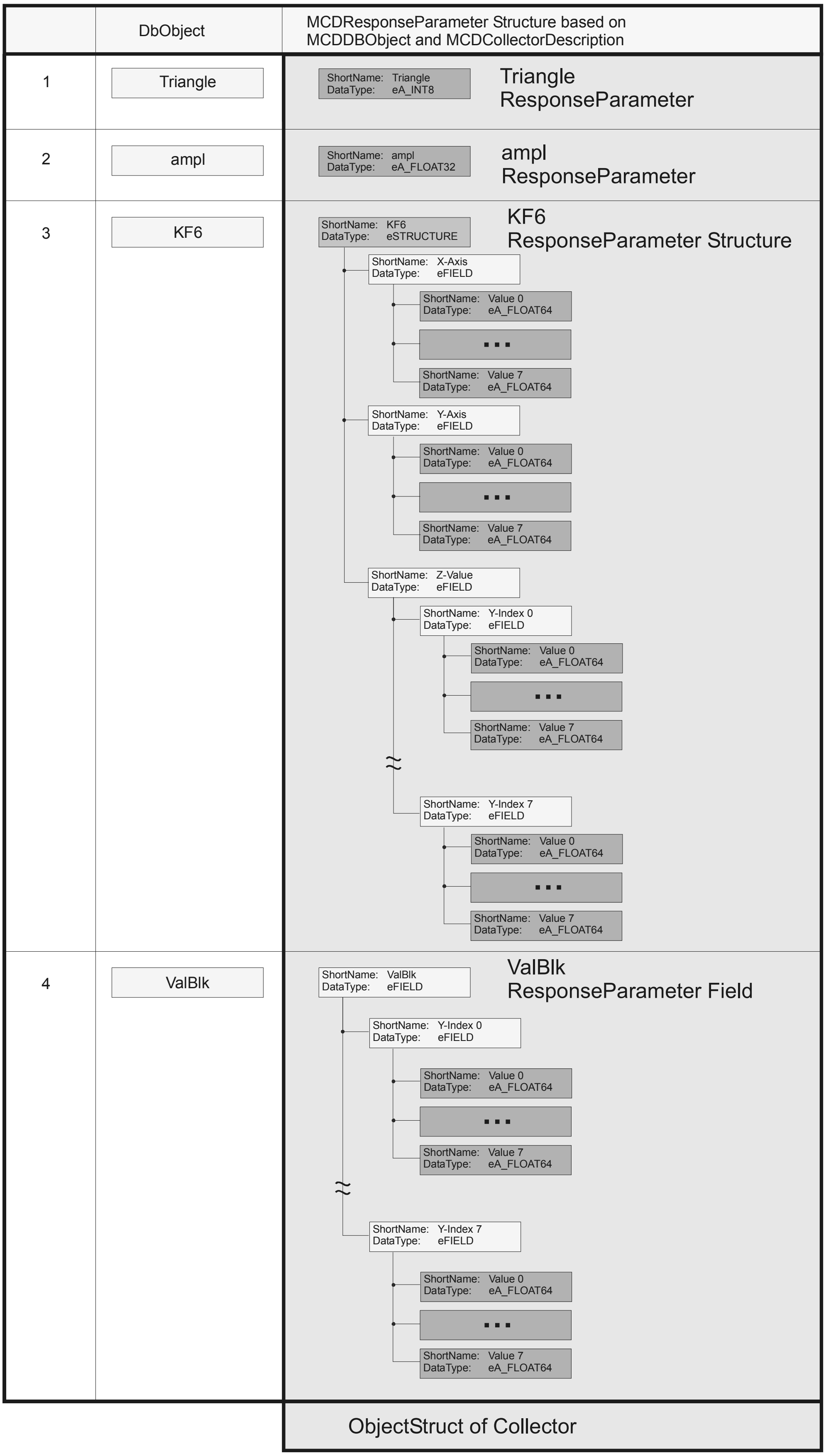
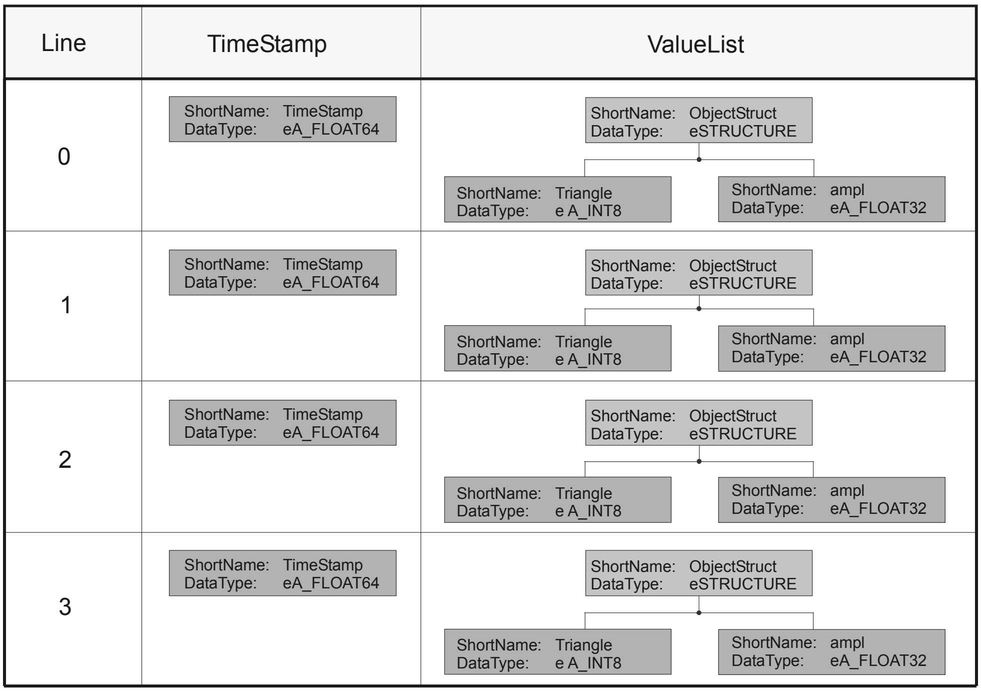

DbObject
MCDResponseParameter Structure based on MCDDBObject and MCDCollectorDescription
1
Triangle
ShortName: Triangle DataType: eA_INT8
Triangle ResponseParameter
2
ampl
ShortName: ampl DataType: eA_FLOAT32
ampl ResponseParameter
3
KF6
ShortName: KF6 DataType: eSTRUCTURE
KF6 ResponseParameter Structure
ShortName: X-Axis DataType: eFIELD
ShortName: Value 0 DataType: eA_FLOAT64
...
ShortName: Value 7 DataType: eA_FLOAT64
ShortName: Y-Axis DataType: eFIELD
ShortName: Value 0 DataType: eA_FLOAT64
...
ShortName: Value 7 DataType: eA_FLOAT64
ShortName: Z-Value DataType: eFIELD
ShortName: Y-Index 0 DataType: eFIELD
ShortName: Value 0 DataType: eA_FLOAT64
...
ShortName: Value 7 DataType: eA_FLOAT64
ShortName: Y-Index 7 DataType: eFIELD
ShortName: Value 0 DataType: eA_FLOAT64
...
ShortName: Value 7 DataType: eA_FLOAT64
4
ValBlk
ShortName: ValBlk DataType: eFIELD
ValBlk ResponseParameter Field
ShortName: Y-Index 0 DataType: eFIELD
ShortName: Value 0 DataType: eA_FLOAT64
...
ShortName: Value 7 DataType: eA_FLOAT64
ShortName: Y-Index 7 DataType: eFIELD
ShortName: Value 0 DataType: eA_FLOAT64
...
ShortName: Value 7 DataType: eA_FLOAT64
ObjectStruct of Collector

Figure 38 — Object list structure

As a Result represents one line of a buffer, in the following for the case that only the Measurements "Triangle" and "ampl" are configured, the principle structure of the Collector Buffer is shown.



Line	TimeStamp	ValueList
0	ShortName: TimeStamp DataType: eA_FLOAT64	ShortName: ObjectStruct DataType: eSTRUCTURE ShortName: Triangle DataType: e A_INT8 ShortName: ampl DataType: eA_FLOAT32
1	ShortName: TimeStamp DataType: eA_FLOAT64	ShortName: ObjectStruct DataType: eSTRUCTURE ShortName: Triangle DataType: e A_INT8 ShortName: ampl DataType: eA_FLOAT32
2	ShortName: TimeStamp DataType: eA_FLOAT64	ShortName: ObjectStruct DataType: eSTRUCTURE ShortName: Triangle DataType: e A_INT8 ShortName: ampl DataType: eA_FLOAT32
3	ShortName: TimeStamp DataType: eA_FLOAT64	ShortName: ObjectStruct DataType: eSTRUCTURE ShortName: Triangle DataType: e A_INT8 ShortName: ampl DataType: eA_FLOAT32

Figure 39 — Collector buffer structure

8.1.4 Collector usage in diagnostics

In a D server context, `MCDCollector` objects are used for collecting bus traffic trace data. The basic collector concepts and mechanisms are identical as for the measurement context. However, a few aspects of the `MCDCollector` functionality have been simplified for the diagnostics context, as the collection of bus traffic data has less elaborate requirements that do not require the full measurement collector functionality.

One of these simplifications concerns the retrieval and representation of collected data. In the diagnostics context, monitored data will be contained in the `MCDDatatypeAsciiStrings` collection returned by the `MCDCollector::fetchMonitoringFrames(…)` method. Each monitored message will be contained in one `MCDDatatypeAsciiString` within that collection. The semantics of these monitoring data strings are explained in detail in Monitoring vehicle bus traffic.

The other simplification concerns the `MCDCollector` states. As the diagnostics use case requires only that the collection of vehicle bus traffic can be started and stopped, a `MCDCollector` that is to be used within a D-server shall implement the state diagram as shown in Figure 40 — A MCDCollector used for diagnostics can only be started and stopped.

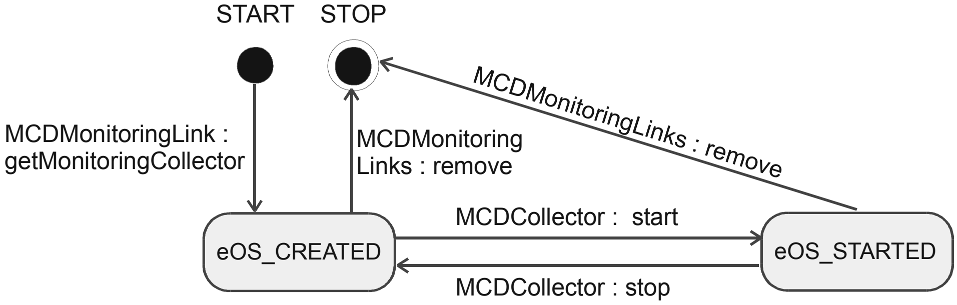

START
STOP
MCDMonitoringLink : getMonitoringCollector
MCDMonitoring Links : remove
MCDMonitoringLinks : remove
MCDCollector : start
MCDCollector : stop
eOS_CREATED
eOS_STARTED

Figure 40 — A MCDCollector used for diagnostics can only be started and stopped.

In a diagnostics context, all `MCDCollector` methods can be called in any `MCDCollector` state. For more detailed information on using the `MCDCollector` in a D-server, please refer to Monitoring vehicle bus traffic.

8.2 Use cases

8.2.1 Measurement with a collector - activation

At first, the Collector is created within the Collector–Collection. After creation, this Collector is empty, that means that there are no objects within the CollectedObject-Collection and the parameters within the Collector and within the buffer hold the default setting. Some of the default settings are taken over from ASAM MCD 2 MC, others are determined by MCD-server. To be able to specifically evaluate the Events for this Collector, an EventHandler for this Collector is registered. After this, the reconfiguring of the Collector properties may take place in case the default values shall not be used. Following this, the objects to be measured by the Collector (Measurements and Characteristics) are included into the CollectedObject-Collection as tupel of DbObject and CollectorDescription. It shall be noted that each DbObject may only exist once within the Collector and Measurements shall always be accepted. It is optional and thus depending on the tool if the Collector is able to measure Characteristics. After the CollectedObjects-Collection has been filled completely, the complete configuration is tested by means of Check() and the Collector transits to the state `eOS_CONFIGURED`. If no Measurements and / or characteristics added and the `MCDCollectedObjects` are empty, the method `check` delivers the error `eRT_COL_CHECK_FAILED`. After configuration the complete Collector may be activated.

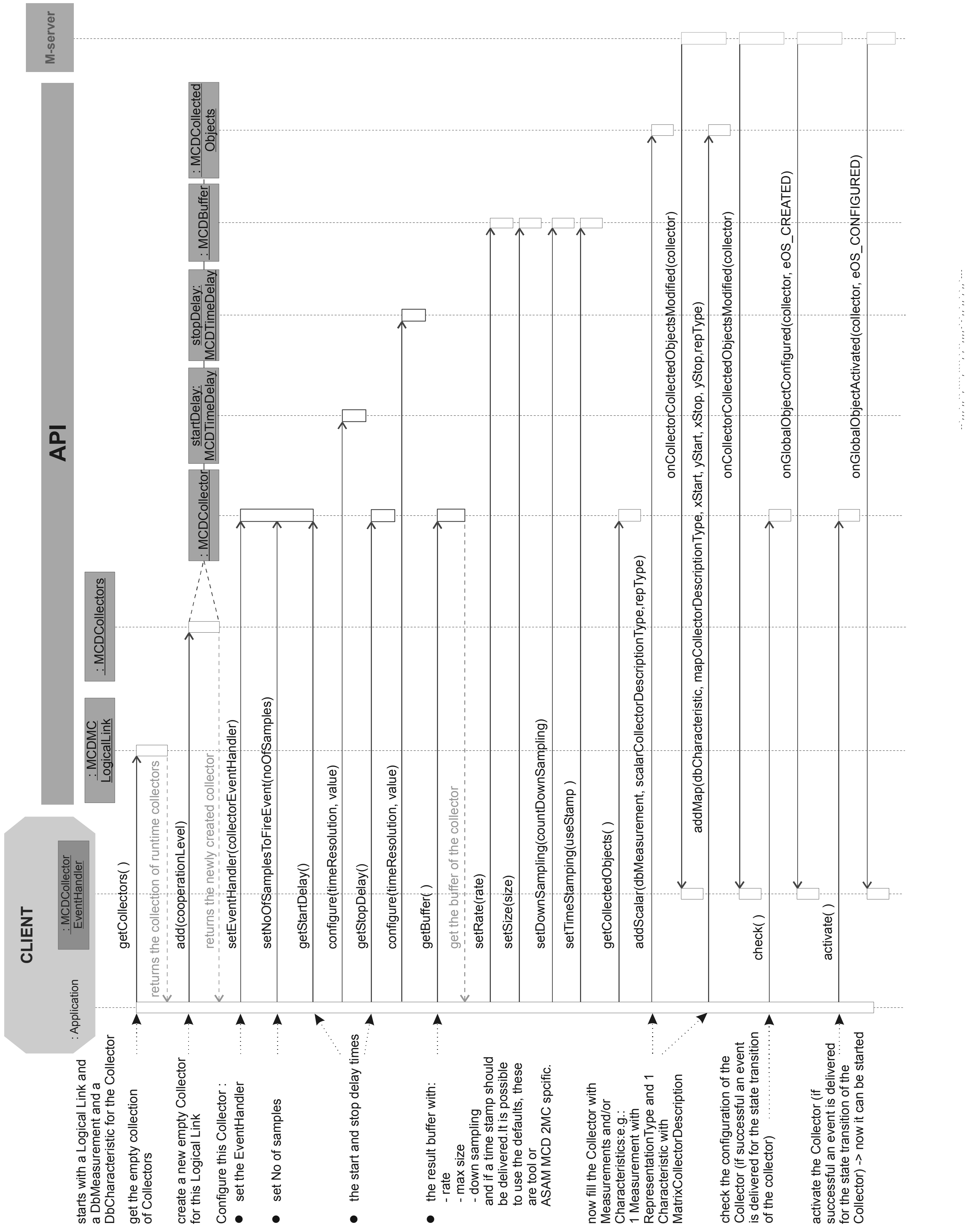

CLIENT
API
M-server
: Application
: MCDCollector EventHandler
: MCDMC LogicalLink
: MCDCollectors
: MCDCollector
startDelay: MCDTimeDelay
stopDelay: MCDTimeDelay
: MCDBuffer
: MCDCollected Objects
getCollectors( )
returns the collection of runtime collectors
add(cooperationLevel)
returns the newly created collector
setEventHandler(collectorEventHandler)
setNoOfSamplesToFireEvent(noOfSamples)
getStartDelay()
configure(timeResolution, value)
getStopDelay()
configure(timeResolution, value)
getBuffer( )
get the buffer of the collector
setRate(rate)
setSize(size)
setDownSampling(countDownSampling)
setTimeStamping(useStamp)
getCollectedObjects( )
addScalar(dbMeasurement, scalarCollectorDescriptionType,repType)
onCollectorCollectedObjectsModified(collector)
addMap(dbCharacteristic, mapCollectorDescriptionType, xStart, yStart, xStop, yStop,repType)
onCollectorCollectedObjectsModified(collector)
check( )
onGlobalObjectConfigured(collector, eOS_CREATED)
activate( )
onGlobalObjectActivated(collector, eOS_CONFIGURED)
starts with a Logical Link and a DbMeasurement and a DbCharacteristic for the Collector
get the empty collection of Collectors
create a new empty Collector for this Logical Link
Configure this Collector :
● set the EventHandler
● set No of samples
● the start and stop delay times
● the result buffer with:
- rate
- max size
- down sampling
and if a time stamp should be delivered.It is possible to use the defaults, these are tool or ASAM MCD 2MC spcific.
now fill the Collector with Measurements and/or Characteristics:e.g.: 1 Measurement with RepresentationType and 1 Characteristic with MatrixCollectorDescription
check the configuration of the Collector (if successful an event is delivered for the state transition of the collector)
activate the Collector (if successful an event is delivered for the state transition of the Collector) -> now it can be started

Figure 41 — Collector activation

8.2.2 Measurement with a collector - result access

After the Collector has been started the occurring of result records is reported by means of events. An event `onCollectorResultReady` will reported, if the number of results is reached configured with `MCDCollector:setNoOfSamplesToFireEvent.` The default value is one. This goes on until the Collector is stopped. Result records are fetched (this means all results which will be transported to the Client will for this Client not remain in the buffer) by using the method `fetchResult()` and are provided as Result Collection.

For multi thread capability each of the Client working with this Collector gets an Id for requesting results from the buffer. This Id will be created for the Client by calling `createBufferIdForPolling` or will be delivered with the event `onCollectorResultReady` which is announcing the result. Calling `MCDCollector::getFillingLevel(bufferID:A_UINT32):A_UINT32` returns the filling level of the buffer depending on this special thread, the total filling level of the buffer over all Clients can be determined with `MCDBuffer::getFillingLevel(error:MCDError):A_UINT32.`

The `getFillingLevel` methods deliver a summary error if at least one of the results which is available in the buffer includes an error or warning. Only the error with the highest severity will be reported.

Thread in the context "MCDCollector buffer ID” has a different meaning from “Client” in the Multi-Client definition. In the MCDCollector, the thread is one of several possible instances within a client, that handles results by calling fetchResults(). For example, one client shows several Views of the data of one collector. It can use a different Buffer ID for each view for easier result handling in the different threads.

For each call to `createBufferIdForPolling`() and for each registered eventhandler, that will receive `onCollectorResultReady` at the collector itself, the client shall call fetchResult to avoid overflows. A mixed mode (registering an eventhandler and not want to receive `onColletorResultReady` event, therefore requesting a buffer management in the collector via `createBufferIdForPolling`) is not supported.

The method `fetchResult()` support with parameter `numReq` different possibilities of result access:

Table 22 — Result access possibilities

Value of numReq	Meaning
- n (n ∈ N, n <> 0)	returns n results. If n > m, where m is the number of available results in the buffer, m results will be delivered. The results in the delivered collection are ordered by their timestamp. The element with the lowest index has got the newest timestamp. All results will be removed from the buffer. In case of n = -1 the Online Value is taken over (last value).
0	returns the whole buffer. After this the buffer is empty. The results in the delivered collection are ordered by their timestamp. The element with the lowest index has got the oldest timestamp.
+ n (n ∈ N, n <> 0)	returns n results. If n > m, where m is the number of available results in the buffer, m results will be delivered. The results in the delivered collection are ordered by their timestamp. The element with the lowest index has got the oldest timestamp. The delivered results will be removed from the buffer.

The structure of the Result Collection, which is harmonized with the D function block, is described in the following.

For each Result within the Result Collection the Collection of Responses is polled. As the Collector may only acquire the measured values from one module, there is only exactly one Response within this Collection at a time. For this Response the contained Response Parameter may be polled. The Response Parameter contains values that have already been broken down to the result structure by the MCD-server. The Response Parameter may be accessed via the name within the Collection. With this, independently from a reconfiguration of the Collector, a more simple access to result elements that have been kept is possible. Each of these parameters contains a type. Depending on this type being a simple type or a complex type, the further processing takes place. In case of simple type the value may be polled, in case of complex type the polling of the Response Parameter Collection of the next level takes place.

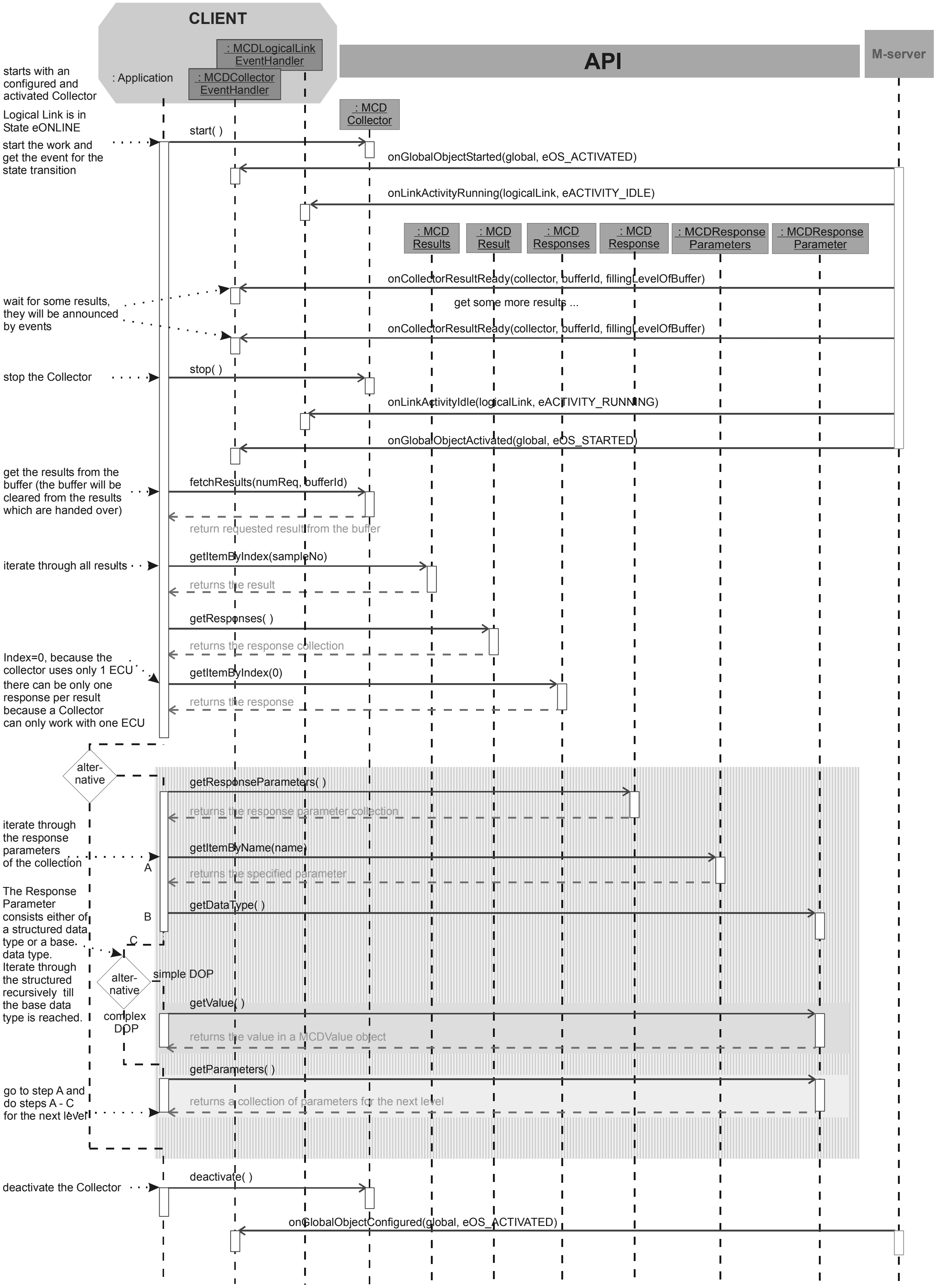

CLIENT
: MCDLogicalLink EventHandler
API
M-server
: Application
: MCDCollector EventHandler
starts with an configured and activated Collector
Logical Link is in State eONLINE
: MCD Collector
start the work and get the event for the state transition
start( )
onGlobalObjectStarted(global, eOS_ACTIVATED)
onLinkActivityRunning(logicalLink, eACTIVITY_IDLE)
: MCD Results
: MCD Result
: MCD Responses
: MCD Response
: MCDResponse Parameters
: MCDResponse Parameter
onCollectorResultReady(collector, bufferId, fillingLevelOfBuffer)
wait for some results, they will be announced by events
get some more results ...
onCollectorResultReady(collector, bufferId, fillingLevelOfBuffer)
stop the Collector
stop( )
onLinkActivityIdle(logicalLink, eACTIVITY_RUNNING)
onGlobalObjectActivated(global, eOS_STARTED)
get the results from the buffer (the buffer will be cleared from the results which are handed over)
fetchResults(numReq, bufferId)
return requested result from the buffer
iterate through all results
getItemByIndex(sampleNo)
returns the result
getResponses( )
returns the response collection
Index=0, because the collector uses only 1 ECU there can be only one response per result because a Collector can only work with one ECU
getItemByIndex(0)
returns the response
alter-native
getResponseParameters( )
returns the response parameter collection
iterate through the response parameters of the collection
getItemByName(name)
A
returns the specified parameter
The Response Parameter consists either of a structured data type or a base data type. Iterate through the structured recursively till the base data type is reached.
getDataType( )
B
C
alter-native
simple DOP
getValue( )
complex DOP
returns the value in a MCDValue object
getParameters( )
go to step A and do steps A - C for the next level
returns a collection of parameters for the next level
deactivate the Collector
deactivate( )
onGlobalObjectConfigured(global, eOS_ACTIVATED)

Figure 42 — Collector result

8.2.3 Measurement with a collector - polling for results

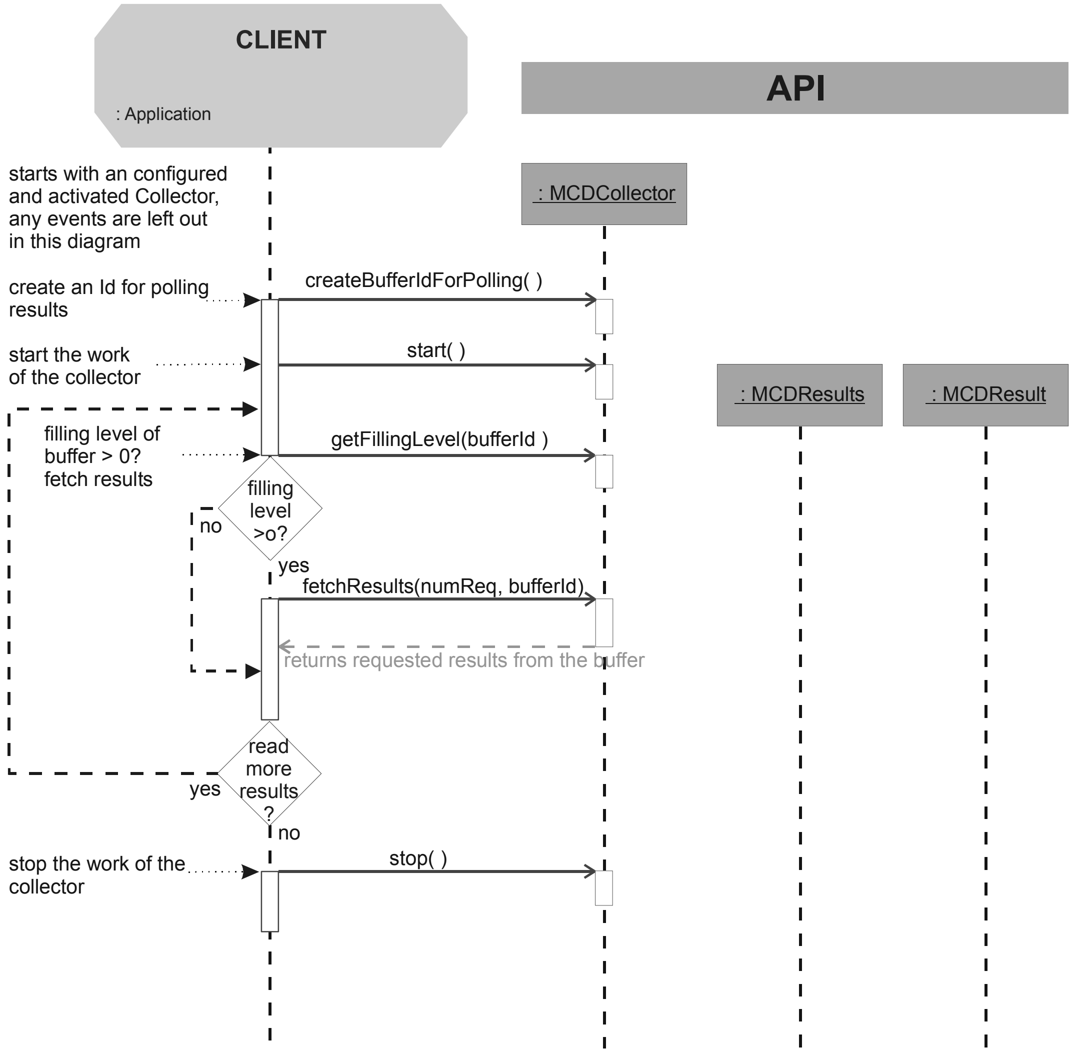

CLIENT
: Application
API
: MCDCollector
: MCDResults
: MCDResult
starts with an configured and activated Collector, any events are left out in this diagram
create an Id for polling results
createBufferIdForPolling( )
start the work of the collector
start( )
filling level of buffer > 0? fetch results
getFillingLevel(bufferId)
filling level >o?
no
yes
fetchResults(numReq, bufferId)
returns requested results from the buffer
read more results ?
yes
no
stop the work of the collector
stop( )

Figure 43 — Collector result polling

For the polling of the results of a Collector a BufferId is necessary, which can be fetched using the method `createBufferIdForPolling()`. The BufferId is necessary for the Thread identification inside the Client. This way it is made sure, that all results are available for the respective Threads without any other Thread being able to delete the results from the buffer.

After this the Collector is started. After starting the Thread begins to poll the Collector for results using its BufferId. In this case the Thread will ignore the Events, but nevertheless for each result the Event `onCollectorResultReady()` will be sent. The results may be fetched with `fetchResult()`. As soon as the Thread decides to take no more results from this Collector, he may stop the Collector by means of suspend. After this the Collector is in state Collector `eOS_ACTIVATED` again.

Result entries within the buffer may only be released by the MCD-server, if all registered Threads have fetched the results. Otherwise the buffer will be filled further on which might lead to an overwriting of the results.

The filling level of the collector result buffer can be asked with `MCDBuffer::getFillingLevel(error:MCDError):A_UINT32` and delivers the filling level independent from all threads which used this collector. Calling `MCDCollector::getFillingLevel (bufferID:A_UINT32):A_UINT32` delivers the filling level of the collector result buffer for the identified Thread.

9 Function block Diagnostics

9.1 Description of Terms

9.1.1 General

The key terms and definitions which apply in this part of ISO 22900 are given in Clause 3. This subclause describes the most important terms in more detail. For every term which is directly related to an ODX element, the corresponding ODX element name is given in parentheses.

9.1.2 Access Key

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. The single AccessKey elements are separated by dots. 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 23 — 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.

9.1.3 Functional Class (FUNCTIONAL-CLASS)

Functional classes are sets of diagnostic services. Each functional class is identified by a unique name A diagnostic service can be part of one or more functional classes. Functional classes are defined and authored in the ODX data pool of a project.

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

A Job is a sequence of diagnostic services and other jobs with a control flow. The control flow may be based on results (return values) of execution of services and jobs. Typical use cases for jobs are ECU (re-) programming, encryption like seed & key algorithms and gateway tests.

9.1.5 Physical Interface Link

A physical interface link is the physical connection between the VCI connector of a VCI and the interface connector. Technically speaking, the VCI connector and the interface connector are the connectors at the two opposite ends of the cable connecting a VCI with a vehicle. In the VCI, a VCI connector is driven by a VCI module (also called interface in the following subclauses). The VCI modules contained in a VCI and their properties are accessed by the D-server via the D-PDU API.

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 D-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.

9.1.6 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 D-server to the ECU in the vehicle.

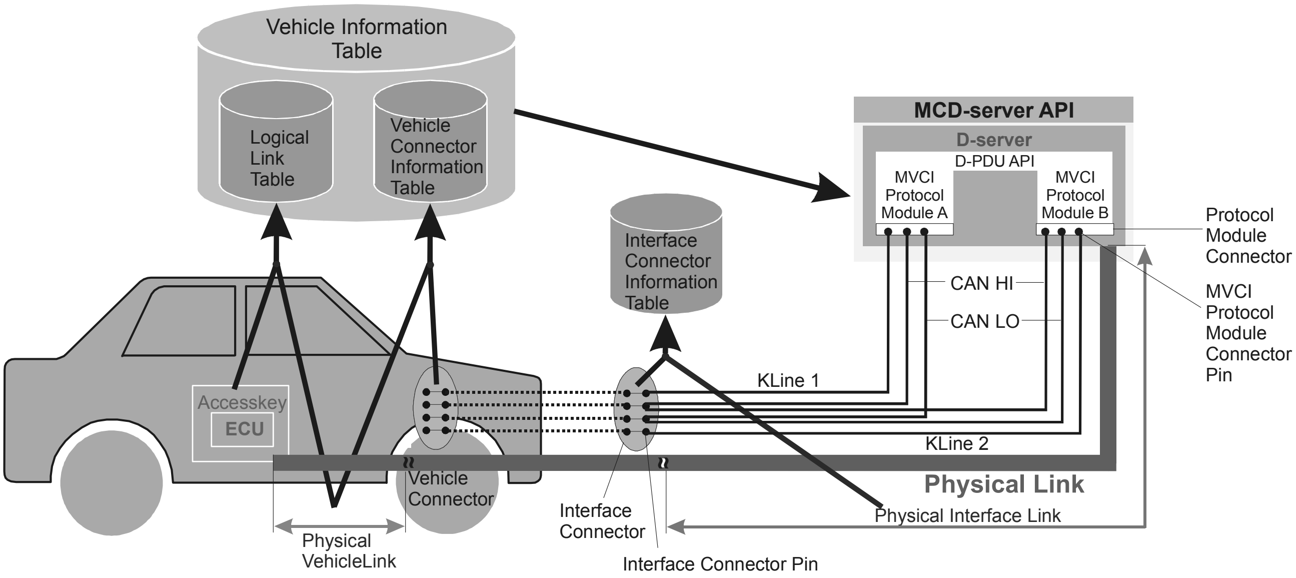


Figure 44 — Overall scheme between different links and tables in D

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

9.1.7 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.

9.2 Structuring of the function block Diagnostics

9.2.1 Separation in database and runtime side

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 the ASAM MCD 2 D 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 D-server. As soon as the project has been selected, all information is available. It does not have to exist within objects at this moment, but the option to access it shall 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 D-server should generate suitable shortnames for shortnames are not obtainable from ODX. These shortnames should have a prefix "#RtGen_" that uniquely classifies these shortnames as generated.

`MCDSystem::getSupportedODXVersions()` lists all possible ASAM MCD 2 D ODX versions which are supported by this D-server implementation.

The communication part contains all interfaces via which the communication with the ECU s takes place. The majority of the runtime objects is created on the basis of database objects and thus has 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 instanced multiply (multiple objects). The multiply instanced runtime objects of an identical database object may be parameterised differently. Singletons are `MCDJobAPI`, `MCDSystem`, `MCDProject` and `MCDEnumValue`.

The runtime objects may be executed (executable), have an internal status and return results.
Following the structure of the MCD-server API and the relations between the interfaces are described.

9.2.2 Relation between Vehicle Connector Information Table and Logical Link Table

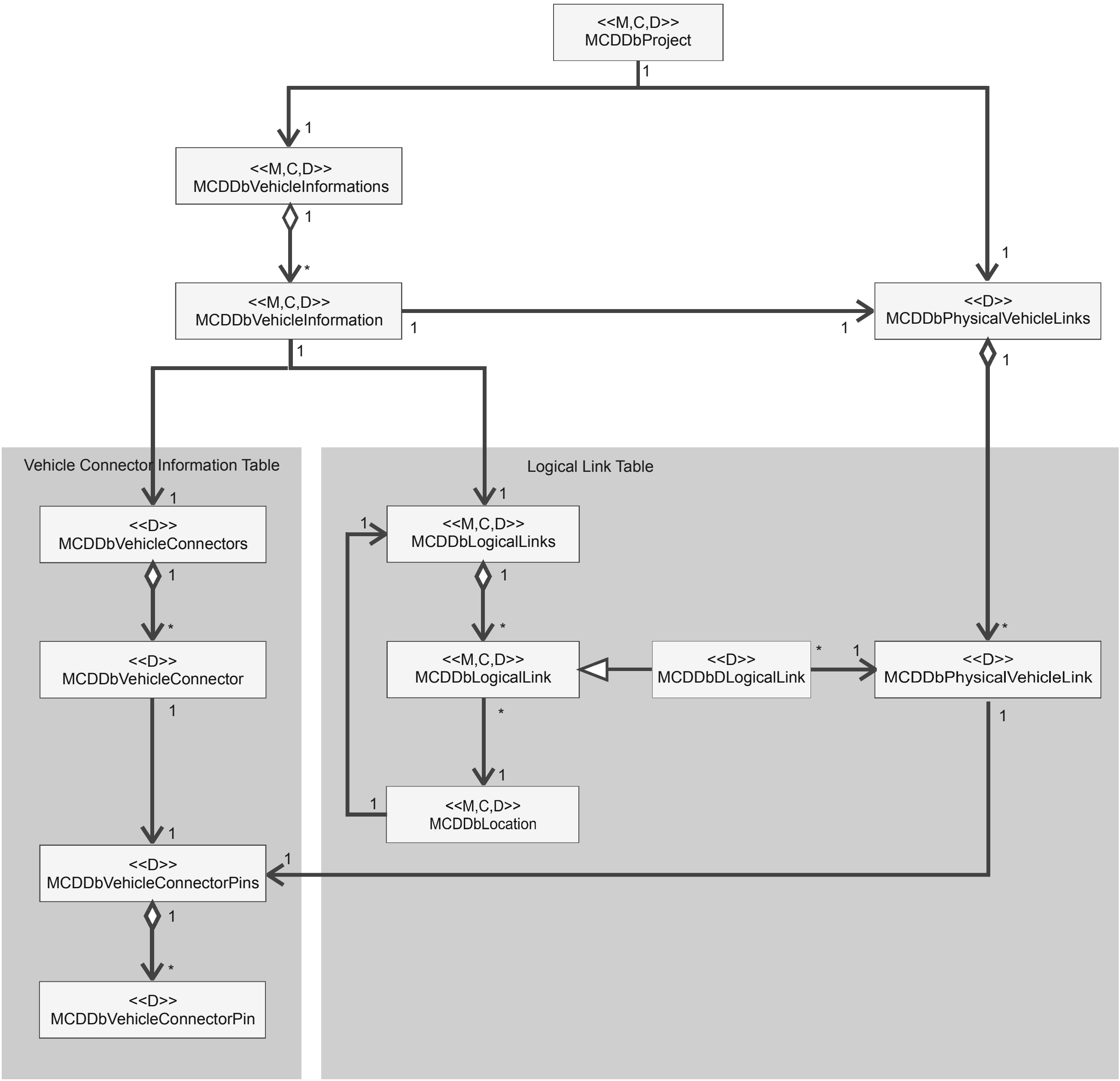


Figure 45 — DbProject and relation to Logical Link and Vehicle Information Table in D

Within this ERD the database relations between project and contained Vehicle Information for ECU's are shown. It is shown, which information may be listed in collections and in which multiplicity the respective objects are allowed to exist within each relation in the database.

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

To each Logical Link belongs exactly one Location and one Physical Vehicle Link, in accordance with the entry in the Logical Link Table. The Vehicle Connectors and the ConnectorPins in accordance with the Physical Vehicle Link can be listed. Within the Vehicle Information Table the VehicleConnectors with its 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.

9.2.3 Hierarchical model

Additionally to the interfaces within the MCD further interfaces exist within the function block Diagnostic.

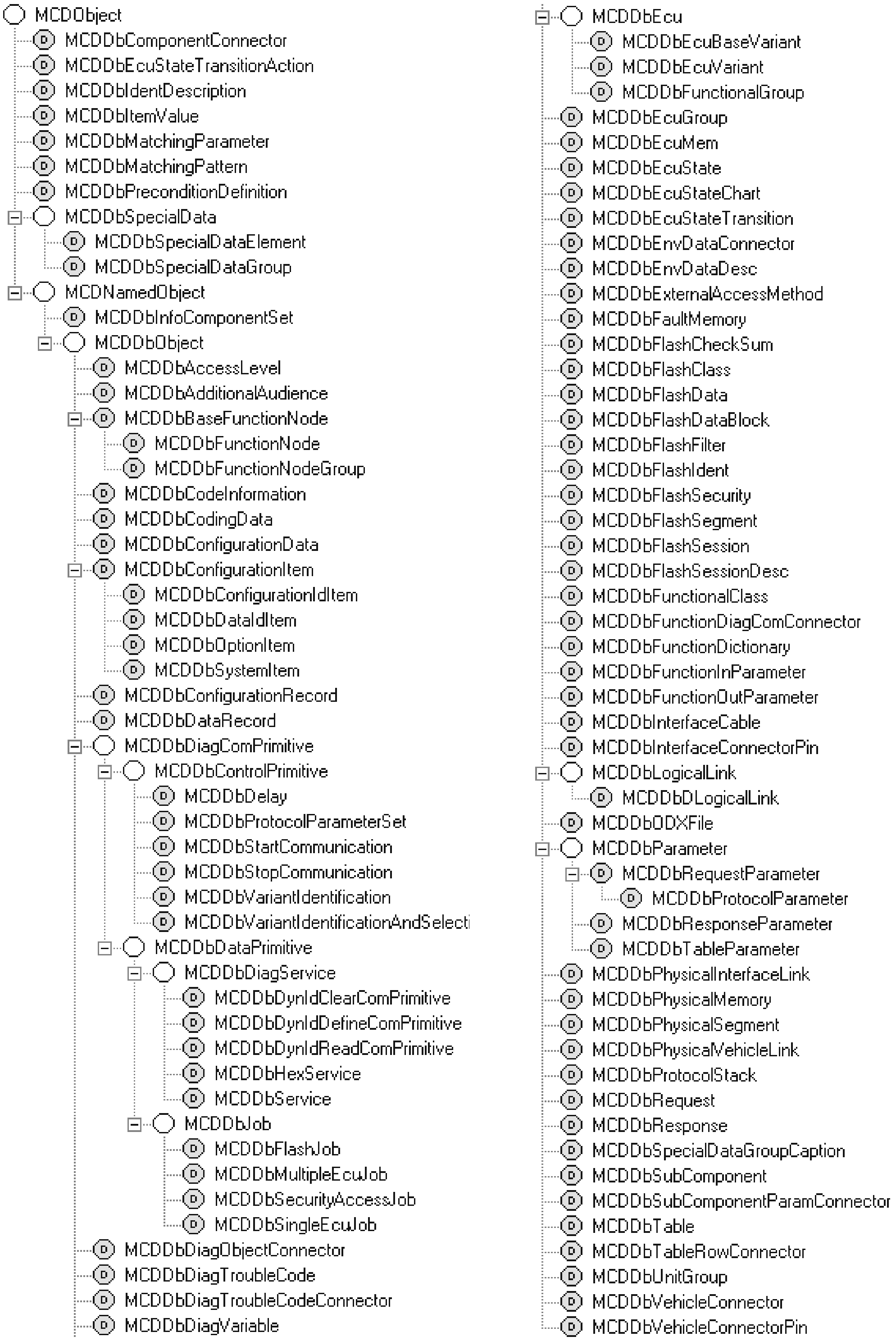


Figure 46 — Hierarchical structure of database in function block D

Within the database part templates for all DiagComPrimitives and their parameters, information for ECU (re-) programming and the information concerning the ECU s 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 template for the runtime communication primitives.

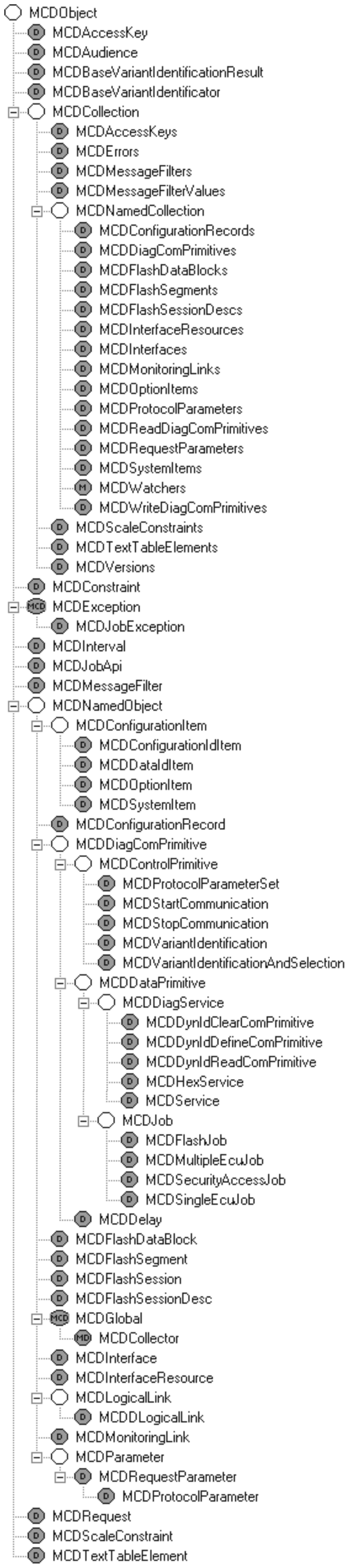


Figure 47 — Hierarchical structure of runtime part in function block D

On runtime side the communication actions are carried out by means of DiagComPrimitives, which also include the Services and all four kinds of jobs (FlashJob, SingleEcuJob, MultipleEcuJob and SecurityAccessJob). A specific feature of the Diagnostic function block is the use of RequestParameters as input for the DiagComPrimitives.

`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 as for example Services and Jobs. Jobs and Services are derived from DataPrimitives and thus they shall be handled alike.

9.2.4 Entity Relationship Diagrams

9.2.4.1 ERD DbLocation

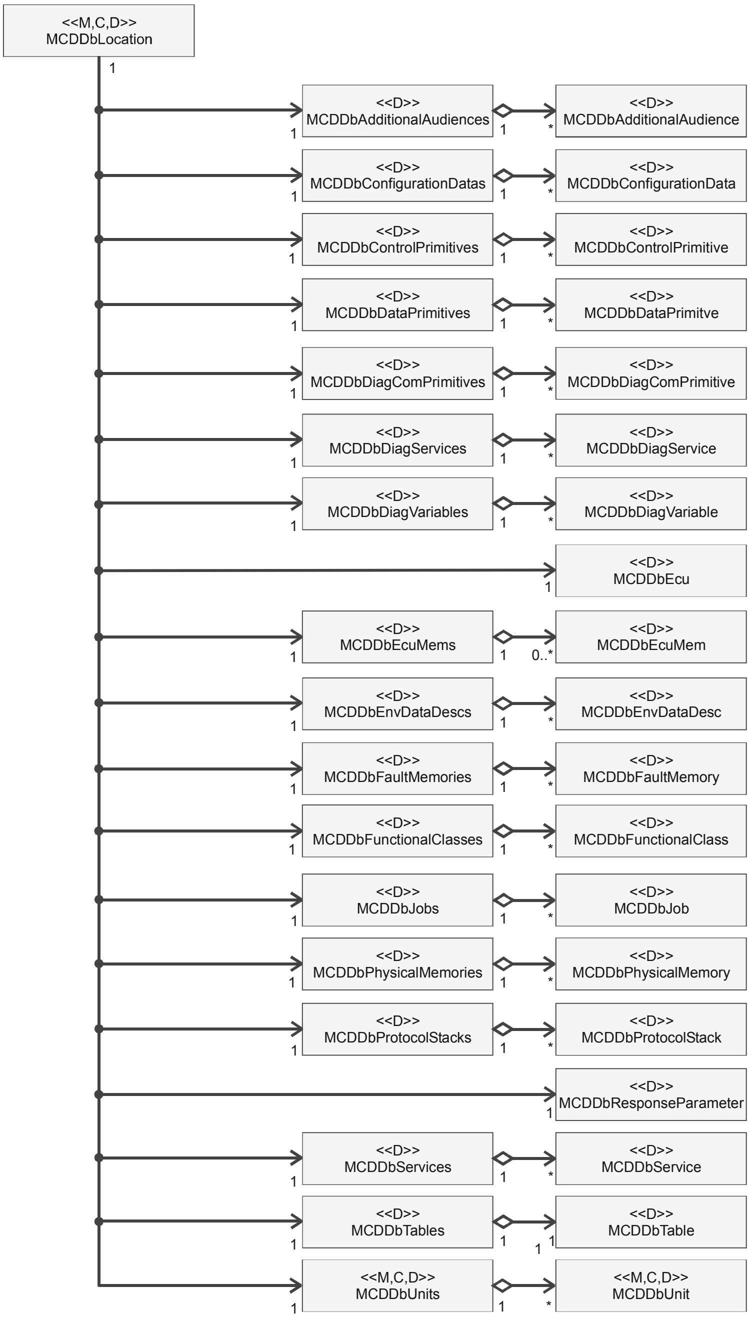


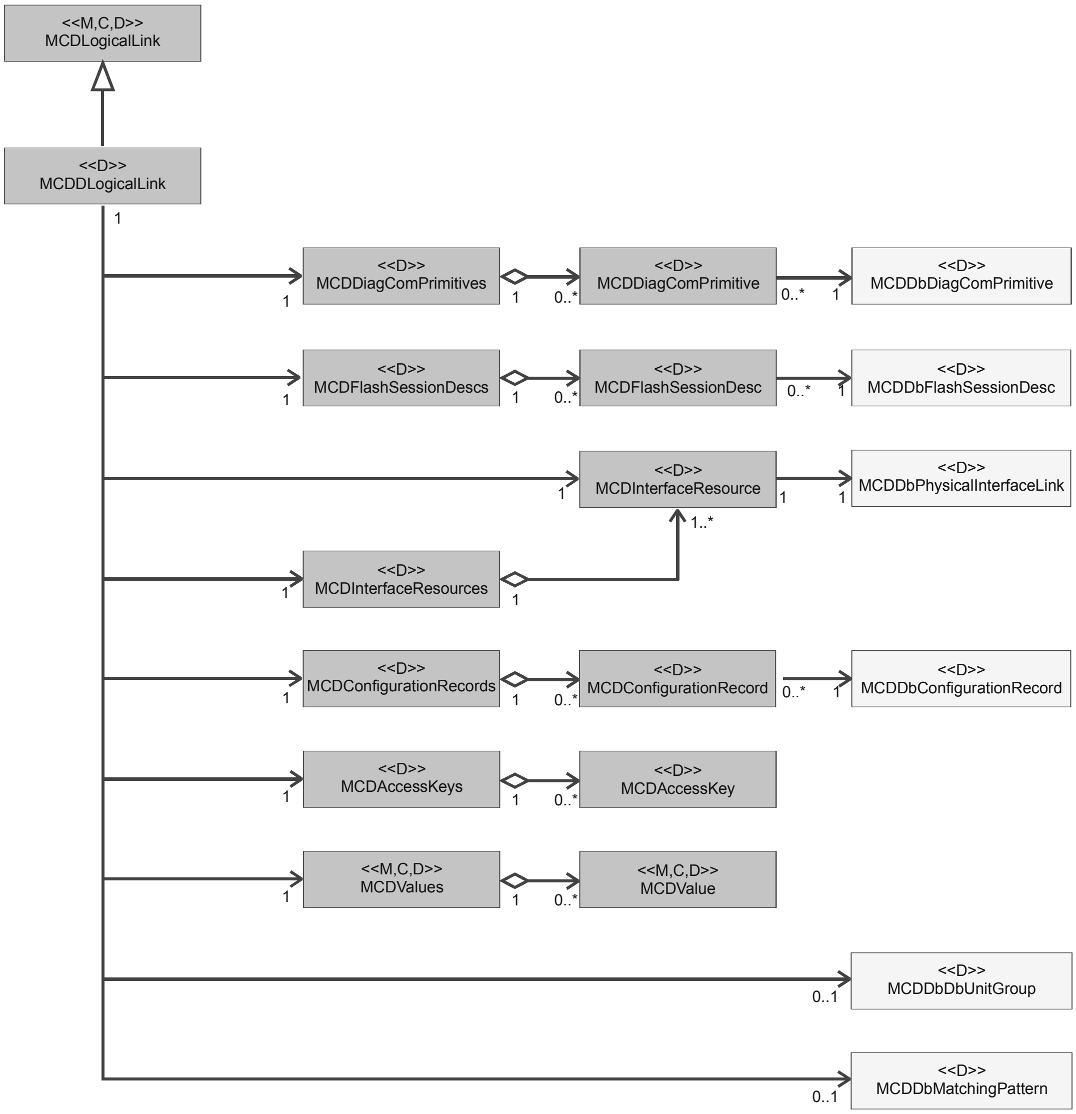
Figure 48 — Location association for D

These associations are created from the information out of ODX description files, whereat `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 a ECU. Every Collection contains 0..n objects of the same class or children of a common super class.

- DiagComPrimitives
- Services
- FunctionalClasses (aggregation of Services)
- Jobs
- FlashSessions and FlashClasses (aggregation of FlashSessions)

9.2.4.2 ERD Logical Link and associated MCD Objects



<<M,C,D>> MCDLogicalLink
<<D>> MCDDLogicalLink
1
<<D>> MCDDiagComPrimitives
1
1
0..*
<<D>> MCDDiagComPrimitive
0..*
1
<<D>> MCDDbDiagComPrimitive
<<D>> MCDFlashSessionDescs
1
1
0..*
<<D>> MCDFlashSessionDesc
0..*
1
<<D>> MCDDbFlashSessionDesc
<<D>> MCDInterfaceResource
1
1
1
<<D>> MCDDbPhysicalInterfaceLink
1..*
<<D>> MCDInterfaceResources
1
1
<<D>> MCDConfigurationRecords
1
1
0..*
<<D>> MCDConfigurationRecord
0..*
1
<<D>> MCDDbConfigurationRecord
<<D>> MCDAccessKeys
1
1
0..*
<<D>> MCDAccessKey
<<M,C,D>> MCDValues
1
1
0..*
<<M,C,D>> MCDValue
0..1
<<D>> MCDDbDbUnitGroup
0..1
<<D>> MCDDbMatchingPattern

Figure 49 — Logical Link and associated MCD Objects in function block D

The Entity Relationship Diagram in Figure 49 — Logical Link and associated MCD Objects in function block D shows the relations between a Logical Link and its associated objects. The most important entities are communication primitives (called DiagComPrimitives) which can be created at a Logical Link. Also shown are the relations between runtime and the database parts of these communication primitives.

For every Logical Link, 0..n different DiagComPrimitives can be created at the same time. Every Logical Link maintains its own collection of runtime DiagComPrimitives. This collection can contain multiple runtime instances of the same DbDiagComPrimitive. The creation mechanism for MCDDiagComPrimitives in the D-server assures that every MCDDiagComPrimitive in the same collection of DiagComPrimitives has a unique name. For this purpose, the D-server generates the ShortNames of the runtime DiagComPrimitives in accordance with the following pattern:

```
#RtGen_<MCDDbDiagComPrimitiveShortName>_<UniqueNumber>
```

By allowing multiple instances of every possible DiagComPrimitive including MCDStartCommunication, MCDStopCommunication, MCDVariantIdentification and MCDVariantIdentificationAndSelection, the different client applications using the same Logical Link can manipulate the local protocol parameters attached to “their” instance of a DiagComPrimitive without interfering with changes of a different client application.

On the database side, MCDDbDiagComPrimitive objects available at a MCDDbLocation include all MCDDbDataPrimitives and all MCDDbControlPrimitives that are relevant for that location. Therefore, the D-server generated database objects of the Control Primitives in the list below are contained in the list of all available MCDDbDiagComPrimitives of a MCDDbLocation:

- START_COMMUNICATION,
- STOP_COMMUNICATION,
- VARIANT_IDENTIFICATION and
- VARIANT_IDENTIFICATION_AND_SELECTION

NOTE START_COMMUNICATION and STOP_COMMUNICATION might be given in the ODX data. In this case, the database template of these communication primitives shows the request and the responses as defined in the ODX data.

In addition, the list of data primitives (and as a result, also the list of DiagComPrimitives) contains a MCDDbDiagComPrimitive representing the generic HEXSERVICE (MCDDbHexService).

Besides the DiagComPrimitives other objects are related to a Logical Link. These are as follows:

- MCDConfigurationRecords (through method getConfigurationRecords()), which provide the entry point to the ECU configuration functionality.
- MCDValues (through method getDefinableDynIds()), which represent the values of the DefinableDynIds available at the Logical Link.
- MCDFlashSessionDescs (through method getFlashSessionDescs()), which provide the entry point to the ECU re-programming functionality.
- MCDAccessKeys (through methods getIdentifiedVariantAccessKeys() and getSelectedVariantAccessKeys()), which show the identified and selected AccessKeys of the Variant, the Logical Link is based on at database side.
- MCDInterfaceResource (through method getInterfaceResource()), which refer to the physical basis of the LogicalLink.
- MCDDbMatchingPattern (through method getMatchedDbEcuVariantPattern()), which give access to the database part of the mechanism of identifying the connected ECU.

9.2.4.3 ERD Request and Response Parameter associations

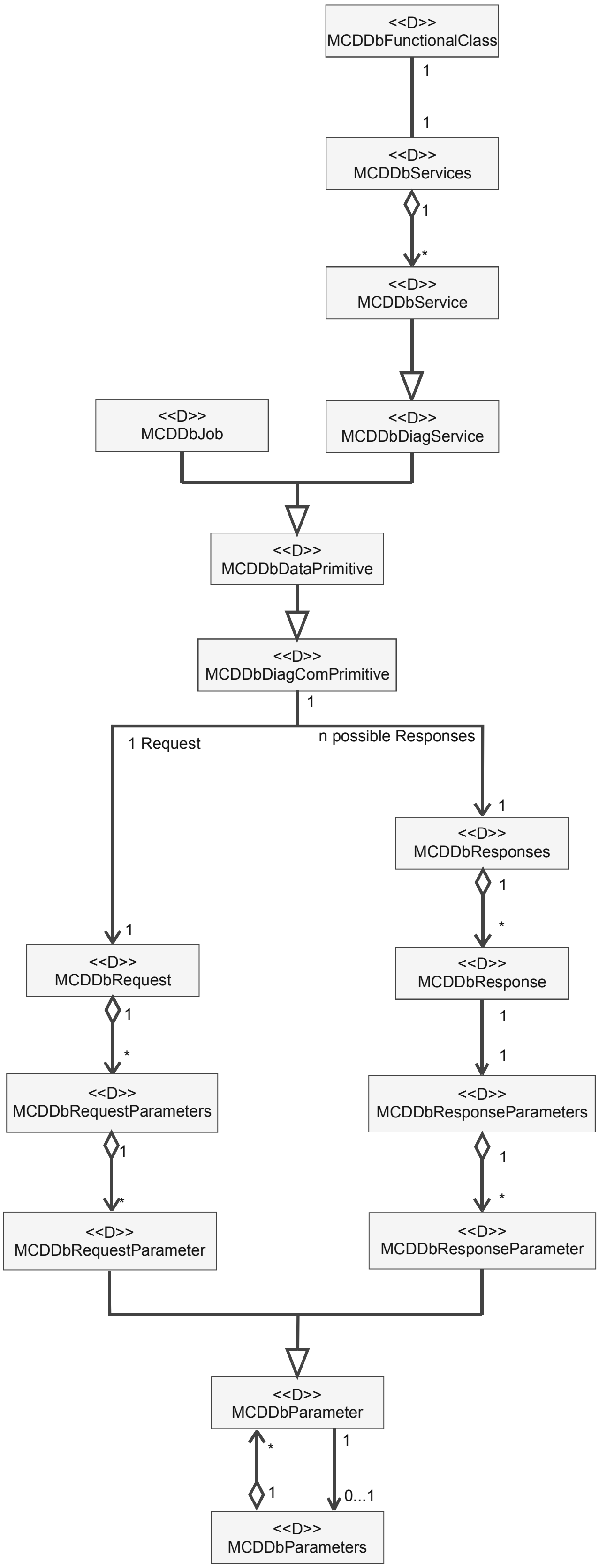


Figure 50 — RequestParameter associations

Within this ERD the relations concerning the kinds of parameters of the object model are shown. The database side is considered in detail. Within the upper part it is shown, that MCDDbFunctionalClass contains a collection of Services which belong to the MCDDbDataComPrimitive.

Each DiagComPrimitive contains a possibly empty set of RequestParameters as well as a possibly empty set of predefined Responses. Every Response contains a possibly empty set of ResponseParameters, which describe the general structure of the Response. This is illustrated in figure Response Structure of DTC (Figure 123 — Result structure DTC).

A RequestParameter and a ResponseParameter may reference a Simple DOP or a Complex DOP. In this way nested structures can be built.

A parameter with data type eTEXTTABLE is used to map a A_UNICODE2STRING (physical value) to an internal value and vice versa. In ODX, the COMPU-METHOD (conversion function) of CATEGORY TEXTTABLE is used in the respective DOP of that parameter. The MCDValue of the parameter is of data type eA_UNICODE2STRING. The D-server shall consider two cases for the conversion from the physical to the internal value:

— The defined interval (COMPU-SCALE) consists of exactly one internal value, e.g. UPPER-LIMIT and LOWER-LIMIT are equal. In this case the internal value is used - irrespective of the COMPU-INVERSE-VALUE.

— The defined interval consists of at least two internal values - the COMPU-INVERSE-VALUE is used. So, the COMPU-INVERSE-VALUE shall be defined in ODX.

Table 24 — Example TextTable

Internal	physical
[0,5)	"OFF"
[5,9)	"ON"
[9,∞)	"OFF"

Table 25 — Example TextTable A and B (Default in grey background)

Example A		Example B	
coded	physical	coded	physical
0	"Unknown"	0	"Unknown"
1	"ON"	1	"ON"
2	"OFF"	2	"OFF"
5	"ON"	3	"Unknown"
Default	“Unknown”	5	“ON”
		Default	“Unknown”

The collection of MCDTextTableElements at MCDDbRequestParameter or MCDDbResponseParameter contains all elements except of the default element of the text table in ODX. The reason is that the default element of a text table is only used when converting from coded to physical value as an exceptional case. For example A, the collection would contain text table elements for (UNKNOWN : 0), (ON : 1), (OFF : 2) and (ON : 5). For example B, the collection of MCDTextTableElements would contain elements for (UNKNOWN : 0), (ON : 1), (OFF : 2), (UNKNOWN : 3) and (ON : 5). The default element is only used by the method MCDDbParameter::getDefaultValue().

In the case the physical value (MCDResponseParameter::getValue()) returned at a response parameter having a text table conversion cannot be found amongst the LongNames of the MCDTextTableElements obtainable at the same parameter, this value must have been the default value of the text table conversion in ODX. In this case, the method MCDResponseParameter::getTextTableElement() throws an exception of type MCDProgramViolationException with error code eRT_ELEMENT_NOT_AVAILABLE.

In ODX, the validity of the internal value can be restricted to a given interval via internal constraints. The physical value can be restricted via physical constraints. The value definition of both kinds of constraints can be obtained by using the methods getInternalConstraint() and getPhysicalConstraint() at MCDDbParameter. With the type of constraint it is defined how the MCDInterval and its limits shall be interpreted. Values regarding the internal constraint definition shall be interpreted by the internal data type. The values regarding the physical constraint definition shall be interpreted by the physical data type.

The implicit valid range of a value is defined in ODX by the integral data type. In the case of internal constraint definition the implicit valid range is additionally restricted by the encoding and the size of the parameter, which is defined either by the number of 1 in a condensed bit mask or the bit length. The explicit valid range is defined by the limits of the outer constraint (internal- or physical constraint), which is always a VALID interval, minus all inner scale constraints (SCALE-CONSTR in ODX) where the attribute VALIDITY is not equal to VALID. The attribute VALIDITY of each constraint definition can take the values VALID, NOT-VALID, NOTDEFINED or NOT-AVAILABLE. The interval of a constraint is defined in ODX by the UPPER-LIMIT and LOWER-LIMIT attribute. If UPPER-LIMIT and/or LOWER-LIMIT are missing, the explicit valid range is not restricted in that direction. In general, the valid range is defined by the intersection of the implicit and the explicit valid range.

If the corresponding interval type is set to eLIMIT_INFINITE, the lower and upper limit are defined by the lowest and highest possible value of the integral data type.

For example the method getUpperLimit() will return the maximum positive value of the corresponding data type if getUpperLimitIntervalType() returns MCDInterval::eLIMIT_INFINITE.

In case of a request, the physical values given by the user or pre-defined in ODX shall be checked against the physical constraint by the D-server. If the check is successful the physical values will be converted to the corresponding internal values. At least (after applying the computational method) the D-server will check the internal values against the internal constraints. If successful, the data can be coded into the request message.

If the value violates the physical constraints in case of a request the computational method will not be applied. MCDRequestParameter::setValue() will throw a MCDParameterizationException with error code ePAR_INVALID_VALUE. In this state the methods getValue() and getCodedValue() will deliver a MCDValue that is not initialized. Thus, calling the method MCDValue::isValid() for that MCDValue returns false.

In case of a response, the internal values extracted from the ECU response message and interpreted by the internal data type shall be checked against the internal constraint by the D-server. If the check is successful, the internal values will be converted to the corresponding physical values by application of the computational method. At last the D-server will check the physical values against the physical constraints.

If the value violates the internal constraints in case of a response the computational method will not be applied. In this state MCDParameter::getValue() will throw an MCDProgramViolationException with error code eRT_ELEMENT_NOT_AVAILABLE and getCodedValue() returns the internal value that violates the internal constraints.

The MCDRangeInfo will be set in accordance with the validity of the value (request/response).

The valid range of a constraint is defined by the intersection of the implicit and the explicit valid range. The implicit valid range of the internal constraint is defined by the coded data type restricted by the encoding and the size of the parameter, which is defined either by the number of 1 in a condensed BIT-MASK or the BIT-LENGTH. The implicit valid range of the physical constraint is defined by the physical data type only. The explicit valid range is defined by UPPER-LIMIT and LOWER-LIMIT of the constraint minus all SCALE-

CONSTR with VALIDITY != VALID. If UPPER-LIMIT and/or LOWER-LIMIT are missing in the ODX data, the explicit valid range is not restricted in that direction.

The validity of a value is determined by the following multiple step process:

— If the value is outside of the intersection of implicit and explicit valid range, ignoring any SCALE-CONSTR definition, the method `MCDParameter::getValueRangeInfo()` delivers `eVALUE_NOT_VALID`.

— If the value is inside of the valid range (intersection of implicit and explicit valid range with respect to the SCALE-CONSTR definitions), the method `MCDParameter::getValueRangeInfo()` delivers `eVALUE_VALID`.

— If the value is inside any SCALE-CONSTR value range the method `MCDParameter::getValueRangeInfo()` returns the validity defined by the SCALE-CONSTR in ODX.

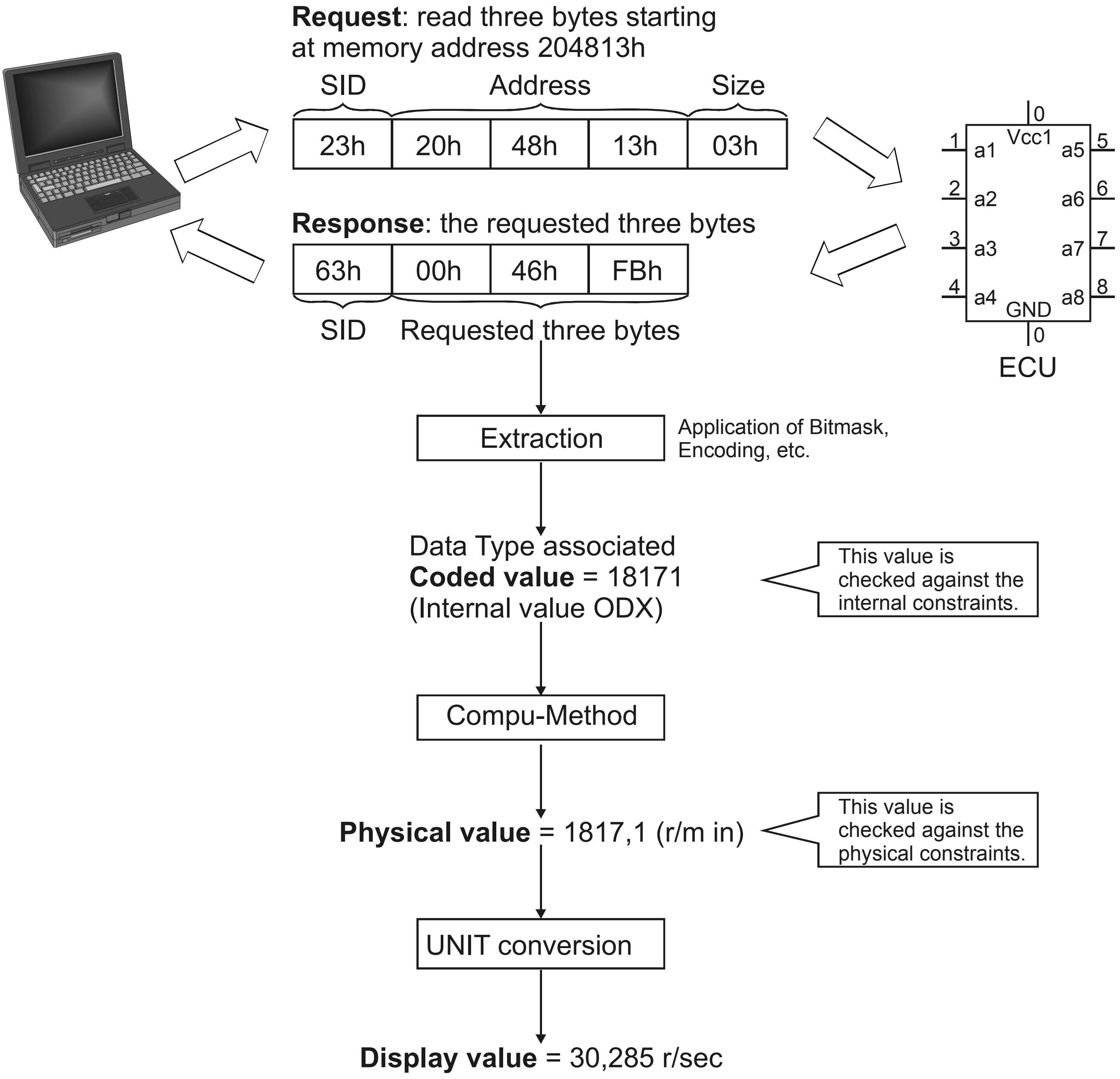

Request: read three bytes starting at memory address 204813h
SID
Address
Size
23h
20h
48h
13h
03h
Response: the requested three bytes
63h
00h
46h
FBh
SID
Requested three bytes
0
Vcc1
1
a1
a5
5
2
a2
a6
6
3
a3
a7
7
4
a4
a8
8
GND
0
ECU
Extraction
Application of Bitmask, Encoding, etc.
Data Type associated
Coded value = 18171
(Internal value ODX)
This value is checked against the internal constraints.
Compu-Method
Physical value = 1817,1 (r/m in)
This value is checked against the physical constraints.
UNIT conversion
Display value = 30,285 r/sec

Figure 51 — Application of Constraints during Data Extraction Process

For the examples following, this ODX specification of DIAG-CODED-TYPE is assumed:

```
<DIAG-CODED-TYPE xsi:type="STANDARD-LENGTH-TYPE" BASE-DATA-TYPE="A_INT32"
  CONDENSED="true" BASE-TYPE-ENCODING="SM">
  <BIT-LENGTH>12</BIT-LENGTH>
  <BIT-MASK>0E07</BIT-MASK>
</DIAG-CODED-TYPE>
```

Within the message the parameter has a length of 12 bit, but the condensed bitmask restricts the length of the signed integer value to actually 6 bit. Because it is encoded as SM (see ODX specification[11] for details of encodings), the implicit valid range is [-31, +31].

In the first example the explicit valid range is (-∞, +∞) and therefore the intersection between the explicit and implicit valid ranges is [-31, +31].

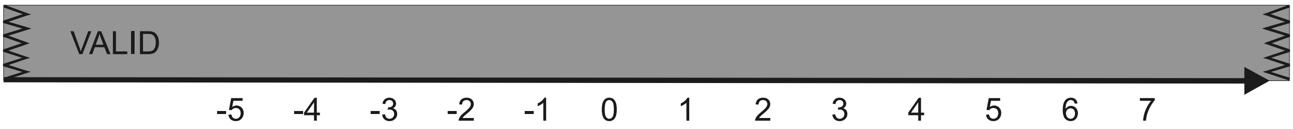


Figure 52 — Use of constraints: No explicit restrictions (no INTERNAL-CONSTR)

In the second example the internal constraint is limited by the lower limit value -3 and the upper limit value 5. Both limits are defined with the interval type CLOSED.

```
<INTERNAL-CONSTR>
    <LOWER-LIMIT INTERVAL-TYPE = "CLOSED">-3</LOWER-LIMIT>
    <UPPER-LIMIT INTERVAL-TYPE = "CLOSED">5</UPPER-LIMIT>
</INTERNAL-CONSTR>
```

The explicit valid range is [-3, +5] and, therefore, the intersection between the explicit and implicit valid ranges is [-3, +5].

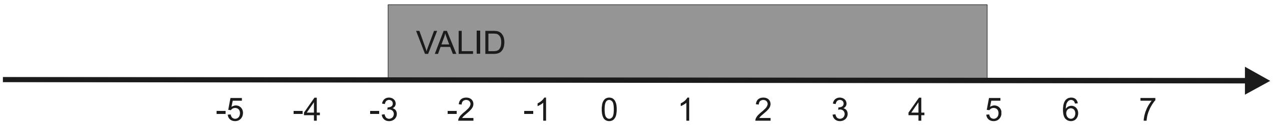


Figure 53 — Use of constraints: One valid interval

In the next example no limits for the internal constraint are defined but two inner scale constraints.

```
<INTERNAL-CONSTR>
    <SCALE-CONSTRS>
        <SCALE-CONSTR VALIDITY = "NOT-DEFINED">
            <LOWER-LIMIT INTERVAL-TYPE = "OPEN">4</LOWER-LIMIT>
            <UPPER-LIMIT INTERVAL-TYPE = "CLOSED">5</UPPER-LIMIT>
        </SCALE-CONSTR>
        <SCALE-CONSTR VALIDITY = "NOT-VALID">
            <LOWER-LIMIT INTERVAL-TYPE = "OPEN">5</LOWER-LIMIT>
            <UPPER-LIMIT INTERVAL-TYPE = "CLOSED">6</UPPER-LIMIT>
        </SCALE-CONSTR>
    </SCALE-CONSTRS>
</INTERNAL-CONSTR>
```

With no limits for the INTERNAL-CONSTR the explicit validity of the internal value is defined in the interval (-∞, +∞). Within this outer interval the interval (4, 5] is declared via SCALE-CONSTRS as NON-DEFINED and the interval (5, 6] is declared as NOT-VALID, which is illustrated in the following figure.

The explicit valid ranges are (-∞, +4] and (+6, +∞), therefore, the intersections between the explicit and implicit valid ranges are [-31, +4] and (+6, +31]. Because the internal type is an integer type, the second range is equivalent to [+7, +31].

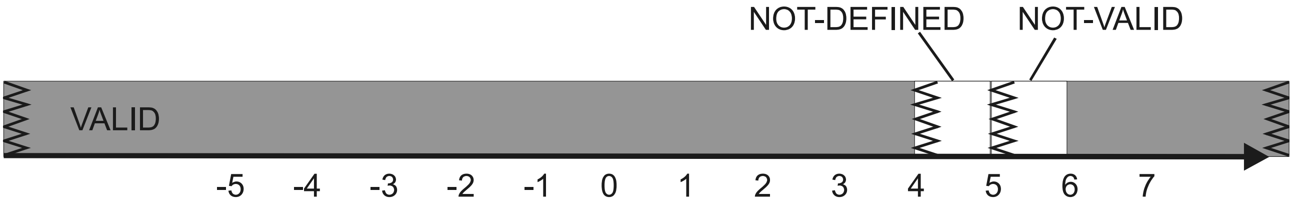


Figure 54 — Use of constraints: No INTERNAL-CONSTR limits

The next and the last example regarding internal constraints is defined by the following ODX data.

```
<INTERNAL-CONSTR>
    <LOWER-LIMIT INTERVAL-TYPE = "CLOSED">0</LOWER-LIMIT>
    <UPPER-LIMIT INTERVAL-TYPE = "INFINITE"/>
    <SCALE-CONSTRS>
        <SCALE-CONSTR VALIDITY = "NOT-DEFINED">
            <LOWER-LIMIT INTERVAL-TYPE = "OPEN">4</LOWER-LIMIT>
            <UPPER-LIMIT INTERVAL-TYPE = "CLOSED">5</UPPER-LIMIT>
        </SCALE-CONSTR>
        <SCALE-CONSTR VALIDITY = "NOT-AVAILABLE">
            <LOWER-LIMIT INTERVAL-TYPE = "OPEN">5</LOWER-LIMIT>
            <UPPER-LIMIT INTERVAL-TYPE = "CLOSED">6</UPPER-LIMIT>
        </SCALE-CONSTR>
    </SCALE-CONSTRS>
</INTERNAL-CONSTR>
```

This ODX data defines the explicit validity of the internal value in the interval [0, +∞) by defining limits for the INTERNAL-CONSTR. Within this outer interval the interval (4, 5] is declared via SCALE-CONSTRS as NON-DEFINED and the interval (5, 6] is declared as NOT-AVAILABLE, which is illustrated in the following figure.

The explicit valid ranges are [0, +4] and (+6, +∞), therefore, the intersections between the explicit and implicit valid ranges are [0, +4] and (+6, +31]. Because the internal type is an integer type, the second range is equivalent to [+7, +31].

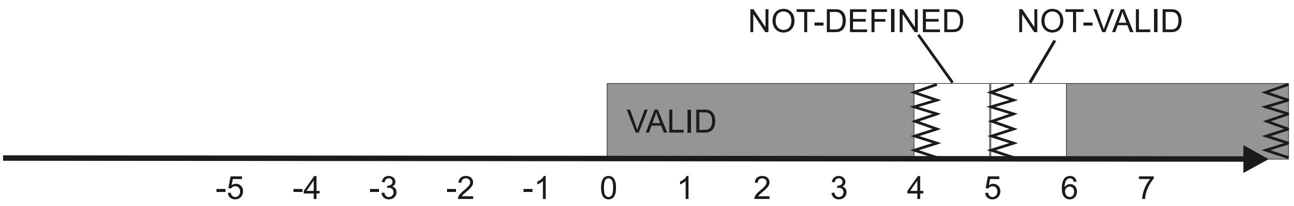


Figure 55 — Use of constraints: Clipping

In a similar way, it is possible to specify constraints for the physical value if the physical type is a numerical type. In this case, the implicit valid range is not restricted by the size or encoding of the internal data type but by the physical data type.

General D-server behaviour for CODED-CONST and PHYS-CONST Parameters:

As CODED-CONST and PHYS-CONST parameters are parameters like any other in ODX, these parameters are also available at the D-server API. That is, these parameters are contained in the collections delivered by `getRequestParameters()` and `getResponseParameters()`. However, it is impossible for a client application or job to change the value of a constant parameter.

Interval information for CODED-CONST and PHYS-CONST Parameters:

CODED-CONST:
The D-server generates a `MCDInterval` object where the return value of the methods `MCDInterval::getLowerLimit()`, `MCDInterval::getUpperLimit()`, and `MCDDbParameter::getDefaultValue()` is the same value, namely the physical value of this constant parameter.

The methods `MCDInterval:: getLowerLimitIntervalType()` and `MCDInterval::getUpperLimitIntervalType ()` both return `MCDLimitType::eLIMIT_CLOSED`.

In case of a CODED-CONST parameter, the methods `MCDInterval::getLowerLimit()` and `MCDInterval::getUpperLimit()` both return the same coded value – the constant coded value.

PHYS-CONST:
In case of a PHYS-CONST parameter, a `MCDInterval` object is returned which corresponds to the information in the ODX data.

The method `getCodedValue()` always delivers an object of type `MCDValue` containing the coded value of the `MCDResponseParameter` object. The types used to store a coded value in a `MCDValue` object are the appropriate coded ODX types.

NRC-CONST
Parameters of type CODED-CONST or PHYS-CONST possess only one possible value and may appear at both request and response parameters. In contrast to that, parameters of type NRC-CONST consist of a collection of possible CODED-CONST values and may only appear at response parameters. Due to that, parameters of type NRC-CONST behave differently from parameters of type CODED-CONST or PHYS-CONST and throw the exception `MCDProgramViolationException` with error code `eRT_VALUE_NOT_AVAILABLE` for the following methods:

- `MCDDbParameter::getDefaultValue()`
- `MCDDbParameter::getCodedDefaultValue()`
- `MCDDbParameter::getInternalConstraint()`
- `MCDDbParameter::getPhysicalConstraint()`

Behaviour of MCDResponseParameter::getCodedValue():

— In case a MCDResponseParameter contains a valid coded value, a copy of the corresponding MCDValue object containing the server-internal coded value is returned.

— In case a MCDResponseParameter does not contain a valid coded value and therefore no coded type, MCDResonseParameter::getCodedValue() returns a MCDValue object of type A_BITFIELD which is initialized with the bits representing the corresponding parameter in the response PDU.

— In case the parameter is of type CODED-CONST, the method getCodedValue() returns the constant coded value as defined in ODX. For coded const parameters, this value is mandatory.

— In case the parameter is of type PHYS-CONST, the method getCodedValue() returns an object of type MCDValue which is initialised with the coded value calculated from the constant physical value.

— In case the parameter is of type RESERVED, the method getCodedValue() returns the coded value as extracted from the response PDU. If no coded value and therefore no coded type is defined in ODX, getCodedValue() returns a MCDValue object of type A_BITFIELD which is initialized with the bits representing the corresponding parameter in the response PDU. Any coded value defined for a reserved response parameter in ODX is ignored.

— For complex parameters, the method getCodedValue() returns the same values as defined for the method MCDParameter::getValue().

Behaviour of MCDRequestParameter::getCodedValue():

The method getCodedValue() always delivers an object of type MCDValue containing the coded value of the MCDRequestParameter object. The types used to store a coded value in a MCDValue object are the appropriate coded ODX types.

— In case a MCDRequestParameter contains a valid coded value (set via setValue() or setCodedValue()), a copy of the corresponding server-internal coded value is returned in form of a MCDValue object.

— In case setValue() or setCodedValue() have not been called before but a physical default value is defined in ODX, an object of type MCDValue is returned which is initialised with the coded value calculated from the physical default value.

— In case the parameter is of type CODED-CONST, the method getCodedValue() returns the constant coded value as defined in ODX. For coded const parameters, this value is mandatory.

— In case the parameter is of type PHYS-CONST, the method getCodedValue() returns an object of type MCDValue which is initialised with the coded value calculated from the constant physical value.

— In case the parameter is of type RESERVED, the method getCodedValue() returns the coded value as defined in ODX. If no coded value is defined in ODX, getCodedValue() returns a correctly typed MCDValue object (coded ODX value type), which is initialized with the value which would be used by the server when sending the request. This means that the server is not to alter the coded value after it has been returned to a client via getCodedValue() once.

— In case setValue() or setCodedValue() have not been called before and no physical default value is defined in ODX, the method getCodedValue() returns a correctly typed MCDValue object (coded ODX value type). The internal state of this MCDValue is “uninitialized” and its value is “undefined”. That is, calling a getXXX()-Method at this MCDValue (where XXX represents the type of this MCDValue) will result in an exception being thrown. The type of this exception is MCDProgramViolationException with error code eRT_VALUE_NOT_INITIALIZED.

— For complex parameters, the method `getCodedValue()` returns the same values as defined for the method `MCDParameter::getValue()`.

Behaviour of MCDRequestParameter::setCodedValue():

— If `MCDRequestParameter::setCodedValue(MCDValue)` is called, the coded value passed is used to alter the server-internal PDU accordingly. Also the physical value of the affected parameter shall be updated (backward conversion), if possible.

— If it is not possible to perform backwards conversion of the coded value to a corresponding physical value (e.g. an appropriate conversion formula is missing), the range information of that parameter is set to `eVALUE_CODED_TO_PHYSICAL_FAILED`. Thereafter, the physical value of that request parameter with range information set to e`VALUE_CODED_TO_PHYSICAL_FAILED` is considered invalid. However, it is still possible to read the coded value via `MCDRequestParameter::getCodedValue()`.

Behaviour of setting values for CODED-CONST and PHYS-CONST Parameters:

— Parameters of type CODED-CONST or PHYS-CONST can not be changed by using the method MCDRequestParameter::setValue(). So, calling method MCDRequestParameter::setValue() leads to an MCDParameterizationException with error code ePAR_INVALID_VALUE.

— The same applies for the method MCDResponseParameter::setValue(), which can be called by a job to fill the response parameter structure. In case a job tries to overwrite a CONST value using this method, a MCDParameterizationException with error code ePAR_INVALID_VALUE will be thrown.

— Reponses returned by an ECU that don't match values for CONST values, similarly to wrong values for other types of parameters in ODX, should be marked as erroneous. This is achieved by a flag which can be queried by hasError() at various classes, e.g. MCDResult::hasError(), MCDResponse::hasError(), and MCDResponseParameter::hasError(). The error code in this case is eRT_VALUE_OUT_OF_RANGE.

9.2.4.4 ERD Jobs

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

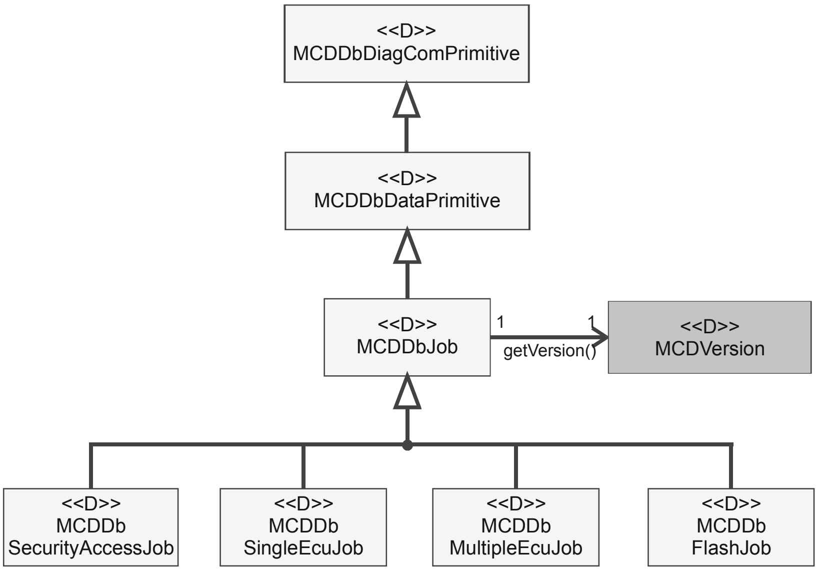


Figure 56 — Job associations

9.3 System Properties

A global mechanism to handle system properties is provided by the methods `getPropertyNames()`, `getProperty(A_ASCIISTRING propertyName)`, `setProperty(A_ASCIISTRING propertyName, MCDValue value)` and `resetProperty(A_ASCIISTRING propertyName)` at the interface `MCDSystem`. The methods allow retrieval and modification of all system properties defined within the MCD-server. System properties are separated into *mandatory* and *optional* properties. For details on currently defined properties, please refer to Table I.1 — System properties "System Properties" in Annex I.

In addition to the properties predefined in the MCD-3 specification (mandatory and optional), every vendor or OEM is free to define his own additional server properties. To avoid namespace conflicts, the shortnames of these properties shall be prefixed in accordance with Sun's rules for java package names, e.g. starting with com.company.<PROPERTY_NAME>.

For all properties defined as mandatory or optional in the MCD-3 specification and for all properties additionally defined by a vendor or OEM, the following information shall be included (either in the MCD-3 specification or in the corresponding server's documentation):

- Name of the property
- Type of the value
- Valid values
- Default value
- Description
- Mandatory / Optional
- Precondition

9.4 Diagnostic DiagComPrimitives and Services

9.4.1 Diagnostic DiagComPrimitives and States

9.4.1.1 Diagnostic DiagComPrimitives

These are the types of DiagComPrimitives declared in the API:

`MCDHexService`	performs a diagnostic hexservice (low level service)
`MCDService`	performs a diagnostic service
`MCDProtocolParameterSet`	provides the set of protocol parameters defined for the location
`MCDStartCommunication`	performs protocol specific initialization
`MCDStopCommunication`	performs protocol specific termination
`MCDVariantIdentification`	performs a variant identification
`MCDVariantIdentificationAndSelection`	performs a variant identification and selects the identified variant
`MCDSingleEcuJob`	performs a job on a single ECU
`MCDMultipleEcuJob`	performs a job on multiple ECUs
`MCDFlashJob`	performs a Flash Job

`MCDSecurityAccessJob` allows to call external libraries, e.g. via the JNI interface, to perform calculations for security access

`MCDDynIdDefineComPrimitive` Define the data structure for a dynamically defined local Id

`MCDDynIdReadComPrimitive` read the service, which used a dynamically defined local Id data structure

`MCDDynIdClearComPrimitive` Delete the DynId and the assigned data structure

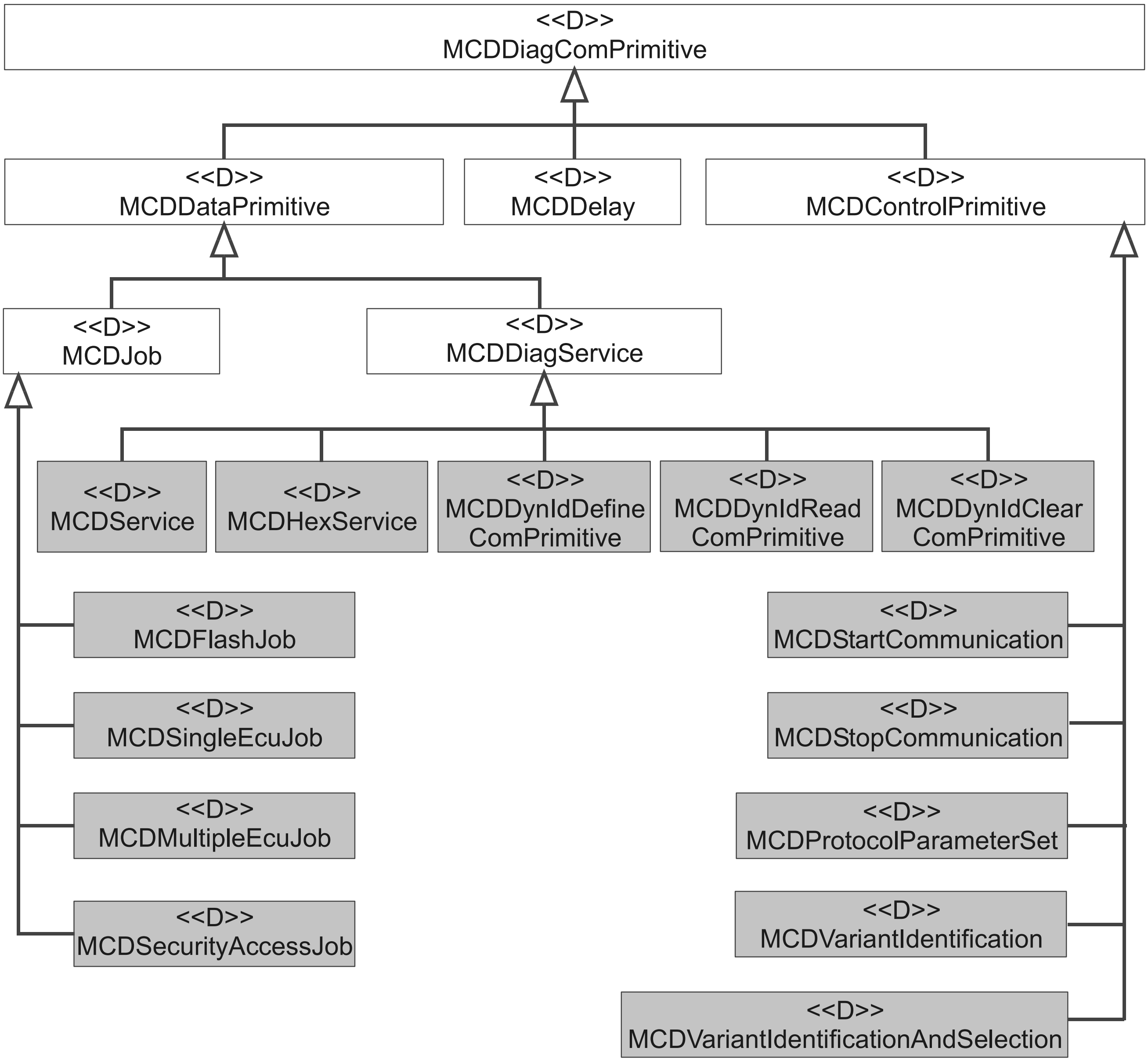

<<D>> MCDDiagComPrimitive
<<D>> MCDDataPrimitive
<<D>> MCDDelay
<<D>> MCDControlPrimitive
<<D>> MCDJob
<<D>> MCDDiagService
<<D>> MCDService
<<D>> MCDHexService
<<D>> MCDDynIdDefineComPrimitive
<<D>> MCDDynIdReadComPrimitive
<<D>> MCDDynIdClearComPrimitive
<<D>> MCDFlashJob
<<D>> MCDSingleEcuJob
<<D>> MCDMultipleEcuJob
<<D>> MCDSecurityAccessJob
<<D>> MCDStartCommunication
<<D>> MCDStopCommunication
<<D>> MCDProtocolParameterSet
<<D>> MCDVariantIdentification
<<D>> MCDVariantIdentificationAndSelection

Figure 57 — Hierarchy of inheritance of DiagComPrimitive

The DiagComPrimitives are divided in two different types: DataPrimitives and ControlPrimitives.

ControlPrimitives perform state transitions, protocol settings or recognize the real ECU. They exist only once per Logical Link. While execution of one of these ControlPrimitives the Logical Link Activity Queue shall be empty and the Activity state shall be `eACTIVITY_IDLE`. They can not be executed asynchronously.

DataPrimitive perform action on the ECU. They are divided in such with PDU information like the service and such without PDU information, the jobs. All DataPrimitives can (if the DataPrimitive carries the IS-CYCLIC flag within the database) executed in cyclic mode and/or asynchronously. Only DataPrimitivesWithPDU can be executed in repetition mode. That is why only DataPrimitive can resize the ResultBuffer.

All communication primitives have the equivalent result structure as diagnostic services. Each non-cyclic DiagComPrimitive has exactly one result record per execution.

In case a DiagComPrimitive, e.g. `MCDDynIdDefineComPrimitive`, `MCDDynIdReadComPrimitive`, or `MCDDynIdClearComPrimitive`, is not available (either in the D-server or in the ODX database), an exception of type eDB_ELEMENT_NOT_AVAILABLE is thrown.

List of MCDObjectTypes for which an object can be created by calling `MCDDiagComPrimitives::addByType` (only object types for which no additional information is required during the creation, that is, no additional parameters):

- eMCDHEXSERVICE
- eMCDPROTOCOLPARAMETERSET
- eMCDSTARTCOMMUNICATION
- eMCDSTOPCOMMUNICATION
- eMCDVARIANTIDENTIFICATION
- eMCDVARIANTIDENTIFICATIONANDSELECTION

If the method is called for a different MCDObjectType than for the ones not in this list an exception of type ePAR_INVALID_OBJECTTYPE_FOR_DIAGCOMPRIMITIVE is thrown.

HexServices represent plain byte streams sent to some ECU on the bus. That is, by using HexServices the majority of security and control functionality of the D-server is bypassed. Therefore, an application using HexServices needs to take of the responsibility for correctness and needs to contain all the necessary mechanisms and control structures to correctly react to whatever is the result of a HexService. Possible results comprise – amongst others – MCDResults, MCDExceptions, or no response at all.

A hex service consists of an empty request and an empty response, that is, both request and response do not contain any parameters. The generated shortnames for these objects are #RtGen_Request and #RtGen_Response.

For setting the request PDU of a hex service, use the `setPDU` method at the `MCDRequest` object associated with the runtime `MCDHexService` object. For retrieving the response PDU, use the `getResponseMessage` method at the `MCDResponse` object associated with the runtime `MCDHexService` object.

9.4.1.2 States of DiagComPrimitives

Each runtime DiagComPrimitive has four states, `eIDLE`, `eREPEATING`, `ePENDING` and `eNOT_EXECUTABLE`. It is created within the state `eIDLE` and may be deleted within the states `eNOT_EXECUTABLE` and `eIDLE` only. Furthermore, within the state `eIDLE` the parameterisation and the evaluation of the results can be carried out. The DiagComPrimitive state changes from `eIDLE` to `ePENDING` is made every time the Client Application starts a method, that goes through the queue till end of this method (e.g. `startRepetition()`), not for repeated execution, `updateRepetitionParameters` and `stopRepetition`. After this or after the execution has been stopped by `cancel()` it returns to the state `eIDLE`. As soon as the DiagComPrimitive starts its execution by `startRepetition()`, it takes the state `eREPEATING`. After finishing the execution within repeated mode by `stopRepetition()`, the Service returns to the state `eIDLE`.

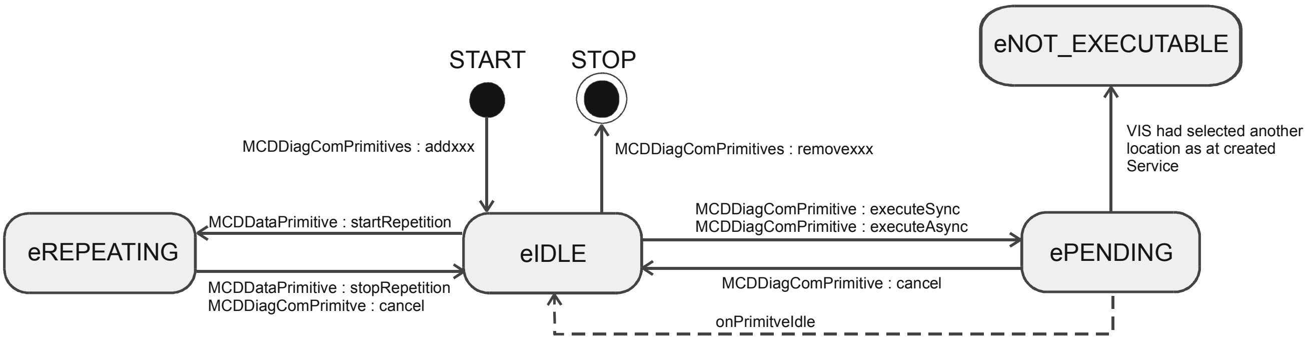

eNOT_EXECUTABLE
START
STOP
MCDDiagComPrimitives : addxxx
MCDDiagComPrimitives : removexxx
VIS had selected another location as at created Service
MCDDataPrimitive : startRepetition
MCDDiagComPrimitive : executeSync
MCDDiagComPrimitive : executeAsync
eREPEATING
eIDLE
ePENDING
MCDDataPrimitive : stopRepetition
MCDDiagComPrimitve : cancel
MCDDiagComPrimitive : cancel
onPrimitveIdle

Figure 58 — State diagram MCDDiagComPrimitive

`MCDService` will be considered in more detail within the next subclause.

Table 26 — DiagCom Primitive states

				MCDDiag Com Primitive		MCDDataPrimitve				MCDProtocol ParameterSet	
System State	LL State	Activity State	DCP State	cancel	executeSync	executeAsync	startRepetition	stopRepetition	updateRepetitionParameter	fetchValue(s)FromInterface	executeSync
eLOGICALLY_CONNECTED	eONLINE & eCOMMUNICATION	IDLE	eNOT_EXECUTABLE								
			eIDLE		X	X	X			X	X
			ePENDING	▨	▨	▨	▨	▨	▨	▨	▨
			eREPEATING	▨	▨	▨	▨	▨	▨	▨	▨
		RUNNING	eNOT_EXECUTABLE								
			eIDLE		X	X	X			X	X
			ePENDING	X							
			eREPEATING	X				X	X		
		SUSPENDED	eNOT_EXECUTABLE								
			eIDLE		X	X	X			X	X
			ePENDING	X							
			eREPEATING	X				X	X		

Key

▨ State could not be reached

NOTE For the execution of DCP the general rule to not throw exceptions in case of feedback state changes has been cancelled as otherwise stability of the D-server is harmed.

9.4.2 Service overview

A diagnostic service will be executed by the Data Processor of the D-server.

Following a subdivision of the diagnostic services is shown.

Table 27 — Types of diagnose services

RuntimeMode (DbDiagService)	RepetitionMode (DbDataPrimitive)	Transmission Mode (DbDiagComPrimitive)	Kind of Execution
eNONCYCLIC	eSINGLE	eSEND_AND_RECEIVE	Async and Sync
		eSEND_OR_RECEIVE	Async and Sync
		eSEND	Async and Sync
		eRECEIVE	Async and Sync
	eREPEATED	eSEND_AND_RECEIVE	Async
		eSEND_OR_RECEIVE	Async
		eSEND	Async
		eRECEIVE	Async
eCYCLIC	eSINGLE	eSEND_AND_RECEIVE	Async
		eSEND_OR_RECEIVE	Async
		eSEND	Async
		eRECEIVE	Async

For `eSEND_OR_RECEIVE` the transmission mode shall be selected before the MCDService is executed. Otherwise the DefaultValue is `eSEND_AND_RECEIVE` is chosen from the server before executing the DiagComPrimitive.

Generally, Jobs are handled like Diag services.

The possible execution modes of a DiagComPrimitive can be obtained by means of the RunTimeMode (`MCDDbDiagService::getRuntimeMode()`), the RepetitionMode (`MCDDbDataPrimitive::getRepetitionMode()`), and the TransmissionMode (`MCDDbDiagComPrimitive::getTransmissionMode()`) of the corresponding database template.

If the DiagComPrimitive reports to allow for repeated execution (`MCDDbDataPrimitive::getRepetitionMode()` returns `eREPEATED`), this DiagComPrimitive can be executed repeatedly by means of `startRepetition()` in addition to `executeSync()` and `executeAsync()`. `eREPEATED` is the default return value of the method `MCDDbDataPrimitive::getRepetitionMode()` if the corresponding information cannot be obtained from ODX.

A non cyclic diagnostic service returns exactly one result record. Non cyclic services end with the returning of the result or are terminated by a Timeout. Non cyclic services may be started synchronously or asynchronously. A non cyclic diagnostic service (no Job !) may be executed in Repetition Mode (if supported by the D-server and permitted by the available data, start by means of the method `startRepetition()`). Thus, after its execution (Event: `onPrimitveHasResult`), the non cyclic diagnostic service is automatically started anew by the D-server. The execution is repeated until the method `stopRepetition()` is called. The time interval between two automatically executions of the non cyclic diagnostic service by the D-server may be set by the Client (RepetitionTime). The Activity Queue of the Logical Link is not bothered by the automatic starting of the non cyclic diagnostic service by the D-server. But however, a delay of the repeated execution may occur, if other methods within the Activity Queue are executed.

Repetition of data primitives is performed by the D-server. However, different parameters contribute to the configuration of a repeated data primitive – request parameters and the repetition time (interval between two executions of a repeated data primitive). If the request parameters of a data primitive are modified while the data primitive is executed repeatedly, the changes shall be confirmed via the method `updateRepetitionParameters()` to guarantee a consistent change of all modified parameters for the next execution of the data primitive. The repetition interval (repetition time) is not a request parameter but a command to the D-server itself. However, to achieve a common behaviour and to support extension with further data affecting the repeated execution of a data primitive, a repetition time set via `setRepetitionTime()` needs to be confirmed by calling the method `updateRepetitionParameter()`, too. That is, if a data primitive is configured to be executed repeatedly, the request parameters are filled and the repetition time is set. Afterwards, both request parameters and repetition time are confirmed for the next execution cycle via `updateRepetitionParameters()`. If the repetition time needs to be modified while a data primitive is already executed repeatedly, the new repetition time is set via `setRepetitionTime()` and needs to be enabled via `updateRepetitionParameters()` for the next execution cycle (see Figure 66 — Repeated service updating parameters). The same applies to a change of request parameters.

Cyclic diagnostic services return complete result records within non equidistant time intervals. The structure of the independent result records is in accordance with the database template. Cyclic diagnostic services usually end with the call of the method `cancel()` (ExecutionState: `eCanceledDuringExecution`). Cyclic diagnostic services may be started asynchronously only.

Diagnostic services which are started by `executeSync()`or by `executeAsync()`, fire an event `onPrimitiveHasResult` to all register Eventhandlers. The execution State can evaluate by means of this event.

For unique identification of executed data primitive e.g. in multi client scenarios, the methods executeAsync and startRepetition deliver an ID for the execution. This ID is reused to connect execution and result. The Result delivered by `onPrimitiveHasResult` or `onPrimitiveHasIntermediateResult` carries this ID , contained in `MCDResult::getExecutionID()`. In case of synchronous execution this information is superfluous but contained for consistency reasons.

The Service is derived from DiagComPrimitive (via DataPrimitive and DiagService) and thus also has the two states `eIDLE` and `ePENDING`, which are used in the same way. Additionally there is the states `eREPEATING`. Within the state `eIDLE` the Service may be parameterised for the execution within the repeated mode. As soon as the Service starts its execution by `startRepetition()`, it takes the state `eREPEATING`. This state isn't changed, if its RequestParameter are set anew. After finishing the repetition by `stopRepetition()`, the Service returns to the state `eIDLE`.

NOTE 1 The instruction queue shown in the following pictures is represented by Activity state.

In some cases, ECUs might signal a delay of their answer to a request by means of a “response pending” message to connected test equipment. Doing so, an ECUs can avoid time-out errors to occur. In general, response pending messages are not forwarded to a D-server by the D-PDU API or any other protocol layer. Instead, the lower layers, e.g. the D-PDU API, process and interpret these messages on their, e.g. by not sending a time-out to the server for a request the answer to which has been signalled as delayed by the corresponding ECU. That is, the protocol layer will wait until the real answer to a request is sent by the ECU and will then forward this answer as the correct answer to a connected D-server.

To disburden a D-server form response pending handling, this should always be performed below the D-server. Otherwise a D-server would need to interpret responses. As a result, a D-server and/or a client using the server would need to implement protocol interpreters, which would cause protocol dependent D-server. Therefore, response pending handling should be performed in the protocol drivers used by a D-server. In addition, the used protocol drivers should not allow switching off response pending handling – neither in the drivers nor anywhere else in the lower layers.

The value of parameters of type `eSYSTEM` is calculated by the MCD-server at creation time of the corresponding request.

All parameters of type defined in the enum `MCDParameterType` are visible for requests/responses (runtime-part and DB-part) and for the PDU. Parameters of type `NRC-CONST` may only appear at response parameters of services. They may not appear in response parameters of jobs.

In case of a receive-only service, the request contains only parameters of type `eCODED-CONST` and `ePHYS-CONST`, e.g. serviceID, subfunctionID. The request is send to the D-PDU API not to an ECU.

If no coded type is given for a reserved parameter in ODX, the method `MCDResponseParameter::getValue()` returns a MCDValue object of type `A_BITFIELD`. This MCDValue is initialized with the bits of the parameter‘s representation in the response-PDU.

If a coded type and coded value is given for a reserved parameter in ODX, the method `MCDResponseParameter::getValue()` returns a MCDValue object of this type. This MCDValue is extracted from the response-PDU in accordance with the general rules for data extraction in ODX.

NOTE 2 In accordance with ODX checker rule 118, coded type and coded value can only exist jointly. That is, they always appear in pairs. However, the coded value is ignored here.

Table 28 — Services

Key

m –	number ECUs in functional group
p –	number of available multi-part responses
n –	number of execution cycles during cyclic or repeated execution

* potentially additional responses can occur, e.g. from ECUs which have not been queried, that is, the response address of which has not been part of the response address table (to be checked)

Service (S) **S = SingleShot DiagService** **C = CyclicDiagService** **R = Repeated DiagService** **J = JavaJob**	**Execution Mode (EM)** **S = Synchronous** **A = Asynchronous**	**Termination (T)** **RT = Response or Timeout** **C = Cancel** **S = Stop Repetition**	**Addressing Mode (AM)** **P = Physical** **F = Functional**	**Enable Intermediate Result (IR)** **Y(es) = Enable** **N(o) = Not Enable** **M(eaningless) = No Influence**	**Termination Events SPR = Single-Part Response**	**Termination Events MPR = Multi-Part Response**
Single-Shot Diag Service	synchronous	Response or Timeout	Physical	no influence	1 x MCDResult (to caller) 1 x onPrimitiveHasResult (to every registered event handler) 1 x onPrimitiveIdle see Figure 59	Not allowed
	synchronous	Timeout	Functional	no influence	1 x MCDResult (to caller) 1 x onPrimitiveHasResult (to every registered event handler, 1 result with <=m responses) 1 x onPrimitiveIdle see Figure K.1	Not allowed
	asynchronous	Response (only if not multi-part) or Timeout	Physical	no influence	1 x onPrimitiveHasResult (1 result with 1 response) 1 x onPrimitiveIdle see Figure 58	p x onPrimitiveHasResult (p results with 1 response each) 1 x onPrimitiveIdle (after Timeout) see Figure K.2
	asynchronous	Cancel	Physical	no influence	1 x onPrimitiveHasResult (empty result which includes execution state eCANCELLED_DURING_EXECUTION) 1 x onPrimitiveIdle	p x onPrimitiveHasResult (number of results until cancel has been processed, 1 response per result) 1 x onPrimitiveHasResult (empty result which includes execution state eCANCELLED_DURING_EXECUTION) 1 x onPrimitiveIdle

Table 28 (*continued*)

Service (S) **S = SingleShot DiagService** **C = CyclicDiagService** **R = Repeated DiagService** **J = JavaJob**	**Execution Mode (EM)** **S = Synchronous** **A = Asynchronous**	**Termination (T)** **RT = Response or Timeout** **C = Cancel** **S = Stop Repetition**	**Addressing Mode (AM)** **P = Physical** **F = Functional**	**Enable Intermediate Result (IR)** **Y(es) = Enable** **N(o) = Not Enable** **M(eaningless) = No Influence**	**Termination Events** **SPR = Single-Part Response**	**Termination Events** **MPR = Multi-Part Response**
	asynchronous	Timeout	Functional	No	1 x onPrimitiveHasResult (1 result containing <=m responses)* 1 x onPrimitiveIdle see Figure K.3	p x onPrimitiveHasResult (where potentially p > m, 1 response per result) 1 x onPrimitiveIdle see Figure K.4
	asynchronous	Cancel	Functional	No	1 x onPrimitiveHasResult (1 result containing <=m responses received until cancel has been processed, result includes execution state eCANCELLED_DURING_EXECUTION) 1 x onPrimitiveIdle see Figure K.6	p x onPrimitiveHasResult (where potentially p > m, 1 response per result) 1 x onPrimitiveHasResult (1 empty result which has the execution state eCANCELLED_DURING_EXECUTION) 1 x onPrimitiveIdle
	asynchronous	Timeout	Functional	Yes	1 x onPrimitiveHasResult (1 result which contains all responses received until the data primitive has been started) <=m x onPrimitiveHasIntermediateResult (1 result containing 1 response) 1 x onPrimitiveIdle see Figure K.7	p x onPrimitiveHasResult (where potentially p > m, 1 response per result) 1 x onPrimitiveIdle
	asynchronous	Cancel	Functional	Yes	1 x onPrimitiveHasResult (1 result which contains all responses received until the data primitive has been started, result has execution state eCANCELLED_DURING_EXECUTION) <=m x onPrimitiveHasIntermediateResult (1 result containing 1 response) 1 x onPrimitiveIdle	p x onPrimitiveHasResult (where potentially p > m, 1 response per result) 1 x onPrimitiveHasResult (1 empty result with execution state eCANCELLED_DURING_EXECUTION) 1 x onPrimitiveIdle
Cyclic Diag Service	asynchronous	Cancel	Physical	no influence	n x onPrimitiveHasResult (n results with 1 response each) 1 x onPrimitiveHasResult (empty result with execution state eCANCELLED_DURING_EXECUTION) 1 x onPrimitiveIdle see Figure 62	$\sum_{i=1}^{n} p_i$ x on PrimitiveHasResult ($\sum_{i=1}^{n} p_i$ results with 1 response each, where potentially p_i >= 1) 1 x onPrimitiveHasResult (result with execution state eCANCELLED_DURING_EXECUTION) 1 x onPrimitiveIdle see Figure K.5
	asynchronous	Cancel	Functional	no influence	$\sum_{i=1}^{n} m_i$ on PrimitiveHasResult ($\sum_{i=1}^{n} m_i$ results containing 1 response each, m_i <= m) 1 x onPrimitiveHasResult (result with execution state eCANCELLED_DURING_EXECUTION) 1 x onPrimitiveIdle	$\sum_{i=1}^{n} p_i$ x on PrimitiveHasResult ($\sum_{i=1}^{n} p_i$ results with 1 response each, where potentially p_i > m) 1 x onPrimitiveHasResult (result with execution state eCANCELLED_DURING_EXECUTION) 1 x onPrimitiveIdle

Table 28 (*continued*)

Service (S) **S = SingleShot DiagService** **C = CyclicDiagService** **R = Repeated DiagService** **J = JavaJob**	**Execution Mode (EM)** **S = Synchronous** **A = Asynchronous**	**Termination (T)** **RT = Response or Timeout** **C = Cancel** **S = Stop Repetition**	**Addressing Mode (AM)** **P = Physical** **F = Functional**	**Enable Intermediate Result (IR)** **Y(es) = Enable** **N(o) = Not Enable** **M(eaningless) = No Influence**	**Termination Events SPR = Single-Part Response**	**Termination Events MPR = Multi-Part Response**
Repeated Diag Service	asynchronous	Stop Repetition	Physical	no influence	n x onPrimitveHasResult(n results containing 1 response each) 1 x onPrimitiveHasResult(empty result with execution state eREPETITION_STOPPED) 1 x onPrimitiveIdle see Figure 63	$\sum_{i=1}^{n} p_i$ x on PrimitiveHasResult ($\sum_{i=1}^{n} p_i$ results with 1 response each, where potentially p_i >= 1) 1 x onPrimitiveHasResult(empty result with execution state eREPETITION_STOPPED) 1 x onPrimitiveIdle
	asynchronous	Stop Repetition	Functional	No	n x onPrimitveHasResult(n results containing <=m responses each) 1 x onPrimitiveHasResult(empty result with execution state eREPETITION_STOPPED) 1 x onPrimitiveIdle	$\sum_{i=1}^{n} p_i$ x on PrimitiveHasResult ($\sum_{i=1}^{n} p_i$ results with 1 response each, where potentially p_i > m) 1 x onPrimitiveHasResult(empty result with execution state eREPETITION_STOPPED) 1 x onPrimitiveIdle
	asynchronous	Stop Repetition	Functional	Yes	$\sum_{i=1}^{n} m_i$ on PrimitiveHasIntermediate Result ($\sum_{i=1}^{n} m_i$ results containing 1 response each, m_i <= m) n x 1 onPrimitiveHasResult (n results containing <= m responses each, results include an error for timeout) 1 x onPrimitiveHasResult(empty result with execution state eREPETITION_STOPPED) 1 x onPrimitiveIdle	$\sum_{i=1}^{n} p_i$ x on PrimitiveHasResult ($\sum_{i=1}^{n} p_i$ results with 1 response each, where potentially p_i > m) 1 x onPrimitiveHasResult(empty result with execution state eREPETITION_STOPPED) 1 x onPrimitiveIdle
	asynchronous	Cancel	Physical	no influence	n x onPrimitveHasResult (n results containing 1 response each) 1 x onPrimitiveHasResult (result with execution state eCANCELLED_DURING_EXECUTION) 1 x onPrimitiveIdle see Figure K.7 and Figure K.8	$\sum_{i=1}^{n} p_i$ x on PrimitiveHasResult ($\sum_{i=1}^{n} p_i$ results with 1 response each, where potentially p_i >= 1) 1 x onPrimitiveHasResult (result with execution state eCANCELLED_DURING_EXECUTION) 1 x onPrimitiveIdle

Table 28 (*continued*)

Service (S) **S = SingleShot DiagService** **C = CyclicDiagService** **R = Repeated DiagService** **J = JavaJob**	**Execution Mode (EM)** **S = Synchronous** **A = Asynchronous**	**Termination (T)** **RT = Response or Timeout** **C = Cancel** **S = Stop Repetition**	**Addressing Mode (AM)** **P = Physical** **F = Functional**	**Enable Intermediate Result (IR)** **Y(es) = Enable** **N(o) = Not Enable** **M(eaningless) = No Influence**	**Termination Events SPR = Single-Part Response**	**Termination Events MPR = Multi-Part Response**
	asynchronous	Cancel	Functional	No	n x onPrimitveHasResult (n results containing <=m responses each) 1 x onPrimitiveHasResult (result with execution state eCANCELLED_DURING_EXECUTION) 1 x onPrimitiveIdle	$\sum_{i=1}^{n} pi$ x on PrimitiveHasResult ($\sum_{i=1}^{n} pi$ results with 1 response each, where potentially p_i > m) 1 x onPrimitiveHasResult (result with execution state eCANCELLED_DURING_EXECUTION) 1 x onPrimitiveIdle
	asynchronous	Cancel	Functional	Yes	$\sum_{i=1}^{n} mi$ on PrimitiveHasIntermediate Result ($\sum_{i=1}^{n} mi$ results containing 1 response each, m_i <= m) n x 1 onPrimitiveHasResult (n results containing <= m responses each, results include an error for timeout) 1 x onPrimitiveHasResult(result with execution state eCANCELLED_DURING_EXECUTION) 1 x onPrimitiveIdle	$\sum_{i=1}^{n} pi$ x on PrimitiveHasResult ($\sum_{i=1}^{n} pi$ results with 1 response each, where potentially p_i > m) 1 x onPrimitiveHasResult (result with execution state eCANCELLED_DURING_EXECUTION) 1 x onPrimitiveIdle
Java-Job	synchronous	Response or Timeout	Physical	no influence	Number of intermediate results controlled by Java-Job x onPrimitiveHasIntermediate Result 1 x MCDResult (to caller) 1 x onPrimitiveResult(1 result containing an arbitrary number of responses) 1 x onPrimitiveIdle see Figure 178	Not allowed
	asynchronous	Response or Timeout	Physical	no influence	Number of intermediate results controlled by Java-Job x onPrimitiveHasIntermediate Result 1 x onPrimitiveResult(1 result containing an arbitrary number of responses) 1 x onPrimitiveIdle see Figure 177	Not allowed
	asynchronous	Cancel	Physical	no influence	Number of intermediate results controlled by Java-Job before cancel x onPrimitiveHasIntermediate Result 1 x onPrimitiveHasResult (empty result which includes execution state eCANCELLED_DURING_EXECUTION) 1 x onPrimitiveIdle	Not allowed
	synchronous		Functional	no influence	Not allowed	Not allowed
	asynchronous		Functional	no influence	Not allowed	Not allowed
	asynchronous	Cancel	Functional	no influence	Not allowed	Not allowed

Table 28 (*continued*)

Service (S) **S = SingleShot DiagService** **C = CyclicDiagService** **R = Repeated DiagService** **J = JavaJob**	**Execution Mode (EM)** **S = Synchronous** **A = Asynchronous**	**Termination (T)** **RT = Response or Timeout** **C = Cancel** **S = Stop Repetition**	**Addressing Mode (AM)** **P = Physical** **F = Functional**	**Enable Intermediate Result (IR)** **Y(es) = Enable** **N(o) = Not Enable** **M(eaningless) = No Influence**	**Termination Events SPR = Single-Part Response**	**Termination Events MPR = Multi-Part Response**
Repeated Java-Job	asynchronous	Stop Repetition	Physical	no influence	n x Number of intermediate results controlled by Java-Job x onPrimitiveHasIntermediateResult m x onPrimitveHasResult(n results containing 1 response each) 1 x onPrimitiveHasResult(empty result with execution state eREPETITION_STOPPED) 1 x onPrimitiveIdle	Not allowed
	asynchronous	Cancel	Physical	no influence	n x Number of intermediate results controlled by Java-Job before cancel x onPrimitiveHasIntermediateResult m x onPrimitveHasResult(n results containing 1 response each) 1 x onPrimitiveHasResult (empty result which includes execution state eCANCELLED_DURING_EXECUTION) 1 x onPrimitiveIdle	Not allowed
	asynchronous	Stop Repetition	Functional	No	Not allowed	Not allowed
	asynchronous	Stop Repetition	Functional	Yes	Not allowed	Not allowed
	asynchronous	Cancel	Functional	no influence	Not allowed	Not allowed

Examples for not further in detailed services can be found in Annex K.

9.4.3 Non cyclic single shot diag service

Executed outside jobs

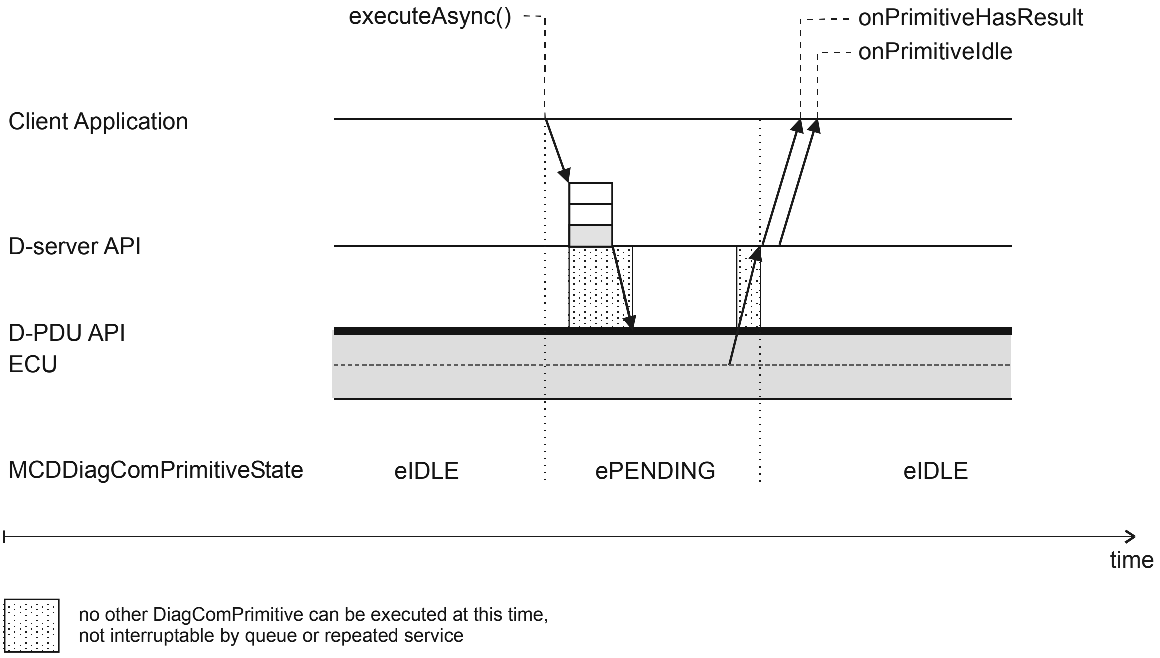


**Figure 59 — Non cyclic single shot diag service (asynchronous executed outside jobs)
S=S; EM=A; T=RT; AM=P; IR=N; SPR**

NOTE 1 After sending the request it is possible to start other services from the queue to the ECU. This is depended from the used protocol, this means it is not necessary to wait till the response is available.

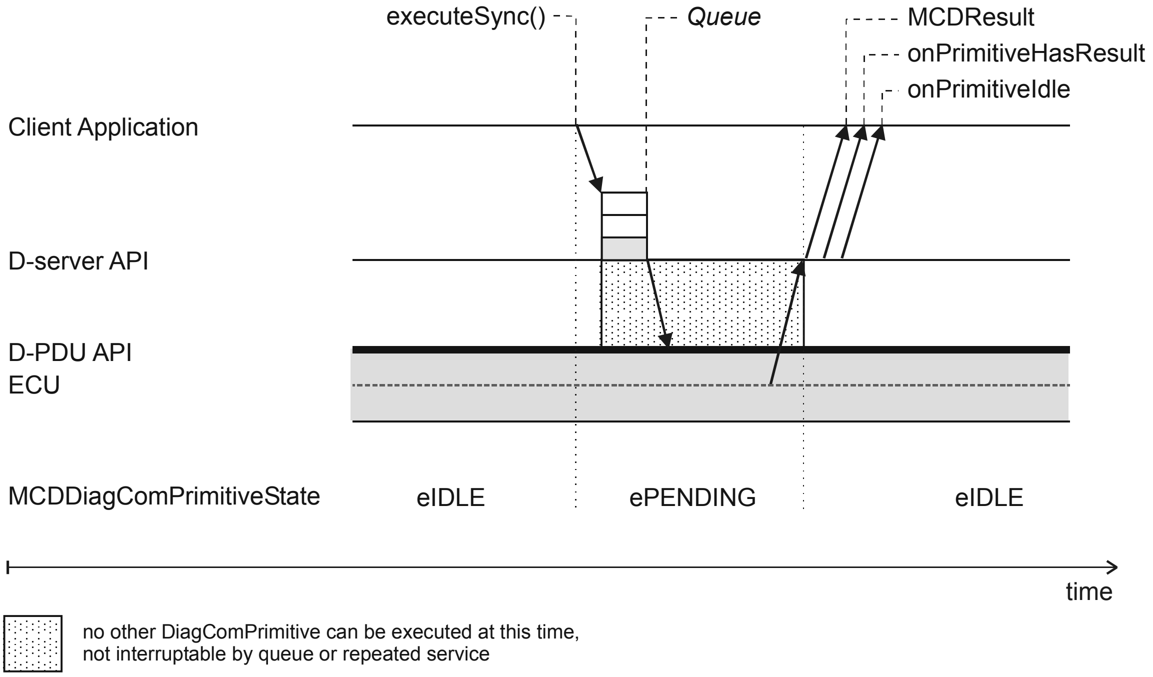


**Figure 60 — Non cyclic single shot diag service (synchronous executed outside jobs)
S=S; EM=S; T=RT; AM=P; IR=N; SPR**

Sample: normal diag service Keyword2000 (e.g. ReadTroubleCode)

Description: This variant of DiagComPrimitive execution is compatible to ASAM MCD 3 specification version 1.1.

DiagComPrimitive method description:

`executeSync()` synchronous start of DiagComPrimitive execution
`executeAsync()` asynchronous start of DiagComPrimitive execution
`cancel()` quit DiagComPrimitive execution as fast as possible or remove it from execution activity queue

States:

The DiagComPrimitive state changes from `eIDLE` (initially; state before starting DiagComPrimitive execution) to `ePENDING` (state while execution) back to `eIDLE` (state after execution). The states of the DiagComPrimitive are set by the D-server.

Results:

There can be only 0 or 1 result. There cannot be intermediate results. The result is the return value of the synchronous execution. The execution state can be requested from the Result itself.
Functional addressing delivers also only one complete result.

Executed inside jobs

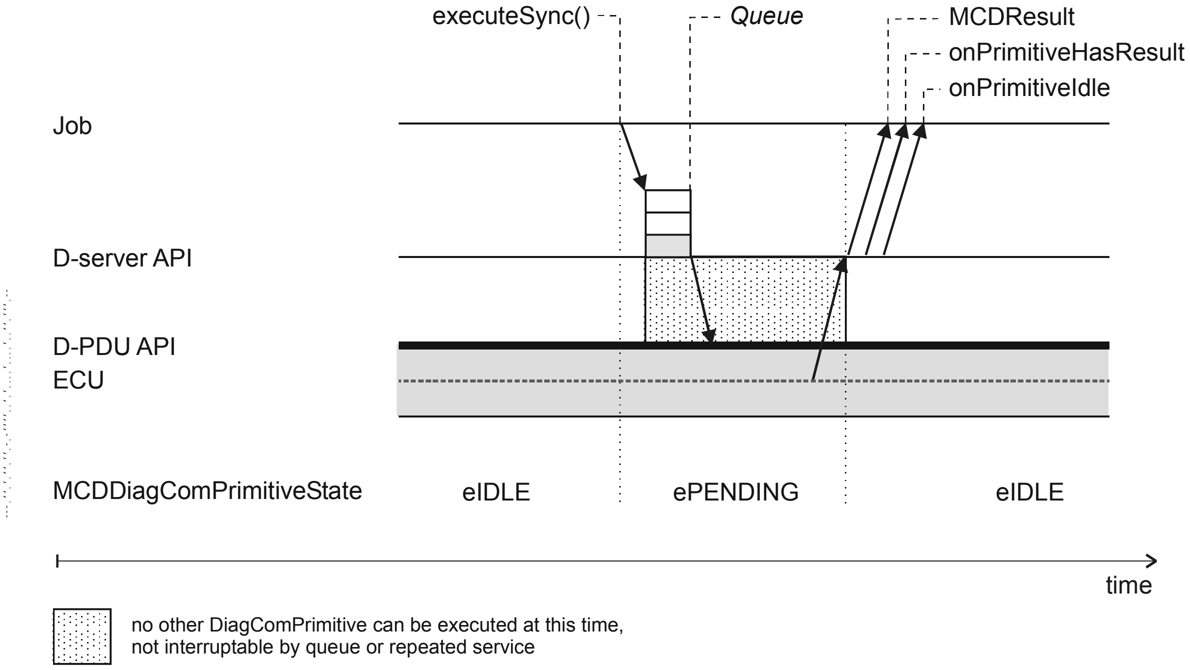


Figure 61 — Non cyclic single shot diag service or job (executed inside jobs)

NOTE 2 Inside a job only non cyclic single diag services and jobs can be executed and these shall be started synchronous.

Results:

There can be only 0 or 1 result. There cannot be intermediate results.
The result is the return value of the synchronous execution. The execution state can be requested from the Result itself.
Functional addressing delivers also only one complete result.
The results of this service will be evaluated inside the job (see 9.22).

9.4.4 Cyclic diag service

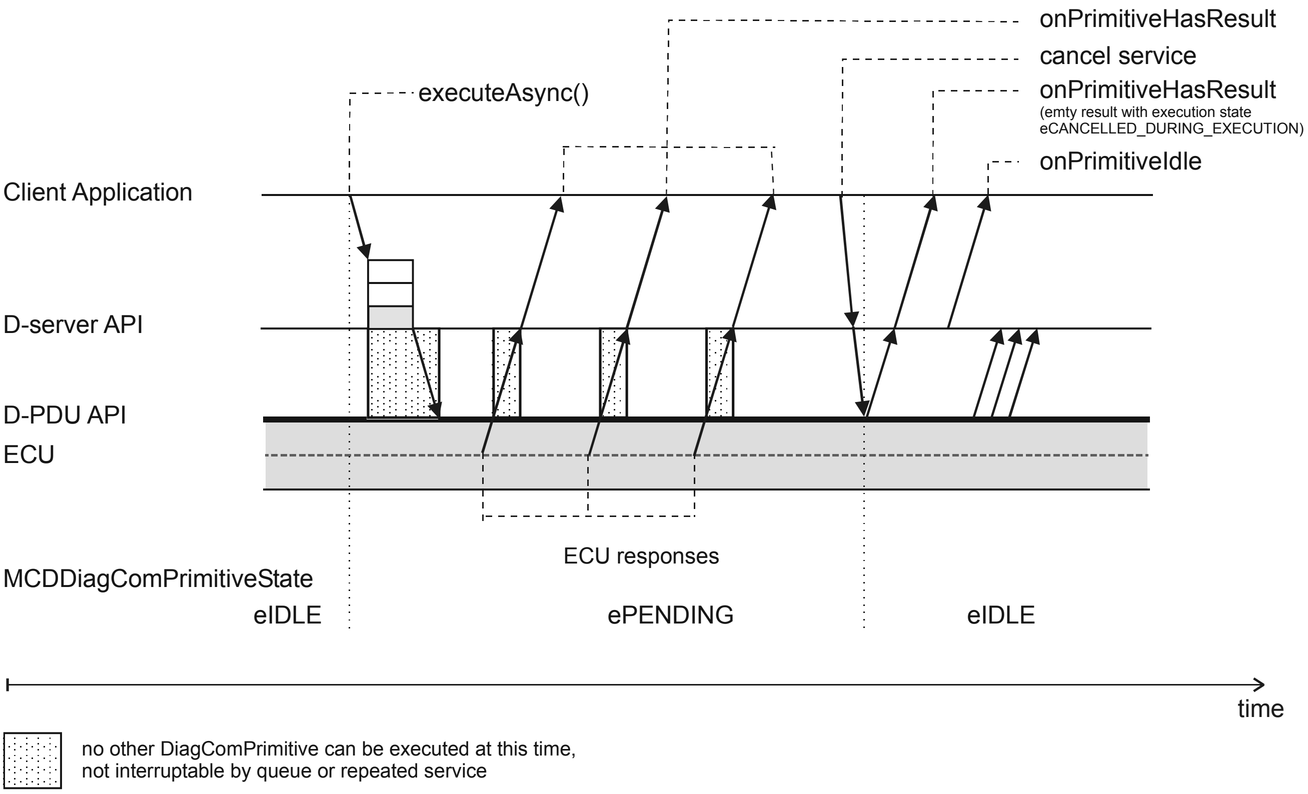


Figure 62 — Cyclic diag services
S=C; EM=A; T=C; AM=P; IR=N; SPR

Sample: Keyword2000 periodic transmission

Description: This variant of DiagComPrimitive execution is compatible to ASAM MCD 3 specification Version 1.1.

DiagComPrimitive method description:

`executeAsync`()	asynchronous start of DiagComPrimitive execution
`cancel`()	quit DiagComPrimitive execution as fast as possible or remove it from execution queue (in version 3D 1.1 = DiagServiceStop())

States:
The DiagComPrimitive state changes from `eIDLE` (initially; state before starting DiagComPrimitive execution) to `ePENDING` (state while execution) back to `eIDLE` (state after execution). The states of the DiagComPrimitive are set by the D-server. While `ePENDING` several results can be reported via event and requested from ring buffer. The execution will normally be terminated by cancel(). But a termination with TimeOut or BusError is also possible and not in all use cases an indication of errors (e.g. in ECU a loop with 100 data registrations). In this case a `onPrimitiveIdle` event will be sent.

Results:
There can be several (zero or more) results, stored in the ring buffer. For every result there is an event sent to the Client Application. The results do not have to occur in equidistant time intervals.
After getting one of the events `onPrimitiveHasResult` the execution state and the number of results can be requested.

9.4.5 Repeated diag service

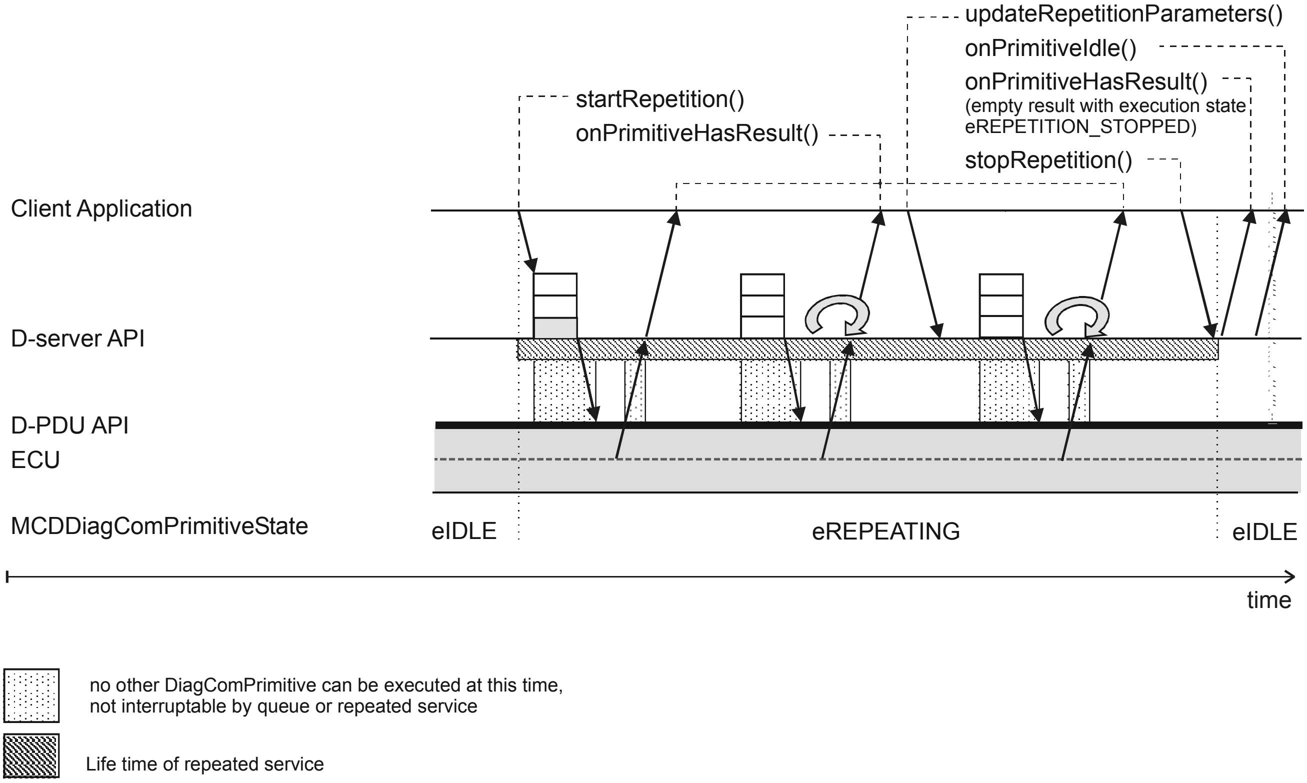

updateRepetitionParameters()
onPrimitiveIdle()
onPrimitiveHasResult()
(empty result with execution state eREPETITION_STOPPED)
stopRepetition()
startRepetition()
onPrimitiveHasResult()
Client Application
D-server API
D-PDU API
ECU
MCDDiagComPrimitiveState
eIDLE
eREPEATING
eIDLE
time
no other DiagComPrimitive can be executed at this time, not interruptable by queue or repeated service
Life time of repeated service

Figure 63 — Repeated diag service
S=R; EM=A; T=S; AM=P; IR=N; SPR

Description: Repeated execution of a NON CYCLIC DIAG SERVICE
The time between two repeated executions will be set by Client Application and is not stored in database.

DiagComPrimitive method description:

`startRepetition()`	start of DiagComPrimitive execution, after passing the queue, the service will live in a loop and start action (not through the queue)
`stopRepetition()`	quit DiagComPrimitive execution
`updateRepetitionParameters ()`	the repetition time of the communication primitive, set by the client application, becomes active for the repeated execution. The new repetition time will take effect after the next execution.

States:
The DiagComPrimitive state changes from `eIDLE` (before `startRepetition()`) to `eREPEATING` (after `startRepetition()` and back to `eIDLE` (after `stopRepetition()` or `cancel()`).

Results:
There can be one or more results, stored in the ring buffer. There cannot be intermediate results. After getting one of the events `onPrimitiveHasResult` the execution state and the number of results can requested.

9.4.6 Repeated send only diag service

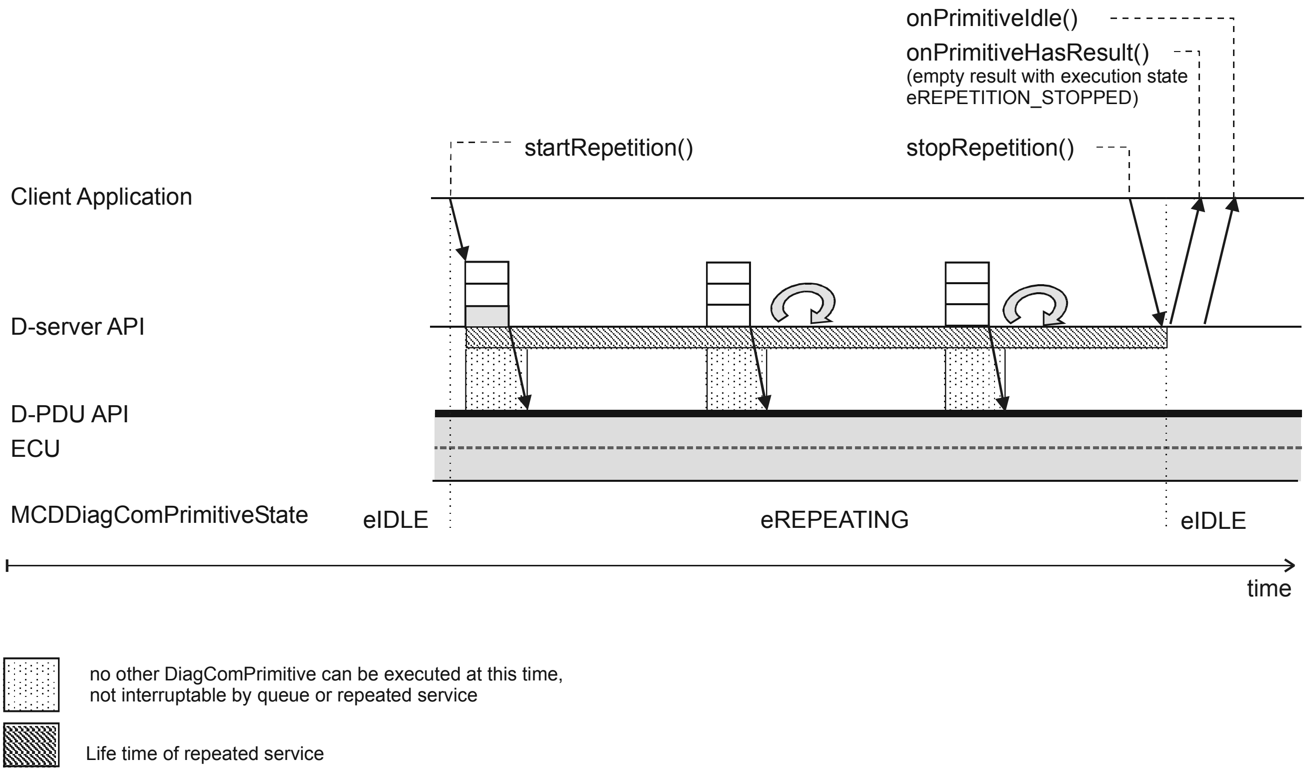

onPrimitiveIdle()
onPrimitiveHasResult()
(empty result with execution state eREPETITION_STOPPED)
startRepetition()
stopRepetition()
Client Application
D-server API
D-PDU API
ECU
MCDDiagComPrimitiveState
eIDLE
eREPEATING
eIDLE
time
no other DiagComPrimitive can be executed at this time, not interruptable by queue or repeated service
Life time of repeated service

Figure 64 — Repeated send only diag service

Sample:
1. Send single CAN frame (diag on CAN, ISO 15765 unacknowledged unsegmented data transfer)
2. RestBus Simulation

Description: Special case of Repeated Diag Service

Results: There are no results.

9.4.7 Repeated receive only diag service

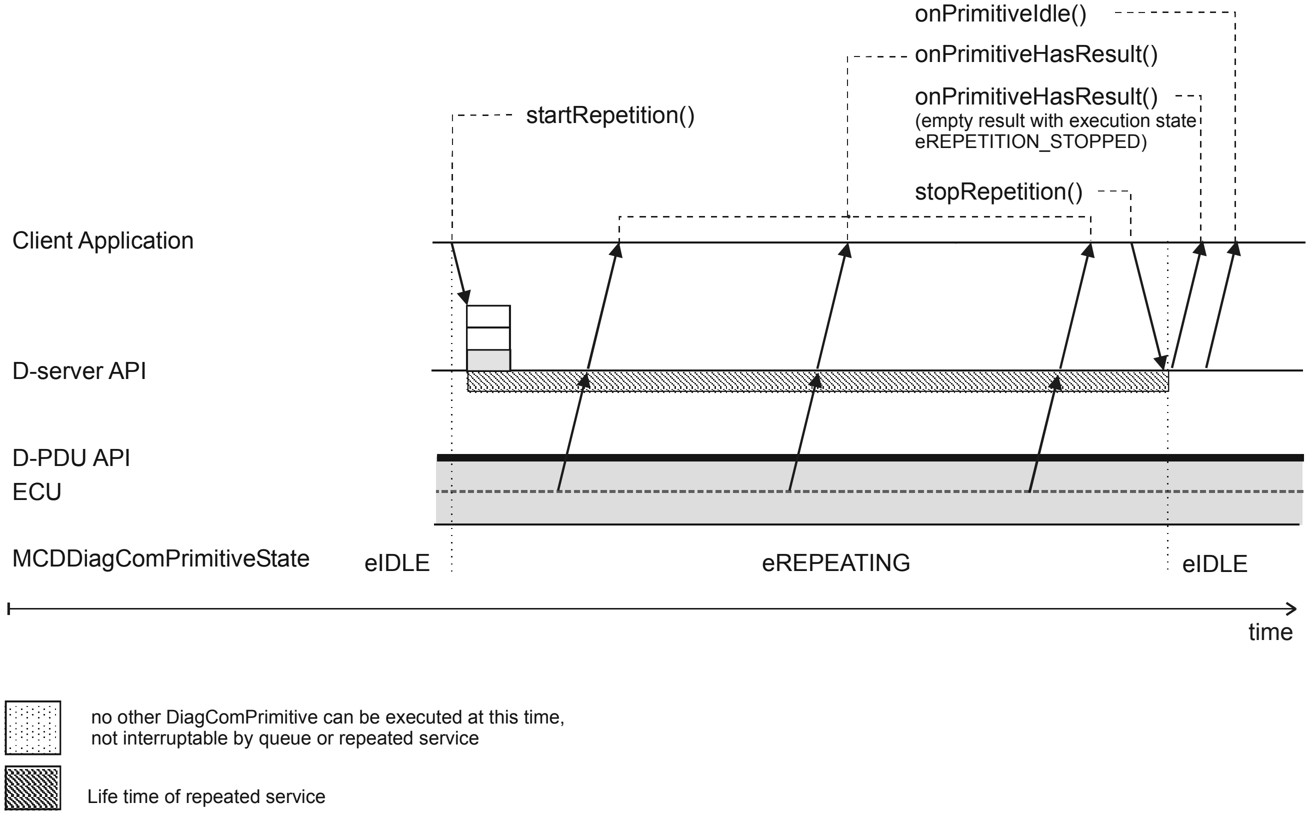

onPrimitiveIdle()
onPrimitiveHasResult()
onPrimitiveHasResult()
(empty result with execution state eREPETITION_STOPPED)
startRepetition()
stopRepetition()
Client Application
D-server API
D-PDU API
ECU
MCDDiagComPrimitiveState
eIDLE
eREPEATING
eIDLE
time
no other DiagComPrimitive can be executed at this time, not interruptable by queue or repeated service
Life time of repeated service

Figure 65 — Repeated receive only diag service

Sample: 1.ISO 15765 receive single CAN frames
2.ISO 14229-1 response on event

Description: Special case of Repeated Diag Service

Results:
There can be one or more results, stored in the ring buffer. There cannot be intermediate results. After getting one of the events `onPrimitiveHasResult` the execution state and the number of results can request.

9.4.8 Updating repetition parameters

Repetition parameters are not only the values of the request parameters but also the repetition time (interval between two executions of a repeated data primitive). These parameters can be changed during execution of the data primitive in repeated mode. The changes need to be confirmed by the call of `updateRepetitionParameters`. The next execution of the data primitive will take the new request parameter values and after this execution the new repetition time will take effect.

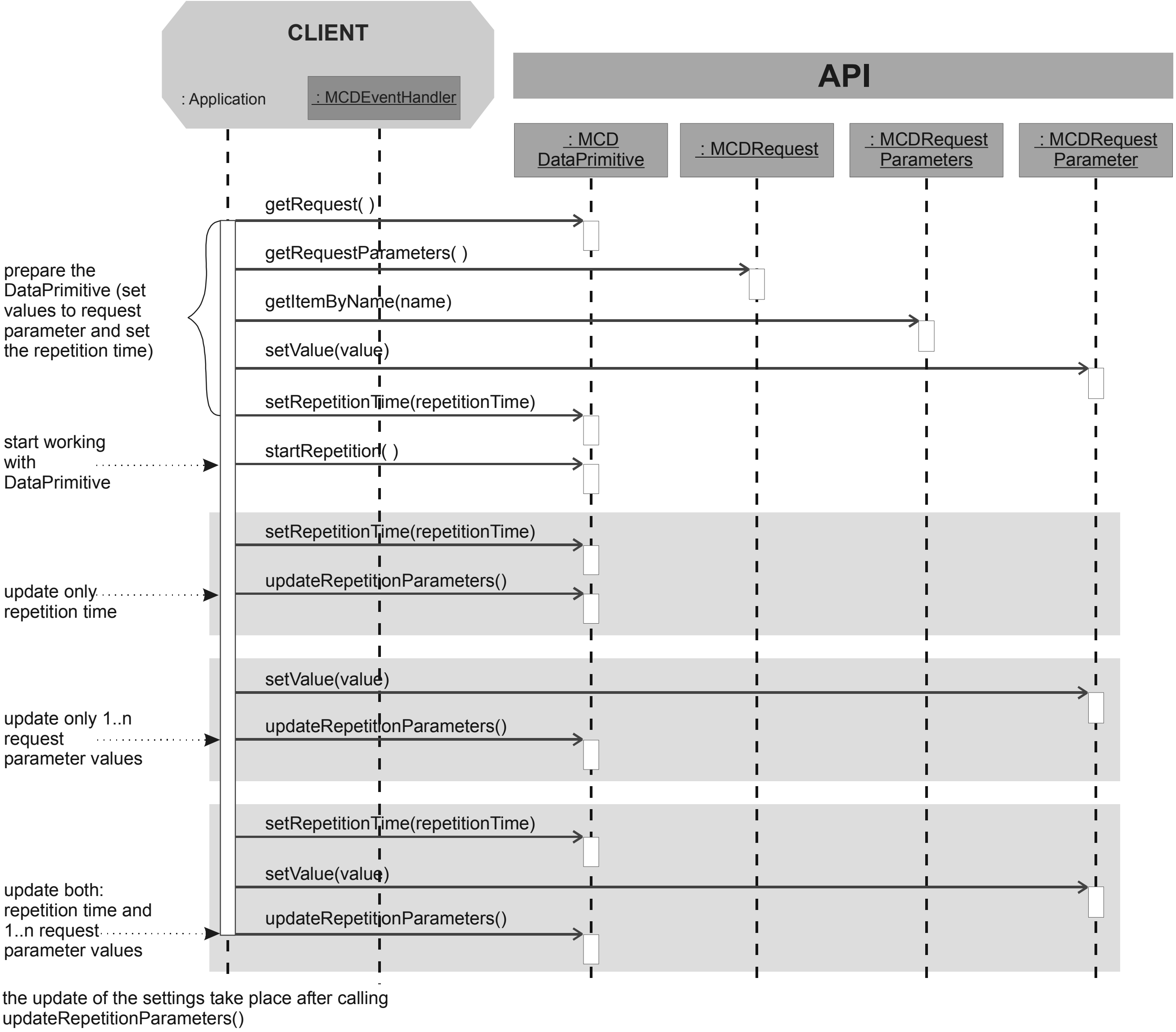

CLIENT
: Application
: MCDEventHandler
API
: MCD DataPrimitive
: MCDRequest
: MCDRequest Parameters
: MCDRequest Parameter
getRequest( )
getRequestParameters( )
getItemByName(name)
setValue(value)
setRepetitionTime(repetitionTime)
startRepetition( )
setRepetitionTime(repetitionTime)
updateRepetitionParameters()
setValue(value)
updateRepetitionParameters()
setRepetitionTime(repetitionTime)
setValue(value)
updateRepetitionParameters()
prepare the DataPrimitive (set values to request parameter and set the repetition time)
start working with DataPrimitive
update only repetition time
update only 1..n request parameter values
update both: repetition time and 1..n request parameter values
the update of the settings take place after calling updateRepetitionParameters()

Figure 66 — Repeated service updating parameters

9.4.9 Summary

Table 29 — Events in case of single or repeated execution of DiagService and Jobs

	Single or Repeated
DiagService	onPrimitiveHasResult
	onPrimitiveIdle
Job	onPrimitiveHasIntermediateResult ②
	onPrimitiveHasResult ①

The Job API's `sendFinalResult` ① raises an event of type `onPrimitiveHasResult` independent of the fact whether a Job has been executed single-shot or repeated. The Job API's `sendIntermediateResult` ② raises an event of type onPrimitiveHasIntermediateResult independent of the fact whether a Job has been executed single-shot or repeated.

If MCDDLogicalLink ::setIntermediateResultForFunctionalAddressing is set to true and the service is executed asynchronously, intermediate results are created for functional services. For each response of an ECU an intermediate result will be created and send via the onPrimitiveHasIntermediateResult event to the client. The handling of final result is not influenced by enabling intermediate results.

Table 30 — Example matrix for assigning MCDExecutionState to MCDResult

	Physical Addressing (service/job)	**Functional Addressing**		
ECU response(s)	**ECU**	**ECU1**	**ECU2**	**ECU3**
eCANCELED_DURING_EXECUTION	canceled during execution	canceled during execution		
eCANCELED_FROM_QUEUE	canceled from queue	canceled from queue		
eREPETITION_STOPPED	repetition stopped	repetition stopped		
eALL_FAILED	failed	failed	failed	failed
eFAILED	**state not allowed in physical addressing mode**	failed	positive negative invalid failed	positive negative invalid
eALL_INVALID_RESPONSE	invalid	invalid	invalid	invalid
eINVALID_RESPONSE	**state not allowed in physical addressing mode**	invalid	positive negative invalid	positive negative
eALL_NEGATIVE	negative	negative	negative	negative
eNEGATIVE	**state not allowed in physical addressing mode**	negative	positive negative	positive
eALL_POSITIVE	positive	positive	positive	positive

NOTE Order of resolution in D-server is from top to bottom.

9.4.10 Protocol parameters

9.4.10.1 General

For a D-server to exchange data with an ECU, it needs to be able to correctly set up the communications link to be used, e.g. regarding baud rate, protocol timings, tester present behaviour and so on. To be able to handle these kinds of settings in a protocol-independent manner, the concept of protocol parameters (also called communication parameters in this part of ISO 22900) has been introduced. Protocol parameters are used to define all settings that are relevant for diagnostic communication, and described by specific elements within an ODX data set. The D-server is agnostic concerning the contents of protocol parameters, and usually passes them on to its protocol layer implementation (e.g. a D-PDU API layer), which contains the according logic to understand and correctly use protocol parameter semantics. This subclause describes how protocol parameters are represented at the D-server API, and how client applications can modify and use protocol parameters to configure a communication link's behaviour.

9.4.10.2 Introduction related to ODX

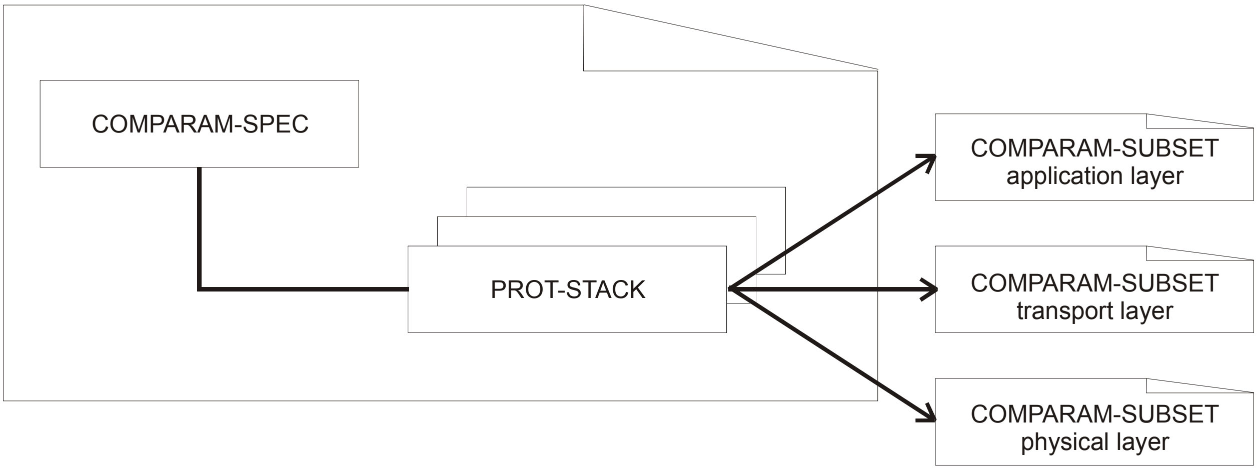

COMPARAM-SPEC
PROT-STACK
COMPARAM-SUBSET application layer
COMPARAM-SUBSET transport layer
COMPARAM-SUBSET physical layer

Figure 67 — PROT-STACKS and its COMPARAM-SUBSETs

Protocol parameters are defined within an ODX part called a PROT-STACK. Within a PROT-STACK, communication parameters are grouped into different COMPARAM-SUBSETs, usually with respect to the OSI layer they belong to (e.g. physical-, transport- or protocol-layer). At the D-server API, communication parameters are not distinguished in accordance with the different COMPARAM-SUBSET/OSI layers they belong to; rather, a superset of all available communication parameters (within the active PROT-STACK) will be delivered to the client application. Exactly one PROT-STACK is used during runtime and defines the active communication parameter set for a communication link. The communication parameter values defined within the PROT-STACK (default values) can be redefined at different contexts like diagnostic layers (DIAG-LAYER), diagnostic service (DIAG-SERVICE), logical link and physical vehicle link. There are dependencies between the different redefinition contexts. For example, the redefinition of a communication parameter at PROTOCOL level is also applied in all the more specialized DIAG-LAYERS (e.g. BASE-VARIANT) using this PROTOCOL. A complete description of redefinition of communication parameters can be found in ASAM MCD 2 D ODX. In general, two types of communication parameter exist in ODX: The simple COMPARAM holding a single value of a simple data type and the COMPLEX-COMPARAM, which contains a complex structure of values. The following list shows the general mapping between communication parameter types defined in ODX and types of MCDDbProtocolParameters.

- A COMPLEX-COMPARAM with ALLOW-MULTIPLE-VALUES set to 'true' results in a protocol parameter with datatype `eFIELD` containing an arbitrary number of COMPLEX-COMPARAMs. The shortnames of the inner structure elements are generated by the D-server in accordance with the following pattern: `#RtGen_<NameOfComplexComParam>_<unique_number>`.

- A COMPLEX-COMPARAM with ALLOW-MULTIPLE-VALUES set to 'false' results in a protocol parameter with datatype `eSTRUCTURE` containing an arbitrary number of simple COMPARAMs or COMPLEX-COMPARAMs.

- A simple COMPARAM results in a protocol parameter with parameter type `eVALUE`. The datatype of such a protocol parameter is defined by the DOP of this parameter in ODX.

In case of a COMPLEX-COMPARAM with ALLOW-MULTIPLE-VALUES set to true (mapped to eFIELD), the COMPLEX-COMPARAM value can be redefined by multiple-value structures within one context, which results in multiple-field items.

Such a COMPLEX-COMPARAM cannot be redefined partially, i.e. it is not possible to overwrite only parts of a COMPLEX-COMPARAM that is inherited from a higher layer. A redefinition will always substitute the entire COMPLEX-COMPARAM, regardless whether the new definition contains less data than the overridden one.

For example, consider the COMPLEX-COMPARAM CP_SessionTimingOverride, which contains an array of structures containing session timing data, one structure for each defined session. If this parameter is redefined in a DIAG-LAYER (overriding a definition that e.g. contains timing data for sessions A and B), the new definition again shall provide array entries for all sessions where a timing override should be applied at runtime. If CP_SessionTimingOverride is redefined for session A but not for session B, no timing override is applied at runtime for session B.

In contrast to COMPLEX-COMPARAMs with ALLOW-MULTIPLE-VALUES set to false, the order of sub-parameters (field items) of a COMPLEX-COMPARAM with ALLOW-MULTIPLE-VALUES set to true doesn’t carry any information. Therefore, a D-server does not have to guarantee the same sequence for such field items as defined in ODX. Redefinition of communication parameter values is done by the ODX element COMPARAM-REF with an attached value definition. Only a COMPLEX-COMPARAM with ALLOW-MULTIPLE-VALUES set to true might be referenced by a COMPARAM-REF without a value definition to reset the communication parameter value to an empty field.

At the D-server API, communication parameters are represented by `MCDDbProtocolParameter` or `MCDProtocolParameter` objects. In the following, the `MCDDbProtocolParameter` and `MCDProtocolParameter` objects are both called ProtocolParameters if there is no need to distinguish between both types.

NOTE The `MCDRequestParameters` of a `MCDProtocolParameterSet` control primitive represent the protocol parameters of the Logical Link.

As an example, the protocol parameter structure at the D-server API level for the complex protocol parameter CP_SessionTimingOverride is illustrated in the figure below, followed by the appropriate protocol parameter data set in ODX.

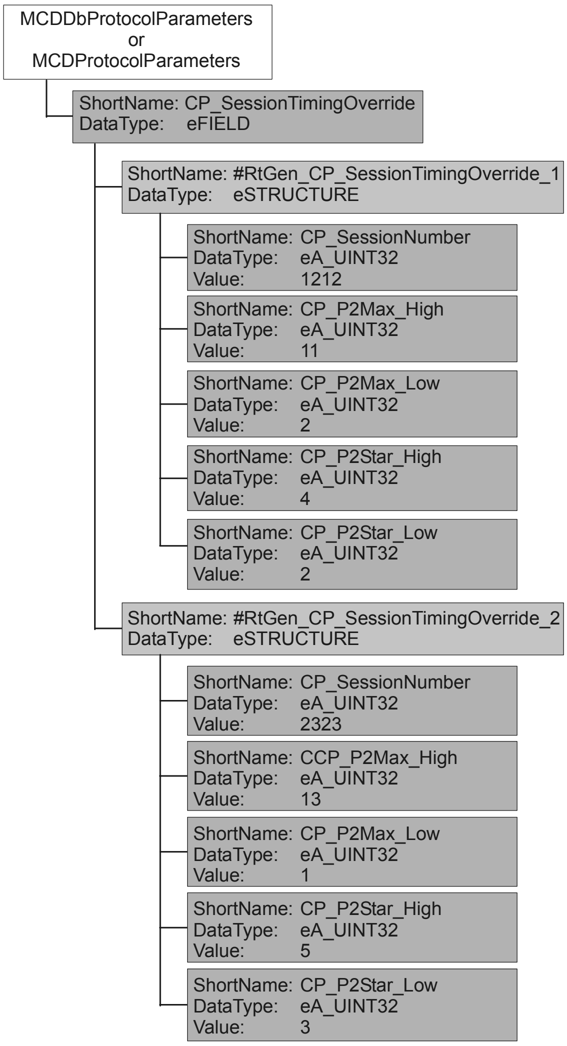


Figure 68 — Complex Comparam CP_SessionTimingOverride at MCD-3D level

```
<COMPARAM-REFS>
    <COMPARAM-REF ID-REF="ISO_15765_3.CP_SessionTimingOverride">
        <COMPLEX-VALUE>
            <SIMPLE-VALUE>1212</SIMPLE-VALUE>
            <SIMPLE-VALUE>11</SIMPLE-VALUE>
            <SIMPLE-VALUE>2</SIMPLE-VALUE>
            <SIMPLE-VALUE>4</SIMPLE-VALUE>
            <SIMPLE-VALUE>2</SIMPLE-VALUE>
        </COMPLEX-VALUE>
    </COMPARAM-REF>
    <COMPARAM-REF ID-REF="ISO_15765_3.CP_SessionTimingOverride">
        <COMPLEX-VALUE>
            <SIMPLE-VALUE>2323</SIMPLE-VALUE>
            <SIMPLE-VALUE>13</SIMPLE-VALUE>
            <SIMPLE-VALUE>1</SIMPLE-VALUE>
            <SIMPLE-VALUE>5</SIMPLE-VALUE>
            <SIMPLE-VALUE>3</SIMPLE-VALUE>
        </COMPLEX-VALUE>
    </COMPARAM-REF>
</COMPARAM-REFS>
```

The ODX definition of the complex protocol parameter CP_SessionTimingOverride used by the example above looks as follows:

```
<COMPLEX-COMPARAM ID="ISO_15765_3.CP_SessionTimingOverride" CPTYPE="OPTIONAL" CPUSAGE="TESTER" PARAM-
     CLASS="TIMING" ALLOW-MULTIPLE-VALUES="true">
     <SHORT-NAME>CP_SessionTimingOverride</SHORT-NAME>
     <COMPARAM ID="ISO_15765_3.CP_SessionNumber" CPTYPE="OPTIONAL" CPUSAGE="TESTER" PARAM-CLASS="TIMING">
         <SHORT-NAME>CP_SessionNumber</SHORT-NAME>
         <PHYSICAL-DEFAULT-VALUE>0</PHYSICAL-DEFAULT-VALUE>
         <DATA-OBJECT-PROP-REF ID-REF="ISO_15765_3.DOP_IDENTICAL_16Bit"/>
     </COMPARAM>
     <COMPARAM ID="ISO_15765_3.CP_P2Max_High" CPTYPE="OPTIONAL" CPUSAGE="TESTER" PARAM-CLASS="TIMING">
         <SHORT-NAME>CP_P2Max_High</SHORT-NAME>
         <PHYSICAL-DEFAULT-VALUE>0</PHYSICAL-DEFAULT-VALUE>
         <DATA-OBJECT-PROP-REF ID-REF="ISO_15765_3.DOP_IDENTICAL_8Bit_1ms"/>
     </COMPARAM>
     <COMPARAM ID="ISO_15765_3.CP_P2Max_Low" CPTYPE="OPTIONAL" CPUSAGE="TESTER" PARAM-CLASS="TIMING">
         <SHORT-NAME>CP_P2Max_Low</SHORT-NAME>
         <PHYSICAL-DEFAULT-VALUE>0</PHYSICAL-DEFAULT-VALUE>
         <DATA-OBJECT-PROP-REF ID-REF="ISO_15765_3.DOP_IDENTICAL_8Bit_1ms"/>
     </COMPARAM>
     <COMPARAM ID="ISO_15765_3.CP_P2Star_High" CPTYPE="OPTIONAL" CPUSAGE="TESTER" PARAM-CLASS="TIMING">
         <SHORT-NAME>CP_P2Star_High</SHORT-NAME>
         <PHYSICAL-DEFAULT-VALUE>0</PHYSICAL-DEFAULT-VALUE>
         <DATA-OBJECT-PROP-REF ID-REF="ISO_15765_3.DOP_LINEAR_8Bit_Resolution_10ms"/>
     </COMPARAM>
     <COMPARAM ID="ISO_15765_3.CP_P2Star_Low" CPTYPE="OPTIONAL" CPUSAGE="TESTER" PARAM-CLASS="TIMING">
         <SHORT-NAME>CP_P2Star_Low</SHORT-NAME>
         <PHYSICAL-DEFAULT-VALUE>0</PHYSICAL-DEFAULT-VALUE>
         <DATA-OBJECT-PROP-REF ID-REF="ISO_15765_3.DOP_LINEAR_8Bit_Resolution_10ms"/>
     </COMPARAM>
  <COMPLEX-PHYSICAL-DEFAULT-VALUE></COMPLEX-PHYSICAL-DEFAULT-VALUE>
</COMPLEX-COMPARAM>
…..
<DATA-OBJECT-PROP ID="ISO_15765_3.DOP_IDENTICAL_8Bit_1ms">
     <SHORT-NAME>DOP_IDENTICAL_8Bit_1ms</SHORT-NAME>
     <LONG-NAME>DOP_IDENTICAL_8Bit_1ms</LONG-NAME>
     <COMPU-METHOD>
         <CATEGORY>IDENTICAL</CATEGORY>
     </COMPU-METHOD>
     <DIAG-CODED-TYPE BASE-DATA-TYPE="A_UINT32" xsi:type="STANDARD-LENGTH-TYPE">
         <BIT-LENGTH>8</BIT-LENGTH>
     </DIAG-CODED-TYPE>
     <PHYSICAL-TYPE BASE-DATA-TYPE="A_UINT32"/>
     <UNIT-REF ID-REF="ISO_15765_3.ms"/>
</DATA-OBJECT-PROP>
<DATA-OBJECT-PROP ID="ISO_15765_3.DOP_LINEAR_8Bit_Resolution_10ms">
     <SHORT-NAME>DOP_LINEAR_8Bit_Resolution_10ms</SHORT-NAME>
     <LONG-NAME>DOP_LINEAR_8Bit_Resolution_10ms</LONG-NAME>
     <COMPU-METHOD>
         <CATEGORY>LINEAR</CATEGORY>
         <COMPU-INTERNAL-TO-PHYS>
             <COMPU-SCALES>
                 <COMPU-SCALE>
                     <COMPU-RATIONAL-COEFFS>
                         <COMPU-NUMERATOR>
                             <V>0</V>
```

```
                            <V>10</V>
                        </COMPU-NUMERATOR>
                        <COMPU-DENOMINATOR>
                            <V>1</V>
                        </COMPU-DENOMINATOR>
                    </COMPU-RATIONAL-COEFFS>
                </COMPU-SCALE>
            </COMPU-SCALES>
        </COMPU-INTERNAL-TO-PHYS>
    </COMPU-METHOD>
    <DIAG-CODED-TYPE BASE-DATA-TYPE="A_UINT32" xsi:type="STANDARD-LENGTH-TYPE">
        <BIT-LENGTH>8</BIT-LENGTH>
    </DIAG-CODED-TYPE>
    <PHYSICAL-TYPE BASE-DATA-TYPE="A_UINT32"/>
    <UNIT-REF ID-REF="ISO_15765_3.ms"/>
</DATA-OBJECT-PROP>
<DATA-OBJECT-PROP ID="ISO_15765_3.DOP_IDENTICAL_16Bit">
    <SHORT-NAME>DOP_IDENTICAL_16Bit</SHORT-NAME>
    <LONG-NAME>DOP_IDENTICAL_16Bit</LONG-NAME>
    <COMPU-METHOD>
        <CATEGORY>IDENTICAL</CATEGORY>
    </COMPU-METHOD>
    <DIAG-CODED-TYPE BASE-DATA-TYPE="A_UINT32" xsi:type="STANDARD-LENGTH-TYPE">
        <BIT-LENGTH>16</BIT-LENGTH>
    </DIAG-CODED-TYPE>
    <PHYSICAL-TYPE BASE-DATA-TYPE="A_UINT32"/>
</DATA-OBJECT-PROP>
```

In ODX, protocol parameters are classified into three different categories:

— MCDProtocolParameterClass (ODX: PARAM-CLASS);

— MCDProtocolParameterType (ODX: CPTYPE);

— MCDProtocolParameterUsage ODX: CPUSAGE).

The enumeration MCDProtocolParameterClass comprises the following items:

— eBUSTYPE: This class of parameters is used to define bus type specific parameters (e.g. baud rate). Most of these parameters affect the physical hardware. These parameters can only be modified by the first Logical Link that acquired the physical resource. When a second Logical Link is created for the same resource, these parameters that were previously set will be active for the new Logical Link.

— `eCOM`: General communication parameters.

— `eERRHDL`: Parameter defining the behaviour of the runtime system in case an error occurred, e.g. the runtime system could either continue communication after a timeout was detected, or stop and reactivate the communication link.

— `eINIT`: Parameters for initiation of communication e.g. trigger address or wakeup pattern. These parameters shall not be overwritten within an ECU-Variant layer in any way.

— `eTESTER_PRESENT`: This type of communication parameter is relevant for the various aspects of tester present functionality, e.g. they determine tester present timeout settings or tester present message contents.

— `eTIMING`: Message flow timing parameters, e.g. inter-byte-time or time between request and response.

- `eUNIQUE_ID`: This type of communication parameter is used to indicate to both the ComLogicalLink and to the application that the information is used for protocol response handling from a physical or functional group of ECUs to uniquely identify an ECU response.

The enumeration `MCDProtocolParameterType` comprises the following items:

- `eSTANDARD`: A communication parameter belonging to a standardized protocol that shall be supported by a runtime system implementing this standardized protocol. Diagnostic data using a protocol not supported by the runtime system cannot be executed by the D-server.

- `eOPTIONAL`: This communication parameter does not have to be supported by the runtime system. If a DIAG-COMM uses an unsupported communication parameter of this type, the parameter can be ignored and the DIAG-COMM can be executed nevertheless.

- `eOEM-SPECIFIC`: The communication parameter is part of a non-standardized OEM-specific protocol; nevertheless it is required to be implemented by the runtime system. Diagnostic data using an OEM-specific protocol not supported by the runtime system cannot be executed by the D-server.

- `eOEM-OPTIONAL`: This communication parameter is a non-standardized, OEM-specific parameter that does not have to be supported by the protocol layer implementation of the runtime system (D-PDU API).

- As stated above, DiagComPrimitives with associated protocol parameters of type STANDARD or OEM-SPECIFIC will not be executed by the D-server if one or more of these protocol parameters are not supported by the protocol driver (D-PDU API). In this case, a `MCDCommunicationException` of type `eCOM_COMPARAM_NOT_SUPPORTED` will be thrown when the client application tries to execute such a DiagComPrimitive. The same applies to the method `MCDDLogicalLink::gotoOnline()` if at least one of the protocol parameters to be set is not supported by the protocol driver.

The enumeration MCDProtocolParameterUsage comprises the following items:

- `eECU-COMM` parameters are relevant for basic properties of the communication channel with an ECU (e.g. timings and addresses).

- `eECU-SOFTWARE` parameters are only used for ECU software generation and configuration. Communication parameters of this type shall be ignored by the D-server.

- `eAPPLICATION` parameters are only evaluated by the client application. Communication parameters of this type shall not be passed down to the D-PDU API by the D-server, but they shall be accessible via the D-server API.

- `eTESTER` parameters are only valid to the tester during diagnostic communication; they are not relevant to the use case of ECU software generation and calibration.

The DISPLAY-LEVEL of COMPARAMs is used to restrict the visibility (and therefore changeability) of the COMPARAMs in a client application. Therefore the D-server only delivers the DISPLAY-Level to the client application, which shall implement any functionality based on display level information.

9.4.10.3 Inheritance of protocol parameters

During runtime a diagnostic layer is linked with a PROT-STACK to import a valid set of communication parameters (COMPARAM and COMPLEX-COMPARAM elements). This unambiguous link is defined via the PROTOCOL or the LOGICAL-LINK. The COMPARAM values are inherited between the different layers. An inherited COMPARAM value is identical to that in the parent layer, i.e. it has the same value. A communication parameter shall be overridden to change its value.

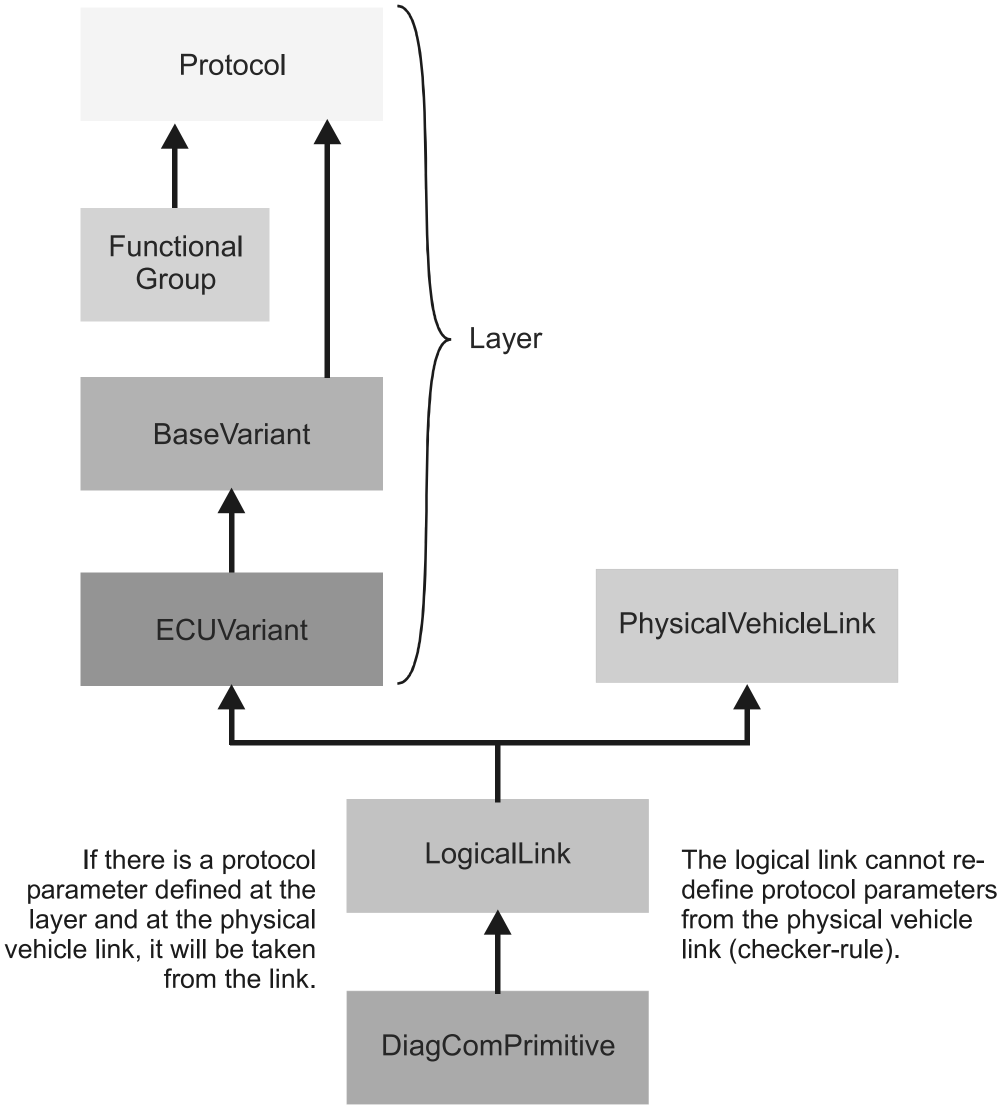

Protocol
Functional Group
BaseVariant
ECUVariant
Layer
PhysicalVehicleLink
LogicalLink
DiagComPrimitive
If there is a protocol parameter defined at the layer and at the physical vehicle link, it will be taken from the link.
The logical link cannot re-define protocol parameters from the physical vehicle link (checker-rule).

Figure 69 — Inheritance of COMPARAMs between different layers

Protocol parameters can be overwritten at different layers to change their value. The following issues shall be considered:

— Overwritten communication parameter values on a FUNCTIONAL-GROUP level are not inherited by lower layers (i.e. the inheriting base variants).

— Base variants inherit their set of communication parameter values directly from the protocol layer, and can then override them with locally defined values.

— Since a logical link always includes one dedicated protocol, all multiple inheritance issues can be resolved unambiguously at runtime.

A simplified inheritance example for simple communication parameters without redefinition of communication parameters at Physical Vehicle Link and Logical Link is illustrated in Figure 70 — UML representation of inheritance of communication parameters (example), which shows a PROTOCOL instance (PROT-A) referencing the PROT-STACK instance PS-B as the active PROT-STACK. As a consequence, the valid set of communication parameters consists of A, B and C. Because COMPARAM A value is overridden within the protocol layer, the valid values in the scope of PROT-A are A=4, B=5 and C=15.

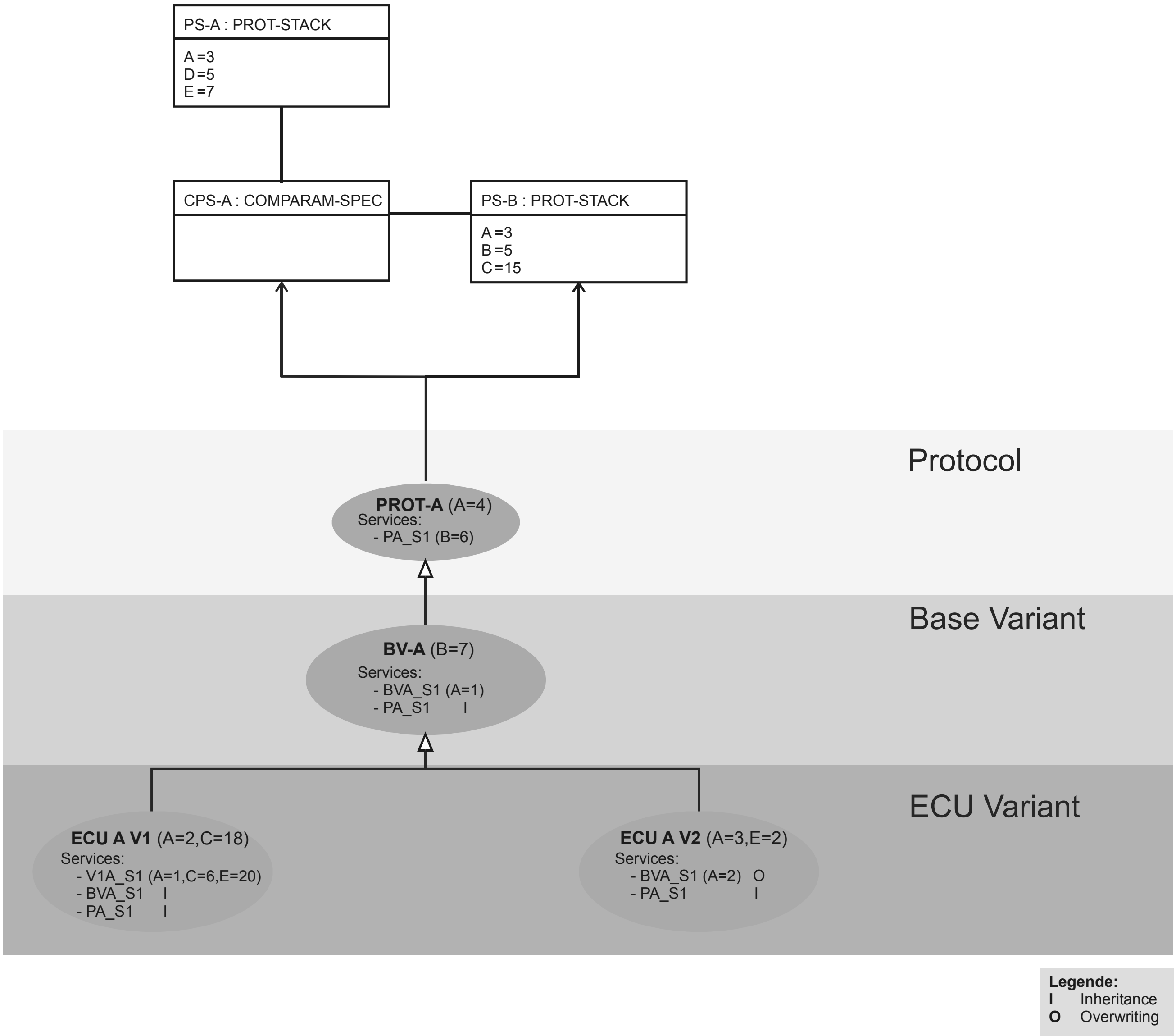


Figure 70 — UML representation of inheritance of communication parameters (example)

These values apply to all the DIAG-COMM that do not override a communication parameter value themselves. The DIAG-SERVICE with the identifier PA_S1 in the PROTOCOL PROT-A shown in the example overrides the COMPARAM B value. This is done within the data by a COMPARAM-REF element. The COMPARAM set for this special DIAG-SERVICE results in [A=4, B=6, C=15].

The BASE-VARIANT instance BV-A overrides the COMPARAM B value. All other COMPARAM values are derived from the protocol layer PROT-A. The COMPARAM set in the scope of BV-A is [A=4, B=7, C=15]. Two DIAG-COMMs are available in this base variant layer. The DIAG-SERVICE BVA_S1 defined within BV-A and the DIAG-SERVICE PA_S1 derived from the parent layer PROT-A. The COMPARAM set for PA_S1 at the base variant level is [A=4, B=6, C=15] and COMPARAM set for BVA_S1 is [A=1, B=7, C=15].

There are two instances of an ECU-VARIANT both inherited from BV-A. For ECU-A-V1 the COMPARAM set is [A=2, B=7, C=18]. There are three DIAG-SERVICEs available at ECU-A-V1. The inherited PA_S1 with [A=2, B=6, C=18], the inherited BVA_S1 with [A=1, B=7, C=18] and locally defined V1A_S1 with [A=1, B=7, C=6]. Because the COMPARAM E is not part of the current communication parameter set the COMPARAM-REF element at V1A_S1 that tries to override the COMPARAM E value exclusively for the DIAG-SERVICE has no affect and can be ignored. The ECU-VARIANT instance ECU-A-V2 applies the COMPARAM values [A=3, B=7, C=15]. E is ignored because it is not part of the active PROT-STACK. The available DIAG-SERVICEs are the value inherited PA_S1 with [A=3, B=6, C=15] and the locally defined BVA_S1 which overrides the BVA_S1 defined at the layer above. The COMPARAM set for BVA_S1 in the scope of ECU-A-V2 is [A=2, B=7, C=15].

9.4.10.4 Database part

This subclause describes how the ODX structures can be retrieved at the D-server API. At a MCDDbLocation, the method getDbProtocolStacks() will deliver a collection of all MCDDbProtocolStack objects, which are associated with the location (ODX: layer). A PROTOCOL location is associated with all protocol stacks defined within a COMPARAM-SPEC if the protocol stack is not explicit referenced by the PROTOCOL location. Additionally a BASE-VARIANT location might inherit from more than one PROTOCOL locations. Therefore getDbProtocolStacks() might return multiple protocol stack objects. In case a BASE-VARIANT location inherits from only one PROTOCOL location and the PROTOCOL location specifies the used protocol stack by an explicit reference, the method getDbProtocolStacks()returns exactly one protocol stack object. Every protocol stack holds a valid set of MCDDbProtocolParameters. In addition MCDDbProtocolStack::getDbProtocolType returns the protocol type as an ASCIISTRING with the format <ApplicationLayerName>_on_<TransportLayerName> as defined in ISO 22900-2:2009, Clause B.1.

A method getDbProtocolParameters() returns all overwritten and inherited protocol parameters which are available at the current MCDDbLocation.

In accordance with the example above (see Figure 70 — UML representation of inheritance of communication parameters (example)) a method call getProtocolStacks()::getItemByIndex(0)::getDbProtocolParameters() at the MCDDbLocation ECUV-A would result the protocol parameter set [A=2, B=7, C=18]. A method call getProtocolStack()::getDbProtocolParameters() at the MCDDbLogicalLink for ECUV-A would result the same protocol parameter set if no protocol parameter is redefined at the Logical Link and the associated Physical Vehicle Link.

The method getDbProtocolParametersByUsage(MCDProtocolParameterUsage) at the MCDDbProtocolStack can be used to retrieve only protocol parameters for a certain usage type. On the database side, all different types of CPUSAGE protocol parameters are accessible, but at runtime only COM-PARAMS with usage types ECU_COMM and TESTER will be delivered to the client application. Communication parameters with the usage type APPLICATION will be handled as read-only information.

The different categories of protocol parameters (see 9.4.10.2) can be accessed as follows:

— MCDDbProtocolParameter::getProtocolParameterClass() delivers an enumeration of type MCDProtocolParameterClass

— MCDDbProtocolParameter::getProtocolParameterType() delivers an enumeration of type MCDProtocolParameterType

— MCDDbProtocolParameter::getProtocolParameterUsage() delivers an enumeration of MCDProtocolParameterUsage

9.4.10.5 Runtime part

Local and global protocol parameters

At runtime, it is distinguished between “local” protocol parameters at diagnostic services or certain control primitives (MCD(Db)DiagService, MCD(Db)StartCommunication and MCD(Db)StopCommunication), and “global” protocol parameters which apply to entire Logical Links. For the ease of reading, we will refer to DiagComPrimitives in the remainder of this subclause instead of explicitly stating that protocol parameters are only available for elements of type MCD(Db)DiagService, MCD(Db)StartCommunication and MCD(Db)StopCommunication.

The local protocol parameter collection of a DiagComPrimitive is a subset of all protocol parameters of the related Logical Link. The values of all Logical Link protocol parameters that are temporarily valid for a DiagComPrimitive at execution time are determined through the rules described in the following paragraphs.

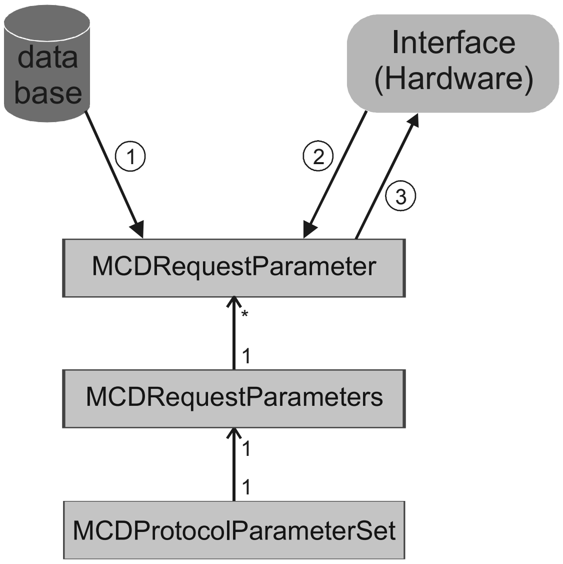
In general, a DiagComPrimitive is executed in the context of the protocol parameter values that are currently valid at the Logical Link – except when the local protocol parameters collection of the DiagComPrimitive is not empty; in this case, the value of each local protocol parameter overwrites the value of the corresponding global protocol parameter. The D-server shall set the DiagComPrimitive specific protocol parameters before executing the DiagComPrimitive, and shall restore the original protocol parameter settings afterwards. These temporary changes of protocol parameters are only valid during this DiagComPrimitive's execution time. All other DiagComPrimitives are executed in their own context of protocol parameters consisting of the protocol parameters currently valid at the Logical Link and their own temporarily overwritten protocol parameters.

Global protocol parameters are set at the logical link by using the `MCDProtocolParameterSet` control primitive.

Local protocol parameters can be obtained from a DiagComPrimitive by using the method `MCDDiagService::getProtocolParameters()`. The D-server will only set those protocol parameters locally which are members of this collection. The server initializes the collection with those parameters that are overwritten for the DiagComPrimitive in the database. By adding protocol parameters to this collection (`MCDProtocolParameters::addByXXX(...)`), changing the values of protocol parameters in the collection (`MCDProtocolParameter::setValue(…)`), or by removing protocol parameters from this collection (`MCDProtocolParameter::removeXXX(…)`), an application can control which local protocol parameters are overwritten by the server for execution of the DiagComPrimitive.

DiagComPrimitives of type `MCDProtocolParameterSet` are not handled differently from other DiagComPrimitives. That is, these ControlPrimitives can only be executed in Logical Link states `eONLINE` and `eCOMMUNICATION`. If a Logical Link needs to be prepared via protocol parameters before going into communication, the client application shall take care to execute the proper `MCDProtocolParameterSet` prior to all other DiagComPrimitives, especially `MCDStartCommunication`.

The protocol parameters for a Logical Link are set via the `MCDProtocolParameterSet::executeSync()` method. The effect is a global change of the protocol parameters of the link the `MCDProtocolParameterSet` primitive was created on. It is not allowed to execute a ControlPrimitive of type `MCDProtocolParameterSet` in case at least one repeated DiagComPrimitive is currently executed on the same Logical Link. In this case, an exception of type `MCDProgramViolationException` with error code `eRT_REPEATED_SERVICE_RUNNING` shall be thrown.



① MCDProtocolParameterSet::resetToDefaultValues()
or
MCDProtocolParameterSet::resetToDefaultValue(parameterName)

② MCDProtocolParameterSet::fetchValuesFromInterface()
or
MCDProtocolParameterSet::fetchValueFromInterface(parameterName)

③ MCDProtocolParameterSet::executeSync()

Figure 71 — Relation between database and hardware for MCDRequestParameter

By means of the method `fetchValueFromInterface()` the current values from the interface will be set to the RequestParameters of the `MCDProtocolParameterSet` primitive.

Behaviour in case of non-exclusive setting of Protocol Parameters:

- The D-server binds the current protocol parameters (current parameters in the interface, plus temporary parameters from ODX) to the DiagComPrimitive during its execution. This binding is valid as long as the DiagComPrimitive is active (the Logical Link the DiagComPrimitive is executed on is in state MCDActivityState::eACTIVITY_RUNNING, where the state change from an to eACTIVITY_IDLE has been caused by the execution of the DiagComPrimitive).

- If the D-server, the protocol engine and the ECU allow nested/parallel execution of DiagComPrimitives, each DiagComPrimitive may have its own set of protocol parameters, and they are treated in the same way as above.

- If these conditions are valid, the setting of protocol parameters can be queued.

- The ProtocolParameterSet control primitive can only be executed synchronously.

When using `MCDProtocolParameterSet::executeSync()`, only those communication parameters will be set which have to be changed at the protocol layer (D-PDU API). That is, the ProtocolParameterSet will always be sent as an incomplete set (set of parameter to be changed) of communication parameters to the protocol layer (D-PDU API).

The D-server maintains the mirror values of the protocol parameters of every Logical Link, and keeps track of all changes to the values since the last execution of the MCDProtocolParameterSet control primitive. On the next execution of MCDProtocolParameterSet control primitive, only the changed values will be written to the interface.

There are two ways for the D-server to deal with protocol parameters that are not supported by its D-PDU API implementation. If the setting of a protocol parameter fails, the D-server can continue operating and executing diagnostic services on the communications link that caused the failure. This behavior is most useful in engineering environments, where bus topologies and diagnostic data sets are still under development. Alternatively, the D-server can refuse the execution of diagnostic services on a link when the setting of a protocol parameter has not been successful, as could be appropriate in more restricted environments (e.g. an after sales workshop), to ensure consistent behavior and functionality of vehicle diagnostics.

It is up to the client application whether to use OEM specific protocol parameters. With the method MCDSystem::isUnsupportedComParametersAccepted() it can be determined if the D-server will be accepting non-standard protocol parameters. The method MCDDLogicalLink::unsupportedComParametersAccepted(A_BOOLEAN)can be used to forbid the usage of OEM-specific protocol parameters for this specific Logical Link, but this will only have an effect if protocol parameters are accepted system wide; It is not possible to allow non-standard protocol parameters for a logical link when they are disallowed globally.

For a D-server implementation, this has the following consequences: as the MCDProtocolParameterSet control primitive only allows the client application to set all protocol parameters that are valid for a link in one single operation, the D-server shall provide protocol parameter consistency analogous to the transaction concept of a database system.

For example, imagine the case when a client application executes a MCDProtocolParameterSet primitive on a logical link, which includes five protocol parameters. For each of these five parameters, the D-server in turn shall call a D-PDU API method, telling the protocol driver layer to set the desired value for each respective protocol parameter. Now, the setting of the third protocol parameter fails because it is not supported by the protocol driver. In case the D-server has been configured to accept unsupported protocol parameters, it can continue with setting the remaining two parameters and then use that particular link for doing diagnostic communication.

However, if the D-server is configured to not accept unsupported protocol parameters, it can’t simply abort the execution of the MCDProtocolParameterSet at this point and return the corresponding error to the client application, because two of the five protocol parameters have already been successfully modified. Instead, it first shall reset the first two protocol parameters to their original values, so that the value set of protocol parameters of that link remains in a consistent state. This implies that in this case, the D-server shall cache the original values of all protocol parameters that will be modified by a MCDProtocolParameterSet primitive, so it can undo any modifications in case the setting of any parameter fails.

NOTE This kind of consistency also is required in case protocol parameters are set implicitly by the D-server, e.g. when opening a new logical link, or when executing a MCDDiagComPrimitive that as associated overwritten protocol parameters.

The class MCDProtocolParameterSet inherits from class MCDControlPrimitive. Therefore, the execution of a MCDProtocolParameterSet primitive needs to generate an appropriate response object. This response object is empty (on the runtime as well as on the database side), that is the response collection of the MCDProtocolParameterSet has zero entries. Instead, the result can be obtained from the result collection, which can deliver the error code eRT_PROTOCOLPARAMETERSET_FAILED if the execution of a MCDProtocolParameterSet failed.

The Execution states in case of MCDProtocolParamterSet are as follows:

— eALL_NEGATIVE: Setting of protocol parameters failed for at least one protocol parameter.

— eALL_POSITIVE: Setting of protocol parameters succeeded for at all protocol parameters.

MCDDbProtocolParameterSet is a runtime-generated object. Therefore the following methods

- getDbRequest()::getDbRequestParameters()
- getDbResponses()
- getDbResponsesByType(…)

all deliver an empty collection.

MCDProtocolParameterSet

If a Java-Jobs needs to alter the currently valid protocol parameters of the Logical Link, it should use and execute a MCDProtocolParameterSet from within its code.

NOTE1 All changes to protocol parameters caused by a MCDProtocolParameterSet executed within a Java-Job will be persistent after this job has terminated – just as if a client application would have issued the changes. So, a clean Job implementation shall restore the protocol parameters at the end of the Job execution.

NOTE2 The usage of MCDProtocolParameterSet in a Java-Job is considered harmful, as it could cause undocumented and therefore unexpected changes to the protocol parameters of a logical link at runtime.

9.4.11 Suppress Positive Response

The suppress positive response feature allows to ask the ECU not to send any positive response to the request of the current DiagComPrimitive. This allows to decrease the load of communication bus, protocol layer, and MCD-server. However, a negative response may be sent by the ECU any time. In addition, the ECU may send a positive response after it had sent a response pending as first answer just before.

If the suppress positive response feature is enabled for an MCDService or MCDStart/Stop Communication primitive, the "suppress positive response" bit in the protocol data stream is set by the protocol layer, e.g. the D-PDU API, if applicable. In an MCD-server implementation, this is achieved by setting the corresponding flags in the communication data structures it passes to the protocol layer. If the protocol layer is a D-PDU API, bit 6 in byte 15 of the PDU header structure needs to be set (see ISO 22900-2 for more information).

The D-server internally applies the bit-mask given in the ODX data to the referenced request parameter (see corresponding ODX structure) before the PDU which describes the service is passed to the D-PDU API for sending.

The solution proposed above has the major benefit that it is independent of the concrete protocol. As a result, the major goal of designing a protocol independent D-server has been met. Furthermore, the solution is also capable of handling future versions of the suppress positive response feature where the bit mask manipulates several non-contiguous bits in possibly multiple contiguous bytes. Finally, this solution does not require to inspect all parameters of a MCDService to find out whether there is a parameter for switching on and off the suppress positive response feature at a service. Instead, this can be directly obtained from the MCDService-object itself.

The suppress positive response feature is not used on level MCDHexServices and MCDDynIdxxxComPrimitives. With MCDHexServices being plain services not controlled by the D-server but by the application, the suppress positive response feature can be used independent of the server. However, as with all other MCDHexServices the application needs to have all the knowledge and all the control structures to react to every possible answer to a MCDHexService by the server, e.g. response message, exception, and timeout. Removing the suppress positive response support from the API for MCDHexServices just prevents inconsistencies between the server internal status and the data contained in a MCDHexService's plain PDU. Furthermore, it prevents the server from interpreting MCDHexService-PDUs to overcome this deficit.

NOTE Interpreting PDUs of a MCDHexService would impose protocol dependencies into the server, which is out of scope by design rule.

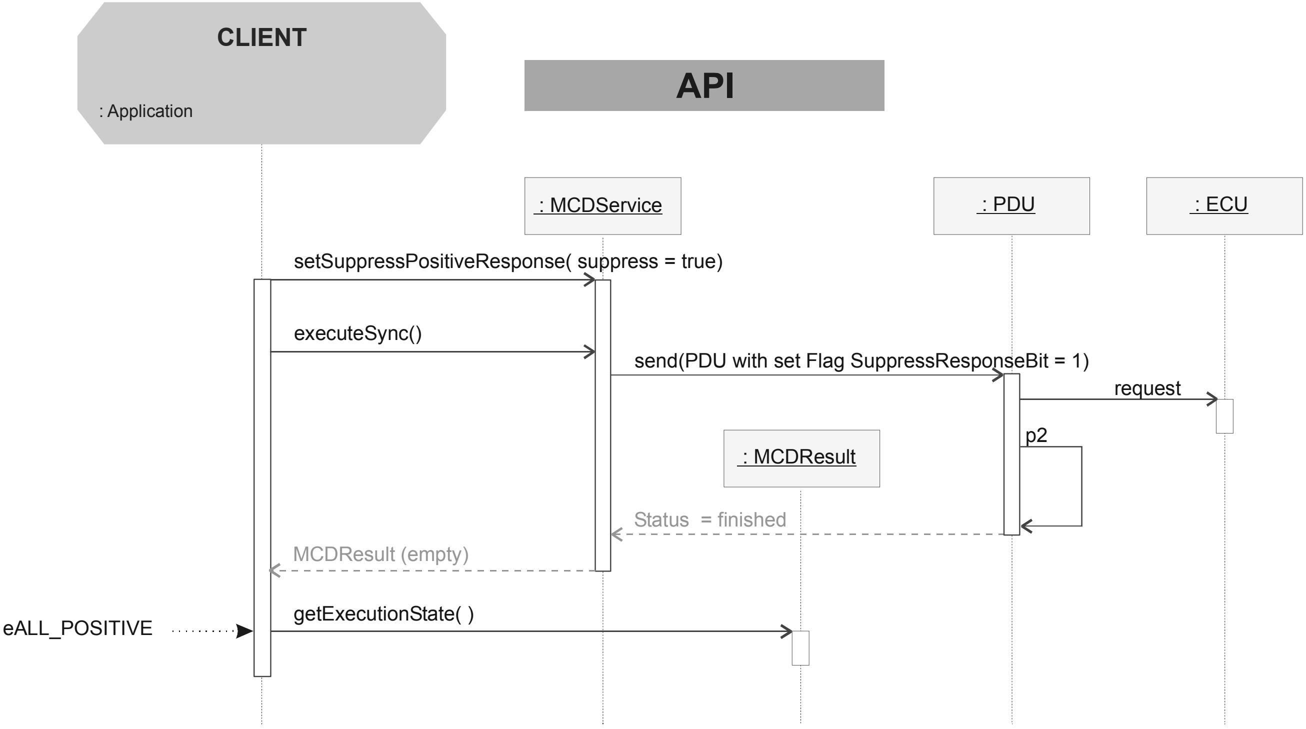


Figure 72 — Suppress Positive Response behaviour (1)

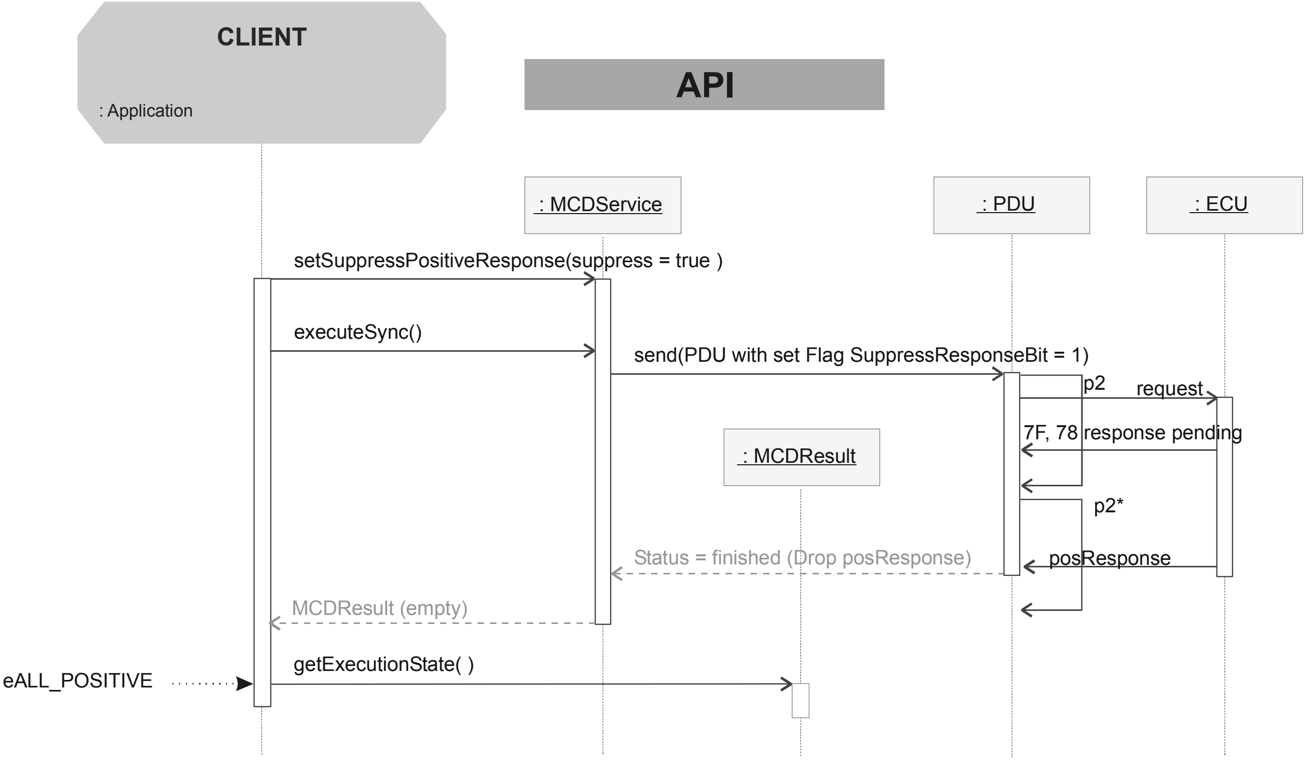


Figure 73 — Suppress Positive Response behaviour (2)

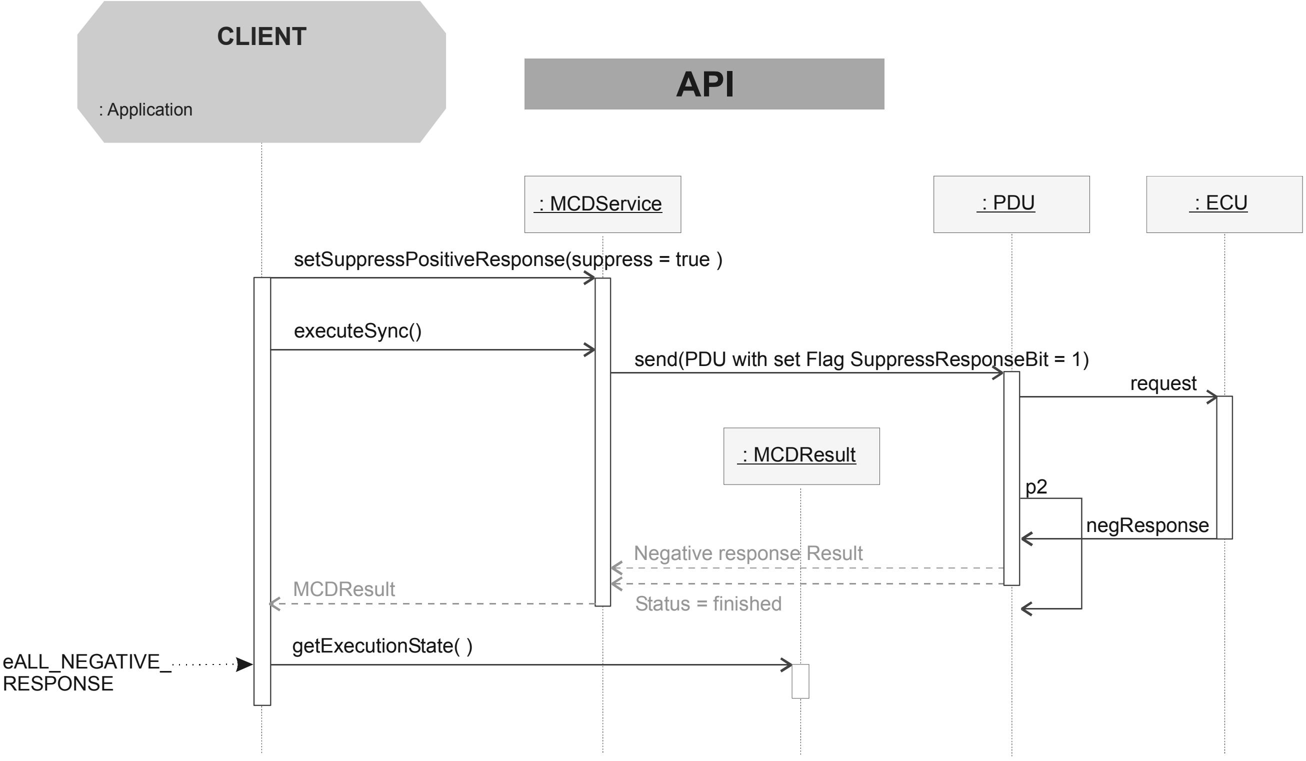


Figure 74 — Suppress Positive Response behaviour (3)

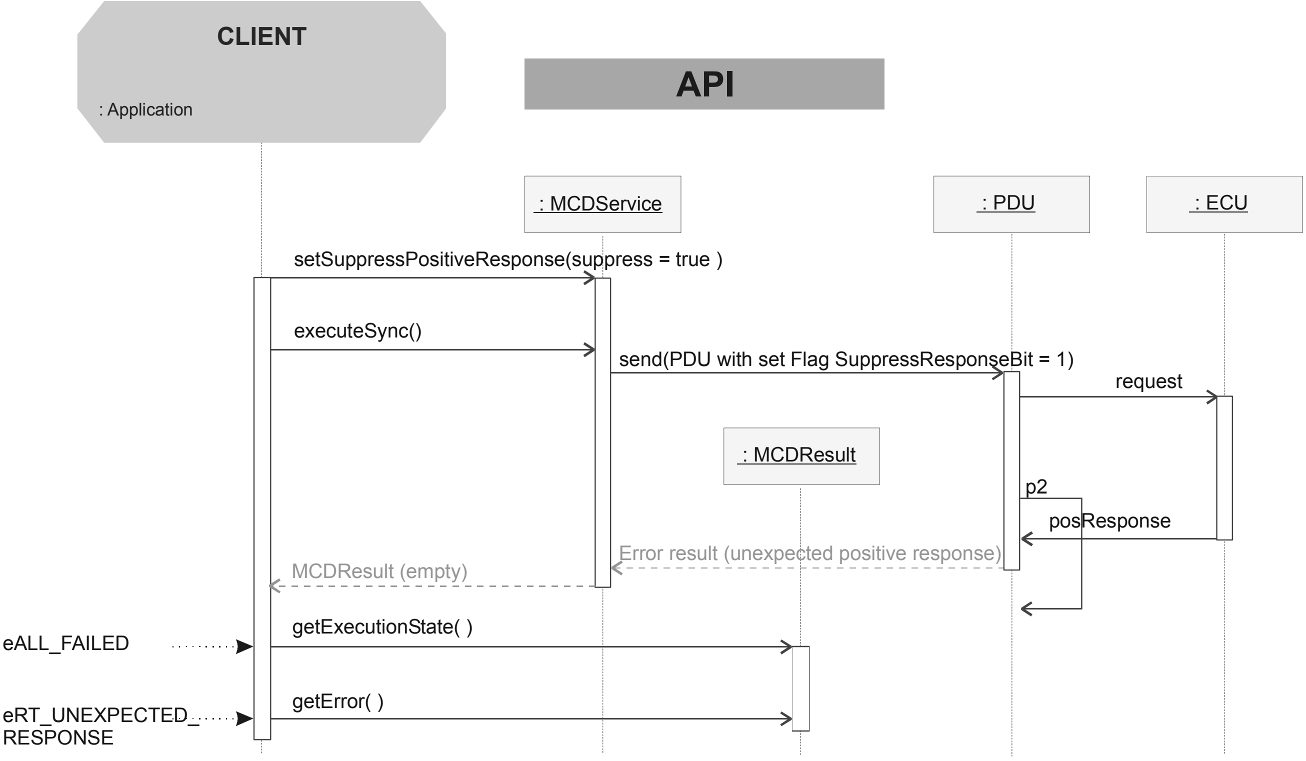
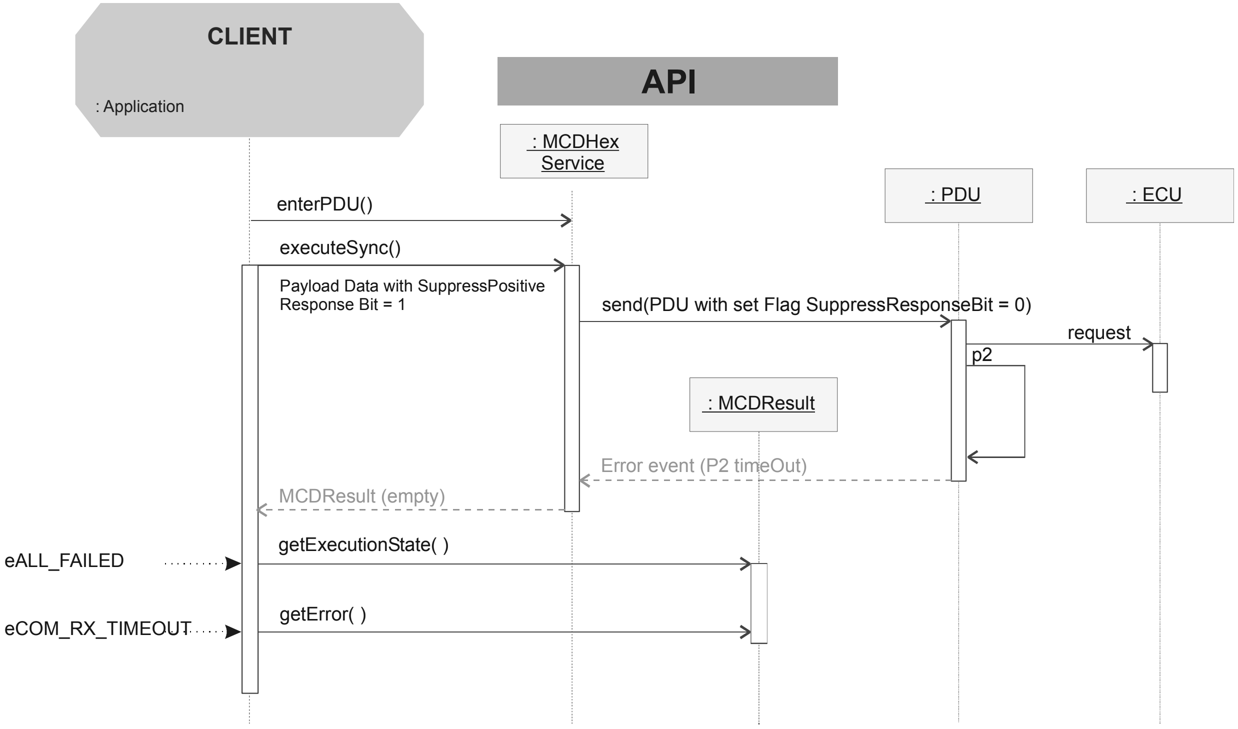


Figure 75 — Suppress Positive Response behaviour (4)



This is the same behaviour as when an ECU does not respond to a Server with a common Diag service

Figure 76 — Suppress Positive Response behaviour (5)

In case an ECU answers with a positive response without response pending to a request with positive response suppression the execution state of the response shall be eALL_FAILED or eFAILED. In case an ECU does not answer (in time) result with the following error will be retuned: eCOM_RX_TIMEOUT.

9.5 Diagnostic variables

In this subclause, the usage of diagnostic variables (diag-variables) is described. The support for diag-variables is optional for a D-server. As described below, only diag-variables of relation type READ are supported.

In the measurement and calibration (MC) use case, the tester has direct access to the ECU memory to read and write data. The ECU provides internal variables to simplify this memory access. An ECU-internal variable maps a logical variable identifier (also called "label") to the memory address of the variable value. This mapping is part of the ECU software; thus, an ECU-internal variable is also called "SW-variable".

In the diagnostic use case (D), another communication paradigm is preferred. Here, diagnostic data is typically transferred via diagnostic services or diagnostic jobs that read/write data from/to the ECU. The ECU memory is not accessed directly.

A runtime system should be capable of utilizing SW-variables as described in the MC use case in the same way as in the diagnostic use case. For this reason the notion of diagnostic variables is introduced in this part of ISO 22900. A diag-variable emulates a SW-variable by calling appropriate diagnostic services for operations like reading or writing a value from/to the ECU.

Because the result of a service can include more than one parameter, the parameter representing the diag-variable shall be extracted from the service response by the D-server.

To enable diag-variables, the following principles are used:

- Provision of an access mechanism to read out all the diag-variables available.
- Usage of a MCDService with a predefined ResultFilter to extract the specific diag-variable.

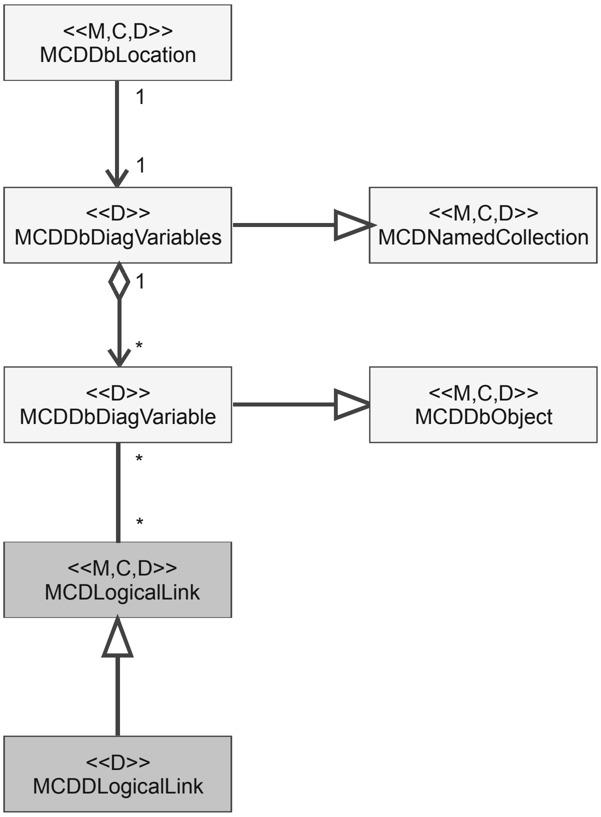


Figure 77 — ERD for diagnostic variables

In this part of ISO 22900, diag-variables are described by the `MCDDbDiagVariable` class. The diag-variables that are defined for a location can be retrieved using the method `MCDDbLocation::getDbDiagVariables()`. A diag-variable has associated attributes describing the usage patterns of the variable. These attributes can be accessed from the class `MCDDbDiagVariable`. These attributes are the variable's collection of `RelationTypes`, its `ValueType`, and a flag determining whether it has the `ReadBeforeWrite` option set in ODX.

The relation type is defined by ODX to be one of READ, WRITE, RETURNCONTROL, or any other value. That is why the relation type is of datatype `MCDDatatypeAsciiString` in the D-server API. Its purpose is to selectively associate a diag-variable with the service used for reading or writing of a diagnostic variable, respectively. As this part of ISO 22900 currently only supports filtering of response parameters, only READ relation type variables can be used by the client application.

The `ValueType` of a diag-variable is defined as an enumeration which contains the following values:

- eCURRENT
- eSTORED
- eSTATIC
- eSUBSTITUTED

The ValueType describes the storage and access strategy used by the D-server when the diag-variable is read or written. For more details, please refer to Table Enumeration "COMM-RELATION-VALUE-TYPE" in ASAM MCD 2 D ODX.

To use a diag-variable at runtime, the method MCDDiagComPrimitives::addDiagVariableServiceByRelationType(MCDDbDiagVariable dbDiagVar, A_ASCIISTRING relationType, MCDCooperationLevel cooperationLevel)) is used. It returns a MCDService object, which can subsequently be used to access the diag-variable. The service has a positive filter preconfigured to filter out all response parameters of the service, except for the one referenced through the diag-variable. The combination of RelationType (as string) and diag-variable always points to a specific service. For more than one service, the same combination is not allowed. Executing the service yields a result containing one response parameter. Since a diag-variable is retrieved through a regular diagnostic service, the result structure is the same as for a service:

- One diag-variable may have more than one result in case of repetitive or cyclic execution.
- Each result consists of one collection of responses which contains exactly one response.
- This response consists of one collection of response parameters which contains exactly one response parameter.
- This response parameter contains the actual value of the diag-variable.

9.6 eEND_OF_PDU as RequestParameter

9.6.1 Database side

If a MCDDbRequestParameter within the Collection of MCDDbRequestParameters has the data type eEND_OF_PDU, the minimum and maximum count of structures can be requested using the methods getMinLength() and getMaxLength(). If the values are not given in ODX, the default value 0 and respectively MAX_UINT32 is returned.

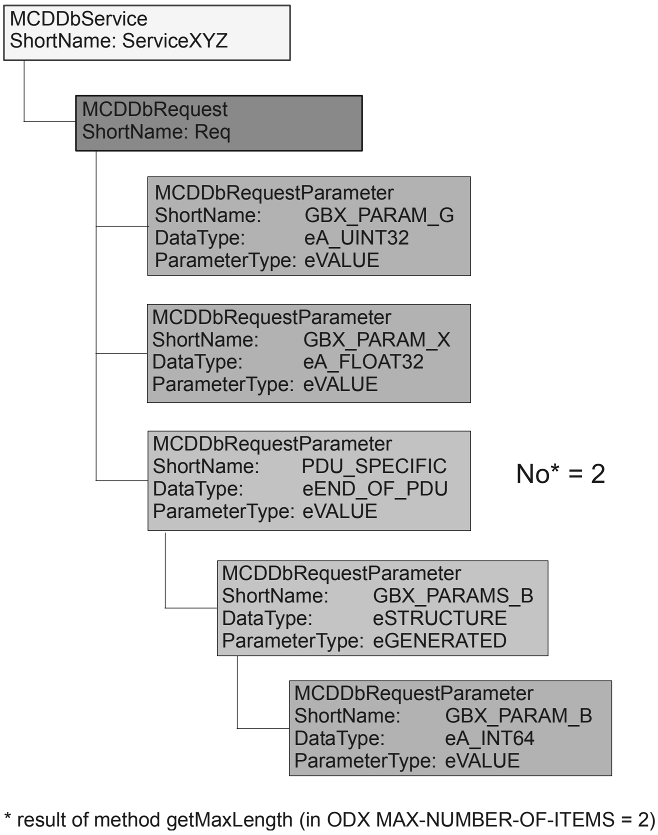


Figure 78 — RequestParameter eEND_OF_PDU on database side

ODX data for the database template

```
<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>
     <PARAM xsi:type="VALUE">
        <SHORT-NAME>GBX_Param_G</SHORT-NAME>
        (…)
     </PARAM>
     <PARAM xsi:type="VALUE">
        <SHORT-NAME>GBX_Param_X</SHORT-NAME>
        (…)
     </PARAM>
     <PARAM xsi:type="VALUE">
        <SHORT-NAME>PDU_Specific</SHORT-NAME>
        (…)
        <DOP-REF ID-REF=“EOP_DOP_ID“/>
     </PARAM>
</REQUEST>

<END-OF-PDU-FIELD ID=“EOP_DOP_ID“>
     <SHORT-NAME>EOP_DOP</SHORT-NAME>
     <BASIC-STRUCTURE-REF ID-REF=”GBX_PARAMS_B_ID”/>
     <MAX-NUMBER-OF-ITEMS>2</MAX-NUMBER-OF-ITEMS>
</END-OF-PDU-FIELD>

<STRUCTURE ID=”GBX_PARAMS_B_ID”>
     <SHORT-NAME>GBX_PARAMS_B</SHORT-NAME>
     <PARAMS>
        <PARAM xsi:type=”VALUE”>
             <SHORT-NAME>GBX_PARAM_B</SHORT-NAME>
             (…)
        </PARAM>
     </PARAMS>
</STRUCTURE>
```

9.6.2 Runtime side

By means of the method `MCDRequestParameter:addParameters (A_UINT32 count)` parameters can be inserted. If the type of the inserted MCDRequestparameter is not eEND_OF_PDU, the exception ePAR_MCD_NO_DYNAMIC_FIELD is thrown. If the parameter count plus the number of already existing parameters (`MCDRequestParameter::getRequestParameters()->getCount())` exceeds the value returned by `getMaxLength()`, the exception ePAR_VALUE_OUT_OF_RANGE is thrown.

After new sets of parameters (several calls `of addParameters()` can be executed in sequence) have been added, the method getParameters() should be used to fetch the whole set of parameters, and deliver a collection that contains an arbitrary number of such elements (zero number of eEND_OF_PDU elements is allowed too), for further processing (fill-in of values etc.)

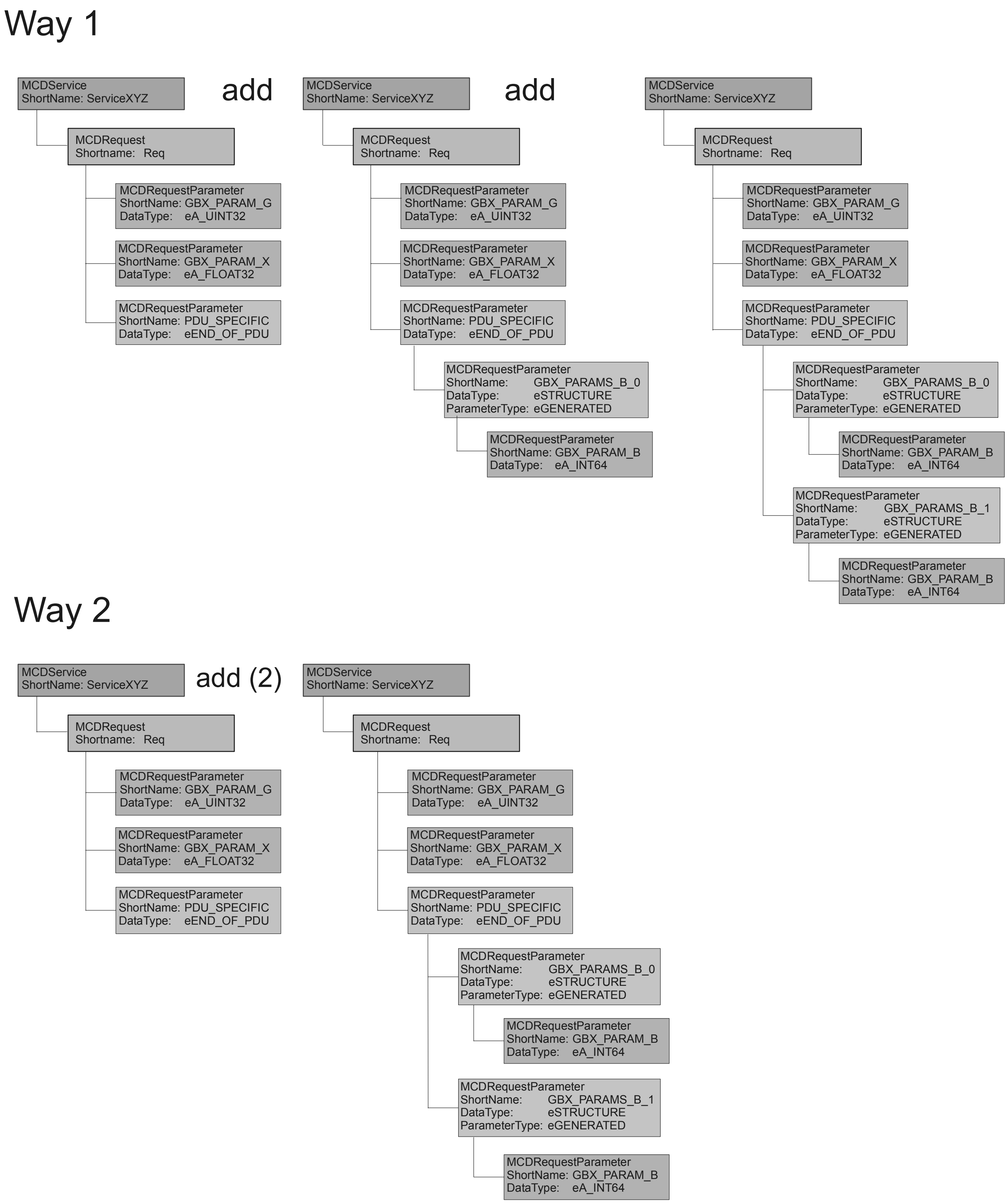


Figure 79 — RequestParameter eEND_OF_PDU on runtime side

9.6.3 COMPUCODE

ODX defines data and algorithms for transforming diagnostic values from their physical into their internal representation and vice versa. This structure is called COMPU-METHOD (see ODX-Spec for more information). There are different categories of COMPU-METHODs defined in ODX. These need to be implemented in the D-server internally. Here, COMPU-METHODs of category COMPUCODE are special. The transformation algorithm of these COMPU-METHODs is coded in the Java programming language and is part of the ODX data.

Only the following standard Java packages are allowed to be used in COMPUCODEs: java.lang; java.math; java.text and java.util. This limitation also applies to all user-defined classes collaborating with the COMPUCODE. These might be classes contained in a JAR file of the COMPUCODE class (if SYNTAX="JAR") or another JAR file added via a LIBRARY element.

A Java class with computational code shall implement the interface I_CompuCode (see Java package asam.d.compucode).

The implementation of Java methods for COMPUCODEs shall be stateless. As the interface definition for COMPOCODEs shows, no parameterization of the Java code is possible. The only way to signal an error condition inside the Java code is to throw an Exception. If a `MCDProgramViolationException` is thrown, a `MCDError` of type `eRT_COMPUCODE_FAILED` is attached to this exception by the D-server to the corresponding `MCDResponseParameter`. For request and response parameters, the mapping of the coded and physical BASE-DATA-TYPE of ODX to datatypes in Java is defined as follows:

- A_INT32 to Integer
- A_UINT32 to Long
- A_FLOAT32 to Float
- A_FLOAT64 to Double
- A_BYTEFIELD to byte[]
- A_ASCIISTRING to String
- A_UTF8STRING to String
- A_UNICODE2STRING to String

The Java code may expect an object of those Java object types as input parameter and shall return the result value as an object of these Java object types. If the result object of a COMPUCODE does not match the expected type and encoding defined in ODX, a `MCDError` is attached to the `MCDResponseParameter` (`eRT_INVALID_COMPUCODE_RESULT`). All objects of the Java object type "String" are always encoded as UCS-2, corresponding to A_UNICODE2STRING.

The Java code for a COMPUCODE is referred by a PROG-CODE data structure in the same way as Java job code. The attributes REVISION and ENCRYPTION at PROG-CODE are not supported by a standard compliant D-server. The support of this attributes is just a user-specific extension of a D-server. There are three storage kinds for computational Java code which are defined by the member SYNTAX: "JAVA" for Java source code, "CLASS" for Java byte code (compiled) and "JAR" for Java archive. In contrast to "CLASS" and "JAR", the support of Java source code is optional. A D-server is free to support "JAVA" source code. When storing as Java archive an ENTRYPOINT specifies the name of the class containing the program code. Identical to the processing of a SINGLE-ECU-JOB the COMPUCODE shall be executed in a separate Java class context (see 9.21). See Annex B.

9.7 Variable length parameters

Some diagnostic protocols (e.g. UDS) provide services that have parameters of variable length. The size of such a parameter is defined by another parameter in the same request message, the so-called length key parameter. In ODX, length key parameters and variable length parameters have a one-to-one relationship, that is, for each variable length parameter, there is one length key parameter. For this length key parameter, the simple `MCDParameterType eLENGTH_KEY` is used in D-server. As defined in ODX, parameters of type `eLENGTH_KEY` code the length of a variable length parameter in bits. However, if a diagnostic protocol requires the length to be given in a different format in the PDU, there needs to be a corresponding conversion between the coded (protocol dependent) and physical value (protocol independent, size in bits) in the ODX data. A D-server shall only consider the physical value of a length key parameter for determining the length of the referenced variable length parameter. At D-server side the data type of a length key parameter is mapped to eA_UINT32.

In case an MCDRequest or MCDResponse object contains a parameter that is of variable length, the corresponding MCDRequestParameter or MCDResponseParameter object of the request's or response's parameter list shall return true when queried with the MCDRequestParameter.isVariableLength() or MCDResponseParameter.isVariableLength() method, respectively. If that method returns true, the method MCDRequestParameter.getLengthKey() or MCDResponseParameter.getLengthKey() can be used to retrieve the parameter with parameter type eLENGTH_KEY that contains the associated parameter length. Both methods isVariableLength() and getLengthKey() are also available at the corresponding database objects MCDDbRequestParameter and MCDDbResponseParameter.

The client is responsible for setting the value of the corresponding length key parameter prior to setting the value of a variable length parameter. Setting the corresponding length key parameter value of a variable length parameter shall result in the following D-server internal actions:

- The D-server checks the new length key value and, if valid, sets the server internal length key value to its new value. In case an invalid length key value is set at an MCDRequestParameter or MCDResponseParameter, an exception is thrown when calling the method setValue() (MCDParameterizationException, ePAR_INVALID_VALUE) and the D-server internal length key value shall not be set to the invalid value.
- The state of the D-server internal MCDValue object of the variable length parameter is set to "uninitialized".
- The D-PDU API position of all subsequent parameters is recalculated as far as possible.

Setting a new value at a variable length parameter that does not match the size defined by the value of its corresponding length key parameter shall result in an exception being thrown by the D-server (MCDParameterizationException, ePAR_INVALID_VALUE). In case the value of the corresponding length key parameter is not yet initialized, a client is not allowed to set the value of the variable length parameter. Doing so shall result in an exception being thrown by the D-server (MCDProgramViolationException, eRT_WRONG_SEQUENCE).

If a length key parameter is constant, the client is not allowed to change the size of the corresponding variable length parameter. In that case, the size of the variable length parameter is determined by the predefined default value of its corresponding length key parameter.

The example in Figure 80 — Example Variable Length Parameters shows the D-PDU of a request message with two variable length parameters. Each of these two variable length parameters has a corresponding length key parameter, which defines the size of its variable length parameter in bits. The bit size of both length key parameters LengthOfMemoryAddress and LengthOfMemorySize is 4 bits. Together they occupy byte #1 of the D-PDU. The physical value n of parameter LengthOfMemoryAddress defines the length of parameter MemoryAddress in bits. The physical value m of parameter LengthOfMemorySize defines the length of parameter MemorySize in bits. In order for parameters MemoryAddress and MemorySize to occupy only complete bytes both n and m shall be devisable by 8 without remainder. It is obvious that in case parameter LengthOfMemoryAddress has no default value assigned to it, the parameter position of parameter MemorySize may only be calculated at runtime.

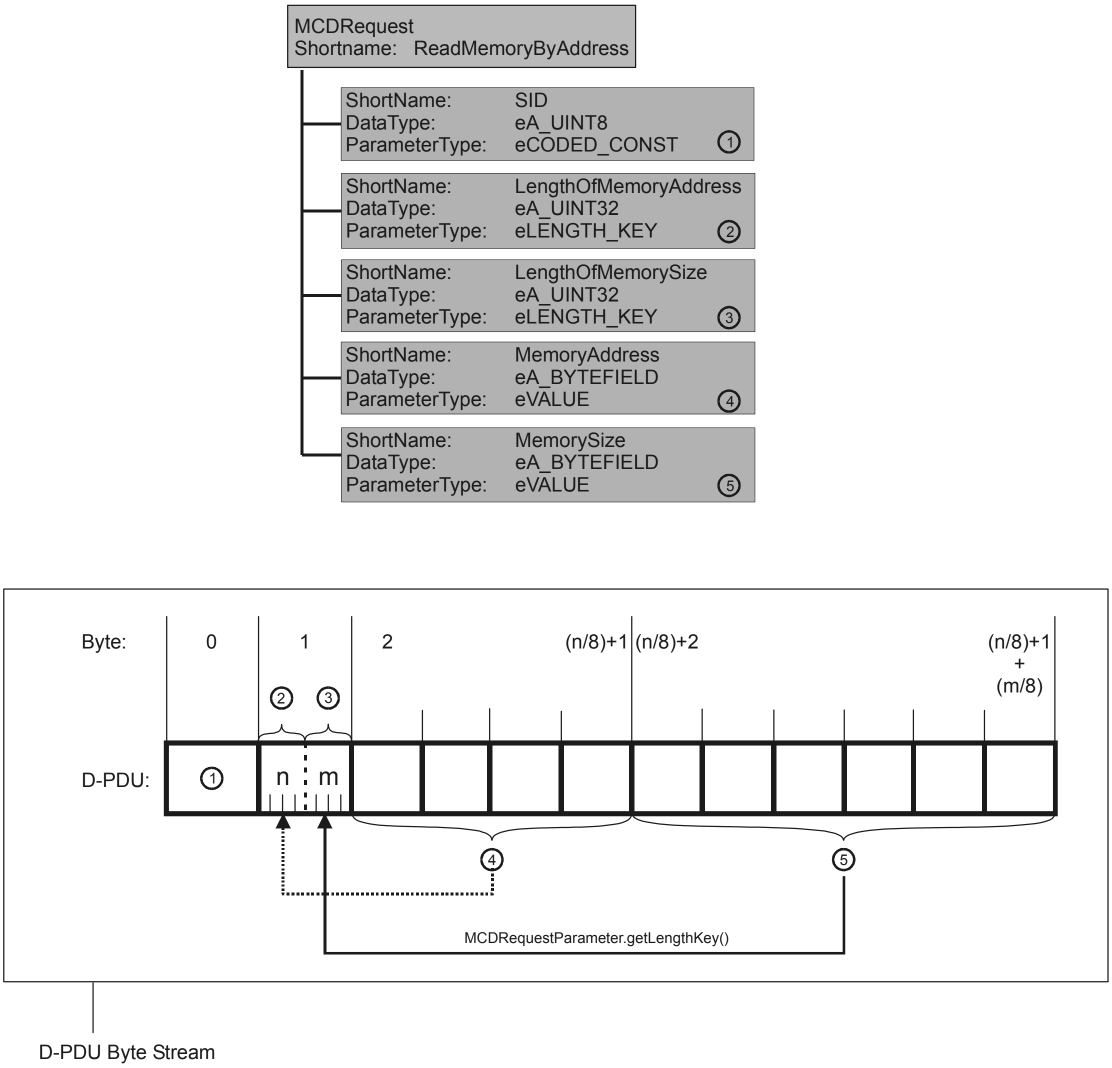


Figure 80 — Example Variable Length Parameters

9.8 Layer inheritance of services

9.8.1 Goal

This subclause describes the basics of the layer inheritance mechanism for diagnostic services. Please refer to the ODX specification for more detailed information on inheritance between diagnostic layers.

9.8.2 Layer inheritance of services

As an introduction to the variant identification subclauses below, this subclause gives a short overview of the layer inheritance basics. These layer inheritance basics are illustrated using the example inheritance hierarchy in Figure 81 — Example configuration of ECUs. This example shows a vehicle configuration which contains four Base Variant layers (BV A, BV B, BV C, BV D) which inherit from a Functional Group layer (FG). Base Variant BV A has an inheriting ECU Variant V A1, BV B has two inheriting ECU Variants V B1 and V B2, and there is one ECU Variant inheriting from BV C (V C1). For the sample scenario, now consider a vehicle electric configuration which contains the ECU Variant V A1 of the ECU represented by the inheritance tree of Base Variant BV A, the ECU Variant V B2 for ECU B2, and the Base Variant D. The ECU represented by Base Variant C and associated ECU Variant V C1 is not present in the sample vehicle electric configuration. This sample setup is used for the variant identification examples below, where the attempt to identify the currently present ECU Variant of Base Variant BV A – marked with the red triangle (V A1) – will deliver a positive (successful) identification result.

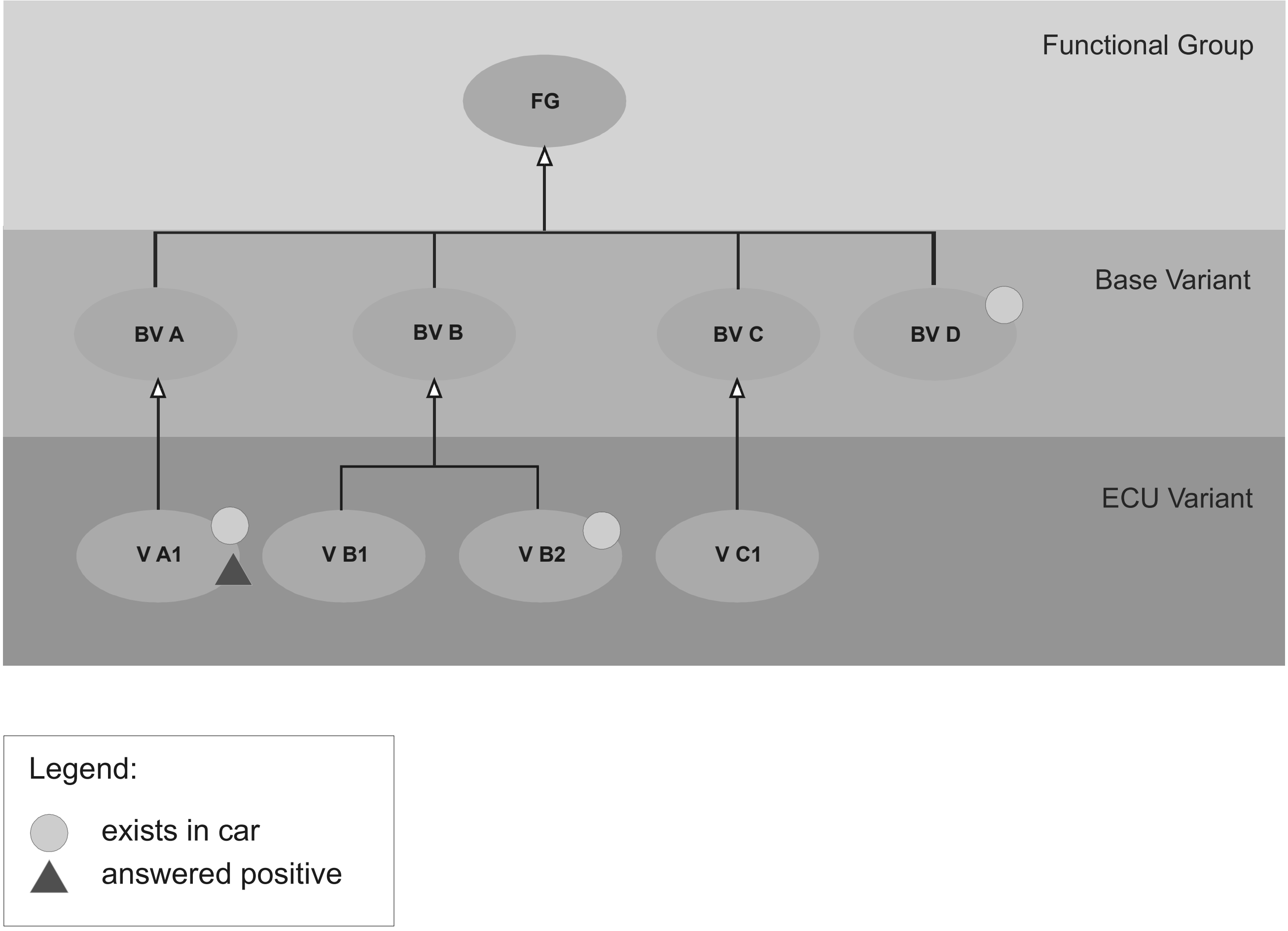


Figure 81 — Example configuration of ECUs

As the D-server shall be able to handle services (DiagComPrimitives) that are coming from the different diagnostic layers, there shall be a clear definition of how the D-server is supposed to resolve services and their parameters when executing a service from a Protocol Layer, a Functional Group layer, a Base Variant layer or an ECU Variant layer. Figure 82 — Interpretation algorithm for responses of services on different diagnostic layers illustrates the resolution algorithm that is used to interpret response data of a DiagService at a Logical Link, depending on the type of the Location of this Logical Link.

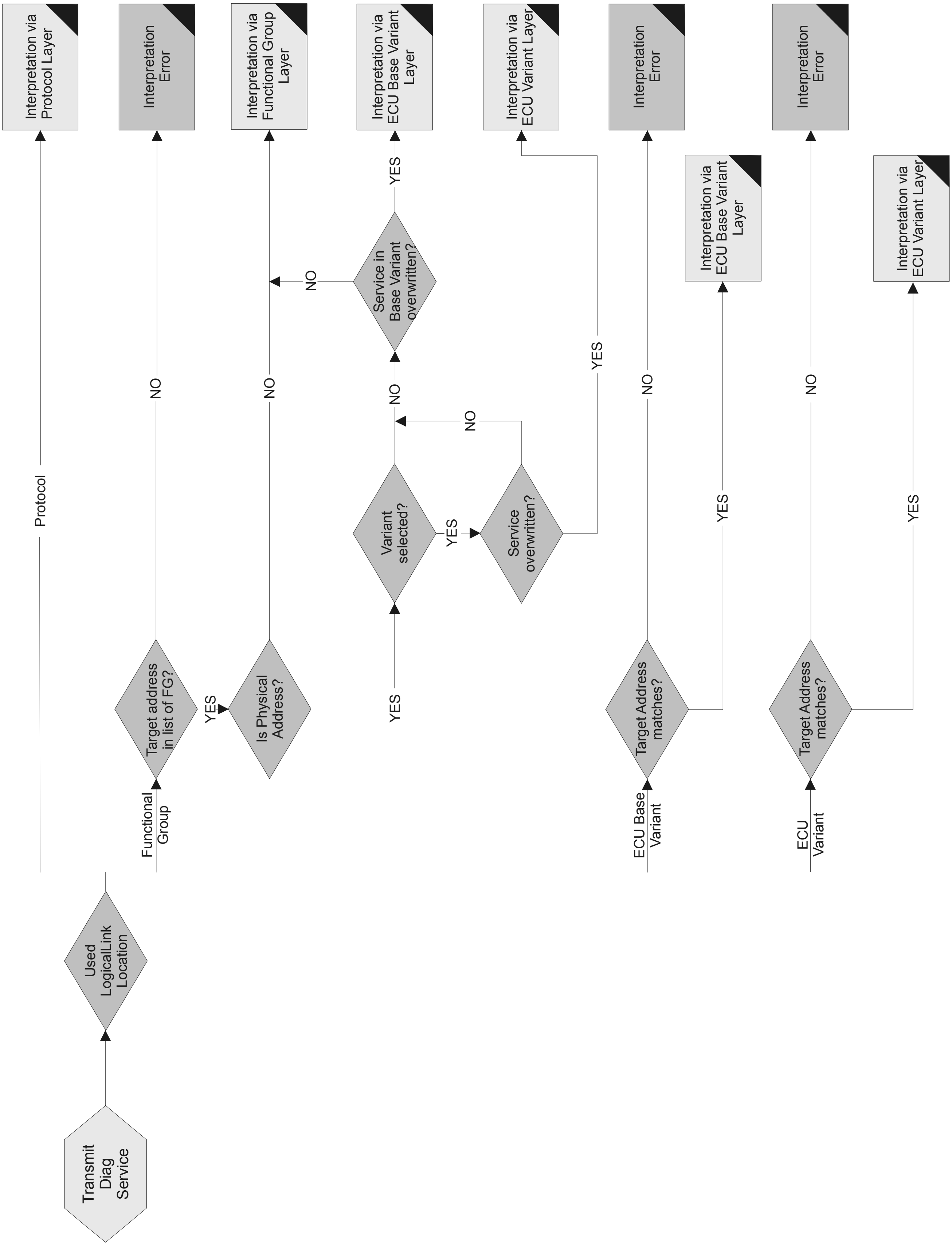


Figure 82 — Interpretation algorithm for responses of services on different diagnostic layers

To reduce the effort of data authoring, ODX allows to overwrite and eliminate services that are inherited from a higher (“more general”) diagnostic layer on the inheriting (more specific) layer. This concept of overwriting and elimination is illustrated in Figure 83 — Example location hierarchy. In the figure, a number of diagnostic services of are inherited (I), overwritten (O) and eliminated (E) through the diagnostic layer structure from a Functional Group named “DoorECUs”, via two ECU Base Variants “DoorECU FrontLeft” and DoorECU FromRight” to their ECU Variants on ECU Variant layer.

NOTE The elimination and overwriting of services breaks the inheritance principle – an ECU Variant that eliminates or changes a set of services inherited from its Base Variant can no longer be used in the same way as the Base Variant. This restriction will play an important role in the variant identification mechanism described later in 9.9.

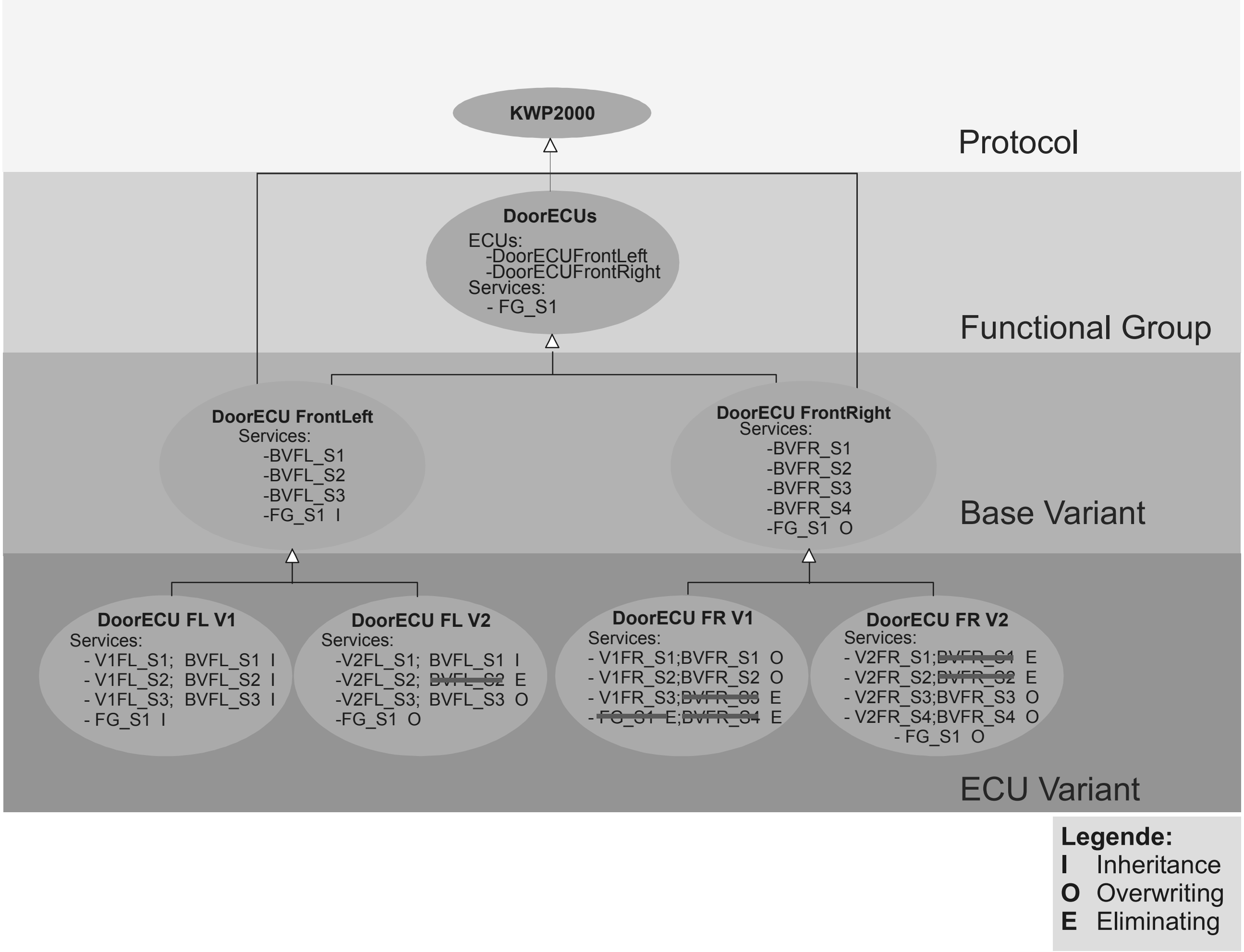


Figure 83 — Example location hierarchy

A second example of the inheritance mechanism for diagnostic services etc. is shown in Figure 84 — Uniqueness of short names in case of inheritance. In this second example, it is illustrated how ambiguity is resolved in case of multiple inheritance, that is, how the required uniqueness of diagnostic service short names is guaranteed by means of elimination when inheriting from multiple ancestors.

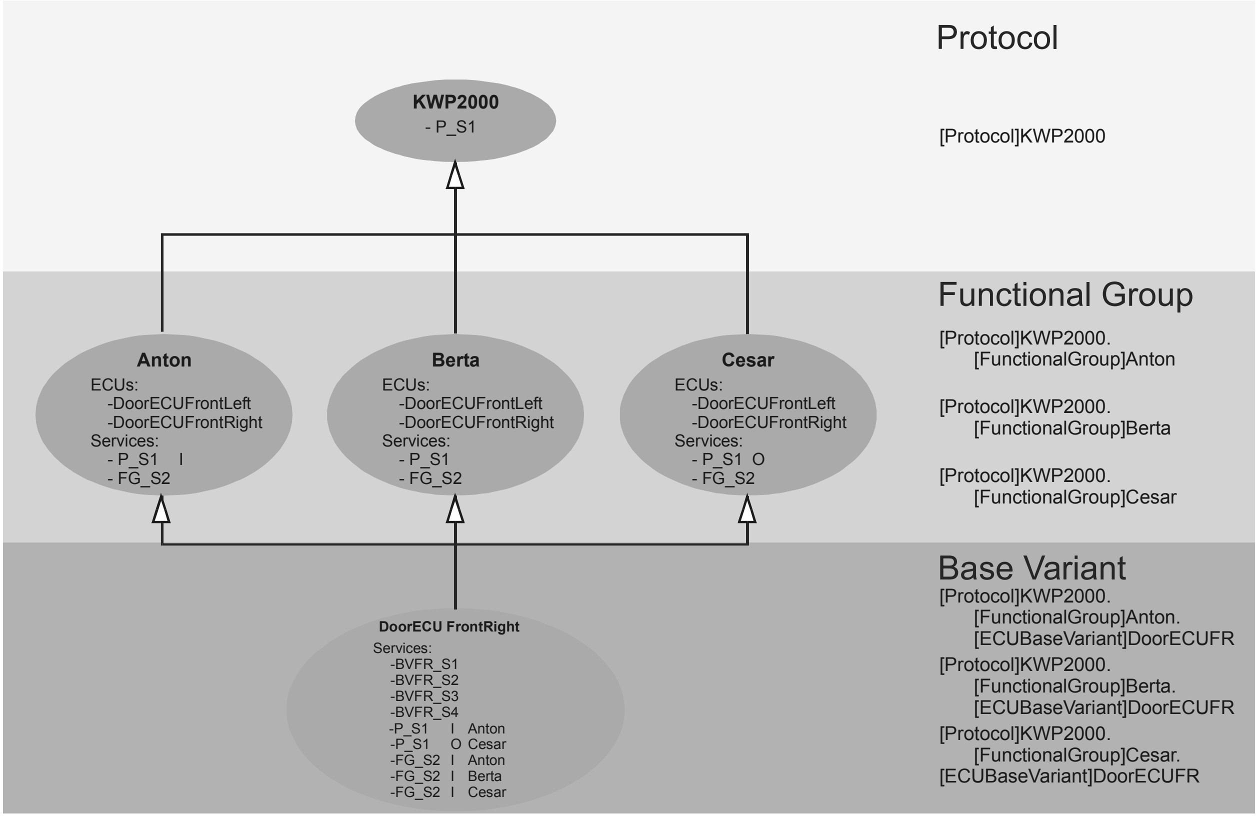
NOTE The possibility of multiple inheritance of diagnostic services is a result of the inheritance hierarchy for diagnostic layers as defined in ODX.

The rules for multiple inheritance are defined as follows:

— For a diagnostic layer, it is forbidden to inherit different data objects with the same short name from different parent layers. This situation shall be resolved by eliminating the offending data objects from all but one of the parent branches (using the NOT-INHERITED paradigm on these objects in the more specialized layer).

— In case a diagnostic layer inherits identical data objects from multiple parent layers, only a single set of these objects should exist in the inheriting layer (e.g. each of two PROTOCOL instances A and B establish a value inheritance relationship to an ECU-SHARED-DATA instance C. All data objects contained within C virtually exist now within A and B at the same level of specialization. If now a BASE-VARIANT D establishes value inheritance relationships to A and B, it seems to get a duplicated set of C's data objects.).

These principles are illustrated in Figure 84 — Uniqueness of short names in case of inheritance where the ECU base variant “DoorECU FrontRight” inherits services from three functional groups “Anton”, “Berta” and “Cesar”. The service “S1” would be inherited from both “Anton” and “Cesar” but the instance inherited from “Anton” is eliminated in the inheritance relationship. The same applies to the service “S2” which is inherited from all three functional groups.



Legende:
I Inheritance
O Overwriting

Remark: Only one Service with identical shortname can inherit from one layer to next layer, otherwise an ODX error occurs

Figure 84 — Uniqueness of short names in case of inheritance

The inheritance of an object which is subject to value inheritance can be prevented in ODX by placing an explicit NOT-INHERITED clause inside the corresponding PARENT-REF. Only objects of classes DIAG-COMM, DIAGVARIABLE, DOP-BASE, TABLE, and GLOBAL-NEG-RESPONSE can be eliminated in ODX in this way.

Eliminated objects are not visible and usable in the inheriting layer or in any layers inheriting from that layer at runtime. Furthermore, eliminated objects are not visible in the MCDDbLocation. Objects with the same short name can be re-created in those layers. However, this redefinition is only valid if no two objects of the same type share the same short name. For services of class VARIANTIDENTIFICATION, any elimination or redefinition on ECU-VARIANT level is forbidden.

9.8.3 Service handling on functional and physical locations

Diagnostic services can be executed on Functional Group, ECU Base Variant and ECU Variant layers. Provided that a fully qualified logical link is defined in the vehicle info spec (address information needs to be set), diagnostic services can also be executed on Protocol level. However, the semantics of the service executions differs between Functional Group and ECU Base Variant / ECU Variant layer (see Figure 85 — Addressing mode in case of physical or functional communication). For Protocol layer, the semantics of service execution is either as for a Functional Group or as for an ECU Base Variant or ECU Variant – depending on the address information set. Therefore, a short description of the characteristics of service execution on physical and functional locations is given in this subclause.

For service execution, two major addressing modes – `PHYSICAL` and `FUNCTIONAL` – are distinguished. The addressing mode `PHYSICAL` means explicit communication with a selected ECU (via ECU Base Variant or ECU Variant) whereas the addressing mode `FUNCTIONAL` denotes a multicast communication via a Functional Group which can contain several ECUs. Internally, the D-server chooses the correct communication mode (`PHYSICAL` or `FUNCTIONAL`) in accordance with the type of DbLocation communication has been set up to.

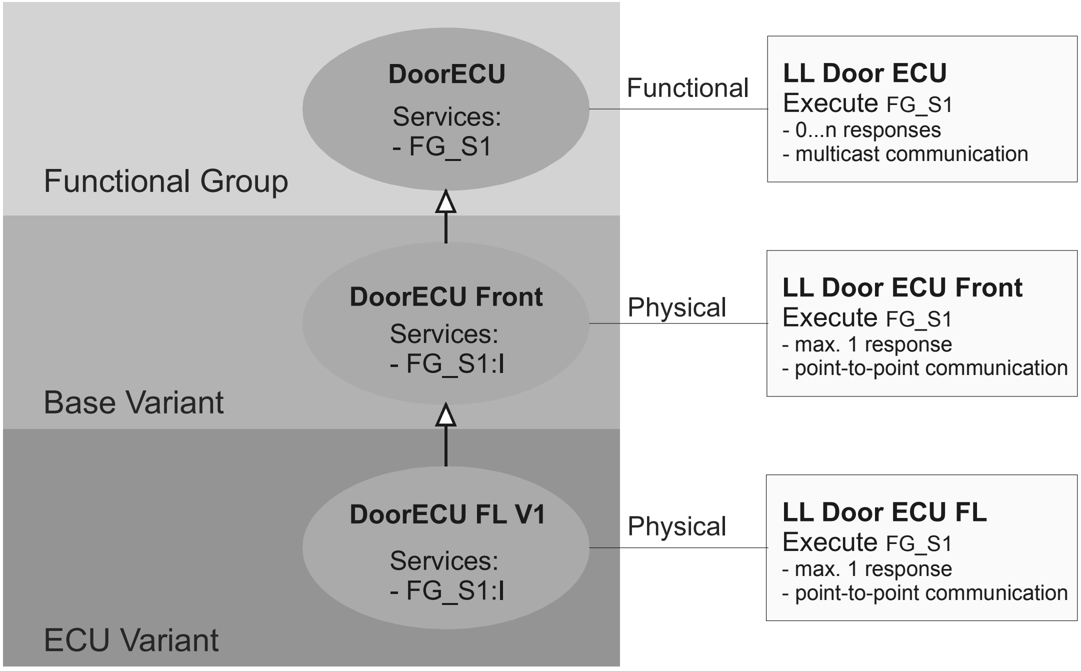


Figure 85 — Addressing mode in case of physical or functional communication

In addition to the communication modes `FUNCTIONAL` and `PHYSICAL`, there is a combined communication mode `FUNCTIONAL-OR-PHYSICAL`. The D-server is able to decide from the data in ODX and the current DbLocation which addressing mode of a service should be used if this service is defined as `FUNCTIONAL-OR-PHYSICAL` in ODX. If the DbLocation is a Functional Group, the service can only be executed functionally. If the DbLocation is an ECU Base Variant or an ECU Variant, the service can only be executed physically.

If functional addressing shall be used, e.g. for testing the reaction of a single ECU to Tester Present or Sleep commands, it is necessary to open a Logical Link to a specific ECU and a second Logical Link to a Functional Group this ECU is contained in. Then, the Logical Link to the Functional Group can be used for functional services while the Logical Link to the ECU is used for physical services.

Functionally executed services are implicitly sent to all ECUs of a Functional Group. Hence, functional addressing can always result in multiple responses, e.g. one response for every ECU contained in the Functional Group. In contrast, if a service is executed physically (on the level of an ECU Base Variant or ECU Variant), the D-server in general only expects a single response.

NOTE 1 There are execution settings where a physically executed diagnostic service also results in multiple responses. However, for the general description of service execution in this subclause, these special settings are not considered.

The request for a `FUNCTIONAL` service is only defined on the Functional Group layer. That is, one single request is sent to all addressed ECUs in case of functional communication. In contrast, a separate request needs to be sent to a set of ECUs if these are to be contacted physically.

If no ECU Base Variants are defined within a Functional Group, the responses to the functional request are resolved on the Functional Group level (compare algorithm sketch in Figure 82 — Interpretation algorithm for responses of services on different diagnostic layers). In case ECU Base Variants are defined for a Functional Group, the (physical) responses are resolved on the level of the different ECU Base Variants. In case an ECU Variant Identification and Selection (VIS) has already been successfully executed for a specific physical ECU in a vehicle, the responses of this specific ECU to a functional request shall be resolved on the ECU Variant level (see Functional Group branch algorithm in Figure 82 — Interpretation algorithm for responses of services on different diagnostic layers). However, if a physical ECU responds with a functional response to a functional request, the response shall be interpreted using the response templates defined on the Functional Group level.

By means of COMPLEX COM-PARAMs, arbitrary sets of response addresses can be defined for a Functional Group. These response addresses can be physical as well as functional response addresses. As a result, functional response addresses which might be used by some ECUs when answering functional requests can be easily added to the list of possible response addresses as defined on the Functional Group level in ODX.

The definition above implies that the request of a functional service inherited from a Functional Group is not overwritten or eliminated by any inheriting ECU Base Variant or ECU Variant layer in the underlying ODX data. Otherwise, the interpretation of the request fails for some ECUs, resulting in a negative response or no response at all, as the D-server will only consider the request as defined in the Functional Group.

NOTE 2 In case of an ECU responds with a response address to a functional request which is not present in the so-called `CP_UniqueRespIdTable` – the table of possible response addresses – of the Functional Group or related ECU Base Variants or ECU Variants, the response is not considered a valid response to the functional request. Therefore, such responses will not be part of the result of the functional request. Indeed, these responses are already dropped in the D-PDU API as no matching response template is available.

In ODX it is allowed to use the same ECU Base Variant definition (and the inheriting ECU Variants) for more than one physical ECU in a vehicle in case they can be diagnosed using the same set of diagnostic services etc. A typical example for such a setting are door ECUs in the same vehicle. Though more than one of these ECUs is present physically in a vehicle, door ECUs usually have the same diagnostic capabilities which can be represented by a single ECU Base Variant definition. To be able to distinguish different physical ECUs sharing the same ECU Base Variant (and ECU Variants) in diagnostics, the address information attached to the Logical Links to these ECUs is used. Although the ECUs share the same ECU Base Variant container, there needs to be a separate Logical Link defined to each of these ECUs. These Logical Links need to have different values for the communication parameters of PARAM-CLASS UNIQUE_ID. More precisely, two Logical Links are considered to point to two different physical ECUs in case at least one of the values for their communication parameters of PARAM-CLASS UNIQUE_ID differs. Otherwise, these Logical Links shall not be used in parallel as they point to the same diagnostic port of the same physical ECU.

However, a unique handle can be calculated from the values of the communication parameters of PARAM_CLASS UNIQUE_ID which are assigned to a Logical Link. This unique handle is shared with the D-PDU API and allows to uniquely assign a response of an ECU to the Logical Link which has been used to send the corresponding request. With the paradigm described in the paragraph before, the Logical Link represents the physical ECU which has sent the response – or at least one access port to a physical ECU.

NOTE 3 Different Logical Links to the same ECU Base Variant might have different status with respect to variant identification, even if they originally pointed to the same DbLocation. While one Logical Link still uses the services from the

DbLocation of the ECU Base Variant, another Logical Link might have already switched to an ECU Variant of the ECU Base Variant by means of variant identification and selection (VIS). Moreover, VIS might have resulted in the selection of different ECU Variants for these two Logical Links. That is, the result of variant identification and selection is specific to a Logical Link. As a result, functional communication to a Functional Group containing an ECU Base Variant which is used for more than one physical ECU in a vehicle usually results in more than one response – one for each physical ECU represented by the same ECU Base Variant (including inheriting ECU Variants).

9.9 Variant Identification and Selection (VI / VIS)

9.9.1 Goal

A D-server provides means to automatically identify or identify and select the ECU Variant of a Base Variant which is built into a vehicle electric configuration of a vehicle. The following subclauses describe the corresponding mechanisms of variant identification (VI) and variant identification and selection (VIS) in detail. This description includes the rules and restrictions that apply, how requests and responses are handled for these services, and gives some example scenarios.

9.9.2 Variant Identification Algorithm

9.9.2.1 Logical Link Basics

In a D-server, the set of diagnostic services, DTCs, flash sessions, etc. available at a certain access point to an ECU is represented by an MCDDbLocation object. More precisely, an MCDDbLocation represents the accumulation of diagnostic data resulting from an inheritance path from a diagnostic protocol to any other diagnostic layer – Protocol, Functional Group, ECU Base Variant or ECU Variant. In the example inheritance hierarchy in Figure 82 — Interpretation algorithm for responses of services on different diagnostic layers, the MCDDbLocation of ECU Base Variant B1 which is labeled “UDS” comprises

- all diagnostic data defined in protocol “UDS” which is not excluded from inheritance in ECU Base Variant B1,
- and all diagnostic data defined specifically for ECU Base Variant B1.

In order to use the diagnostic data represented by a MCDDbLocation at runtime, a so-called Logical Link is required. In general, the D-server uses the Logical Links predefined in the ODX data. These predefined Logical Links are represented by MCDDbDLogicalLink objects which reference exactly one MCDDbLocation object (see Figure 82 — Interpretation algorithm for responses of services on different diagnostic layers). For vehicle diagnosis, a runtime Logical Link (MCDDLogicalLink) needs to be created which is based on exactly one MCDDbDLogicalLink. In addition, a D-server supports the creation of runtime Logical Links not being based on a MCDDbDLogicalLink on demand for engineering purposes.

The example in Figure 82 — Interpretation algorithm for responses of services on different diagnostic layers shows an ECU base variant “B1” which inherits from two protocols “UDS” and “KWP2000”. As a result, “B1” has two DbLocations – one for the inheritance path starting at protocol “UDS” and one resulting from the inheritance path starting at protocol “KWP2000”. For each of the two DbLocations, Logical Links have been defined. A single Logical Link “DBLogicalLink_B1.KWP2000.CANHS” is defined for DbLocation “KWP2000” and two Logical Links “DBLogicalLink B1.UDS.CANHS_125” and “DBLogicalLink B1.UDS.CANHS_500” are defined for DbLocation “UDS”.

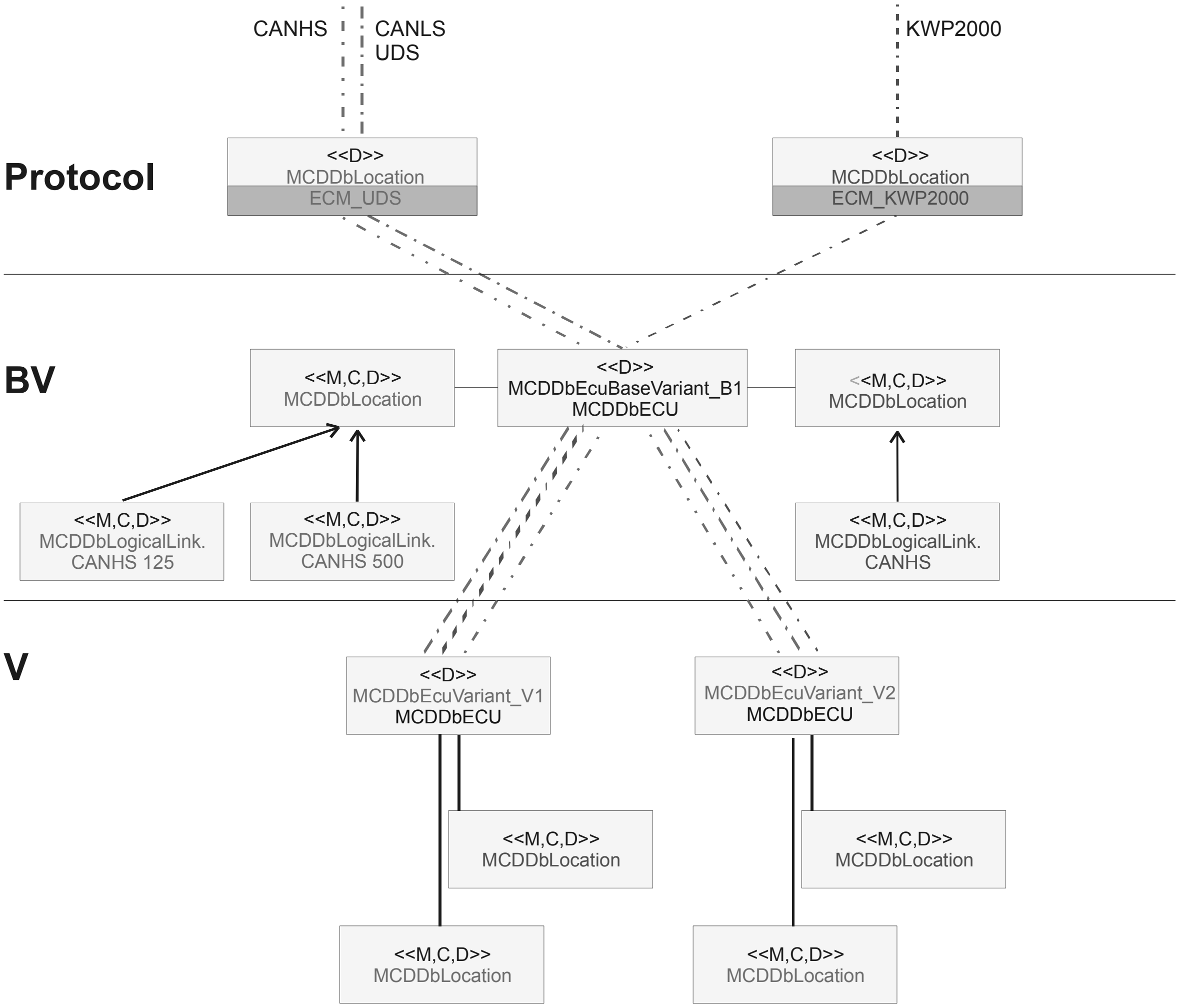


Figure 86 — Possible cardinalities between MCDDbLocation, MCDDbDLogicalLink and MCDDLogicalLink on different diagnostic layers by example

Different Logical Links to the same DbLocation are typically defined if

— alternative Logical Links with different settings for the communication parameters are required, e.g. 125 kBaud for engineering purposes and 500 kBaud when built into a vehicle. Here, the address information stored in the protocol parameters attached to the Logical Links are identical for alternative Logical Links using the same physical links.

— different ECUs in the vehicle are based on the same ECU base variant, e.g. all door ECUs in a vehicle provide identical diagnostic capabilities represented by the DbLocation. Here, the address information stored in the protocol parameters attached to the Logical Links is different for Logical Links to different ECUs.

— an ECU can be accessed via different bus and transport layers. Here, the physical link type and the address information stored in the protocol parameters attached to the Logical Links are typically different for these Logical Links.

NOTE It is only permitted to have one runtime Logical Link for each MCDDbLogicalLink (not counting ‘proxy’ objects for sharing the same runtime Logical Link amongst several clients).

As a result, it is not possible to have a runtime Logical Link to an ECU Base Variant and a runtime Logical Link to any ECU Variant of the ECU Base Variant at the same time if these links are based on the same MCDDbDLogicalLink object (see above). However, in a multi-client scenario, two clients may have access to the same runtime Logical Link by means of corresponding proxy objects. D-server internally, both Logical Link proxies are mapped to the same runtime Logical Link (MCDLogicalLink).

9.9.2.2 Principles of Variant Identification and Variant Identification And Selection

In ODX, Logical Links – represented by `MCDDbDLogicalLinks` in MCD-3 – do never reference an ECU Variant. Instead, ECU Variant access can only be achieved at D-server runtime by either creating a direct Logical Link to an ECU Variant (`MCDLogicalLinks::addByAccessKeyXXX()`) or by asking the D-server to identify (and select) the correct ECU Variant. For the latter case, the variant identification (and selection) algorithm has been defined. This algorithm will be explained in detail in the remainder of this subclause.

From a black box perspective not considering the details of VI and VIS, there is a difference with respect to the status of a Logical Link after performing variant identification (VI) or identification and selection (VIS). After performing VI on a Logical Link, this Logical Link still points to the same DbLocation (see upper part of Figure 87 — Difference between variant identification (VI) and variant identification and selection (VIS)) as before. Here, the identified ECU Variant is only reported to the user of the Logical Link, e.g. ECU Variant V1 in the example, by means of an event (`MCDLogicalLinkEventHandler::onLinkVariantIdentified()`) – provided that the user has registered an event handler. In addition, the identified ECU Variant can be obtained by calling the method MCDDLogicalLink::getIdentifiedVariantAccessKeys() at the corresponding runtime Logical Link.

In contrast to VI, the result of a successful VIS is that the MCDDbDLogicalLink is switched to a DbLocation of the matching ECU Variant (lower part of Figure 87 — Difference between variant identification (VI) and variant identification and selection (VIS)). The identified and selected ECU Variant is reported to the user of the Logical Link by means of the event `MCDLogicalLinkEventHandler:: onLinkVariantSelected()` – provided that the user has registered an event handler. In addition, the selected ECU Variant can be obtained by calling the method MCDDLogicalLink::getSelectedVariantAccessKeys() at the corresponding runtime Logical Link.

NOTE Variant Identification (VI) and Variant Identification and Selection (VIS) may only be performed on ECU Base Variant and ECU Variant level. An execution on Protocol or Functional Group is not possible.

As a result, the D-server offers the control primitives `MCDDbVariantIdentification` and `MCDDbVariantIdentificationAndSelection` only at Locations on ECU Base Variant and ECU Variant level.

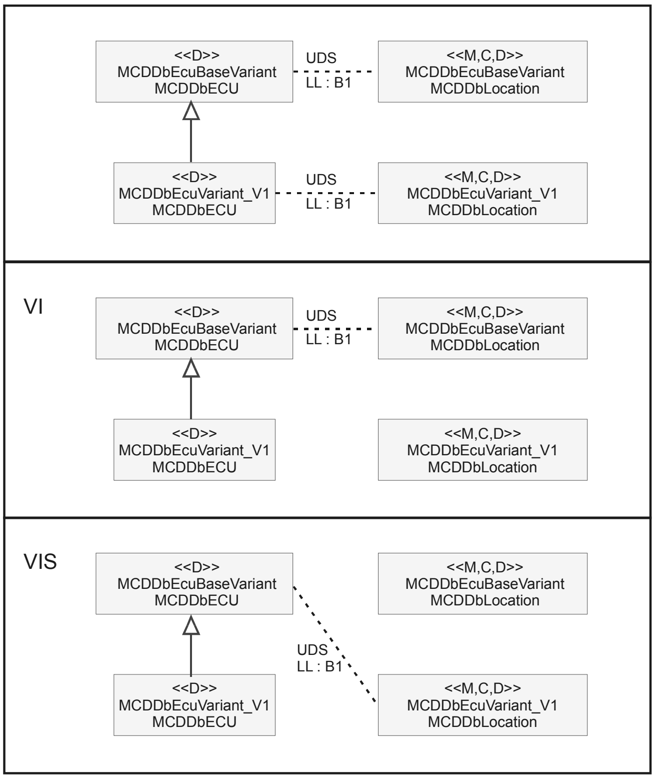


Figure 87 — Difference between variant identification (VI) and variant identification and selection (VIS)

Though Figure 87 — Difference between variant identification (VI) and variant identification and selection (VIS) only illustrates the result of VIS in case of a Logical Link to an ECU Base Variant, it is also possible to switch from one ECU Variant to another ECU Variant by means of VIS. In this case, the client application has, e.g., opened a Logical Link to an ECU Variant which is not built into the currently diagnosed vehicle. Now, VIS can be used to switch to the correct ECU Variant. Technically, the `MCDDbDLogicalLink` is switched from the `MCDDbLocation` of the wrong ECU Variant to the corresponding MCDDbLocation of the correct ECU Variant. Again, the newly selected ECU Variant is reported to the users of a Logical Link by means of the event `MCDLogicalLinkEventHandler::onLinkVariantIdentified()`. And the identified ECU Variant can be obtained by calling the method MCDDLogicalLink::getIdentifiedVariantAccessKeys() at the corresponding runtime Logical Link.

As there can only be one `MCDDLogicalLink` per `MCDDbDLogicalLink` but several proxy objects for the same `MCDDLogicalLink` (multi-client scenario), the results of VI and VIS are immediately visible to all client applications via their corresponding Logical Link proxy. This includes that one client application executing VIS on a shared runtime Logical Link influences other client applications using the same runtime Logical Link.

Furthermore, a client application that is opening a Logical Link will directly communicate with a specific ECU Variant if this Logical Link has already been present in the Logical Link table and if this Logical Link has been switched to this ECU Variant by means of VIS before. Here, the D-server will simply return a new reference (Logical Link proxy) to the already existing runtime Logical Link object. As a result, a runtime Logical Link will not start on the ECU Base Variant for an additional client application if another client application created this Logical Link and executed VIS successfully for this Logical Link before. All proxies to the same Logical Link share the same state. As a result, it is not possible to open Logical Links to two different ECU Variants of the same ECU Base Variant if both Logical Links are based on the same `MCDDbDLogicalLink`.

In case VI was successfully executed for a Base Variant or ECU Variant Logical Link, the method `MCDDLogicalLink::getIdentifiedVariantAccessKeys()` returns a non-empty collection, containing the AccessKey that represents the identified ECU Variant. The method `MCDDLogicalLink::getSelectedVariantAccessKeys()` does only return a non-empty collection if an ECU Variant has already been selected for this Logical Link – either because the Logical Link has been created using the methods `MCDLogicalLinks::addByAccessKeyAndXXX()` or

MCDLogicalLinks::addByVariant(), or by having successfully executed a VIS before. This means that in case of VI, the two collections can contain different sets of AccessKeys.

In case VIS was successfully executed for a Base Variant or ECU Variant Logical Link, the method MCDDLogicalLink::getIdentifiedVariantAccessKeys() returns a nonempty collection of AccessKeys, containing an AccessKey that represents the identified ECU Variant. In this case, the method MCDDLogicalLink::getSelectedVariantAccessKeys() returns a nonempty collection as well, containing the same AccessKey as the first collection. In case of a successful execution of VIS, the two collections need to be identical.

In case VI or VIS fails for a Logical Link (no ECU Variant was identified), the method MCDDLogicalLink::getIdentifiedVariantAccessKeys() returns an empty collection of AccessKeys. However, the collection returned by MCDDLogicalLink::getSelectedVariantAccessKeys() can still be non-empty. This is, e.g., the case when a link to one of the ECUs reachable via the Logical Link that the VI/VIS is executed on has been created by means of one of the methods MCDLogicalLinks::addByAccessKeyAndXXX() or MCDLogicalLinks::addByVariant(). A second example is the case where VIS fails on a LogicalLink pointing to an ECU variant. Here, the LogicalLink remains unchanged – it still points to the same ECU variant afterwards – and the result of the failed VIS is as described above.

The following execution states are defined for the control primitives used execute a VI or VIS:

— eALL_NEGATIVE: No ECU variant has been identified. A negative response is returned by the D-server.

— eALL_POSITIVE: An ECU variant has been identified. A positive response is returned by the D-server.

9.9.2.3 Variant Patterns and Matching Parameters

Technically, variant identification is performed by the D-server by executing a set of DiagComPrimitives, and then matching the ECU responses to a set of patterns that are defined in the ODX data set. Here, positive and negative responses can be used for matching parameters. Which services have to be executed and which parameters are required to evaluate to which values is defined in so-called *matching patterns* (ODX: ECU-VARIANT-PATTERN). In MCD-3, a matching pattern is represented by an object of type MCDDbMatchingPattern (see Figure 88 — Matching Patterns for VI and VIS). An ordered collection of such matching patterns can be obtained from each MCDDbEcuVariant. Every matching pattern references an ordered collection of *matching parameters* (ODX: MATCHING-PARAMETER). Every matching parameter – represented by an object of type MCDDbMatchingPattern in the D-server – refers to a DiagComPrimitive, a Response Parameter of this DiagComPrimitive, and an expected value for this Response Parameter.

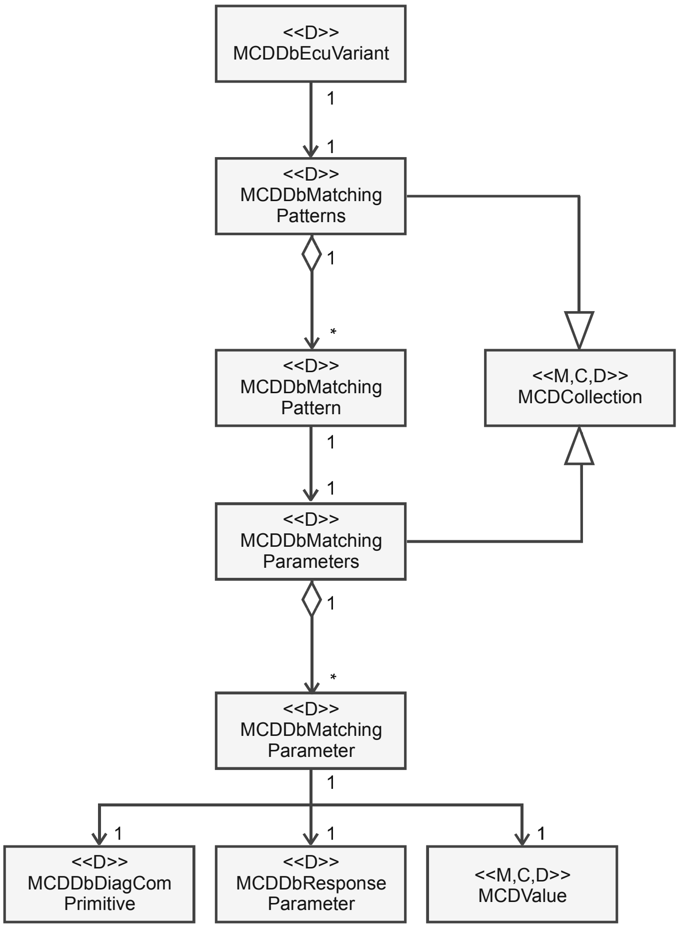


Figure 88 — Matching Patterns for VI and VIS

From the DiagComPrimitives referenced from the matching patterns obtainable from a MCDDbLocation of a MCDDbEcuVariant, the D-server internally generates the Control Primitives MCDDbVariantIdentification and MCDDbVariantIdentificationAndSelection. By default the DbRequest of these Control Primitives is empty, that is, it does not contain any response parameters. Similarly, the collection of DbResponses does only contain a single, empty positive response and a single, empty local negative response. This means that these responses do not have any response parameters as well. For engineering purposes, it can be useful to have more information on the request parameters and response parameters of the control primitives for VI and VIS. Therefore, the D-server can be instructed to generate full request and response templates (DB-part and runtime part). For more information see 9.9.5.

In contrast to the layer inheritance principles described above (see 9.8.2), the following restriction applies to DiagComPrimitives which are used for VI or VIS: The behaviour of these DiagComPrimitives shall not be changed on ECU Variant level. That is, these DiagComPrimitives may neither be overwritten nor eliminated through the NOT-INHERITED mechanism in ODX. This includes the service itself, referenced DOPs and so on. These DiagComPrimitives shall be marked “IS_FINAL = true” and “IS_MANDATORY = true” in ODX.

No DiagComPrimitive to be executed during VI/VIS shall be marked cyclic in ODX! Furthermore, the runtime mode (eCYCLIC, eNONCYCLIC) of any MCDDiagService executed during VI/VIS is ignored by the D-server.

9.9.2.4 Identification Algorithm

In order to perform Variant Identification or Variant Identification and Selection, a runtime instance of the corresponding control primitive – MCDVariantIdentification or MCDVariantIdentificationAndSelection – needs to be created at the Logical Link the VI or VIS is to be executed on. Then, this control primitive is executed synchronously by means of MCDDiagComPrimitive::executeSync(). Now, the D-server internally executes the DiagComPrimitives referenced from the matching parameters in the matching patterns that have been used to generate the variant identification (and selection) control primitive in a certain order. More details on this order and the algorithm can be found below.

One important requirement for the execution of a VI(S) is that it needs to be quick, as the concept of an automated variant identification feature only retains its usefulness when the execution time is kept as low as possible – after all, deliberately complex and elaborate usage scenarios can be handled, e.g., by a Java job and should not be part of a general standard. For this reason, the following rules apply for the execution of a variant identification (and selection) control primitive by a D-server.

VI(S) pattern matching shall be done in a specific order which is defined by the matching patterns and matching parameters in the ODX data. In principle, this matching is a three-step process:

- All ECU Variants inheriting from the ECU Base Variant which is to be resolved to a variant shall be considered.

- For each ECU Variant, the associated Variant Patterns (matching patterns) defined in ODX shall be checked.

- For each Variant Pattern, all Matching Parameters defined in ODX shall be checked.

ECU Variants shall be tested during VI or VIS in ascending alphabetical order of their short names. For each ECU Variant, the D-server then considers the ECU-VARIANT-PATTERNs of the ECU-VARIANT in the order they are defined in the processed ODX data. The first matching ECU-VARIANT-PATTERN determines that this ECU Variant is present in the vehicle. In this case, the ECU Variant this matching pattern belongs to is considered identified. After a match, the other ECU-VARIANT-PATTERNs of an ECU Variant will not to be tested any more (short-cut resolution). As an ECU Variant has been identified successfully, no further ECU Variants' matching patterns will be tested. In case of VIS, the identified ECU Variant is automatically selected. That is, the MCDDbDLogicalLink referenced from the MCDDLogicalLink used for VIS is switched to an MCDDbLocation of the identified MCDDbEcuVariant [see Figure 87 — Difference between variant identification (VI) and variant identification and selection (VIS)].

The MATCHING-PARAMETERs defined in an ECU-VARIANT-PATTERN need to be tested in the order they are defined in ODX. Furthermore, all parameters referenced from a set of MATCHING-PARAMETERs shall be checked, short-cut resolution is forbidden here. All MATCHING-PARAMETERs of an ECU-VARIANT-PATTERN need to match in order to consider the pattern as matching the current ECU Variant.

For the execution of diagnostic services for performing a VI(S) in accordance with the ordering rules described above, the D-server shall use the following algorithm:

- For each ECU Variant (in ascending alphabetical order of their short names), iterate through the ECU-VARIANT-PATTERNs of the ECU Variant valid for the current Logical Link.
  - For each ECU-VARIANT-PATTERN, iterate the MATCHING-PARAMETERs in the order they are defined in ODX.
    - Execute the DIAG-COMM referenced from the current MATCHING-PARAMETER if it has not been executed in the context of a previous ECU-VARIANT-PATTERN or MATCHING-PARAMETER of the same execution cycle of VI(S).
    - Match the value of the response parameter referenced from this MATCHING-PARAMETER with its expected value. Either use a temporarily stored response of a previous execution of the respective DIAG-COMM in of the same execution cycle of VI(S) or use the current response. Store the result of the match until the current ECU-VARIANT-PATTERN has been processed and evaluated completely.
    - Store the result of the execution of the DIAG-COMM (including its responses) for the rest of the current execution cycle of VI(S).

    - Continue with the next MATCHING-PARAMETER in the order defined in the ODX data.

  - Check whether all MATCHING-PARAMETERs of the current ECU-VARIANT-PATTERN have matched their expected values.

    - If the current ECU-VARIANT-PATTERN has matched completely, mark this corresponding ECU Variant as identified and exit the variant identification algorithm completely. In case of VIS, switch the Logical Link to the identified ECU Variant.

    - If the current ECU-VARIANT-PATTERN has **not** matched completely, continue with the next ECU-VARIANT-PATTERN in the order defined in the ODX data.

  - Repeat pattern matching as described above (mind the MATCHING-PARAMETER ordering as defined in ODX) with the next ECU Variant in the ascending order of short names if no ECU Variant has been identified yet. As soon as an ECU-VARIANT-PATTERN matches completely, the ECU Variant is considered as successfully identified, and the VI(S) can be stopped.

- If VI(S) terminates successfully (an ECU Variant has been identified), a positive response is to be generated by the D-server. This positive response is then returned within a corresponding result to the client application – either as return value to the method call MCDDiagComPrimtive::executeSync() or as parameter to the event MCDDiagComPrimitiveHandler::onPrimitiveHasResult() for all registered event handlers.

- If VI(S) terminates without having successfully identified an ECU Variant, a negative response is to be generated by the D-server. This negative response is then returned within a corresponding result to the client application – either as return value to the method call MCDDiagComPrimtive::executeSync() or as parameter to the event MCDDiagComPrimitiveHandler::onPrimitiveHasResult() for all registered event handlers.

For engineering use cases, it needs to be possible for the client application to identify which variant pattern has matched during the ECU variant identification process. If the variant identification fails, it can be important to know for an engineering tester which variant-patterns are available in ODX. To this end, the MCD-3 API provides means to access the variant patterns available for an ECU Variant through the method `MCDDbEcuVariant::getDbVariantPatterns()`. In addition, the variant patterns which have matched (after execution of VI or VIS) can be obtained from the D-server by using the method `MCDDLogicalLink::getMatchedDbEcuVariantPattern()` (see Figure 88 — Matching Patterns for VI and VIS).

A short digression concerning the demand that diagnostic services shall be executed in a defined order depending on individual matching parameter order for each ECU variant pattern and each ECU Variant as found in the ODX data: From the D-server's point of view, this approach is not feasible for several reasons and would cripple the VI/VIS feature for the large majority of use cases. First, there is the fact that the D-server shall build request and result templates for VI/VIS services (if enabled by the corresponding system property). As it is now, the D-server will have to include, for example, five request structures in the result object, one for each of the five executed DiagComs as defined by an exemplary ODX data set in accordance with the rules described above and in 9.9.5. As the execution order of these services is not defined, they will be executed only once on ECU Base Variant level for data gathering while VI/VIS is executed, and the request and response object structures will remain manageable and execution time will be short. In contrast to that, the demand that the execution order of VI/VIS DiagComs shall conform to the order of the matching parameters in each of the ECU variant patterns of all possible ECU Variants for a Base Variant link would lead to the following scenario: First of all, in the worst case (no matching ECU Variant is built into the vehicle), the five DiagComs from the example above would have to be executed anew for each possible permutation of ECU Variants, variant patterns, and matching parameter sets. For example, for a Base Variant with 50 ECU Variants, with 10 variant patterns each and five matching parameters per variant pattern, that would mean that 50*10*5=2500 services would have to be executed for a single VI/VIS. Secondly, when assuming that any of the services that is used for variant identification could cause an ECU to change its state/behaviour, it would be necessary to perform the appropriate reset/state changing action after (unsuccessful) testing of each matching parameter set such that such an ECU would be in the correct state for the next test iteration. Hence, an ECU reset or similar service would have to be part of each VI/VIS pattern. Imagining the time that would be necessary to perform a VI/VIS that executes 2500 services, where each fifth service causes an ECU reset, is left as an exercise to the reader. On top of these obvious problems, there is a more subtle one: as the D-server shall provide request and result structures for VI/VIS, it would have to provide different request/result structures for each of the possible permutations of service order as described above. When imagining an ECU with 50 variants, 10 variant patterns, 5 matching parameter sets/associated DiagComs, with an average of 3 parameters each, that would lead to a request template containing 50*10*5*3=7500 parameters! When doing the same calculation for the result structure, there are not only possible positive responses (and response parameters) to take into account but also negative and global negative response structures. Again, the exercise of imagining the resulting response templates is left to the reader. The general conclusion is, however, that defining an execution order for VI/VIS DiagComs on the matching parameter level is not a desirable solution for a D-server and will therefore not be part of a standard solution. For the few cases where this kind of behaviour is needed for variant identification, the user can always write an appropriate Java job or application logic to handle these specific use cases.

9.9.3 General VI/VIS handling considerations

After successful execution of a VIS on a LogicalLink, the D-server needs to take care that all DiagComPrimitives which have been created on that link before are invalidated by setting their state to `eNOT_EXECUTABLE`. All DiagComPrimitives created on a logical link after it has been switched to an ECU Variant, e.g. by means of variant identification and selection, start in state eIDLE as usual.

If variant identification and selection (VIS) is executed on a logical link which has already been switched to an ECU Variant and if this variant identification and selection results in the same ECU Variant being identified and selected, no state change is issued. Therefore, no DiagComPrimitive is invalidated for this logical link.

If variant identification and selection (VIS) is executed on a logical link which has already been switched to an ECU Variant and if this variant identification and selection results in a different ECU Variant being identified and selected, a state change is issued. The event `onLinkVariantSelected()` is sent to the client(s). Therefore, all DiagComPrimitives that have been created on this logical link before need to be invalidated (state change to `eNOT_EXECUTABLE`), as the target of the logical link has changed.

All DiagComPrimitives that have been created on Base Variant level or a different ECU Variant become invalid after the ECU Variant has been identified and selected for a logical link (logical link is switched to ECU Variant level). This means that every DiagComPrimitive that has been created on a logical link before a VIS has been executed is set to `eNOT_EXECUTABLE` when a new variant is selected for that link, regardless of whether that service is actually overwritten/removed on the new link variant or not. If a clients tries to execute a DiagComPrimitive which is in state `eNOT_EXECUTABLE`, the methods `executeSync()`, `executeAsync()`, and `startRepetition()` throw an exception of type `MCDProgramViolationException` with error code `eRT_NOT_ALLOWED_IN_THIS_OBJECT_STATE`.

The `MCDDLogicalLink` is not changed when executing a successful VIS, but a call to `MCDDbLogicalLink.getDbLocation()` will return a different DbLocation object after the VIS than before. After a successful VIS, the client shall retrieve the DbObjects from the new DbLocation again. Using DbObjects that the client received before a successful VIS can result in runtime errors.

NOTE A VI behaves differently than a VIS with respect to the description above. As the logical link is not switched to a different ECU Variant after a successful VI, no DiagComPrimitives are invalidated (`eNOT_EXECUTABLE`), and no switching of the LogicalLink's `DbLocation` takes place.

For variant identification, the server shall not reuse existing DiagComPrimitives to avoid events being sent to listening clients. That is, no task performed by the server to realize VI or VIS shall be exposed to any client.

Implementation hint: For example, this is guaranteed if the D-server always creates new and purely internal instances of all DiagComPrimitives required during VI/VIS: No DiagComPrimitives will be added to the collection of DiagComPrimitives at the corresponding logical link. As a result, no client will be able to access those instances of class DiagComPrimitive, and no events will be sent to any listening client.

NOTE 1 When a DiagComPrimitive has been queued by a client, the DiagComPrimitive can still be invalidated by a currently running VIS.

In case of a successful VIS, the DiagComPrimitive's state changes to `eNOT_EXECUTABLE` while it is waiting in the execution queue. In that case an event of type `onPrimitiveHasResult` will be raised by the server containing an empty result with the execution state `eCANCELLED_FROM_QUEUE`.

NOTE 2 In case of synchronous execution, the return value of `executeSync()` is the same empty result as mentioned above.

The execution of VIS is only allowed if there is no DiagComPrimitive currently running on the corresponding logical link that is marked cyclic, or has been executed by the `startRepetition()` method. The reason for this is, that the D-server does not know how to stop these services if the VIS is successful (because in this case all DiagComPrimitives on that link will be invalidated, as described in more detail in 9.9.4). Therefore, VIS can only be executed on a Logical Link which is in ActivityState `eACTIVITY_IDLE`. This restriction does not apply to Variant Identification (VI). As a successful VI does not change the Logical Link it is being executed on, it can be executed at any time. The Logical Link shall be in state `eONLINE` or `eCOMMUNICATION` to be able to execute either VI or VIS.

The `VariantIdentification` and `VariantIdentificationAndSelection` primitives are executed like diagnostic services and are only available at ECU Base Variant and ECU Variant locations. Though VI and VIS are handled like primitives, they actually represent a set of diagnostic services that are executed to retrieve ECU data relevant for identification of ECU Variants. This is specified in more detail later in this subclause.

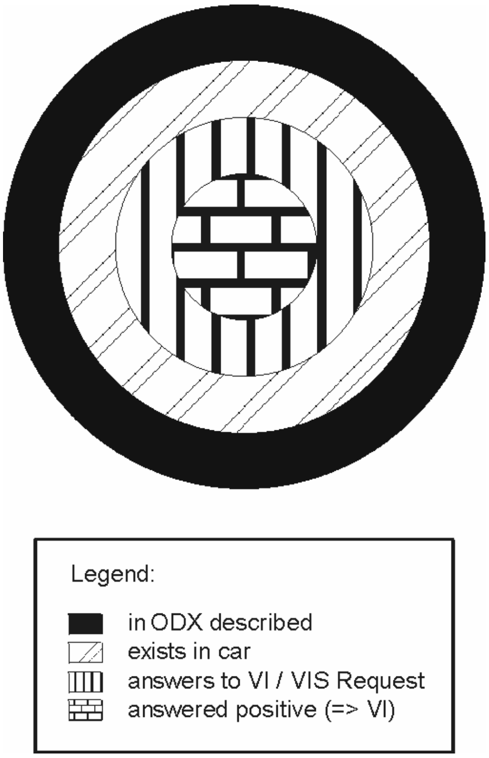


Figure 89 — Defined in ODX data, built into Vehicle and identified quantities of ECUs

Usually, the number of ECUs described in the ODX data, the number of ECUs built into a concrete vehicle, the number of ECUs answering to VI and VIS, and the number of ECU sending a positive response in case of VI and VIS are different. The relation of these numbers is illustrated in Figure 89 — Defined in ODX data, built into Vehicle and identified quantities of ECUs. The number of ECUs described in the ODX data is greater or equal to the number of ECUs built into a concrete vehicle. The number of ECUs built into a concrete vehicle is greater or equal to the number of ECUs answering to VI(S). The number of ECUs answering to VI(S) is greater or equal to the number of ECUs returning a positive response in case of VI(S).

9.9.4 Deselecting of selected variants

Generally, it is not possible to change the selected variant of a logical link by any other means than by execution of a VI/VIS control primitive. The only explicit way for a client application to get a logical link to the base variant of an already selected ECU variant is to remove the existing ECU variant link and then create a new logical link pointing to the ECU base variant. One reason for this behaviour stems from the existence of SingleECUJobs: Imagine the case when a SingleECUJob, which is part of an ECU variant location, is executed on a logical link pointing to this ECU variant. Allowing the SingleECUJob to reset the link it is being executed on to base variant (where the Job is not even available) could lead to undesirable results at D-server runtime.

9.9.5 Request and Response parameters of VI and VIS

9.9.5.1 Goal

The communication primitives `MCDVariantIdentification` and `MCDVariantIdentificationAndSelection` typically consist of a number of diagnostic services marked with the special DIAGNOSTIC-CLASS `VARIANTIDENTIFICATION`. Because VI and VIS are represented as single communication primitives at the API, they can only have one request parameter structure, one positive response structure and one negative response structure. That is the reason why there is a need to combine the request parameters of the used services to a new request structure and to combine the responses of the used services to a new response structure.

The request and response structures of VI(S) control primitives are generated by the D-server in accordance with the database templates of the individual DiagComPrimitives which are used for VI(S) on the relevant database layer. For the request parameter collection, only DiagComPrimitives which actually have request parameters are considered. The response parameter collection needs to contain all DiagComPrimitives which are executed in the context of this VI(S).

NOTE The DB-response templates of the DiagComPrimitives executed during VI/VIS in the D-server need to be created in any case, independent of the fact that they are required in the DB-response template of the runtime-generated VI/VIS service. Thus, the runtime and memory overhead should not be that critical. However, it is possible to define whether the D-server shall create a full set of request and response parameters in the DB-templates of VI(S) control primitives. Please see 9.9.5.4 for more details.

If the generated VI/VIS service for variant identification has not been able to uniquely identify an ECU Variant, a negative response is returned. The result object containing the negative response has an error code of type `eRT_NO_VARIANT_IDENTIFIED`.

The DB-template of the negative response is exactly the same as for the positive response (see creation definition in 9.9.5.4). That is, both the positive response template and the negative DB-response template are built from multiplexers containing all possible positive and negative DB-response templates of the (real) services used to obtain the information for variant identification in the server internally. However, there needs to be a separate `MCDDbResponse` object for a negative response (response type is `eLOCAL_NEGATIVE_RESPONSE`) of a VI/VIS service that also contains a positive response, as the MCDDbResponse object is the lowest level where negative and positive responses are differentiated. As a result, a generated VI/VIS service can have two separate `MCDDbResponse` objects – one for the positive responses and one for the local negative response(s).

The response of a VI/VIS control primitive does never contain any PDU – neither in the request nor in the response.

In case one of the DiagComPrimitives executed during VI/VIS terminates with a timeout, the corresponding MUX parameter in the runtime response of VI/VIS does not contain any branch. Instead, this MUX parameter will have an attached `MCDError` with the error code `eCOM_BUS_ERROR`.

In case one of the DiagComPrimitives executed during VI/VIS returns a response which could not be interpreted, the corresponding MUX parameter in the runtime response of VI/VIS does not contain any branch. 2Instead, this MUX parameter is attached a `MCDError` with error code `eRT_INVALID_RESPONSE`.

9.9.5.2 Request Parameter Structure of VI and VIS

The request parameter collection of a communication primitive (also valid for VI and VIS) should contain all required values to execute the communication primitive. In case of VI and VIS, this means that all request parameters required for the DiagComPrimitive to be executed during VI/VIS should be merged into a single collection of request parameters. This can be done by the D-server as the ODX database provides the possibility of structuring request parameters.

NOTE The short names of objects within one collection shall be unique. The short names of objects in different collections need not to be unique, even if they belong to the same service.

Generation Rule for the Request Structure of a VI(S) Control Primitive

- For each DiagComPrimitive used by a VI/VIS control primitive which has own request parameters, a `MCDDbRequestParameter` of parameter type `eSTRUCTURE` is added to the collection of request parameters of the VI/VIS primitive. The short name of this request parameter shall be the short name of the `MCDDbService` object it represents in the VI/VIS primitive.
- The collection of request parameters of this structure parameter shall be the same as the collection of request parameters in the request of the corresponding DiagComPrimitive.
- The VI/VIS primitive contains a collection of request parameters containing these structures, in alphabetic order of their short names.

An example of the generation of request parameter structures in the DB-template of a VI/VIS control primitive is shown in Figure 90 — RequestParameter of VI / VIS.

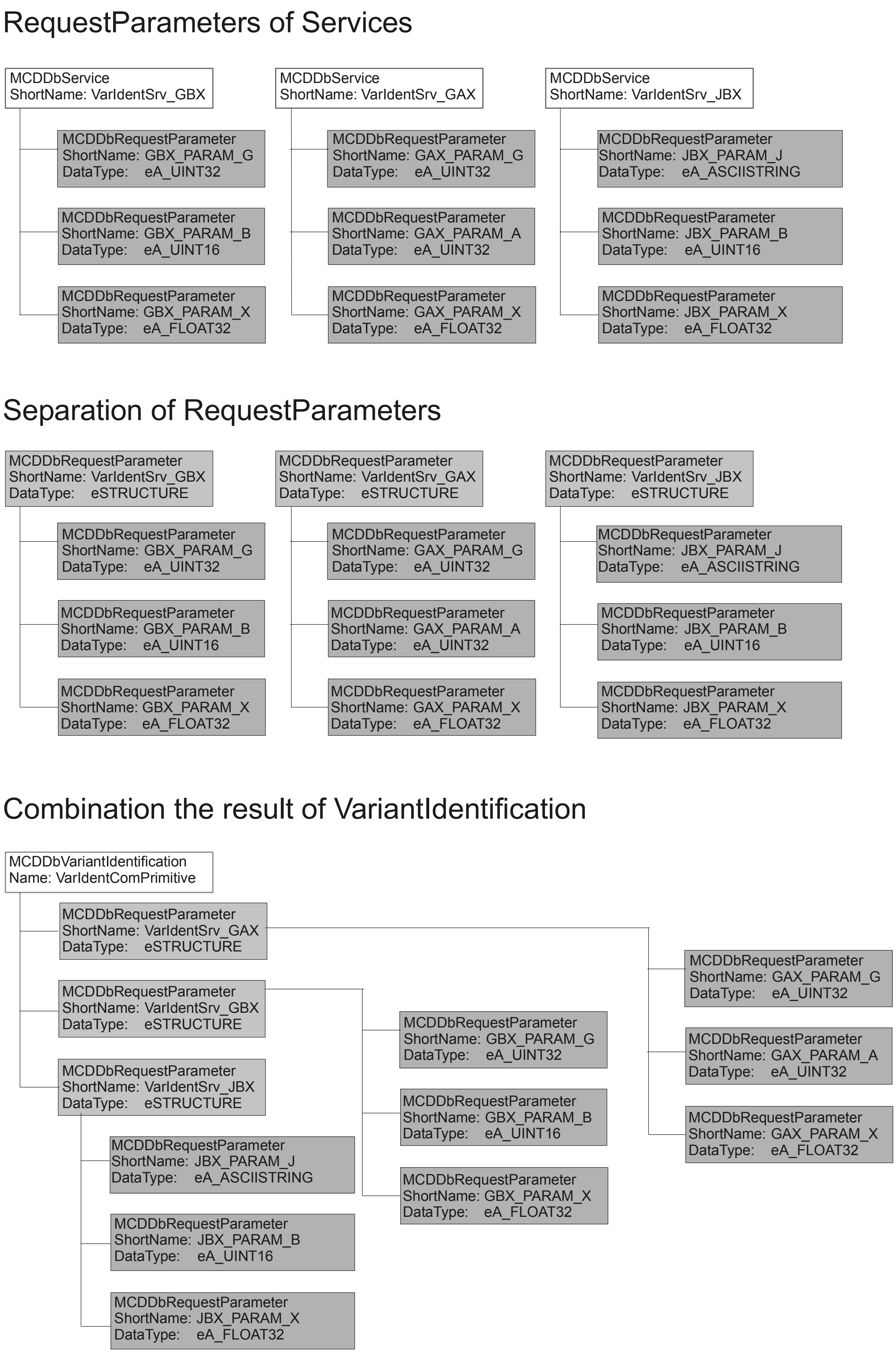

RequestParameters of Services
MCDDbService ShortName: VarIdentSrv_GBX
MCDDbRequestParameter ShortName: GBX_PARAM_G DataType: eA_UINT32
MCDDbRequestParameter ShortName: GBX_PARAM_B DataType: eA_UINT16
MCDDbRequestParameter ShortName: GBX_PARAM_X DataType: eA_FLOAT32
MCDDbService ShortName: VarIdentSrv_GAX
MCDDbRequestParameter ShortName: GAX_PARAM_G DataType: eA_UINT32
MCDDbRequestParameter ShortName: GAX_PARAM_A DataType: eA_UINT32
MCDDbRequestParameter ShortName: GAX_PARAM_X DataType: eA_FLOAT32
MCDDbService ShortName: VarIdentSrv_JBX
MCDDbRequestParameter ShortName: JBX_PARAM_J DataType: eA_ASCIISTRING
MCDDbRequestParameter ShortName: JBX_PARAM_B DataType: eA_UINT16
MCDDbRequestParameter ShortName: JBX_PARAM_X DataType: eA_FLOAT32
Separation of RequestParameters
MCDDbRequestParameter ShortName: VarIdentSrv_GBX DataType: eSTRUCTURE
MCDDbRequestParameter ShortName: GBX_PARAM_G DataType: eA_UINT32
MCDDbRequestParameter ShortName: GBX_PARAM_B DataType: eA_UINT16
MCDDbRequestParameter ShortName: GBX_PARAM_X DataType: eA_FLOAT32
MCDDbRequestParameter ShortName: VarIdentSrv_GAX DataType: eSTRUCTURE
MCDDbRequestParameter ShortName: GAX_PARAM_G DataType: eA_UINT32
MCDDbRequestParameter ShortName: GAX_PARAM_A DataType: eA_UINT32
MCDDbRequestParameter ShortName: GAX_PARAM_X DataType: eA_FLOAT32
MCDDbRequestParameter ShortName: VarIdentSrv_JBX DataType: eSTRUCTURE
MCDDbRequestParameter ShortName: JBX_PARAM_J DataType: eA_ASCIISTRING
MCDDbRequestParameter ShortName: JBX_PARAM_B DataType: eA_UINT16
MCDDbRequestParameter ShortName: JBX_PARAM_X DataType: eA_FLOAT32
Combination the result of VariantIdentification
MCDDbVariantIdentification Name: VarIdentComPrimitive
MCDDbRequestParameter ShortName: VarIdentSrv_GAX DataType: eSTRUCTURE
MCDDbRequestParameter ShortName: VarIdentSrv_GBX DataType: eSTRUCTURE
MCDDbRequestParameter ShortName: VarIdentSrv_JBX DataType: eSTRUCTURE
MCDDbRequestParameter ShortName: JBX_PARAM_J DataType: eA_ASCIISTRING
MCDDbRequestParameter ShortName: JBX_PARAM_B DataType: eA_UINT16
MCDDbRequestParameter ShortName: JBX_PARAM_X DataType: eA_FLOAT32
MCDDbRequestParameter ShortName: GBX_PARAM_G DataType: eA_UINT32
MCDDbRequestParameter ShortName: GBX_PARAM_B DataType: eA_UINT16
MCDDbRequestParameter ShortName: GBX_PARAM_X DataType: eA_FLOAT32
MCDDbRequestParameter ShortName: GAX_PARAM_G DataType: eA_UINT32
MCDDbRequestParameter ShortName: GAX_PARAM_A DataType: eA_UINT32
MCDDbRequestParameter ShortName: GAX_PARAM_X DataType: eA_FLOAT32

Figure 90 — RequestParameter of VI / VIS

9.9.5.3 Response Parameter Structure of VI and VIS

The result of a communication primitive (also valid for VI and VIS) contains one response for each affected ECU. In case of VI/VIS, these ECUs can be addressed by more than one of the executed DiagComPrimitives. Each D-server using the same ODX database shall built the database response for VI and VIS in the same way to guarantee exchangeability of the D-server. The responses of all DiagComPrimitives executed during VI/VIS are contained in the result of the generated VI/VIS service – independent of the fact whether the response(s) has/have been negative or positive. Hence, the DB-template of the positive response is defined as follows:

Generation Rule for the Response Structure of a VI(S) Control Primitive

- For each service, a response parameter of type eMUX is placed on the top-level of the response parameter structure of the VI/VIS service. This also applies if only a single DiagComPrimitive is executed during VI/VIS.
- The shortname of this MUX parameter conforms to the pattern "#RtGen_<shortname_of_diagcomprimitive>".
- Every branch of the MUX parameter references one of the DB-responses of the corresponding DiagComPrimitive. First, the positive DB-responses are added as MUX branches to the MUX parameter. Then, the local negative DB-responses are added as MUX branches. Finally, the global negative DB-responses are added as MUX branches to the MUX parameter.
- The collections of response parameters within these MUX branches shall be the same as the collections of the response parameters from the individual DiagComPrimitives the response is coming from.

The response of the VI/VIS contains a collection of MUX branches containing responses containing response parameters, in alphabetic order of their short names.

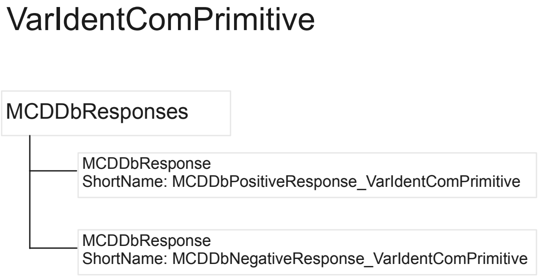

VarIdentComPrimitive
MCDDbResponses
MCDDbResponse
ShortName: MCDDbPositiveResponse_VarIdentComPrimitive
MCDDbResponse
ShortName: MCDDbNegativeResponse_VarIdentComPrimitive

Figure 91 — Result of VI / VIS

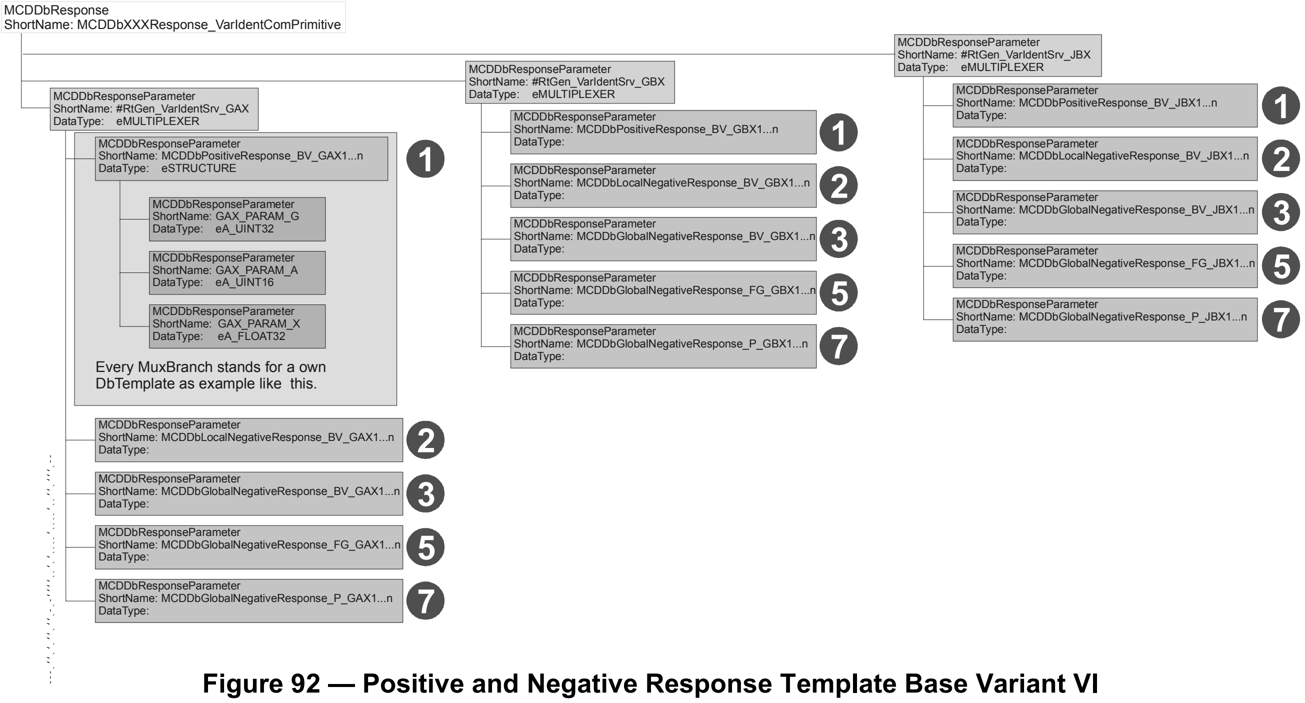

MCDDbResponse
ShortName: MCDDbXXXResponse_VarIdentComPrimitive
MCDDbResponseParameter
ShortName: #RtGen_VarIdentSrv_GAX
DataType: eMULTIPLEXER
MCDDbResponseParameter
ShortName: MCDDbPositiveResponse_BV_GAX1...n
DataType: eSTRUCTURE
MCDDbResponseParameter
ShortName: GAX_PARAM_G
DataType: eA_UINT32
MCDDbResponseParameter
ShortName: GAX_PARAM_A
DataType: eA_UINT16
MCDDbResponseParameter
ShortName: GAX_PARAM_X
DataType: eA_FLOAT32
Every MuxBranch stands for a own DbTemplate as example like this.
MCDDbResponseParameter
ShortName: MCDDbLocalNegativeResponse_BV_GAX1...n
DataType:
MCDDbResponseParameter
ShortName: MCDDbGlobalNegativeResponse_BV_GAX1...n
DataType:
MCDDbResponseParameter
ShortName: MCDDbGlobalNegativeResponse_FG_GAX1...n
DataType:
MCDDbResponseParameter
ShortName: MCDDbGlobalNegativeResponse_P_GAX1...n
DataType:
MCDDbResponseParameter
ShortName: #RtGen_VarIdentSrv_GBX
DataType: eMULTIPLEXER
MCDDbResponseParameter
ShortName: MCDDbPositiveResponse_BV_GBX1...n
DataType:
MCDDbResponseParameter
ShortName: MCDDbLocalNegativeResponse_BV_GBX1...n
DataType:
MCDDbResponseParameter
ShortName: MCDDbGlobalNegativeResponse_BV_GBX1...n
DataType:
MCDDbResponseParameter
ShortName: MCDDbGlobalNegativeResponse_FG_GBX1...n
DataType:
MCDDbResponseParameter
ShortName: MCDDbGlobalNegativeResponse_P_GBX1...n
DataType:
MCDDbResponseParameter
ShortName: #RtGen_VarIdentSrv_JBX
DataType: eMULTIPLEXER
MCDDbResponseParameter
ShortName: MCDDbPositiveResponse_BV_JBX1...n
DataType:
MCDDbResponseParameter
ShortName: MCDDbLocalNegativeResponse_BV_JBX1...n
DataType:
MCDDbResponseParameter
ShortName: MCDDbGlobalNegativeResponse_BV_JBX1...n
DataType:
MCDDbResponseParameter
ShortName: MCDDbGlobalNegativeResponse_FG_JBX1...n
DataType:
MCDDbResponseParameter
ShortName: MCDDbGlobalNegativeResponse_P_JBX1...n
DataType:
1
2
3
5
7

Figure 92 — Positive and Negative Response Template Base Variant VI

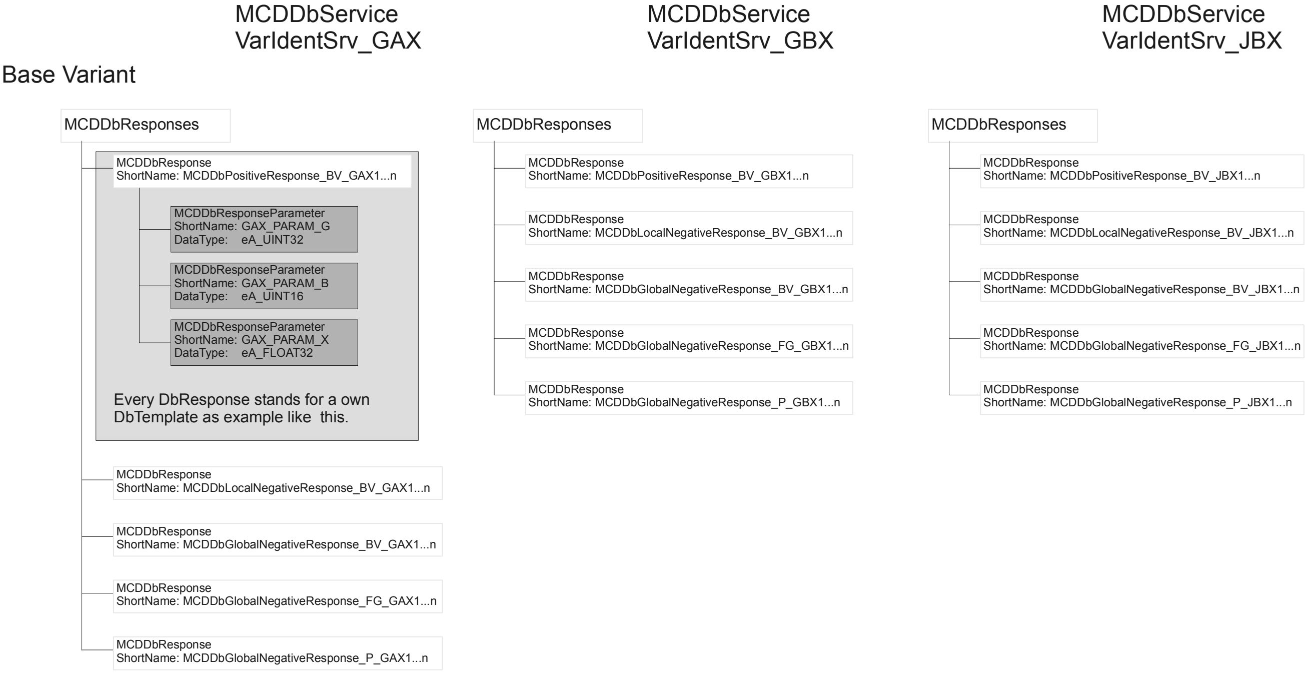

MCDDbService
VarIdentSrv_GAX
MCDDbService
VarIdentSrv_GBX
MCDDbService
VarIdentSrv_JBX
Base Variant
MCDDbResponses
MCDDbResponse
ShortName: MCDDbPositiveResponse_BV_GAX1...n
MCDDbResponseParameter
ShortName: GAX_PARAM_G
DataType: eA_UINT32
MCDDbResponseParameter
ShortName: GAX_PARAM_B
DataType: eA_UINT16
MCDDbResponseParameter
ShortName: GAX_PARAM_X
DataType: eA_FLOAT32
Every DbResponse stands for a own DbTemplate as example like this.
MCDDbResponse
ShortName: MCDDbLocalNegativeResponse_BV_GAX1...n
MCDDbResponse
ShortName: MCDDbGlobalNegativeResponse_BV_GAX1...n
MCDDbResponse
ShortName: MCDDbGlobalNegativeResponse_FG_GAX1...n
MCDDbResponse
ShortName: MCDDbGlobalNegativeResponse_P_GAX1...n
MCDDbResponses
MCDDbResponse
ShortName: MCDDbPositiveResponse_BV_GBX1...n
MCDDbResponse
ShortName: MCDDbLocalNegativeResponse_BV_GBX1...n
MCDDbResponse
ShortName: MCDDbGlobalNegativeResponse_BV_GBX1...n
MCDDbResponse
ShortName: MCDDbGlobalNegativeResponse_FG_GBX1...n
MCDDbResponse
ShortName: MCDDbGlobalNegativeResponse_P_GBX1...n
MCDDbResponses
MCDDbResponse
ShortName: MCDDbPositiveResponse_BV_JBX1...n
MCDDbResponse
ShortName: MCDDbLocalNegativeResponse_BV_JBX1...n
MCDDbResponse
ShortName: MCDDbGlobalNegativeResponse_BV_JBX1...n
MCDDbResponse
ShortName: MCDDbGlobalNegativeResponse_FG_JBX1...n
MCDDbResponse
ShortName: MCDDbGlobalNegativeResponse_P_JBX1...n

Figure 93 — Positive and Negative Response Template Single Service

9.9.5.4 Switching Database Template Generation On and Off

The DB templates for the control primitives VI and VIS can become very large. As a result, the construction of these DB templates consumes a lot of time and memory. Because these complex DB templates will only be used in engineering departments, a system property is used to switch the creation of the DB-templates for the control primitives VI and VIS on or off.

In case the system property “createVIDbTemplate” is set to false (default value), the MCDDbRequest of the control primitives MCDDbVariantIdentification and MCDDbVariantIdentificationAndSelection is empty. The generation of detailed request and response templates does not take place as described above. That is, the

VI and VIS control primitives have exactly one empty positive response and one empty local negative response (MCDDbResponse with no response parameters) in case the value of the system property is set to false. Otherwise, the database template generation takes place as described above.

9.9.6 Example Scenarios for VI and VIS

Legend for the examples for Variant Instantiation and Identification / Selection

In general, the figures in this subclause shall be read from the top to bottom following their time axis and from left to right (order of instancing elements). The method calls from client application to D-server are coloured in black (Create, VariantIdentification, VariantIdentificationAndSelection). If necessary, it is shown which reference on the client application's side issued the method call. Next to the create methods, a small box shows which reference has been created by the method call. After the call, the newly created 'reference-box' is located within the Client while the new Logical Link instance is shown within the D-server – labelled in accordance with what has been identified and/or created by the client application's calls.

Events are coloured in light grey and show which client application is receiving the corresponding event. Within the references, the type of the ECU (e.g. Door ECUFR) is given below the identification (letters: A ... D), and the interpretation layer (e.g. ECU Base Variant) below that.

The presentation is similar on the D-server side. Here, the descriptions are placed directly below the lines 'Logical Link Queue'. It shall be noted that there may be only one instance per ECU on the D-server's side. In addition, in case of a Location of type Functional Group, the interpretation levels for the data transfer direction from the ECU for both control units are given separately.

For each new example, it is stated in the top-right corner which ECU Variant of the ECUs actually are present in the imaginary vehicle (Door ECU FR is Variant1 and Door ECU FL is Variant 2).

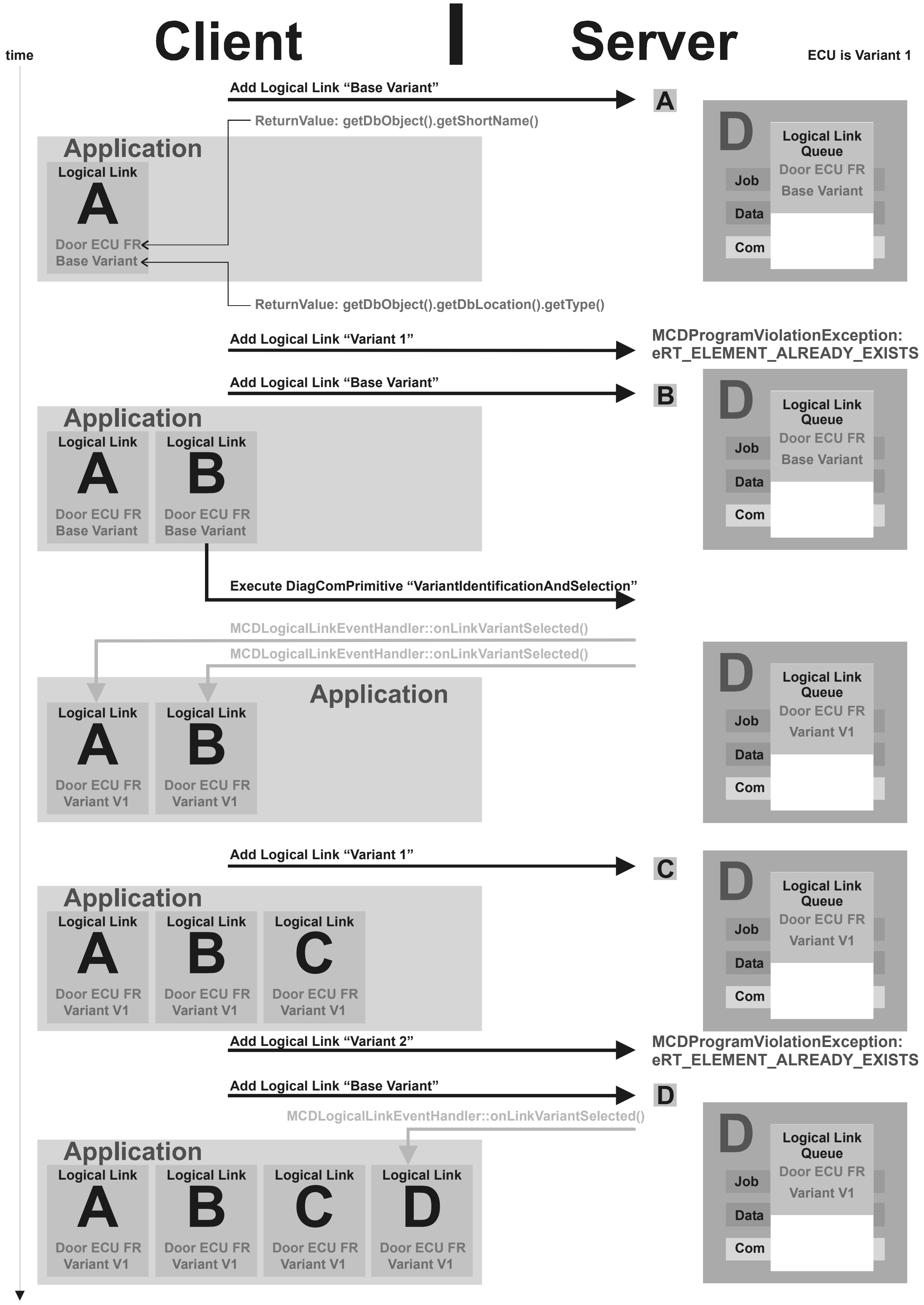

time
Client
Server
ECU is Variant 1
Add Logical Link "Base Variant"
A
ReturnValue: getDbObject().getShortName()
ReturnValue: getDbObject().getDbLocation().getType()
Application
Logical Link
A
Door ECU FR
Base Variant
D
Logical Link Queue
Door ECU FR
Base Variant
Job
Data
Com
Add Logical Link "Variant 1"
MCDProgramViolationException: eRT_ELEMENT_ALREADY_EXISTS
Add Logical Link "Base Variant"
B
Execute DiagComPrimitive "VariantIdentificationAndSelection"
MCDLogicalLinkEventHandler::onLinkVariantSelected()
MCDLogicalLinkEventHandler::onLinkVariantSelected()
Variant V1
Add Logical Link "Variant 1"
C
Add Logical Link "Variant 2"
MCDProgramViolationException: eRT_ELEMENT_ALREADY_EXISTS
Add Logical Link "Base Variant"
D
MCDLogicalLinkEventHandler::onLinkVariantSelected()

Figure 94 — Example variant instantiation and identification/ selection 1

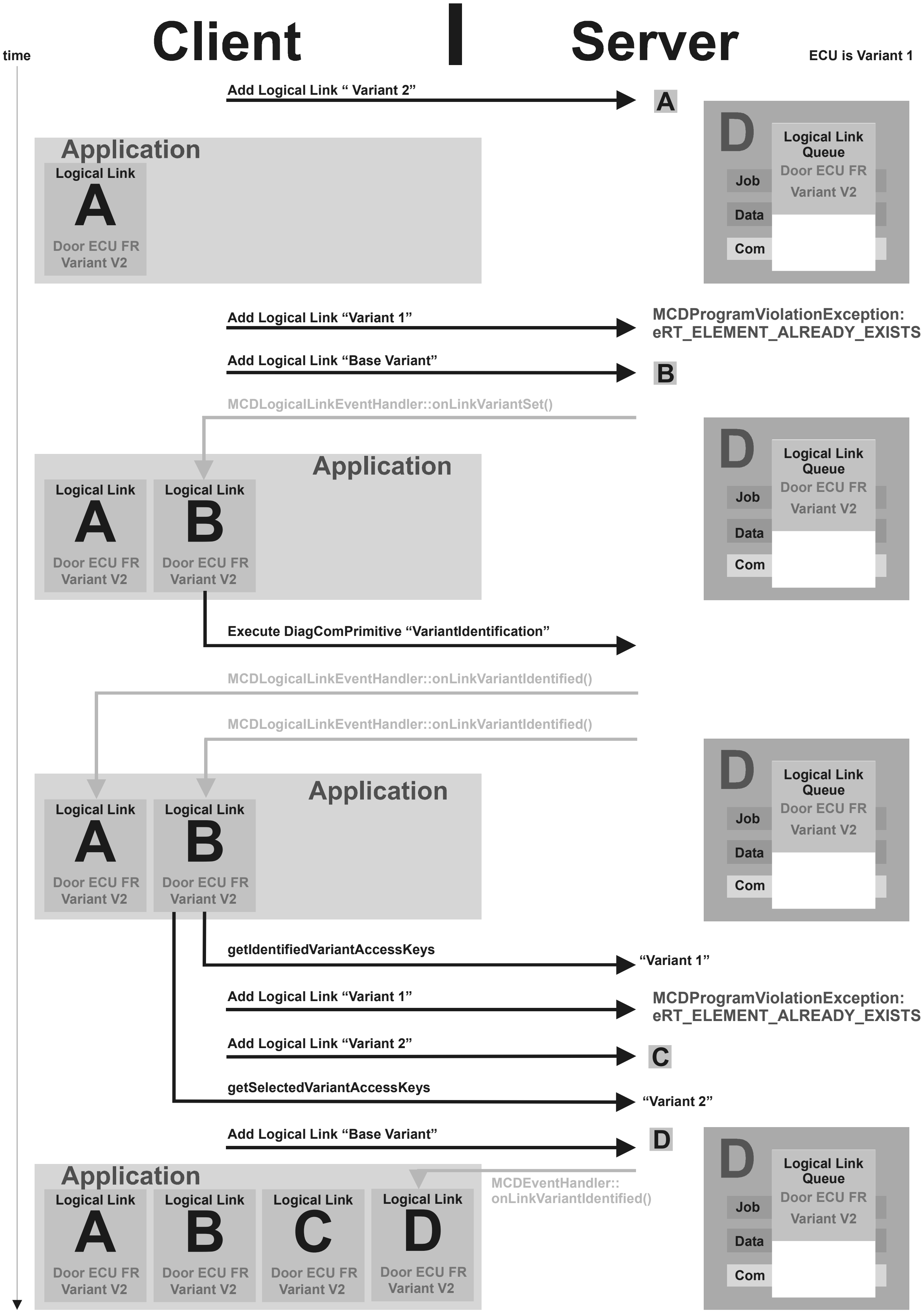

time
Client
Server
ECU is Variant 1
Add Logical Link " Variant 2"
A
D
Logical Link Queue
Door ECU FR
Variant V2
Job
Data
Com
Application
Logical Link
A
Door ECU FR
Variant V2
Add Logical Link "Variant 1"
MCDProgramViolationException:
eRT_ELEMENT_ALREADY_EXISTS
Add Logical Link "Base Variant"
B
MCDLogicalLinkEventHandler::onLinkVariantSet()
Application
Logical Link
A
Door ECU FR
Variant V2
Logical Link
B
Door ECU FR
Variant V2
Execute DiagComPrimitive "VariantIdentification"
MCDLogicalLinkEventHandler::onLinkVariantIdentified()
MCDLogicalLinkEventHandler::onLinkVariantIdentified()
getIdentifiedVariantAccessKeys
"Variant 1"
Add Logical Link "Variant 1"
MCDProgramViolationException:
eRT_ELEMENT_ALREADY_EXISTS
Add Logical Link "Variant 2"
C
getSelectedVariantAccessKeys
"Variant 2"
Add Logical Link "Base Variant"
D
MCDEventHandler::
onLinkVariantIdentified()
Application
Logical Link
A
Logical Link
B
Logical Link
C
Logical Link
D
Door ECU FR
Variant V2

Figure 95 — Example variant instantiation and identification/ selection 2

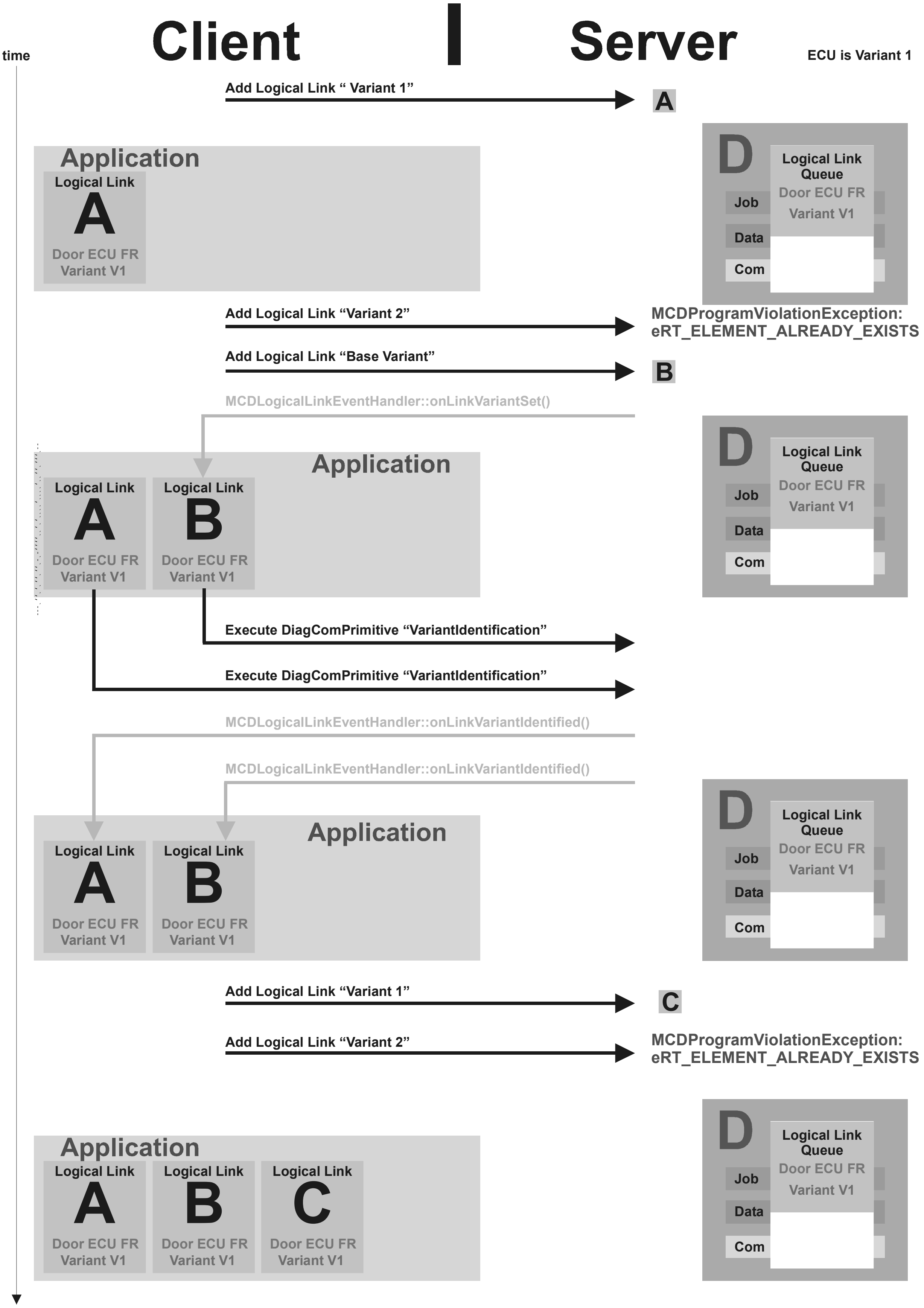

time
Client
Server
ECU is Variant 1
Add Logical Link " Variant 1"
A
Application
Logical Link
A
Door ECU FR
Variant V1
D
Logical Link
Queue
Door ECU FR
Variant V1
Job
Data
Com
Add Logical Link "Variant 2"
MCDProgramViolationException:
eRT_ELEMENT_ALREADY_EXISTS
Add Logical Link "Base Variant"
B
MCDLogicalLinkEventHandler::onLinkVariantSet()
Application
Logical Link
A
Door ECU FR
Variant V1
Logical Link
B
Door ECU FR
Variant V1
Execute DiagComPrimitive "VariantIdentification"
Execute DiagComPrimitive "VariantIdentification"
MCDLogicalLinkEventHandler::onLinkVariantIdentified()
MCDLogicalLinkEventHandler::onLinkVariantIdentified()
Add Logical Link "Variant 1"
C
Add Logical Link "Variant 2"
MCDProgramViolationException:
eRT_ELEMENT_ALREADY_EXISTS
Logical Link
C
Door ECU FR
Variant V1

Figure 96 — Example variant instantiation and identification/ selection 3

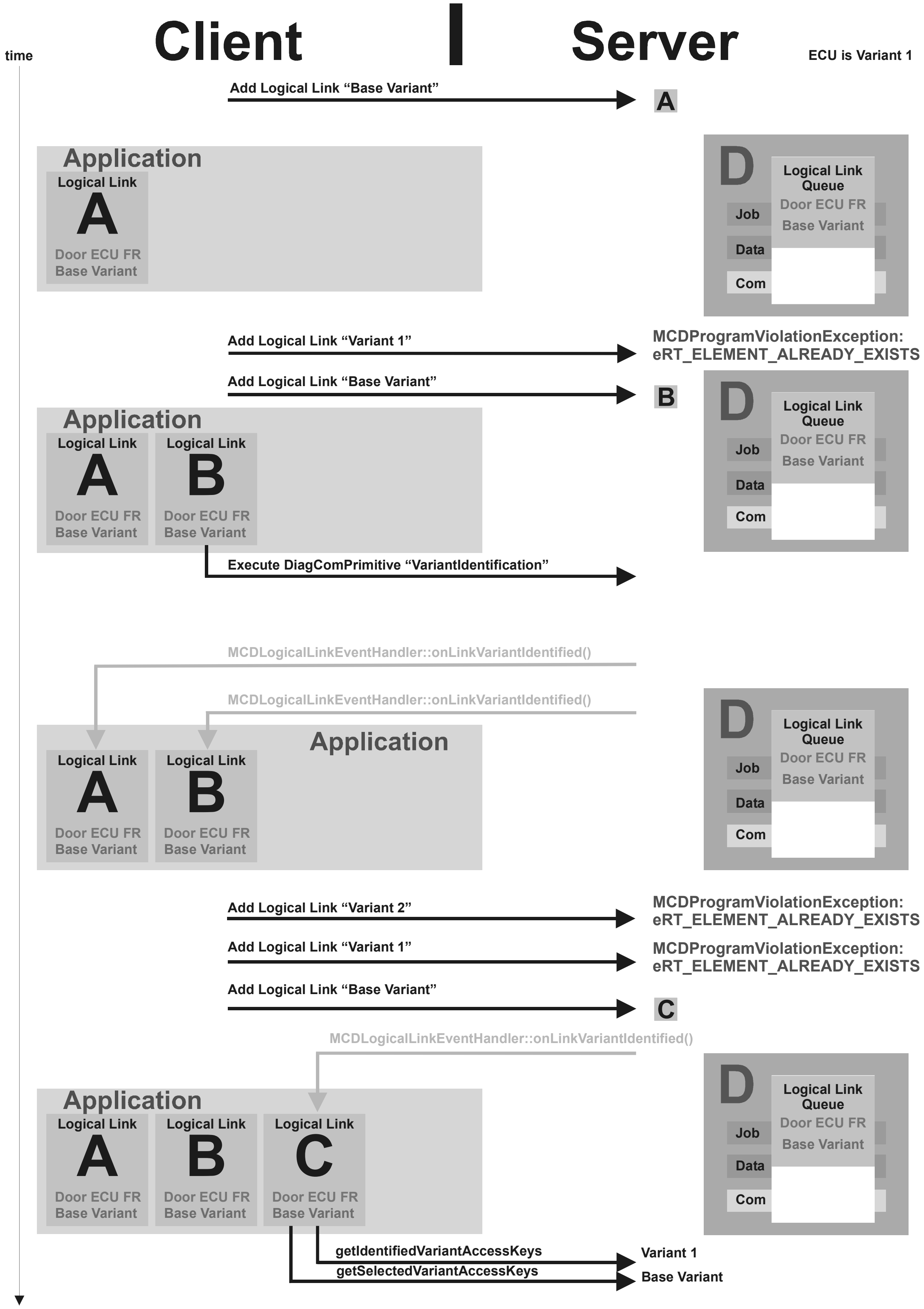

time
Client
Server
ECU is Variant 1
Add Logical Link “Base Variant”
A
Application
Logical Link
A
Door ECU FR
Base Variant
D
Logical Link
Queue
Door ECU FR
Base Variant
Job
Data
Com
Add Logical Link “Variant 1”
MCDProgramViolationException:
eRT_ELEMENT_ALREADY_EXISTS
Add Logical Link “Base Variant”
B
Logical Link
B
Execute DiagComPrimitive “VariantIdentification”
MCDLogicalLinkEventHandler::onLinkVariantIdentified()
MCDLogicalLinkEventHandler::onLinkVariantIdentified()
Add Logical Link “Variant 2”
MCDProgramViolationException:
eRT_ELEMENT_ALREADY_EXISTS
Add Logical Link “Variant 1”
MCDProgramViolationException:
eRT_ELEMENT_ALREADY_EXISTS
Add Logical Link “Base Variant”
C
MCDLogicalLinkEventHandler::onLinkVariantIdentified()
Logical Link
C
getIdentifiedVariantAccessKeys
getSelectedVariantAccessKeys
Variant 1
Base Variant

Figure 97 — Example variant instantiation and identification/ selection 4

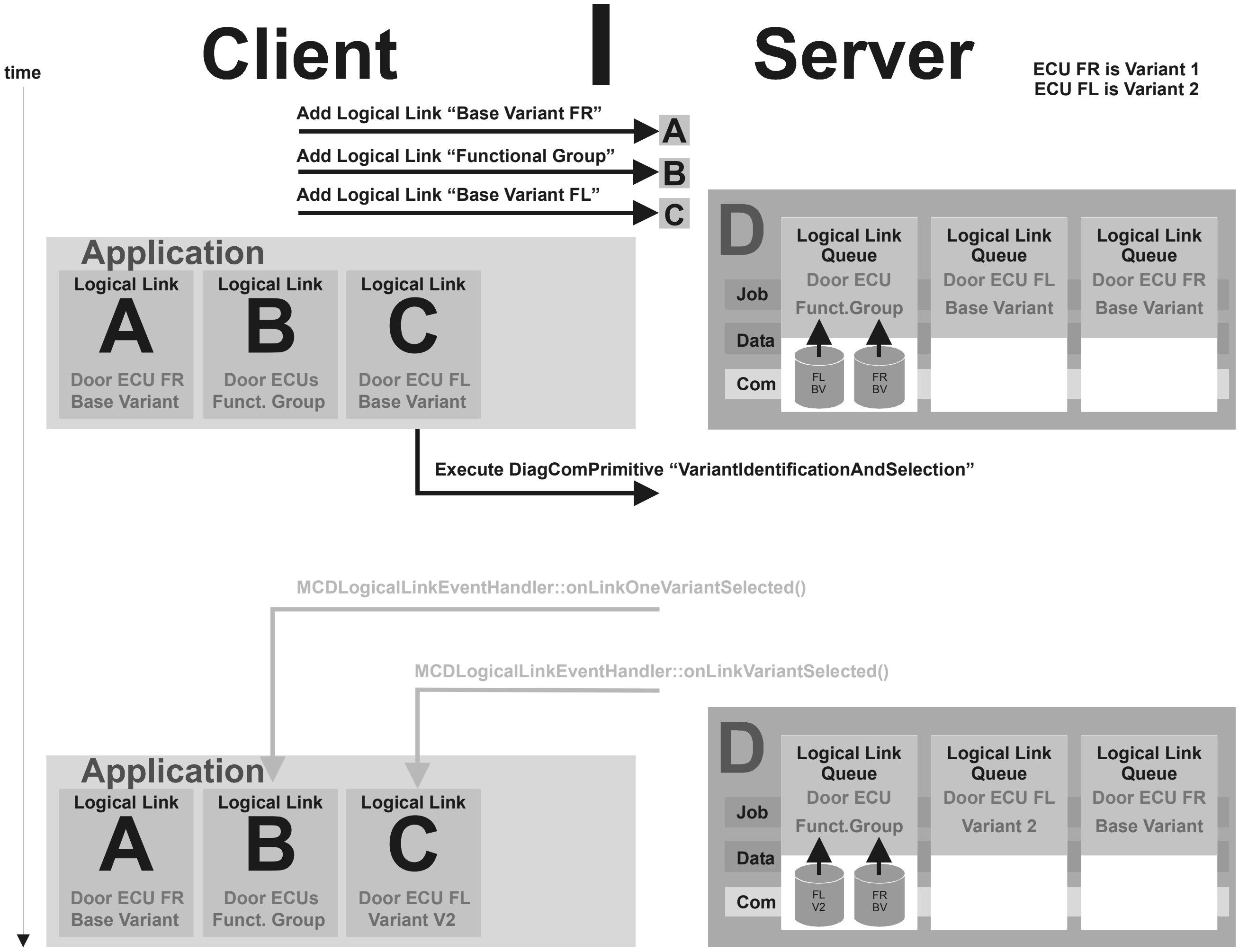


Figure 98 — Example variant instantiation and identification/ selection 5

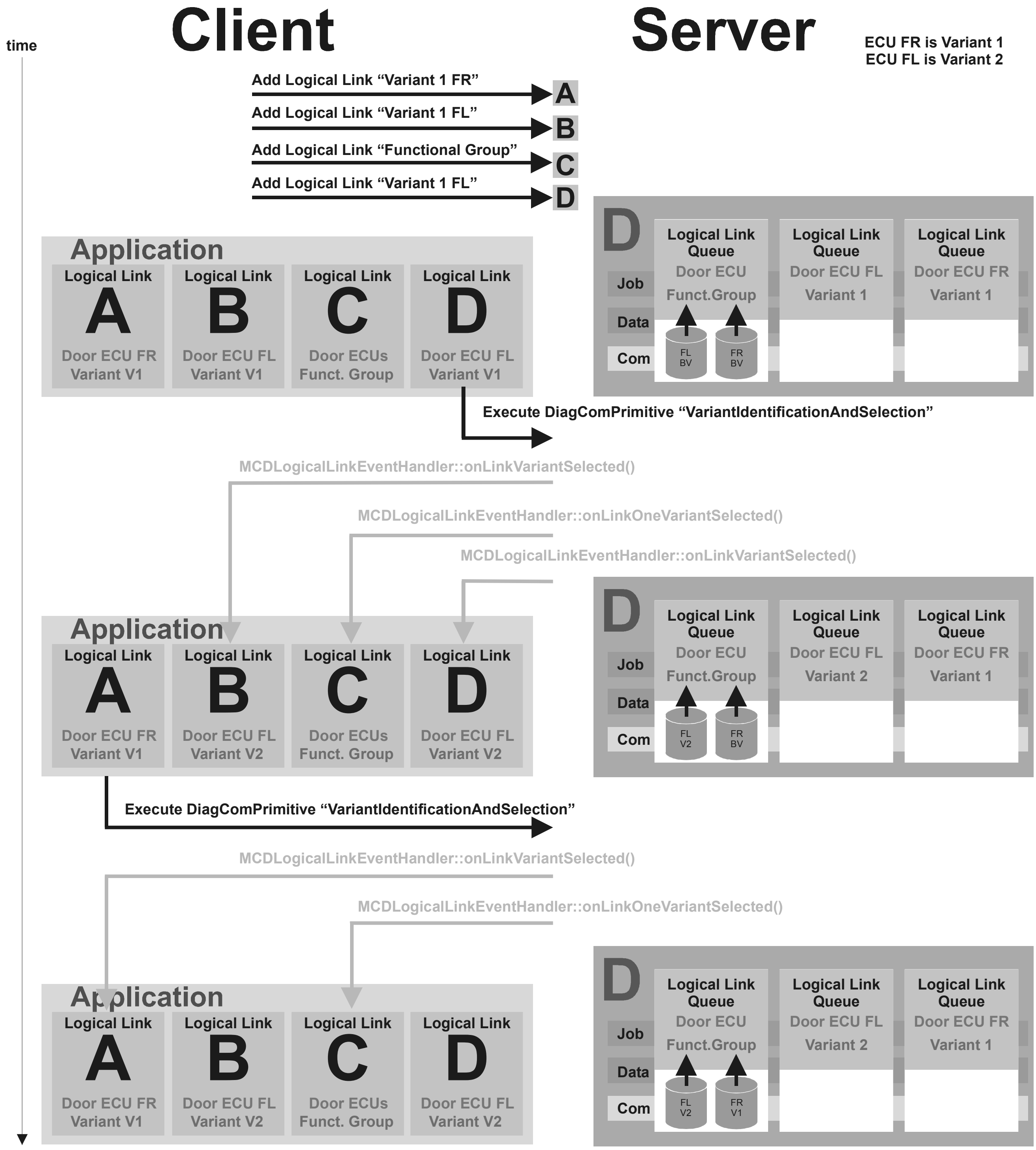

time
Client
Server
ECU FR is Variant 1
ECU FL is Variant 2
Add Logical Link “Variant 1 FR”
A
Add Logical Link “Variant 1 FL”
B
Add Logical Link “Functional Group”
C
Add Logical Link “Variant 1 FL”
D
Application
Logical Link A Door ECU FR Variant V1
Logical Link B Door ECU FL Variant V1
Logical Link C Door ECUs Funct. Group
Logical Link D Door ECU FL Variant V1
D
Job
Data
Com
Logical Link Queue Door ECU Funct.Group
FL BV
FR BV
Logical Link Queue Door ECU FL Variant 1
Logical Link Queue Door ECU FR Variant 1
Execute DiagComPrimitive “VariantIdentificationAndSelection”
MCDLogicalLinkEventHandler::onLinkVariantSelected()
MCDLogicalLinkEventHandler::onLinkOneVariantSelected()
MCDLogicalLinkEventHandler::onLinkVariantSelected()
Application
Logical Link A Door ECU FR Variant V1
Logical Link B Door ECU FL Variant V2
Logical Link C Door ECUs Funct. Group
Logical Link D Door ECU FL Variant V2
D
Job
Data
Com
Logical Link Queue Door ECU Funct.Group
FL V2
FR BV
Logical Link Queue Door ECU FL Variant 2
Logical Link Queue Door ECU FR Variant 1
Execute DiagComPrimitive “VariantIdentificationAndSelection”
MCDLogicalLinkEventHandler::onLinkVariantSelected()
MCDLogicalLinkEventHandler::onLinkOneVariantSelected()
Application
Logical Link A Door ECU FR Variant V1
Logical Link B Door ECU FL Variant V2
Logical Link C Door ECUs Funct. Group
Logical Link D Door ECU FL Variant V2
D
Job
Data
Com
Logical Link Queue Door ECU Funct.Group
FL V2
FR V1
Logical Link Queue Door ECU FL Variant 2
Logical Link Queue Door ECU FR Variant 1

Figure 99 — Example variant instantiation and identification/ selection 6

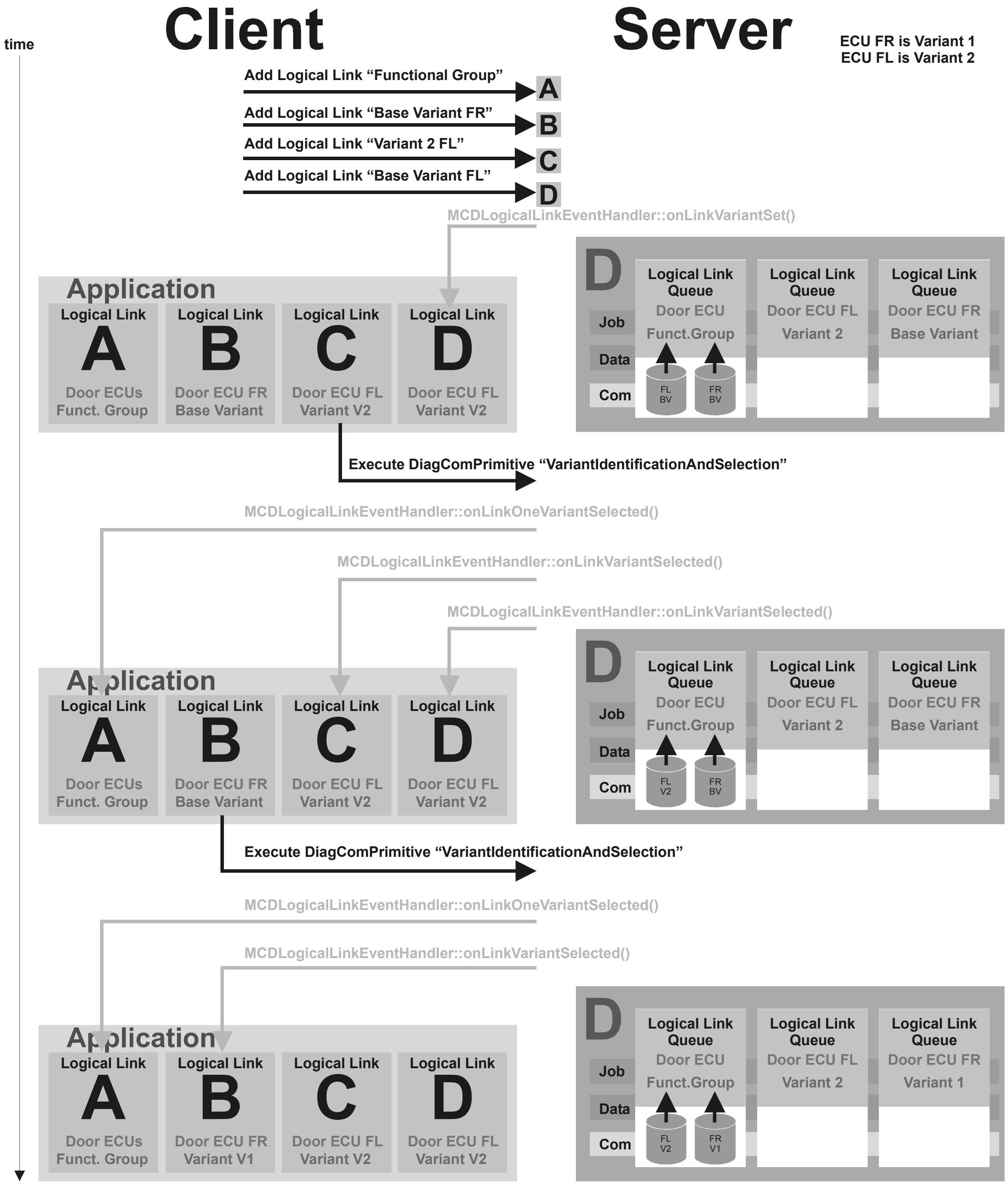


Figure 100 — Example variant instantiation and identification/ selection 7

Table 31 — Variant Identification (VI) and Variant Identification And Selection (VIS)

Instantiated Logical Link	Variant access	Interpret levels for all Procs	Allowed Logical Links	Not allowed Logical Links (error by instantiation)
Door ECU s		BV	Door ECU s	
			Door ECU FL	Door ECU FL V1 Door ECU FL V2
			|| Door ECU FL V1	Door ECU FL V2
			|| Door ECU FL V2	Door ECU FL V1
			Door ECU FR	Door ECU FR V1 Door ECU FR V2
			|| Door ECU FR V1	Door ECU FR V2
			|| Door ECU FR V2	Door ECU FR V1
	VI	BV	Door ECU s	
			Door ECU FL	Door ECU FL V1 Door ECU FL V2
			|| Door ECU FL V1	Door ECU FL V2
			|| Door ECU FL V2	Door ECU FL V1
			Door ECU FR	Door ECU FR V1 Door ECU FR V2
			|| Door ECU FR V1	Door ECU FR V2
			|| Door ECU FR V2	Door ECU FR V1
	VIS	FL V1 FR V2	Door ECU s Door ECU FL Door ECU FR Door ECU FL V1 Door ECU FR V2	Door ECU FL V2 Door ECU FR V1
Door ECU FL		BV	Door ECU s Door ECU FL	Door ECU FL V1 Door ECU FL V2
	VI	BV	Door ECU s Door ECU FL	Door ECU FL V1 Door ECU FL V2
	VIS	FL V1	Door ECU s Door ECU FL Door ECU FL V1	Door ECU FL V2

Table 31 (*continued*)

Instantiated **Logical Link**	**Variant access**	**Interpret levels for all Procs**	**Allowed Logical Links**	**Not allowed Logical Links (error by instantiation)**
Door ECU FL V1		FL V1	Door ECU s Door ECU FL Door ECU FL V1	Door ECU FL V2
	VI	FL V1	Door ECU s Door ECU FL Door ECU FL V1	Door ECU FL V2
	VIS	FL V1	Door ECU s Door ECU FL Door ECU FL V1	Door ECU FL V2
Door ECU FL V2		FL V2	Door ECU s Door ECU FL Door ECU FL V2	Door ECU FL V1
	VI	FL V2	Door ECU s Door ECU FL Door ECU FL V2	Door ECU FL V1
	VIS	FL V1	Door ECU s Door ECU FL Door ECU FL V1	Door ECU FL V2
Door ECU FR		BV	Door ECU s Door ECU FR	Door ECU FR V1 Door ECU FR V2
	VI	BV	Door ECU s Door ECU FR	Door ECU FR V1 Door ECU FR V2
	VIS	FR V2	Door ECU s Door ECU FR Door ECU FR V2	Door ECU FR V1
Door ECU FR V1		FR V1	Door ECU s Door ECU FR Door ECU FR V1	Door ECU FR V2
	VI	FR V1	Door ECU s Door ECU FR Door ECU FR V1	Door ECU FR V2
	VIS	FR V1	Door ECU s Door ECU FR Door ECU FR V2	Door ECU FR V1

Table 31 (*continued*)

Instantiated **Logical Link**	**Variant access**	**Interpret levels for all Procs**	**Allowed Logical Links**	**Not allowed Logical Links (error by instantiation)**
Door ECU FR V2		FR V2	Door ECU s Door ECU FR Door ECU FR V2	Door ECU FR V1
	VI	FR V2	Door ECU s Door ECU FR Door ECU FR V2	Door ECU FR V1
	VIS	FR V2	Door ECU s Door ECU FR Door ECU FR V2	Door ECU FR V1

Assumption: DoorECU FL V1 and DoorECU FR V2 really exist in the imaginary vehicle.

9.10 Base Variant Identification and Selection

A use case which is related to ECU identification is the base variant identification and selection (BVIS) functionality. BVIS is used to determine which ECU of a set of possibly equivalent ECUs is present in a specific vehicle. As an example, consider engine control ECUs where only a single ECU of a wide range of potentially very different ECUs is actually built into a specific vehicle (see Figure 101 — Alternative ECUs in a vehicle modeled using ECU Groups). A typical example of such a scenario are the engine ECUs available for a vehicle platform. Most often, there are several different engines available for a vehicle platform which usually require different engine ECUs, e.g. one for the diesel engines and one for the gas engines. However, a real vehicle will only have one of these ECUs built-in as there is only one engine present.

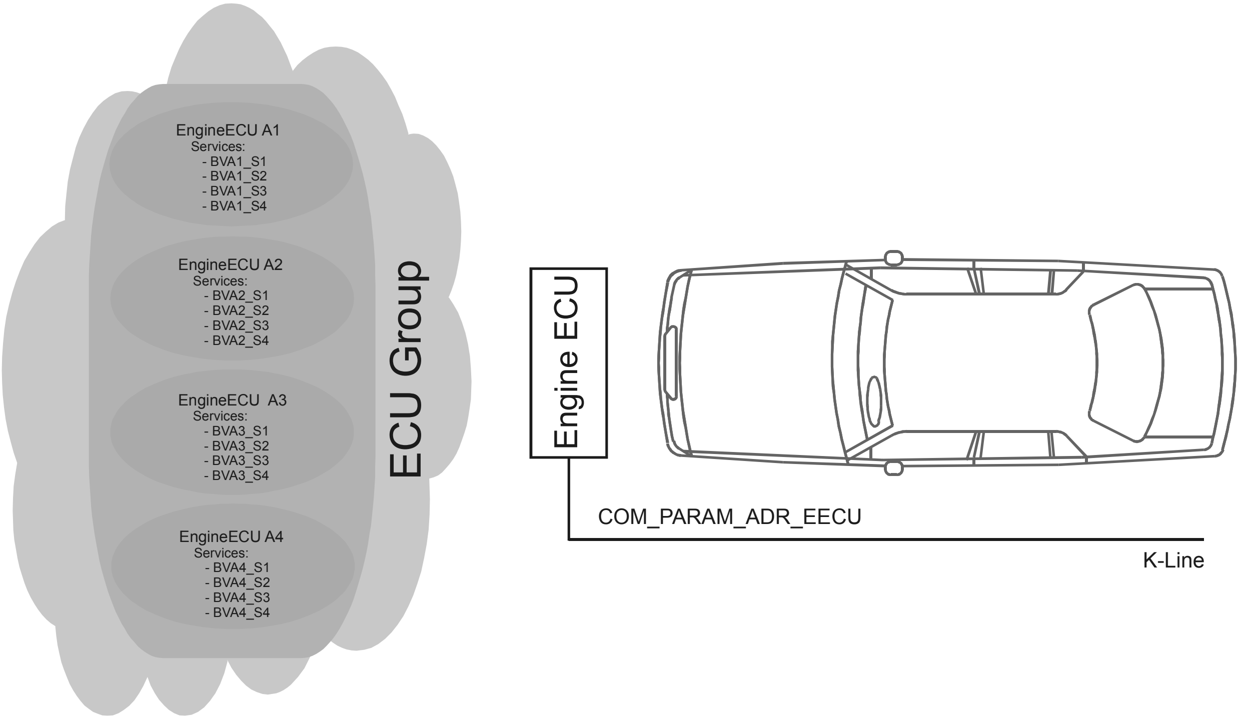


Figure 101 — Alternative ECUs in a vehicle modeled using ECU Groups

To model such a setup, the concept of ECU Groups has been introduced in ODX. An ECU Group allows to group ECU Base Variants which represent alternative ECUs in a vehicle platform. In this part of ISO 22900, an ECU Group is represented by an object of type `MCDDbEcuGroup`. A `MCDDbEcuGroup` references a set of `MCDDbEcuBaseVariants` which describe the different ECUs this group is composed of. Furthermore, ECU Groups allow for identifying the ECU Base Variant of the physical ECU which is actually present in a vehicle. For this purpose, each `MCDDbEcuBaseVariant` may include a set of matching patterns – similarly to the patterns used for variant identification (VI/VIS). These matching patterns can be used by the D-server to identify which of the ECU Base Variants being part of an ECU Group is actually present in the vehicle.

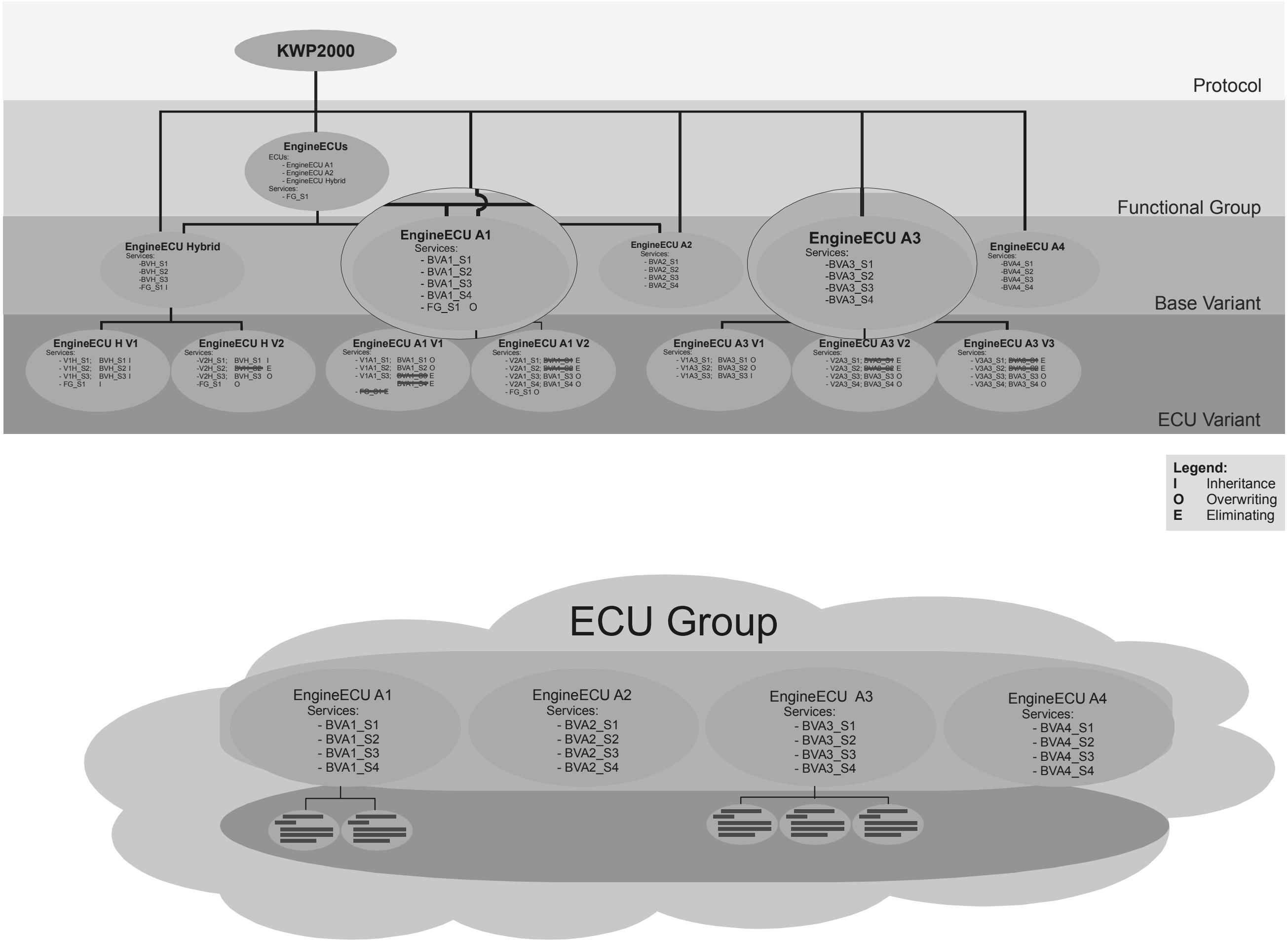


Figure 102 — An example engine ECU data pool is used to build an ECU Group

Figure 102 — An example engine ECU data pool is used to build an ECU Group illustrates how an ECU Group is logically assembled from several ECU Base Variants. ECU Base Variants can be part of an ECU Group regardless of whether they are contained in a Functional Group or not. Even more, if one of the ECU Base Variants in an ECU Group is contained in a Functional Group, it is not required that the rest of the ECU Base Variants in the same ECU Group is also contained in this Functional Group. In the example in Figure 102 — An example engine ECU data pool is used to build an ECU Group, the ECU Base Variants EngineECU A1 and EngineECU A2 are part of the Functional Group EngineECUs while EngineECU A3 and EngineECU A4 are not.

However, in case more than one ECU Base Variant being part of the same ECU Group is also contained in the same Functional Group, it is recommended to perform a Base Variant Identification and Selection (BVIS) before executing any functional communication via the Functional Group. Otherwise, it may not be possible to correctly resolve the responses to functional services. The reason is that ECU Base Variants contained in the same ECU Group typically represent alternative ECUs sharing the same address information. If a functional service would now be executed on Functional Group level, the D-server might use the “wrong” ECU Base Variant to resolve the responses, i.e. the ECU Base Variant of an ECU which is not built into the vehicle currently diagnosed. Here, BVIS can easily be used to identify and exclude those ECU Base Variants from response resolution which are not present in a vehicle. In the example of Figure 101 — Alternative ECUs in a vehicle modeled using ECU Groups, the Base Variant Identification would be necessary to determine which engine ECU is actually present in the vehicle (EngineECU A1, EngineECU A2, EngineECU A3 or EngineECU A4) before any response received when communication functionally via the Functional Group EngineECUs can really be interpreted.

ECU Groups are only designed to represent the alternative ECU Base Variants for one physical ECU in a vehicle. This means that in case several similar physical ECUs are present in a vehicle, there needs to be a separate ECU Group containing the alternative ECU Base Variants for each physical ECU. For example, a vehicle may contain four physical door ECUs – one for each of the four doors. And there may be two different suppliers A and B for these door ECUs. To support that each of the four doors can have a door ECU of either of these two suppliers (in arbitrary order), there need to be four different ECU Groups – one for each door ECU. This example is illustrated in Figure 103 — Physical ECUs with alternative ECU Base Variants require separate ECU Groups.

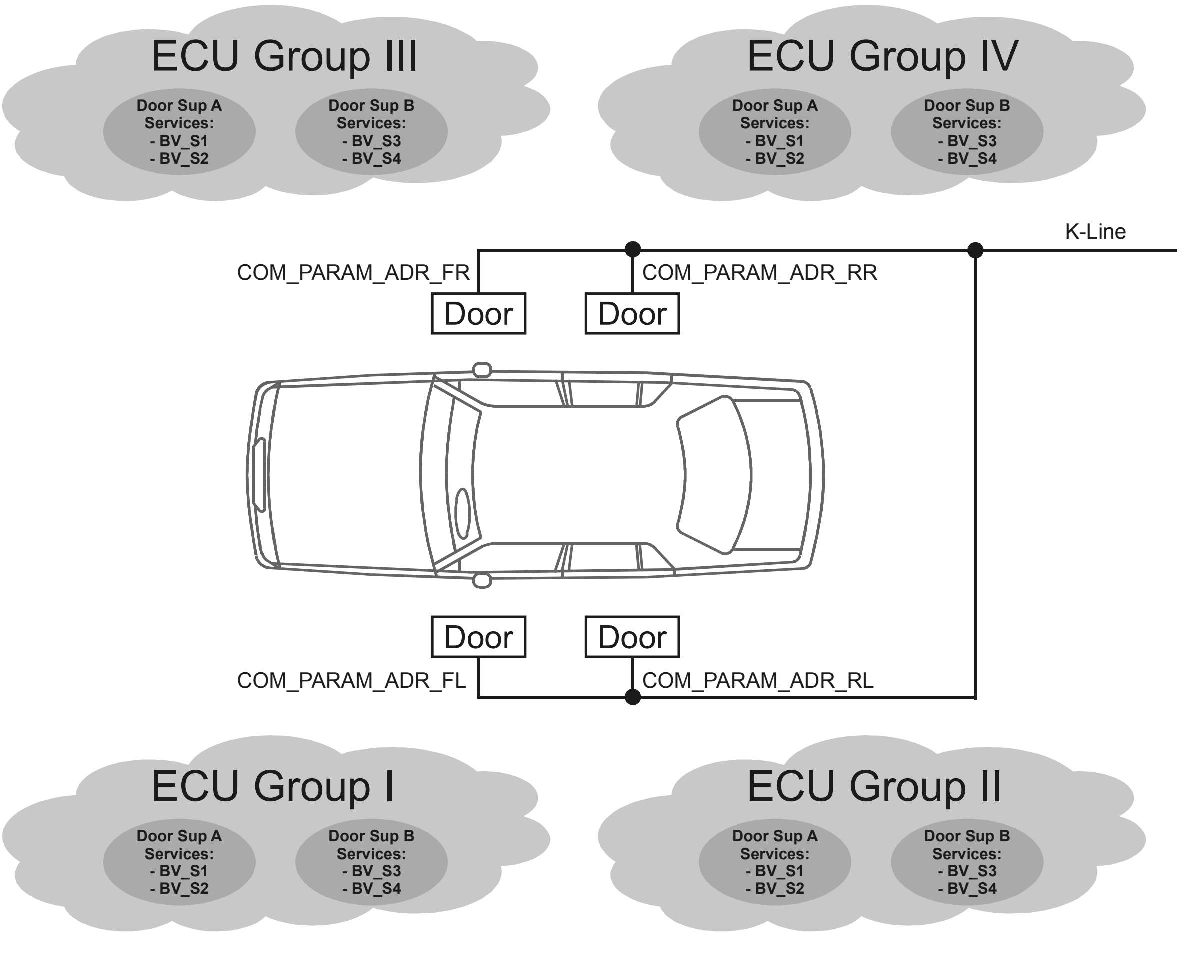

ECU Group III
Door Sup A Services: - BV_S1 - BV_S2
Door Sup B Services: - BV_S3 - BV_S4
ECU Group IV
Door Sup A Services: - BV_S1 - BV_S2
Door Sup B Services: - BV_S3 - BV_S4
K-Line
COM_PARAM_ADR_FR
COM_PARAM_ADR_RR
Door
Door
Door
Door
COM_PARAM_ADR_FL
COM_PARAM_ADR_RL
ECU Group I
Door Sup A Services: - BV_S1 - BV_S2
Door Sup B Services: - BV_S3 - BV_S4
ECU Group II
Door Sup A Services: - BV_S1 - BV_S2
Door Sup B Services: - BV_S3 - BV_S4

Figure 103 — Physical ECUs with alternative ECU Base Variants require separate ECU Groups

The one major difference between BVI(S) and the variant identification scenarios described in 9.9 is that in case of a BVI(S), the type of ECUs contained in the same ECU Group can be quite different, while VI(S) just determines the present “flavor” of the same physical ECU. For example, a basic vehicle radio controller might have nothing in common with a full-blown multimedia system. But both can be part of the same ECU Group. This is in contrast to the relationship between ECU base variants and ECU variants, where the ECU variants (at least in theory) share the functionality of their base variants. This has two implications on the mechanism of BVI(S):

— There is no ‘common’ base variant (with an accompanying logical link) to use for variant identification. Instead, when trying to identify a base variant, each ECU of an ECU Group can have its own logical link and associated diagnostic layer data, completely independent of the other ECUs in the ECU Group.

— For this reason, when performing a BVI(S), the D-server shall instantiate a new logical link for each member of the ECU Group member which is to be resolved. Furthermore, potentially different services need to be executed on each of these Logical Links in order to identify the corresponding ECUs or mark them as absent. This is in contrast to VI(S) where the set of ECU variants to be identified share the same Logical Link and the same set of diagnostic services that are used for variant identification (as they are all based on the same ECU base variant).

The representation of ECU Groups in the D-server API is depicted in Figure 4. A `MCDDbEcuGroups` collection can be retrieved via an object of type `MCDDbVehicleInformation`. This `MCDDbEcuGroups` collection contains those `MCDDbEcuGroup` objects which are available in this vehicle information table.

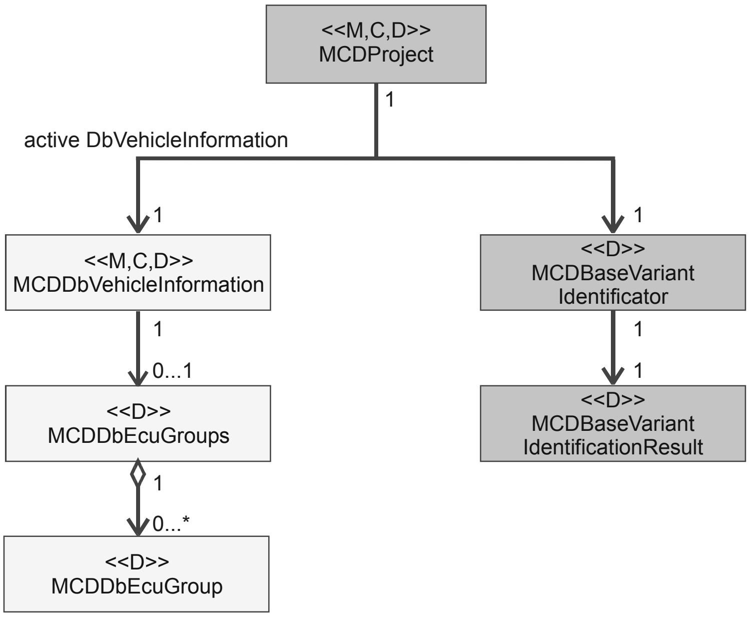


Figure 104 — ECU Groups and Base Variant Identificator are part of the D-server API

Furthermore, Figure 104 — ECU Groups and Base Variant Identificator are part of the D-server API shows the classes that are used by a client application to perform the actual base variant identification – the `MCDBaseVariantIdenfiticator`, which can be retrieved at the `MCDProject`, and a `MCDBaseVariantIdentificationResult`, which carries the result of a base variant identification operation.

When a BVI(S) is executed by the client application (via the methods `MCDBaseVariantIdentificator::executeBaseVariantIdentification()` or `MCDBaseVariantIdentificator::executeBaseVariantIdentificationAndSelection()`), the following steps should be performed by the D-server:

Step 1: The ECU base variants within the ECU Group to be resolved are iterated in their order of occurrence in the ODX data (order of references to ECU Base Variants in ECU Group).

Step 2: For the current ECU base variant, iterate through the ECU base variant’s matching patterns in the order they are defined in the ODX data. Currently, there is only a single BASE-VARIANT-PATTERN per ECU Base Variant in ODX.

Step 3: For the current matching pattern, iterate through the matching parameters in the order they are defined in the matching pattern.

Step 4: Execute the DiagComPrimitive associated with the matching parameter. If the matching parameter is supposed to be executed functionally (ODX: USE-PHYSICAL-ADDRESSING = false), use the functional logical link (ODX: FUNCTIONAL-RESOLUTION-LINK) referenced from the ECU Group for executing the service. If the matching supposed to be executed physically, use the physical logical link (ODX: PHYS-RESOLUTION-LINK) referenced from the ECU Group. If the flag USE-PHYSICAL-ADDRESSING is not defined in the ODX data explicitly, its default value 'true' is to be assumed and physical resolution is to be performed.

Step 5: If the execution of the DiagComPrimitive fails because of a communication error on the associated logical link, continue with the next ECU base variant step 1. The current ECU base variant is considered not to be present in the currently diagnosed vehicle.

Step 6: If the execution of the DiagComPrimitive fails because the logical link to be used is inaccessible (e.g. locked by another client), BVI(S) will fail with a MCDProgramViolationException with error code eRT_BASE_VARIANT_IDENTIFICATION_ABORTED.

Step 7: If the matching parameter associated with the currently executed DiagComPrimitive does not match with the value of the corresponding response parameter in the current response, the current ECU base variant is considered as not identified. Continue with the next ECU base variant in step 1.

Step 8: As soon as all matching parameters within the current ECU base variant's matching pattern have matched, this ECU base variant is considered as identified. Continue in step 9.

Step 9: If an ECU base variant was identified successfully, return a `MCDBaseVariantIdentificationResult` object which contains the `MCDDbDLogicalLink` that points to the identified ECU base variant. In case of ECU Base Variant Identification and Selection (BVIS), the D-server additionally blocks creation of logical links to ECU (Base) Variants which have been discarded (not matched) and invalidates logical links to such ECU (Base) Variants if these already exist in the logical links collection of the current project. If an ECU Base Variant has been discarded, this also applies to all of its ECU Variants.

Step 10: If no ECU base variant was identified for an ECU Group, requesting the identified base variant logical link will result in a MCDProgramViolationException with the error code eRT_NO_BASE_VARIANT_IDENTIFIED.

The BVI(S) methods will return a `MCDBaseVariantIdentificationResult` object which contains the `MCDDbDLogicalLink` object that has been identified (if any). It is distinguished between two categories of errors that occur during a BVI(S):

— In case communication on a logical link fails (execution of a communication primitive results in a communication error), the D-server will continue the BVI(S) with the next member in the ECU Group. The logical links which caused communication errors can be retrieved by the client application by using the method `MCDBaseVariantIdentificationResult::getFailedDbLogicalLinks()`.

— In case the link to be used for executing a communication primitive for BVI(S) is available but not useable by the D-server (e.g. locked by another client or already opened by another client with a cooperation level of `eNO_COOPERATION` or `eREAD-ONLY`), the BVI(S) will be aborted. In this case, a call to `MCDBaseVariantIdentificationResult::getIdentifiedBaseVariantDbLogicalLink()` will result in a `MCDProgramViolationException` with the error code eRT_NO_BASE_VARIANT_IDENTIFIED. The abortion is necessary to avoid that the wrong ECU base variant is identified by the D-server by accident. If BVI(S) would not be aborted here, a subsequently tested ECU base variant could result in a valid match although the previously tested ECU base variant with the inaccessible logical link would actually have been the correct ECU base variant. The logical link that caused this error can be retrieved by the client application by using the `MCDBaseVariantIdentificationResult::getFailedDbLogicalLinks()` method.

In order to discard the results of Base Variant Identification and Selection for a specific ECU Group, e.g. to reactivate other ECU Base Variants after a broken ECU has been replaced in a vehicle in the workshop, it is sufficient to execute Base Variant Identification and Selection for the ECU Group again. The D-server then re-matches all Base Variant Patterns again and such that the matching process may identify and select a different ECU Base Variant. Again, the D-server blocks creation of logical links to ECU (Base) Variants which have been discarded (not matched) and invalidates logical links to such ECU (Base) Variants if these already exist in the logical links collection of the current project. If an ECU Base Variant has been discarded, this also applies to all of its ECU Variants.

NOTE 1 Any DiagComPrimitive defined at an invalidated logical link is also invalidated and cannot be executed any more (MCDDiagComPrimitiveState = eNOT_EXECUTABLE).

There are some special considerations to take into account when performing functional communication while ECU Groups are present in the ODX data. Consider the situation where a functional group contains an ECU base variant which is also part of an ECU Group. In this case, the D-server ideally would have to know which member of the ECU Group is actually present in the vehicle in order to be able to resolve any physical responses to the functional requests that came from ECUs within the ECU Group. There are two options for the D-server to handle such a case. These options depend on whether a base variant identification (BVI) or a base variant identification and selection (BVIS) has been executed before[2] (through the `MCDBaseVariantIdentificator::executeBaseVariantIdentification()` or `MCDBaseVariantIdentificator::executeBaseVariantIdentificationAndSelection()` methods).

- If a service is executed on a Functional Group layer which contains ECU base variants being part of an ECU Group and if no Base Variant Identification and Selection (BVIS) has been executed before, the D-server considers all members of this Functional Group to resolve any response returned from a physical ECU. This includes any ECU Base Variant being part of a non-resolved ECU Group and might result into mismatches during response resolution.

- If a service is executed on a Functional Group layer which contains ECU base variants being part of an ECU Group and if a Base Variant Identification and Selection (BVIS) has been executed before, the D-server will only consider those members of the Functional Group to resolve any response returned from a physical ECU:

  - which are either not part of any resolved ECU Group or

  - which represent the ECU Base Variant which has been identified and selected for a specific ECU Group by means of Base Variant identification and Selection (BVIS) before.

To guarantee meaningful results in functional communications, it is recommended to resolve at least those ECU Groups which share at least one ECU Base Variant with the Functional Group before any functional communication is started. For this purpose, the client application is to determine which ECU Groups have a non-empty intersection with the Functional Group to be used for functional communication. Then, the client application shall start Base Variant Identification and Selection for all of these ECU Groups. Leaving the decision of which ECU Groups need to be resolved to the client application is compromise which allows for maximum flexibility and improves the performance of client applications for production purposes.

In ODX, every matching parameter associated with an ECU base variant can also have an attribute (ODX: USE-PHYSICAL-ADDRESSING) which determines whether the associated DiagComPrimitive to retrieve the actual value from the vehicle is to be executed using physical or functional addressing. If the referenced DiagComPrimitive only supports one addressing mode, the value of this attribute is to be ignored. If the addressing mode of the DiagComComPrimitive is functional-or-physical, the value of this attribute determines whether the DiagComPrimitive for that matching parameter should be executed functionally or physically.

NOTE 2 If a functional service executed by the D-server as part of a BVI(S) belongs to a Functional Group that in turn includes a member of the ECU Group that was supposed to be identified in the first place, the D-server will directly use the ECU base variant referenced by the ECU Group member that included the functional service in its matching patterns for resolution of the response to that functional service. In this case, the D-server does not iterate through the other group members for resolution of the response.

[2] Actually, the behavior of the D-server in case of a BVI is the same as if no BVI had been executed at all.

The cases result in a mismatch with respect to a matching parameter:

a) A matching parameter references a physical DiagComPrimitive (addressing mode = ePHYSICAL) but only a functional Logical Link is defined in the ODX data.

b) A matching parameter references a functional DiagComPrimitive (addressing mode = eFUNCTIONAL) but only a physical Logical Link is defined in the ODX data.

To allow client applications access to the results of the DiagComPrimitives executed by the D-server during a BVI(S), the `MCDBaseVariantIdentificator` class offers the possibility to register a `MCDDiagComPrimitiveEventHandler`. If event handlers registered at such an object are not explicitly released by the client, they will be released by the D-server when the `MCDProject::releaseEcuBaseVariantIdentificator()` method is called.

NOTE 3 All objects created by the D-server during a BVI(S) (e.g. logical links and communication primitives) will be visible throughout the D-server. The objects created by the D-server will belong to the client that executed the BVI(S), and will have the same cooperation level that the client specified when first retrieving the `MCDBaseVariantIdentificator` by using the `MCDProject::createBaseVariantIdentificator()` method. After the execution of the BVI(S), the client application can either release these objects individually, or use the `MCDProject::releaseBaseVariantIdentificator()` method to allow the D-server to free resources created during the BVI(S).

NOTE 4 A base variant identification can be a time-consuming process. Imagine an ECU Group containing 10 ECUs, with 10 diagnostic services to be executed for each ECU. Now, if only the last ECU is actually present in the vehicle, the D-server will iterate through the first 9 ECUs, executing all 10 services for each of these 9 ECUs. When (in the worst case) all of these 90 services result in a timeout because the actual ECU is not present in the vehicle, leading to an execution time of at least 180 seconds if each timeout takes 2 seconds.

9.11 Use Cases

9.11.1 Creation of LogicalLink and usage of DiagComPrimitives

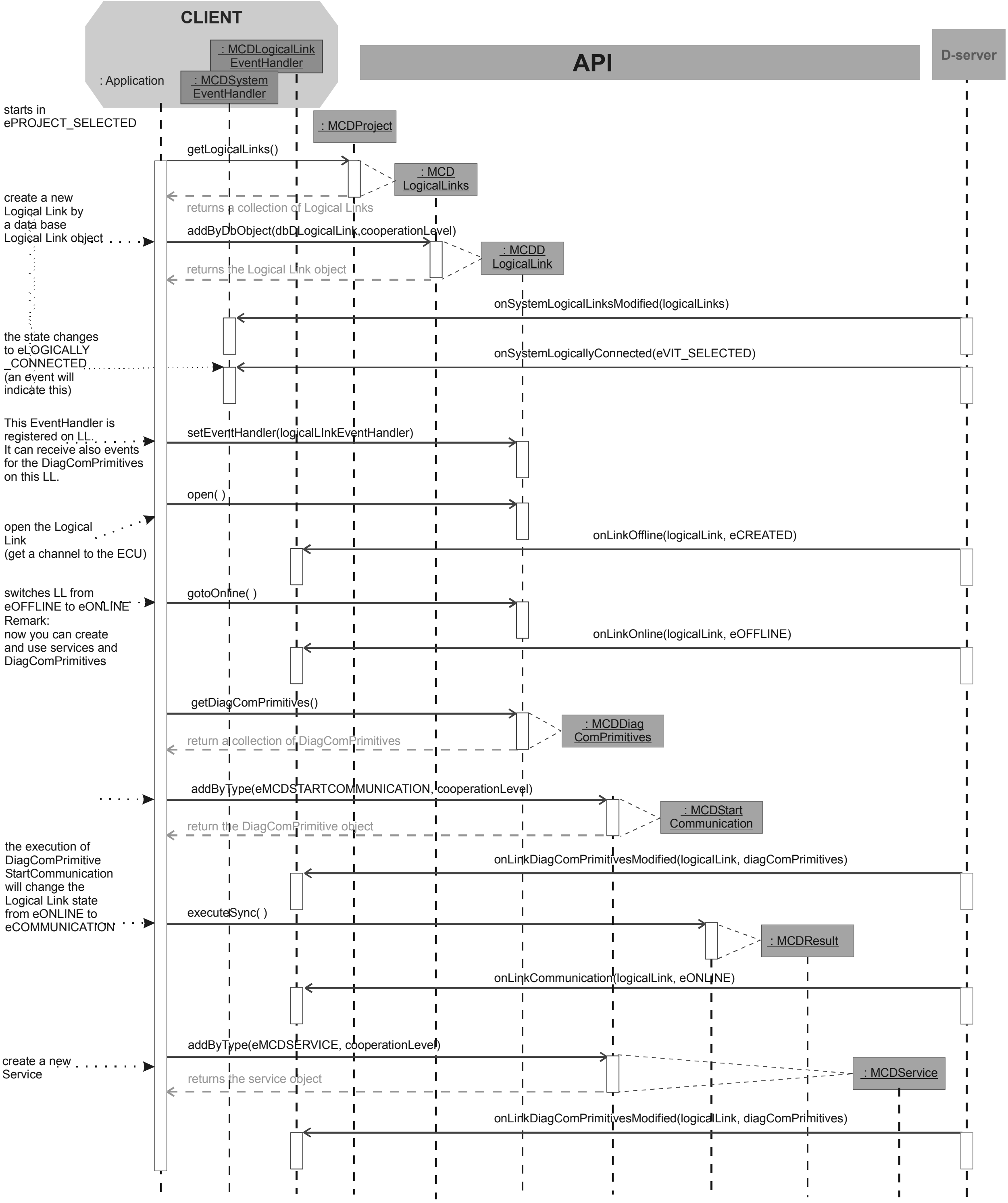

CLIENT
API
D-server
: MCDLogicalLink EventHandler
: Application
: MCDSystem EventHandler
: MCDProject
: MCD LogicalLinks
: MCDD LogicalLink
: MCDDiag ComPrimitives
: MCDStart Communication
: MCDResult
: MCDService
starts in ePROJECT_SELECTED
getLogicalLinks()
returns a collection of Logical Links
create a new Logical Link by a data base Logical Link object
addByDbObject(dbDLogicalLink,cooperationLevel)
returns the Logical Link object
onSystemLogicalLinksModified(logicalLinks)
the state changes to eLOGICALLY _CONNECTED (an event will indicate this)
onSystemLogicallyConnected(eVIT_SELECTED)
This EventHandler is registered on LL. It can receive also events for the DiagComPrimitives on this LL.
setEventHandler(logicalLInkEventHandler)
open( )
open the Logical Link (get a channel to the ECU)
onLinkOffline(logicalLink, eCREATED)
switches LL from eOFFLINE to eONLINE Remark: now you can create and use services and DiagComPrimitives
gotoOnline( )
onLinkOnline(logicalLink, eOFFLINE)
getDiagComPrimitives()
return a collection of DiagComPrimitives
addByType(eMCDSTARTCOMMUNICATION, cooperationLevel)
return the DiagComPrimitive object
the execution of DiagComPrimitive StartCommunication will change the Logical Link state from eONLINE to eCOMMUNICATION
onLinkDiagComPrimitivesModified(logicalLink, diagComPrimitives)
executeSync( )
onLinkCommunication(logicalLink, eONLINE)
create a new Service
addByType(eMCDSERVICE, cooperationLevel)
returns the service object
onLinkDiagComPrimitivesModified(logicalLink, diagComPrimitives)

Figure 105 — Instantiation of Logical Links and DiagComPrimitive (Construction)

At first, the Client creates the runtime Logical Links in accordance with the selected database templates and opens the connection of the Logical Link for the communication. The state of the Logical Link after open is eOFFLINE. As soon as the first Logical Link has been created, the MCDSystem object takes the state eLOGICALLY_CONNECTED.

The first DiagComPrimitive created and executed in this diagram is MCDStartCommunication which performs the state transition to eCOMMUNICATION for the Logical Link. Subsequently, the runtime DiagComPrimitives and Services can be created for the Logical Links in accordance with their database templates.

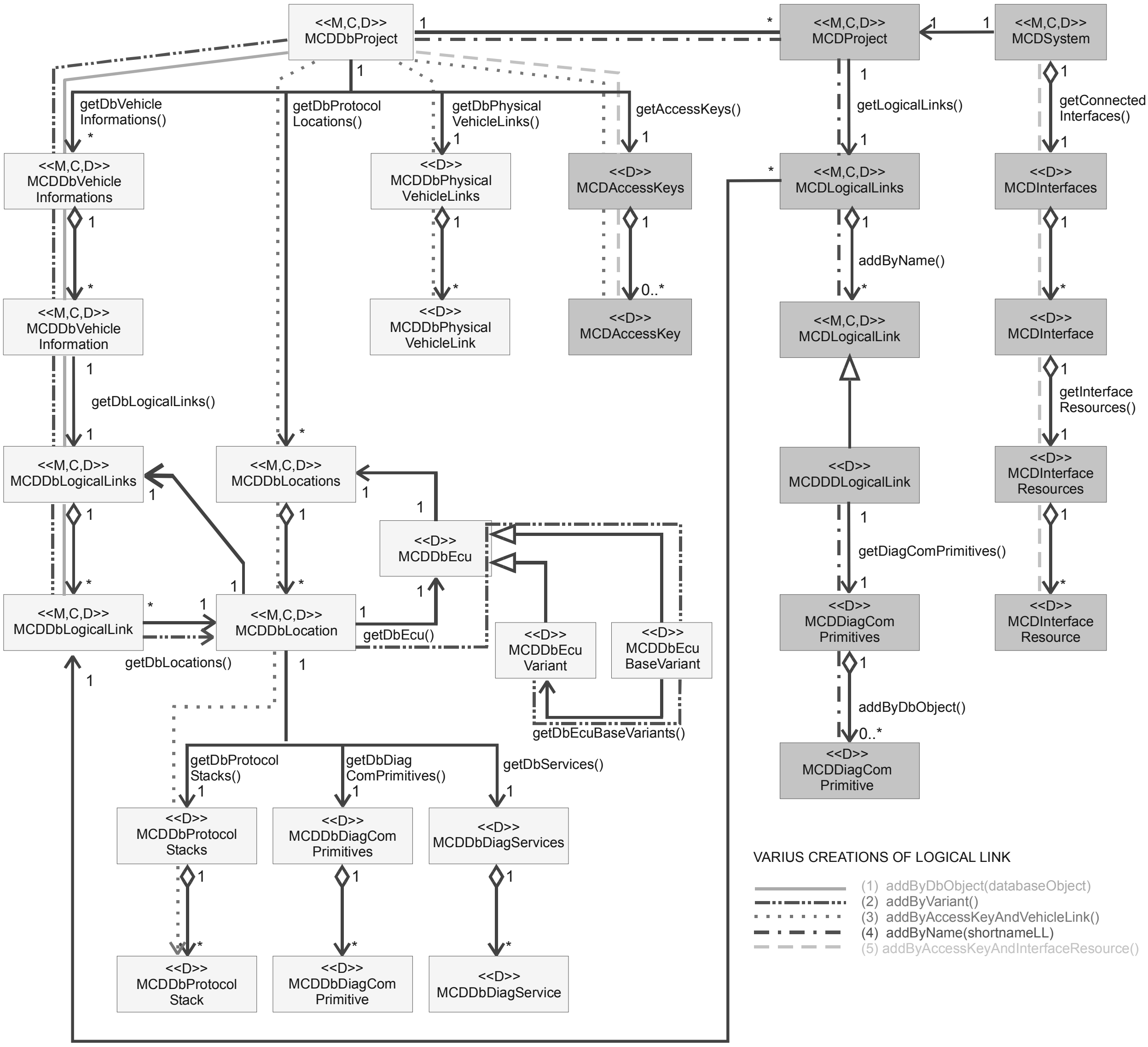

<<M,C,D>> MCDDbProject
<<M,C,D>> MCDProject
<<M,C,D>> MCDSystem
getDbVehicle Informations()
getDbProtocol Locations()
getDbPhysical VehicleLinks()
getAccessKeys()
getLogicalLinks()
getConnected Interfaces()
<<M,C,D>> MCDDbVehicle Informations
<<D>> MCDDbPhysical VehicleLinks
<<D>> MCDAccessKeys
<<M,C,D>> MCDLogicalLinks
<<D>> MCDInterfaces
addByName()
<<M,C,D>> MCDDbVehicle Information
<<D>> MCDDbPhysical VehicleLink
<<D>> MCDAccessKey
<<M,C,D>> MCDLogicalLink
<<D>> MCDInterface
getDbLogicalLinks()
getInterface Resources()
<<M,C,D>> MCDDbLogicalLinks
<<M,C,D>> MCDDbLocations
<<D>> MCDDDLogicalLink
<<D>> MCDInterface Resources
<<D>> MCDDbEcu
getDiagComPrimitives()
<<M,C,D>> MCDDbLogicalLink
<<M,C,D>> MCDDbLocation
getDbEcu()
getDbLocations()
<<D>> MCDDbEcu Variant
<<D>> MCDDbEcu BaseVariant
<<D>> MCDDiagCom Primitives
<<D>> MCDInterface Resource
addByDbObject()
getDbEcuBaseVariants()
getDbProtocol Stacks()
getDbDiag ComPrimitives()
getDbServices()
<<D>> MCDDiagCom Primitive
<<D>> MCDDbProtocol Stacks
<<D>> MCDDbDiagCom Primitives
<<D>> MCDDbDiagServices
<<D>> MCDDbProtocol Stack
<<D>> MCDDbDiagCom Primitive
<<D>> MCDDbDiagService
1
*
0..*
VARIUS CREATIONS OF LOGICAL LINK
(1) addByDbObject(databaseObject)
(2) addByVariant()
(3) addByAccessKeyAndVehicleLink()
(4) addByName(shortnameLL)
(5) addByAccessKeyAndInterfaceResource()

Figure 106 — ERD Possibilities for creation of Logical Links

In this ERD-like diagram_only the possibilities for information acquisition from the database for the purpose of Logical Link creation are shown. The four separate possibilities are each marked with different colours.

For each physical ECU, there might be multiple objects of type MCDDbEcu in the D-server. As a result, objects of type MCDDbEcu (parent class of MCDDbEcuBaseVariant and MCDDbEcuVariant) cannot be used to identify, e.g., all logical links targeting at the same physical ECU.

DbECU is an abstract class. MCDDbLocation::getDbECU returns the reference of a ECU object as MCDDbBEcu interface. This can either be a MCDDbEcuBaseVariant, a MCDDbEcuVariant or a MCDDbFunctionalGroup object.

An MCDDbEcuBaseVariant deliver with getDbEcuBaseVariants all Variants of these Base Variant.

MCDDbECU::getDbLocations delivers for the different Logical Links (physical connections) all relevant Locations.

With the access key always the exact location is specified.

9.11.2 Removal of communication objects

The deleting of all objects used for the communication takes place in reverse order to the setting up. First to change the state from eCOMMUNICATION to eONLINE execute the DiagComPrimitive MCDStopCommunication. Then remove each of the Services and DiagComPrimitives of this Logical Link with the method MCDDLogicalLink::remove (comPrimitive: MCDDiagComprimitive). This action will take place for each Logical Link. Then disconnected the Logical Link on hardware side from the ECU by means of close() and get the event onLinkCreated(...) for the Logical Link.

After this, every Logical Link will be removed from the project by means of MCDLogicalLinks::(link:MCDDLogicalLink). If all Logical Links have been deleted, the state transition of the MCDSystem to ePROJECT_SELECTED is announced by the event onSystemVehicleInformationSelected(...).

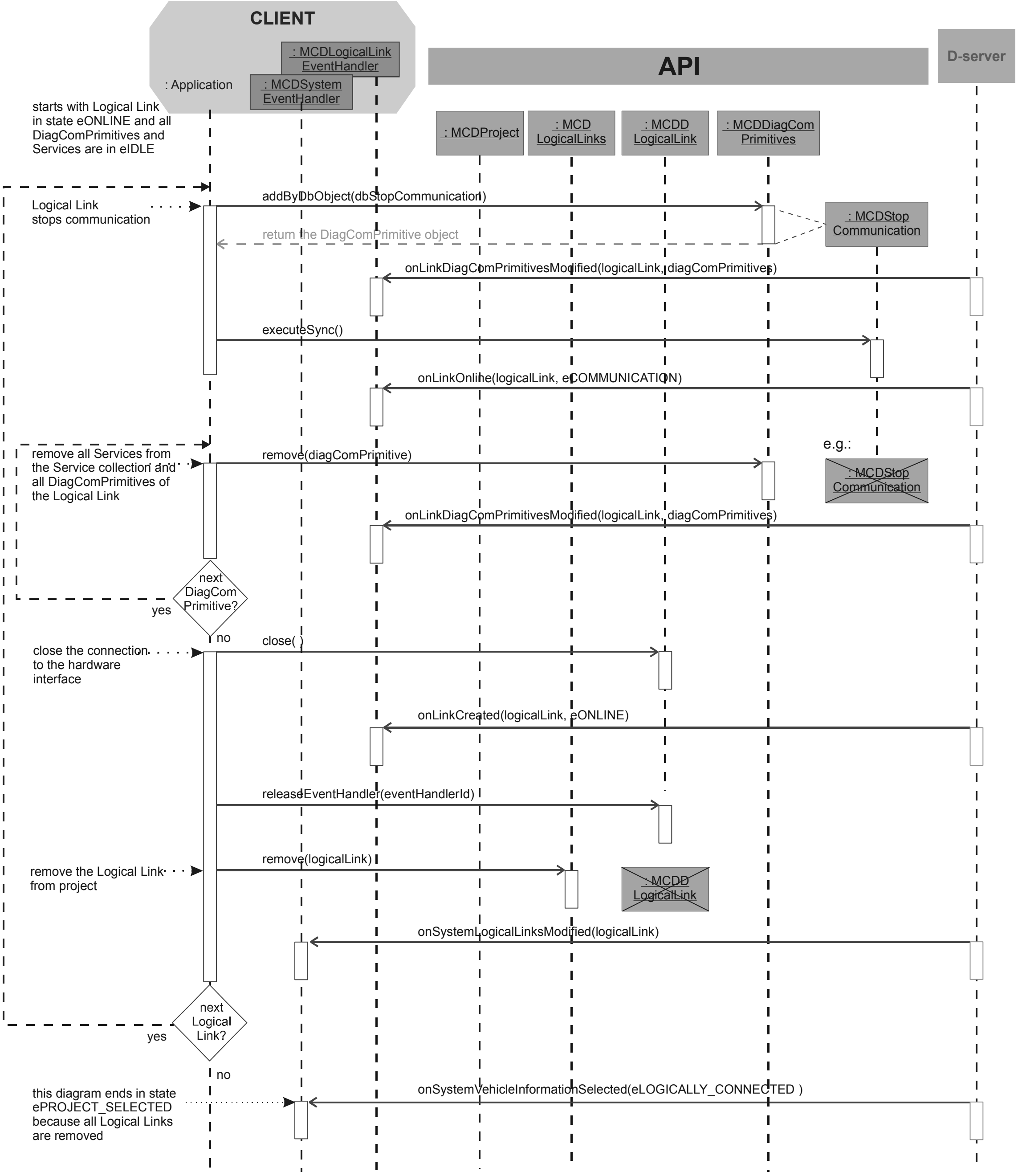

CLIENT
: MCDLogicalLink EventHandler
: Application
: MCDSystem EventHandler
API
D-server
: MCDProject
: MCD LogicalLinks
: MCDD LogicalLink
: MCDDiagCom Primitives
starts with Logical Link in state eONLINE and all DiagComPrimitives and Services are in eIDLE
Logical Link stops communication
addByDbObject(dbStopCommunication)
: MCDStop Communication
return the DiagComPrimitive object
onLinkDiagComPrimitivesModified(logicalLink, diagComPrimitives)
executeSync()
onLinkOnline(logicalLink, eCOMMUNICATION)
e.g.:
remove all Services from the Service collection and all DiagComPrimitives of the Logical Link
remove(diagComPrimitive)
: MCDStop Communication
onLinkDiagComPrimitivesModified(logicalLink, diagComPrimitives)
next DiagCom Primitive?
yes
no
close the connection to the hardware interface
close( )
onLinkCreated(logicalLink, eONLINE)
releaseEventHandler(eventHandlerId)
remove the Logical Link from project
remove(logicalLink)
: MCDD LogicalLink
onSystemLogicalLinksModified(logicalLink)
next Logical Link?
yes
no
this diagram ends in state ePROJECT_SELECTED because all Logical Links are removed
onSystemVehicleInformationSelected(eLOGICALLY_CONNECTED)

Figure 107 — Remove of DiagComPrimitives and Logical Links (Destruction)

9.11.3 Service Handling

9.11.3.1 Non cyclic diag service execution

This Sequence Diagram shows the usual sequence of a non cyclic diag services in asynchronous execution. Firstly, the Request Parameters of the Service are set, in case the default values preset by the database shall not be used. After this, the Service execution starts with the method `executeAsync()`. Following this, the Service will be put into the execution queue of the Logical Link and normally will be executed within the D-server within a finite period of time. While executing the Service, the D-server creates an object for the Result. After finishing the Service, the D-server sends the event `onPrimitiveTerminated(primitive:MCDDiagComPrimitive, link:MCDDLogicalLink, result:MCDResult)` to the EventHandler of the Client. The result Object is asked for the ExecutionState. In case of the correct execution of the Service, the Result-Collection with one Result object is polled by the Service and analysed.

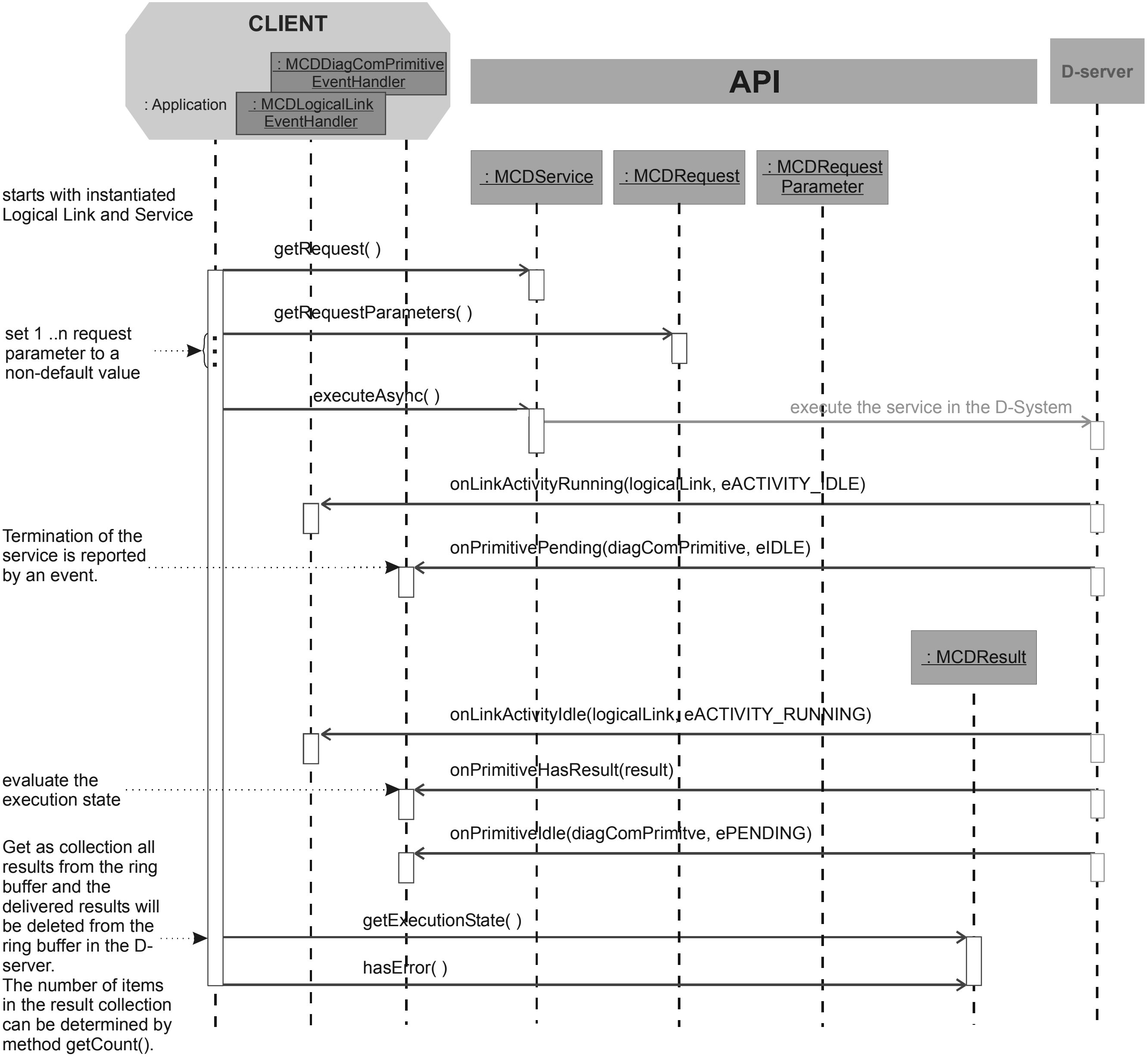

CLIENT
: MCDDiagComPrimitive EventHandler
: Application
: MCDLogicalLink EventHandler
API
D-server
: MCDService
: MCDRequest
: MCDRequest Parameter
starts with instantiated Logical Link and Service
getRequest( )
getRequestParameters( )
set 1 ..n request parameter to a non-default value
executeAsync( )
execute the service in the D-System
onLinkActivityRunning(logicalLink, eACTIVITY_IDLE)
Termination of the service is reported by an event.
onPrimitivePending(diagComPrimitive, eIDLE)
: MCDResult
onLinkActivityIdle(logicalLink, eACTIVITY_RUNNING)
onPrimitiveHasResult(result)
evaluate the execution state
onPrimitiveIdle(diagComPrimitve, ePENDING)
Get as collection all results from the ring buffer and the delivered results will be deleted from the ring buffer in the D-server.
The number of items in the result collection can be determined by method getCount().
getExecutionState( )
hasError( )

Figure 108 — Non cyclic diag service execution (asynchronous)

In case of synchronous execution of a non cyclic diag service the result but no event is delivered as return value.

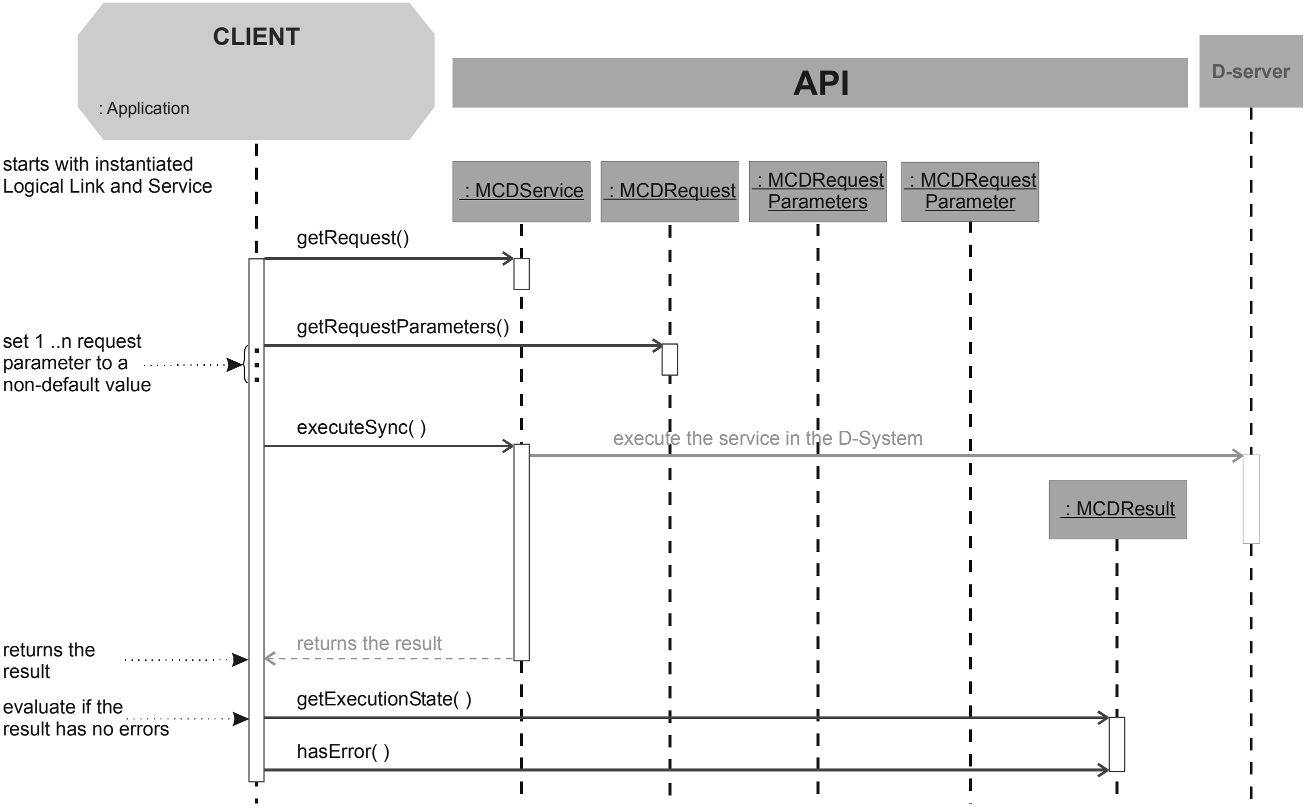


Figure 109 — Non cyclic diag service execution (synchronous)

9.11.3.2 Cyclic diag service execution

This Sequence Diagram shows the usual sequence of a cyclic diag service in asynchronous execution. At first, the Request Parameters of the Service are set with the values necessary for execution, in case the default values preset by the database shall not be used. After this, the service execution is started with the method `executeAsync()`. Following this, the Service will be put into the execution queue of the Logical Link and normally will be executed within the D-server a infinite period of time. It will normally stopped by method `cancel()`. While executing the Service, the D-server creates objects to store the cyclically occurring Results. The generated Results are stored to a ring buffer with defined size. For each Result the D-server sends the event `onPrimitiveHasResult (result:MCDResult)` with the current `MCDExecutionState` to the client. In case of the correct execution of the Service, the already generated ResultObject(s) is(are) polled by the Service and analysed.

The Execution State located within these objects shows how the execution has been running so far. As soon as any error has cropped up at any time the Execution State can not be `eALL_POSITIVE` anymore. To find out if any error has occurred within the current result, this can directly be polled from the each read in `MCDResult`.

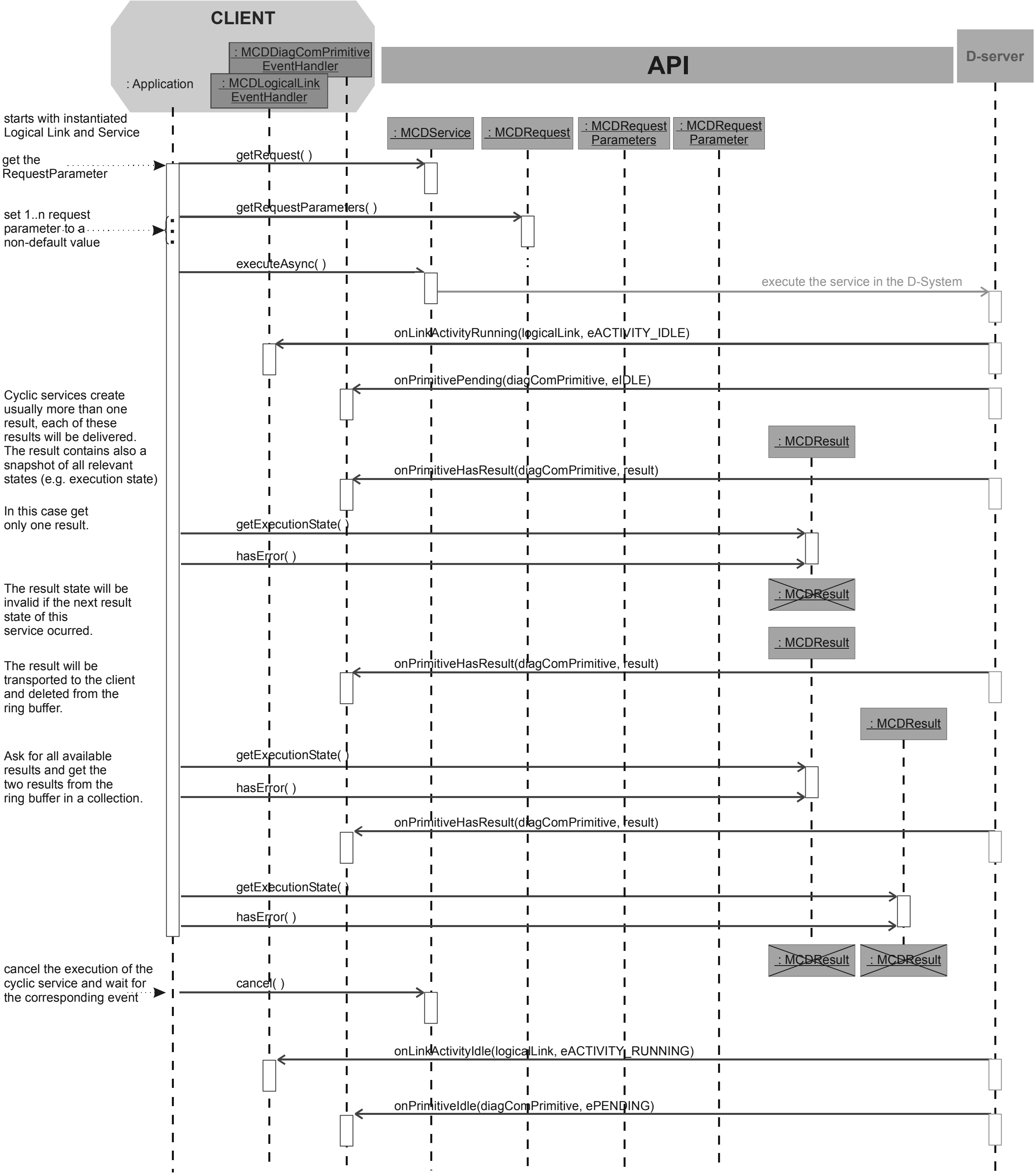

CLIENT
API
D-server
: Application
: MCDDiagComPrimitive EventHandler
: MCDLogicalLink EventHandler
: MCDService
: MCDRequest
: MCDRequest Parameters
: MCDRequest Parameter
starts with instantiated Logical Link and Service
get the RequestParameter
getRequest( )
set 1..n request parameter to a non-default value
getRequestParameters( )
executeAsync( )
execute the service in the D-System
onLinkActivityRunning(logicalLink, eACTIVITY_IDLE)
onPrimitivePending(diagComPrimitive, eIDLE)
Cyclic services create usually more than one result, each of these results will be delivered. The result contains also a snapshot of all relevant states (e.g. execution state)
: MCDResult
onPrimitiveHasResult(diagComPrimitive, result)
In this case get only one result.
getExecutionState( )
hasError( )
The result state will be invalid if the next result state of this service ocurred.
: MCDResult
: MCDResult
The result will be transported to the client and deleted from the ring buffer.
onPrimitiveHasResult(diagComPrimitive, result)
: MCDResult
Ask for all available results and get the two results from the ring buffer in a collection.
getExecutionState( )
hasError( )
onPrimitiveHasResult(diagComPrimitive, result)
getExecutionState( )
hasError( )
: MCDResult
: MCDResult
cancel the execution of the cyclic service and wait for the corresponding event
cancel( )
onLinkActivityIdle(logicalLink, eACTIVITY_RUNNING)
onPrimitiveIdle(diagComPrimitive, ePENDING)

Figure 110 — Cyclic diag service execution

9.11.4 Result access

9.11.4.1 General

After each execution of a Service or ComPrimitive the D-server creates a result. The following Sequence Diagram shows the evaluation of a result of a Service. The structure of the result and the iteration through the result is harmonized with the MC part.

To make clear, wherefrom which information on the execution exists at the end of the Service, the asynchronous start of the Service has been chosen as starting point for the diagram.

As soon as one of the events `onPrimitiveHasResult(MCDResult result)` or `onPrimitiveHasIntermediateResult(MCDResult result)`is reported to the Client, the evaluation can be started. The status of the execution is detected by asking the `MCDExecutionState`. If this is `eALL_POSITIVE` as assumed in the sequence diagram, the result collection is analysed step by step. Within the example exactly one result exists.

Now, the result object may be polled for the created results. Using method `getResponses()`, the Client gets a response collection with one response for each ECU belonging to the Logical Link. Usually this will be exactly one ECU, in the special case Functional Group several ECU's will answer. The method `MCDResponse::getAccessKeyOfLocation()` can only be realized in the MCD-server if the PDU API delivers, e.g. CAN-Ids back to the MCD-server. The name of an ECU is coded in the shortname of a Base Variant. Each single response consists of a collection with at least one response parameter, within which the actual structuring of the result in accordance with the tree concept has been set up. The leaves of the tree contain the actual values, the nods of the tree symbolize the structure elements. To get to the single values of the result, it shall be iterated through the tree and each element is asked for type and name (`getType()` / `getShortName()`). Node elements are polled for sub-elements (`getResponseParameters()`) and leaf elements for the value (`getValue()`). Thus, the Client can analyse result of any structure.

MCDResult objects are singleton and stateless.

9.11.4.2 Result buffer

The results of asynchronous execution for each Service/ DiagComPrimitive are stored to a ring buffer.

The buffer size of a result buffer is DataPrimitive-specific. For DataPrimitives, the buffer size defaults to zero (D-server wide value).

One default value for the size in the whole D-server:

- for every Service different values can be set
- Range: >= 1, max 2^{32} - 1
- read results are deleted from the ring buffer (decrementing number of results).

With respect to DataPrimitives, the server does only keep result objects in a result buffer if and only if at least one client has increased the buffer size for a specific DataPrimitive. Otherwise, the server internal result object can be destroyed after all interested clients (event listener attached) have been provided with the result. In case a buffer size greater zero is defined, the server keeps the result objects in the buffer until

- the clients requesting the queue have called fetchResults() or
- until the queue overflows (cyclic queue, oldest object is overridden and deleted).

The maximum buffer size supported by a MCD-server can be obtained by `MCDSystem::getMaximumBufferSize()::A_UINT32`. The maximum buffer size shall be at least one, that is, a MCD-server needs to support buffers for scripted clients. In case a client wants to set a buffer size greater than `MaxBufferSize`, an exception of type is `ePAR_VALUE_OUT_OF_RANGE` thrown and the buffer size is not modified.
The result ring buffer is internal to the D-server and can contain 0 to N result entries. If the ring buffer is full, any further result will overwrite the oldest result.

The ring buffer will not be emptied if a new execution starts and so all results from former executions which had not been fetched by the Client or had not been overwritten (in case of buffer overflow) are in the ring buffer. The results can be fetched at any time the Client wants to fetch.

The method `fetchResult()` supports with parameter `numReq` different possibilities of result access:

Table 32 — Result access possibilities

Value of numReq	Meaning
- n (n ∈ N, n <> 0)	returns n results. If n > m, where m is the number of available results in the buffer, m results will be delivered. The results in the delivered collection are ordered by their timestamp. The element with the lowest index has got the newest timestamp. All results will be removed from the queue.
0	returns the whole buffer. After this the buffer is empty. The results in the delivered collection are ordered by their timestamp. The element with the lowest index has got the oldest timestamp.
+ n (n ∈ N, n <> 0)	returns n results. If n > m, where m is the number of available results in the buffer, m results will be delivered. The results in the delivered collection are ordered by their timestamp. The element with the lowest index has got the oldest timestamp.

Results delivered to a client in various use cases different

— A client is executing `DiagComPrimitives` (more precisely `DataComPrimitives`) asynchronously. In this case, the result of the execution is delivered by event.

— A scripted client (scripting language without event handling concept) executes services. Currently, the corresponding results can be obtained from the corresponding DataPrimitive by calling its method `MCDDataPrimitive::fetchResults()`.Even more, the scripting client executes a cyclic service which results in multiple responses to the same request.

Result objects are returned in form of an instance of the class `MCDResult`. `MCDResult` objects are static and stateless.

In case `fetchResults()` has been called, the client is provided with a collection of type `MCDResults` containing all the `MCDResult` objects this client has not fetched before.

NOTE 1 These MCDResult objects may not be modified once they have been associated with such a `MCDResults` collection. For example, copies of the `MCDResult` objects could be put into the collections.

The server maintains a single result buffer for every DataPrimitive. For every client, the server needs to maintain pointers (head, tail) to the elements in the buffer that have not yet been fetched. These pointers can be stored in the respective proxy objects. The pointers are only modified if `fetchResults()` is called. Sending results via events does not affect the pointers to the result buffer.

No event is sent if the buffer starts overriding elements as the results have already been delivered to all clients that accept events. Clients that cannot receive events cannot receive exception events either.

Overflows are not considered errors, that is, an overflow does not result in an Error object being attached to any `MCDResult` object or `MCDResults` collection. Here, only the boolean flag obtainable via `MCDResults::hasOverflow()` should be used.

The ownership of a `MCDResult` object delivered to the client via the event is transferred to the client. That is, copies might be delivered. However, the server is no longer responsible for keeping track of these `MCDResult` objects.

Client receives a copy of the DiagComPrimitive after execution which includes the result to be fetched via fetchResult().

Each execution of a DiagComPrimitive which has been fired by its execute()-method shall be handled like on own, self-contained object. That is, multiple execution shall be reflected by multiple DiagComPrimitive objects in the execution queue! Therefore, the event can deliver copies of a self-contained DiagComPrimitive after its execution, which contain the corresponding result structure.

NOTE 2 If the Service is executed synchronously, the result is immediately delivered to the Client as the return value and not stored in the ring buffer. So that are only results from asynchronous execution will be stored in the ring buffer.

9.11.4.3 Error handling in results

Within the object hierarchy contained in an MCDResult, errors may occur at objects of type MCDResult, MCDResponse, and MCDResponseParameter. As there are errors which do not abort the execution of a DiagComPrimitive, there may be errors in the result structure of a DiagComPrimitive which have not been delivered by event of exception before.

To query whether a result structure contains objects having an error attached, the method `MCDResult::hasChildError()` can be used. This method returns true if at least one of the MCDResponse or MCDResponseParameter objects in the result structure has an error. The method `MCDResult::hasChildError()` returns false if the error is only attached to the MCDResult itself while all contained objects do not have an error. As a result, there can be situations where `hasChildError()` returns true while `hasError()` returns false for the same object and vice versa.

NOTE Every execution of a DiagComPrimitive returns a MCDResult object. In case a DiagComPrimitive has terminated with an error, an empty MCDResult shall be returned which only contains an MCDError object. The collection of responses contained in this result is potentially empty, and the execution state is to be determined by the MCD-server in accordance with the definitions for the different execution states. The error information can be transferred by an appropriate MCDError object which is attached to the MCDResult. The type of termination of a DiagComPrimitive is reflected by the MCDExecutionState attached to the MCDResult.

In case an MCDResult, an MCDResponse, or MCDResponseParameter does not have an error the corresponding getError()-method returns an empty MCDError object. The empty MCDError object is defined as follows:

```
<M,C,D>MCDError ::getCode                    =eNO_ERROR
<M,C,D>MCDError ::getCodeDescription         =""
                                             /* empty string, i.e. zero chars */
<M,C,D>MCDError ::getSeverity                =eMESSAGE
<M,C,D>MCDError ::getVendorCode              =0; /* The numerical value 0 */
<M,C,D>MCDError ::getVendorCodeDescription   ="" /* empty string */
```

If an ECU returns a response which cannot be interpreted for a DiagComPrimitive because none of its response patterns matches, the execution state of the corresponding `MCDResult` shall be `eALL_INVALID_RESPONSE` or `eINVALID_RESPONSE` (see rules for execution state). In case of a NRC-CONST parameter, NRCs missing in the value list of this parameter in ODX can result into a timeout as all messages with a different NRC will be ignored. They are not matched by any response pattern.

In case of `eALL_INVALID_RESPONSE`, the result only contains responses with an empty collection of response parameters. Each of the responses is attached an error of type `eRT_INVALID_RESPONSE`. Furthermore, the corresponding response message is contained in each response object.

In case of `eINVALID_RESPONSE`, the result contains the responses of all ECUs. In case of an invalid response (response did not match database templates), the corresponding response object contains an empty collection of response parameters and the corresponding response message. Furthermore, the error `eRT_INVALID_RESPONSE` is returned by the method `MCDResponse::getError()` at such the response object of an invalid response.

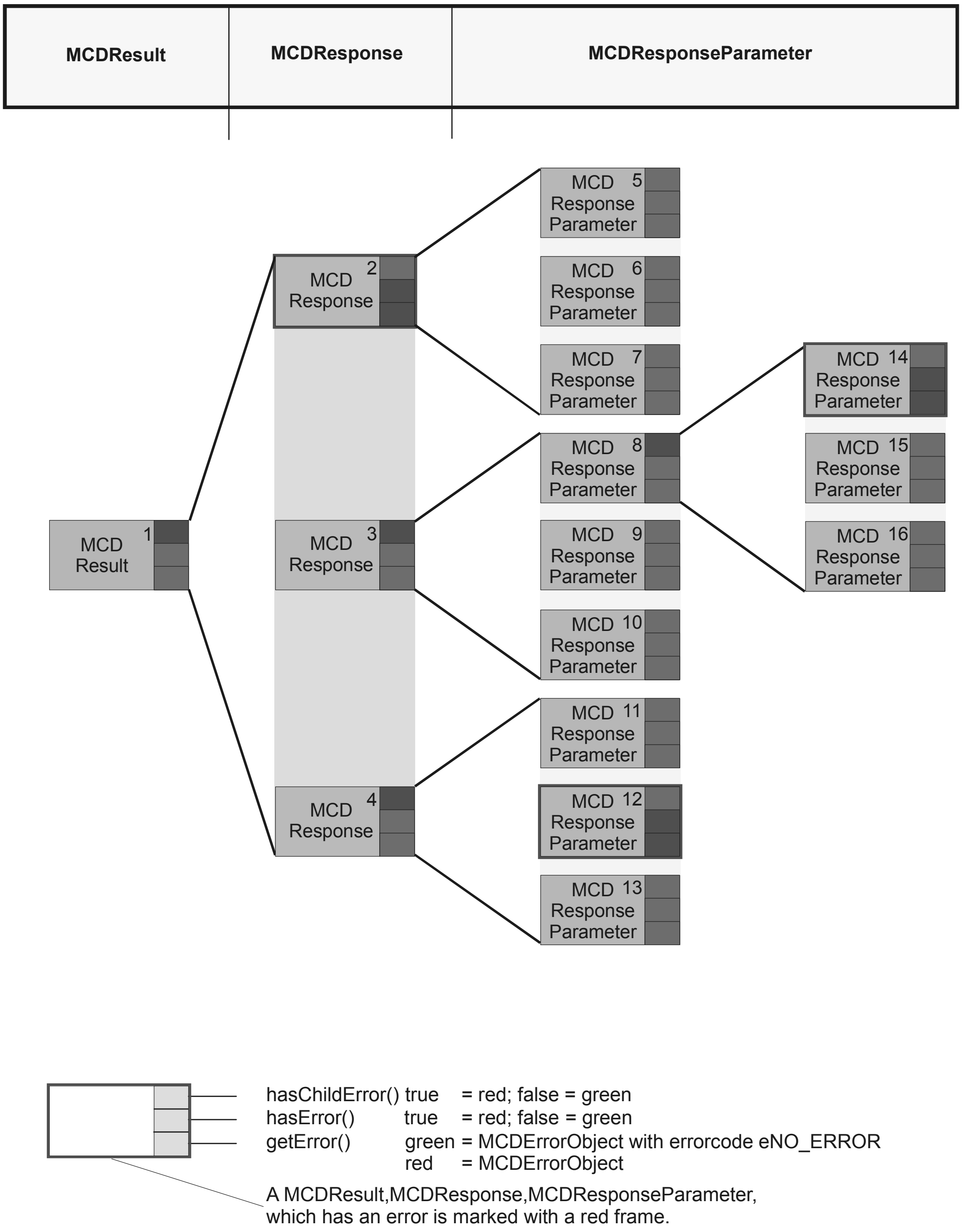

MCDResult
MCDResponse
MCDResponseParameter
MCD Result 1
MCD Response 2
MCD Response 3
MCD Response 4
MCD Response Parameter 5
MCD Response Parameter 6
MCD Response Parameter 7
MCD Response Parameter 8
MCD Response Parameter 9
MCD Response Parameter 10
MCD Response Parameter 11
MCD Response Parameter 12
MCD Response Parameter 13
MCD Response Parameter 14
MCD Response Parameter 15
MCD Response Parameter 16
hasChildError() true = red; false = green
hasError() true = red; false = green
getError() green = MCDErrorObject with errorcode eNO_ERROR
red = MCDErrorObject
A MCDResult,MCDResponse,MCDResponseParameter, which has an error is marked with a red frame.

Figure 111 — Error Handling

9.11.4.4 Result matching for database templates

The result structure of a service or job consists of a `MCDResult` object with one `MCDResponse` object for each ECU. An ECU may answer with a positive or negative response (e.g. a service requires the execution of another service before its execution). In ODX there is a list of positive responses and a list of negative responses at the DIAG-LAYER. The correspondent object of the DIAG-LAYER at MCD-server API is the `MCDDbLocation`.

In ASAM MCD 2 D ODX, the collections of positive, negative, and global negative responses are ordered. In D-server, this information is not available at the API as the ordered collections are currently mapped onto unordered collections.

The method `MCDDbDiagComPrimitive::getDbResponses()` returns a MCDDbResponses collection. This collection contains all responses that are defined in ODX for the corresponding DiagComPrimitive. That is, the collection contains all MCDDbResponse objects representing a positive response, all MCDDbResponse objects representing a local negative response, and all MCDDbResponse objects representing a global negative response to this DiagComPrimitive. Therefore, `MCDDbDiagComPrimitive::getDbResponses()` returns the superset of all MCDDbReponse objects obtainable via `MCDDbDiagComPrimitive::getDbResponsesByType()`.

The collection of MCDDbResponse objects at the class MCDDbDiagComPrimitive

- send-only-service: contains at least the global negative responses defined at the DiagComPrimitive's MCDDbLocation. This set of global negative responses may be empty.

- other services: contains at least one response per ECU - a positive response.

First, the positive responses are matched against the current response in accordance with their ordering. In case none of the positive response templates matches, the specific (local) negative responses are matched against the current response in accordance with their ordering. If no specific negative response matches, the global negative responses are matched against the current response. Again their ordering is considered. In addition, the inheritance hierarchy of global negative responses shall be considered. If inherited global negative responses and locally defined global negative responses are both valid in a specific location, the locally defined global negative responses are evaluated before the inherited ones. In case of multiple inheritance, the global negative responses are evaluated in the alphabetical order of the SHORTNAME of the location (DIAG-LAYER) they have been inherited from.

In the following examples only one of the potentially possible DB response templates is shown. The status before a response PDU has been received is not presented.

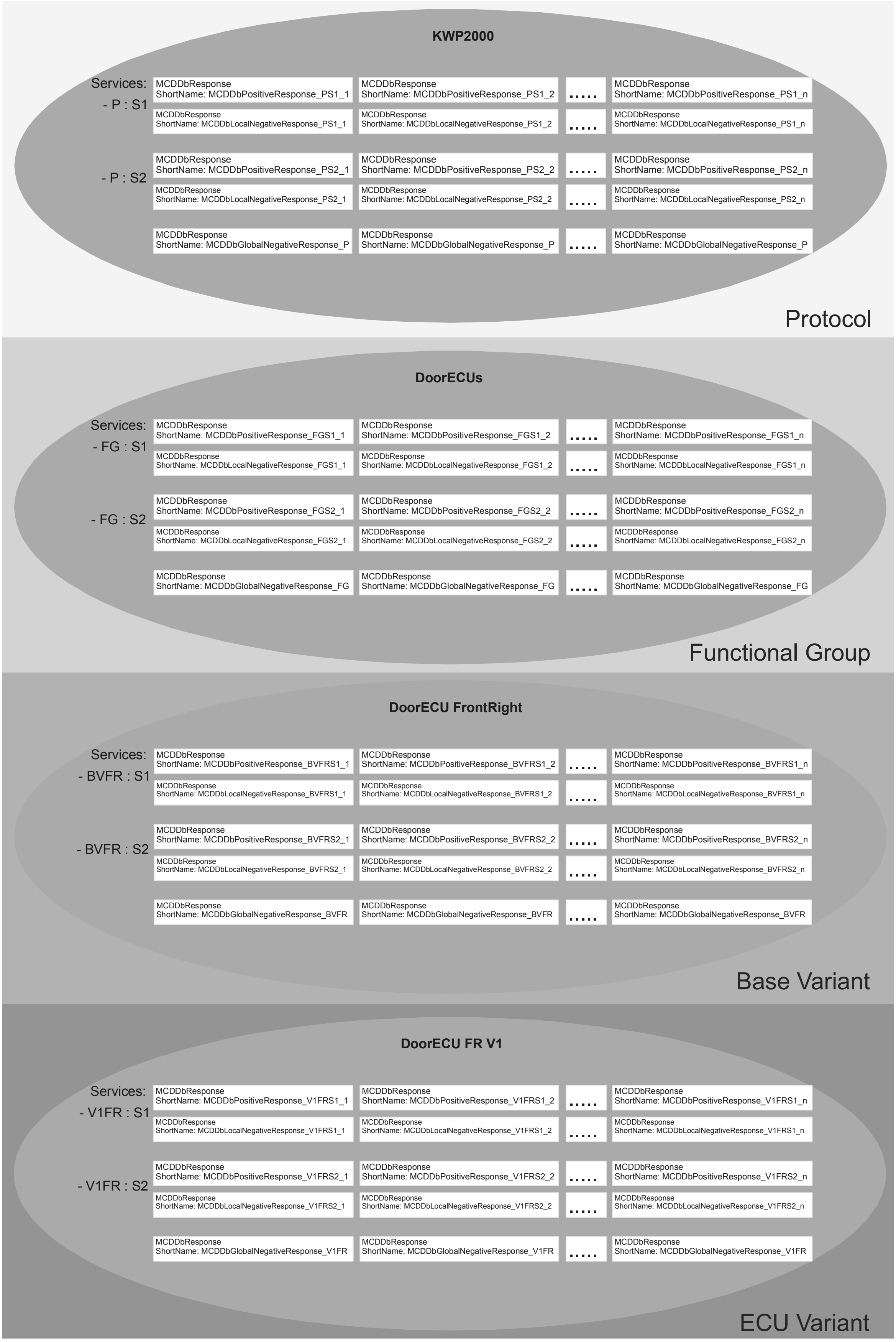

KWP2000
Services:
- P : S1
MCDDbResponse ShortName: MCDDbPositiveResponse_PS1_1
MCDDbResponse ShortName: MCDDbPositiveResponse_PS1_2
.....
MCDDbResponse ShortName: MCDDbPositiveResponse_PS1_n
MCDDbResponse ShortName: MCDDbLocalNegativeResponse_PS1_1
MCDDbResponse ShortName: MCDDbLocalNegativeResponse_PS1_2
MCDDbResponse ShortName: MCDDbLocalNegativeResponse_PS1_n
- P : S2
MCDDbResponse ShortName: MCDDbPositiveResponse_PS2_1
MCDDbResponse ShortName: MCDDbPositiveResponse_PS2_2
MCDDbResponse ShortName: MCDDbPositiveResponse_PS2_n
MCDDbResponse ShortName: MCDDbLocalNegativeResponse_PS2_1
MCDDbResponse ShortName: MCDDbLocalNegativeResponse_PS2_2
MCDDbResponse ShortName: MCDDbLocalNegativeResponse_PS2_n
MCDDbResponse ShortName: MCDDbGlobalNegativeResponse_P
Protocol
DoorECUs
- FG : S1
MCDDbResponse ShortName: MCDDbPositiveResponse_FGS1_1
MCDDbResponse ShortName: MCDDbPositiveResponse_FGS1_2
MCDDbResponse ShortName: MCDDbPositiveResponse_FGS1_n
MCDDbResponse ShortName: MCDDbLocalNegativeResponse_FGS1_1
MCDDbResponse ShortName: MCDDbLocalNegativeResponse_FGS1_2
MCDDbResponse ShortName: MCDDbLocalNegativeResponse_FGS1_n
- FG : S2
MCDDbResponse ShortName: MCDDbPositiveResponse_FGS2_1
MCDDbResponse ShortName: MCDDbPositiveResponse_FGS2_2
MCDDbResponse ShortName: MCDDbPositiveResponse_FGS2_n
MCDDbResponse ShortName: MCDDbLocalNegativeResponse_FGS2_1
MCDDbResponse ShortName: MCDDbLocalNegativeResponse_FGS2_2
MCDDbResponse ShortName: MCDDbLocalNegativeResponse_FGS2_n
MCDDbResponse ShortName: MCDDbGlobalNegativeResponse_FG
Functional Group
DoorECU FrontRight
- BVFR : S1
MCDDbResponse ShortName: MCDDbPositiveResponse_BVFRS1_1
MCDDbResponse ShortName: MCDDbPositiveResponse_BVFRS1_2
MCDDbResponse ShortName: MCDDbPositiveResponse_BVFRS1_n
MCDDbResponse ShortName: MCDDbLocalNegativeResponse_BVFRS1_1
MCDDbResponse ShortName: MCDDbLocalNegativeResponse_BVFRS1_2
MCDDbResponse ShortName: MCDDbLocalNegativeResponse_BVFRS1_n
- BVFR : S2
MCDDbResponse ShortName: MCDDbPositiveResponse_BVFRS2_1
MCDDbResponse ShortName: MCDDbPositiveResponse_BVFRS2_2
MCDDbResponse ShortName: MCDDbPositiveResponse_BVFRS2_n
MCDDbResponse ShortName: MCDDbLocalNegativeResponse_BVFRS2_1
MCDDbResponse ShortName: MCDDbLocalNegativeResponse_BVFRS2_2
MCDDbResponse ShortName: MCDDbLocalNegativeResponse_BVFRS2_n
MCDDbResponse ShortName: MCDDbGlobalNegativeResponse_BVFR
Base Variant
DoorECU FR V1
- V1FR : S1
MCDDbResponse ShortName: MCDDbPositiveResponse_V1FRS1_1
MCDDbResponse ShortName: MCDDbPositiveResponse_V1FRS1_2
MCDDbResponse ShortName: MCDDbPositiveResponse_V1FRS1_n
MCDDbResponse ShortName: MCDDbLocalNegativeResponse_V1FRS1_1
MCDDbResponse ShortName: MCDDbLocalNegativeResponse_V1FRS1_2
MCDDbResponse ShortName: MCDDbLocalNegativeResponse_V1FRS1_n
- V1FR : S2
MCDDbResponse ShortName: MCDDbPositiveResponse_V1FRS2_1
MCDDbResponse ShortName: MCDDbPositiveResponse_V1FRS2_2
MCDDbResponse ShortName: MCDDbPositiveResponse_V1FRS2_n
MCDDbResponse ShortName: MCDDbLocalNegativeResponse_V1FRS2_1
MCDDbResponse ShortName: MCDDbLocalNegativeResponse_V1FRS2_2
MCDDbResponse ShortName: MCDDbLocalNegativeResponse_V1FRS2_n
MCDDbResponse ShortName: MCDDbGlobalNegativeResponse_V1FR
ECU Variant

Figure 112 — Services in Figure 83 — Example location hierarchy

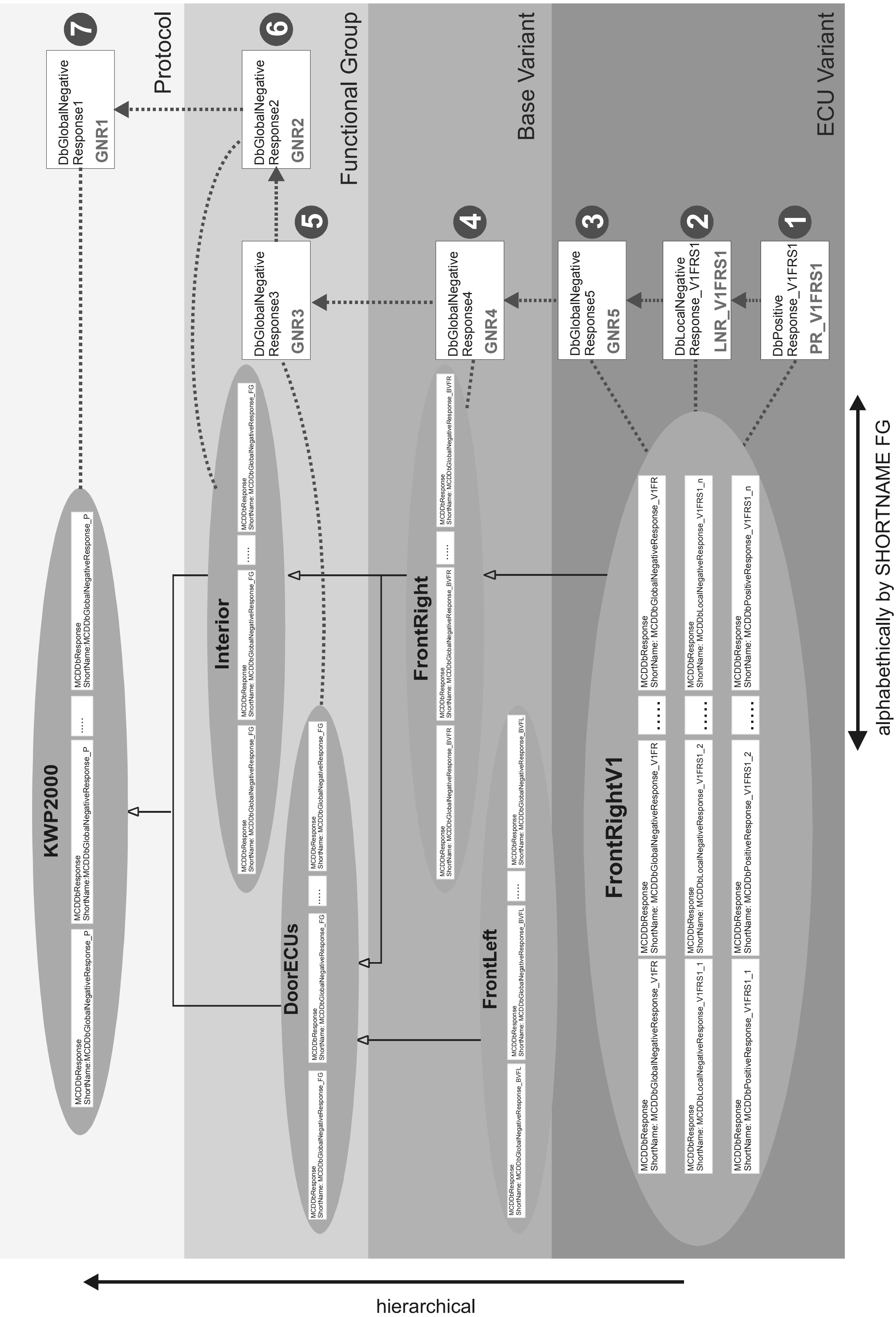

ECU Variant
Base Variant
Functional Group
Protocol
1
DbPositive Response_V1FRS1
PR_V1FRS1
2
DbLocalNegative Response_V1FRS1
LNR_V1FRS1
3
DbGlobalNegative Response5
GNR5
4
DbGlobalNegative Response4
GNR4
5
DbGlobalNegative Response3
GNR3
6
DbGlobalNegative Response2
GNR2
7
DbGlobalNegative Response1
GNR1
FrontRightV1
MCDDbResponse ShortName: MCDDbGlobalNegativeResponse_V1FR
MCDDbResponse ShortName: MCDDbLocalNegativeResponse_V1FRS1_1
MCDDbResponse ShortName: MCDDbLocalNegativeResponse_V1FRS1_2
MCDDbResponse ShortName: MCDDbLocalNegativeResponse_V1FRS1_n
MCDDbResponse ShortName: MCDDbPositiveResponse_V1FRS1_1
MCDDbResponse ShortName: MCDDbPositiveResponse_V1FRS1_2
MCDDbResponse ShortName: MCDDbPositiveResponse_V1FRS1_n
.....
FrontRight
MCDDbResponse ShortName: MCDDbGlobalNegativeResponse_BVFR
FrontLeft
MCDDbResponse ShortName: MCDDbGlobalNegativeResponse_BVFL
Interior
MCDDbResponse ShortName: MCDDbGlobalNegativeResponse_FG
DoorECUs
MCDDbResponse ShortName: MCDDbGlobalNegativeResponse_FG
KWP2000
MCDDbResponse ShortName:MCDDbGlobalNegativeResponse_P
alphabethically by SHORTNAME FG
hierarchical

Figure 113 — Sequence for matching DbTemplate

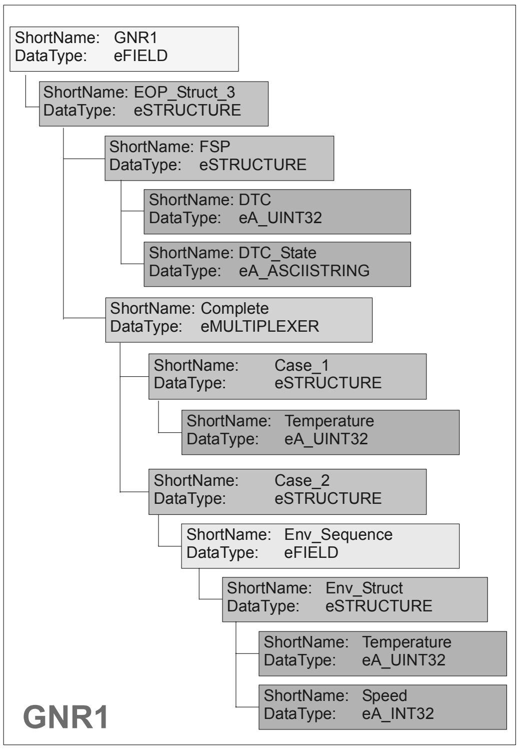


Figure 114 — Result template for Global Negative Response GNR1

ODX data “GNR1” (extract)

```
<NEG-RESPONSE ID=”NegRsp_ID”>
     <SHORT-NAME>NegRsp</SHORT-NAME>
     <PARAM xsi:type="VALUE">
        <SHORT-NAME>GNR1</SHORT-NAME>
        <DOP-REF ID-REF=“EOP_DOP_7_ID“/>
     </PARAM>
</NEG-RESPONSE>
<END-OF-PDU-FIELD ID=“EOP_DOP_7_ID“>
     <SHORT-NAME>EOP_DOP_7</SHORT-NAME>
     <BASIC-STRUCTURE-REF ID-REF=”EOP_STRUCT_3_ID”/>
</END-OF-PDU-FIELD>

<STRUCTURE ID=”EOP_STRUCT_3_ID”>
     <SHORT-NAME>EOP_Struct_3</SHORT-NAME>
     <PARAMS>
        <PARAM xsi:type=”VALUE”>
             <SHORT-NAME>FSP</SHORT-NAME>
             <STRUCTURE-REF ID-REF="FSP_DOP_5_ID"/>
        </PARAM>
     <PARAM xsi:type=”VALUE”>
        <SHORT-NAME>Complete</SHORT-NAME>
        <STRUCTURE-REF ID-REF="MUX_DOP_ID"/>
     </PARAM>
     </PARAMS>
</STRUCTURE>

<STRUCTURE ID=”FSP_DOP_5_ID”>
     <SHORT-NAME>FSP_Struct_5</SHORT-NAME>
             <PARAMS>
                <PARAM xsi:type=”VALUE”>
                    <SHORT-NAME>DTC</SHORT-NAME>
                </PARAM>
                <PARAM xsi:type=”VALUE”>
                    <SHORT-NAME>DTC_State</SHORT-NAME>
                </PARAM>
             </PARAMS>
        </STRUCTURE>

        <MUX ID="MUX_DOP_ID">
             <SHORT-NAME>Mux_DOP</SHORT-NAME>
             <SWITCH-KEY>
                <DATA-OBJECT-PROP-REF ID-REF="SwitchKey_DOP_ID"/>
             </SWITCH-KEY>
             <CASES>
                <CASE>
                    <SHORT-NAME>Case_1</SHORT-NAME>
                    <STRUCTURE-REF ID-REF="Case1_DOP_ID"/>
                </CASE>
                <CASE>
                    <SHORT-NAME>Case_2</SHORT-NAME>
                    <STRUCTURE-REF ID-REF="Case2_DOP_ID"/>
                </CASE>
             </CASES>
        </MUX>

        <STRUCTURE ID="Case1_DOP_ID">
             <SHORT-NAME>StructCase_1</SHORT-NAME>
             <PARAMS>
                <PARAM xsi:type="VALUE">
                    <SHORT-NAME>Temperature</SHORT-NAME>
                </PARAM>
             </PARAMS>
        </STRUCTURE>
        <STRUCTURE ID="Case2_DOP_ID">
             <SHORT-NAME>StructCase_2</SHORT-NAME>
             <PARAMS>
                <PARAM xsi:type="VALUE">
                    <SHORT-NAME>Env_Sequence</SHORT-NAME>
                    <DOP-REF ID-REF="Field_DOP_ID"/>
                </PARAM>
             </PARAMS>
        </STRUCTURE>

        <STRUCTURE ID="Env_DOP_ID">
             <SHORT-NAME>Env_Struct</SHORT-NAME>
             <PARAMS>
                <PARAM xsi:type="VALUE">
                    <SHORT-NAME>Temperature</SHORT-NAME>
                </PARAM>
                <PARAM xsi:type="VALUE">
                    <SHORT-NAME>Speed</SHORT-NAME>
                </PARAM>
             </PARAMS>
        </STRUCTURE>
```

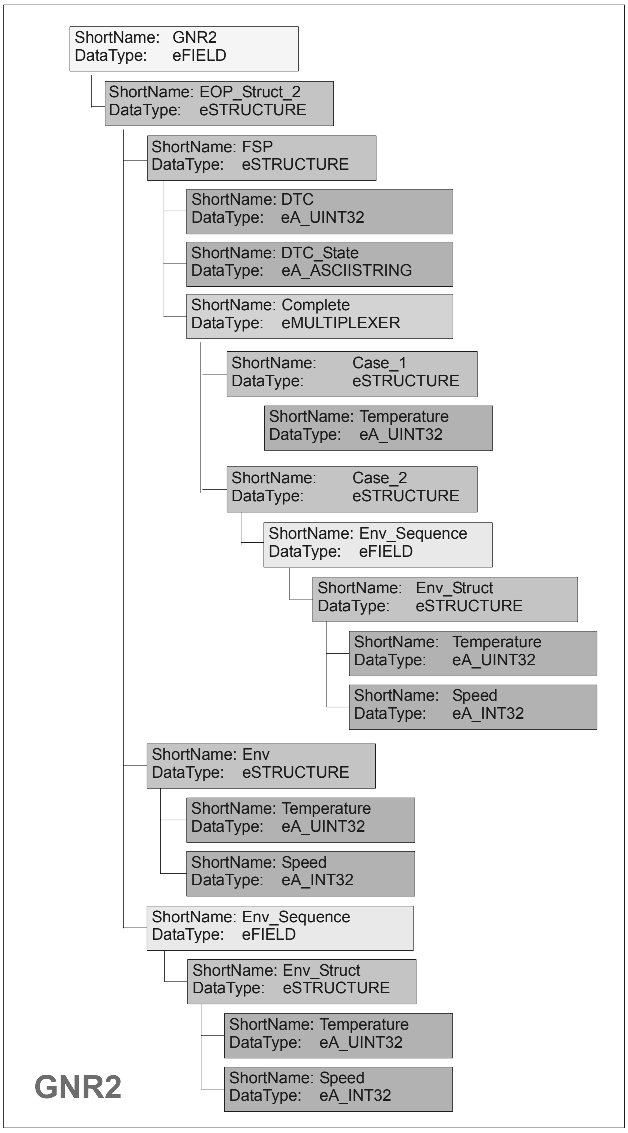


Figure 115 — Result template for Global Negative Response GNR2

ODX data “GNR2” (extract)

```
<NEG-RESPONSE ID=”NegRsp_ID”>
     <SHORT-NAME>NegRsp</SHORT-NAME>
     <PARAM xsi:type="VALUE">
        <SHORT-NAME>GNR2</SHORT-NAME>
        <DOP-REF ID-REF=“EOP_DOP_6_ID“/>
     </PARAM>
</NEG-RESPONSE>
<END-OF-PDU-FIELD ID=“EOP_DOP_6_ID“>
     <SHORT-NAME>EOP_DOP_6</SHORT-NAME>
     <BASIC-STRUCTURE-REF ID-REF=”EOP_STRUCT_2_ID”/>
</END-OF-PDU-FIELD>
<STRUCTURE ID=”EOP_STRUCT_2_ID”>
     <SHORT-NAME>EOP_Struct_2</SHORT-NAME>
     <PARAMS>
        <PARAM xsi:type=”VALUE”>
             <SHORT-NAME>FSP</SHORT-NAME>
             <STRUCTURE-REF ID-REF="FSP_DOP_1_ID"/>
                </PARAM>
                <PARAM xsi:type=”VALUE”>
                    <SHORT-NAME>Env</SHORT-NAME>
                    <STRUCTURE-REF ID-REF="Env_DOP_ID"/>
                </PARAM>
             <PARAM xsi:type=”VALUE”>
                <SHORT-NAME>Env_Sequence</SHORT-NAME>
                <STRUCTURE-REF ID-REF="Field_DOP_ID"/>
             </PARAM>
             </PARAMS>
        </STRUCTURE>
        <STRUCTURE ID=”FSP_DOP_1_ID”>
             <SHORT-NAME>FSP_Struct_1</SHORT-NAME>
             <PARAMS>
                <PARAM xsi:type=”VALUE”>
                    <SHORT-NAME>DTC</SHORT-NAME>
                </PARAM>
                <PARAM xsi:type=”VALUE”>
                    <SHORT-NAME>DTC_State</SHORT-NAME>
                </PARAM>
                <PARAM xsi:type=”VALUE”>
                    <SHORT-NAME>Complete</SHORT-NAME>
                    <DOP-REF ID-REF="MUX_DOP_ID"/>
                </PARAM>
             </PARAMS>
        </STRUCTURE>
        <MUX ID="MUX_DOP_ID">
             <SHORT-NAME>Mux_DOP</SHORT-NAME>
             <SWITCH-KEY>
                <DATA-OBJECT-PROP-REF ID-REF="SwitchKey_DOP_ID"/>
             </SWITCH-KEY>
             <CASES>
                <CASE>
                    <SHORT-NAME>Case_1</SHORT-NAME>
                    <STRUCTURE-REF ID-REF="Case1_DOP_ID"/>
                </CASE>
                <CASE>
                    <SHORT-NAME>Case_2</SHORT-NAME>
                    <STRUCTURE-REF ID-REF="Case2_DOP_ID"/>
                </CASE>
             </CASES>
        </MUX>
        <STRUCTURE ID="Case1_DOP_ID">
             <SHORT-NAME>StructCase_1</SHORT-NAME>
             <PARAMS>
                <PARAM xsi:type="VALUE">
                    <SHORT-NAME>Temperature</SHORT-NAME>
                </PARAM>
             </PARAMS>
        </STRUCTURE>
        <STRUCTURE ID="Case2_DOP_ID">
             <SHORT-NAME>StructCase_2</SHORT-NAME>
             <PARAMS>
                <PARAM xsi:type="VALUE">
                    <SHORT-NAME>Env_Sequence</SHORT-NAME>
                    <DOP-REF ID-REF="Field_DOP_ID"/>
                </PARAM>
     </PARAMS>
</STRUCTURE>
<STRUCTURE ID="Env_DOP_ID">
     <SHORT-NAME>Env_Struct</SHORT-NAME>
     <PARAMS>
        <PARAM xsi:type="VALUE">
             <SHORT-NAME>Temperature</SHORT-NAME>
        </PARAM>
        <PARAM xsi:type="VALUE">
             <SHORT-NAME>Speed</SHORT-NAME>
        </PARAM>
     </PARAMS>
</STRUCTURE>
```

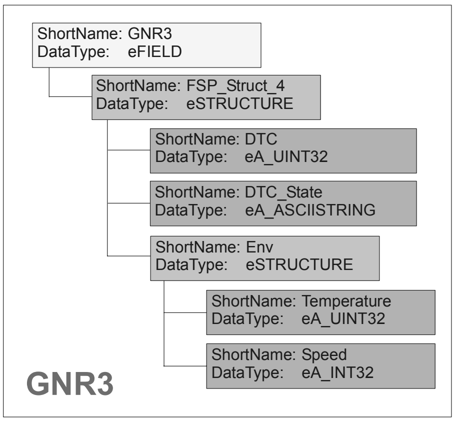


Figure 116 — Result template for Global Negative Response GNR3

ODX data “GNR3” (extract)

```
<GLOBAL-NEG-RESPONSE ID=”GNRsp_ID_3”>
     <SHORT-NAME>GNRsp_3</SHORT-NAME>
     <PARAM xsi:type="VALUE">
        <SHORT-NAME>GNR3</SHORT-NAME>
        <DOP-REF ID-REF=“EOP_DOP_5_ID“/>
     </PARAM>
</GLOBAL-NEG-RESPONSE>

<END-OF-PDU-FIELD ID=“EOP_DOP_5_ID“>
     <SHORT-NAME>EOP_DOP_5</SHORT-NAME>
     <BASIC-STRUCTURE-REF ID-REF=”FSP_DOP_4_ID”/>
</END-OF-PDU-FIELD>

<STRUCTURE ID=”FSP_DOP_4_ID”>
     <SHORT-NAME>FSP_Struct_4</SHORT-NAME>
     <PARAMS>
        <PARAM xsi:type=”VALUE”>
             <SHORT-NAME>DTC</SHORT-NAME>
        </PARAM>
        <PARAM xsi:type=”VALUE”>
             <SHORT-NAME>DTC_State</SHORT-NAME>
        </PARAM>
        <PARAM xsi:type=”VALUE”>
```

```
                <SHORT-NAME>Env</SHORT-NAME>
                <DOP-REF ID-REF="Env_DOP_ID"/>
            </PARAM>
        </PARAMS>
</STRUCTURE>

<STRUCTURE ID="Env_DOP_ID">
        <SHORT-NAME>Env_Struct</SHORT-NAME>
        <PARAMS>
            <PARAM xsi:type="VALUE">
                <SHORT-NAME>Temperature</SHORT-NAME>
            </PARAM>
            <PARAM xsi:type="VALUE">
                <SHORT-NAME>Speed</SHORT-NAME>
            </PARAM>
        </PARAMS>
</STRUCTURE>
```

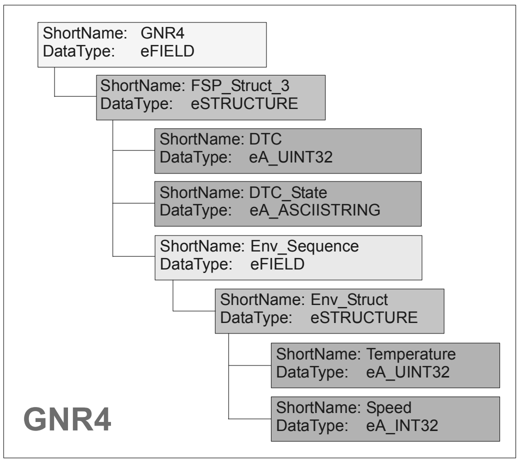


Figure 117 — Result template for Global Negative Response GNR4

ODX data “GNR4” (extract)

```
<GLOBAL-NEG-RESPONSE ID=”GNRsp_ID_4”>
        <SHORT-NAME>GNRsp_4</SHORT-NAME>
        <PARAM xsi:type="VALUE">
            <SHORT-NAME>GNR4</SHORT-NAME>
            <DOP-REF ID-REF=“EOP_DOP_4_ID“/>
        </PARAM>
</GLOBAL-NEG-RESPONSE>

<END-OF-PDU-FIELD ID=“EOP_DOP_4_ID“>
        <SHORT-NAME>EOP_DOP_4</SHORT-NAME>
        <BASIC-STRUCTURE-REF ID-REF=”FSP_DOP_3_ID”/>
</END-OF-PDU-FIELD>

<STRUCTURE ID=”FSP_DOP_3_ID”>
        <SHORT-NAME>FSP_Struct_3</SHORT-NAME>
        <PARAMS>
            <PARAM xsi:type=”VALUE”>
                <SHORT-NAME>DTC</SHORT-NAME>
        </PARAM>
        <PARAM xsi:type=”VALUE”>
            <SHORT-NAME>DTC_State</SHORT-NAME>
        </PARAM>
        <PARAM xsi:type=”VALUE”>
            <SHORT-NAME>Env_Sequence</SHORT-NAME>
            <DOP-REF ID-REF="Field_DOP_ID"/>
        </PARAM>
     </PARAMS>
</STRUCTURE>

<DYNAMIC-ENDMARKER-FIELD ID="Field_DOP_ID">
    <SHORT-NAME>Field_DOP</SHORT-NAME>
    <BASIC-STRUCTURE-REF ID-REF="Env_DOP_ID"/>
</DYNAMIC-ENDMARKER_FIELD>

<STRUCTURE ID="Env_DOP_ID">
     <SHORT-NAME>Env_Struct</SHORT-NAME>
     <PARAMS>
        <PARAM xsi:type="VALUE">
            <SHORT-NAME>Temperature</SHORT-NAME>
        </PARAM>
        <PARAM xsi:type="VALUE">
            <SHORT-NAME>Speed</SHORT-NAME>
        </PARAM>
     </PARAMS>
</STRUCTURE>
```

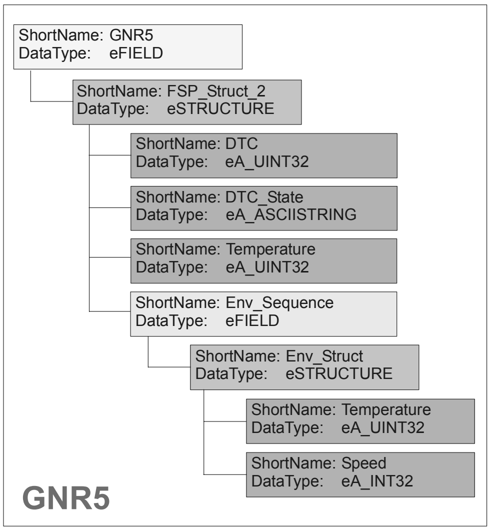


Figure 118 — Result template for Global Negative Response GNR5

ODX data “GNR5” (extract)

```
<GLOBAL-NEG-RESPONSE ID=”GNRsp_ID_5”>
     <SHORT-NAME>GNRsp_5</SHORT-NAME>
     <PARAM xsi:type="VALUE">
        <SHORT-NAME>GNR5</SHORT-NAME>
        <DOP-REF ID-REF=“EOP_DOP_3_ID“/>
     </PARAM>
```

```
</GLOBAL-NEG-RESPONSE>

<END-OF-PDU-FIELD ID=“EOP_DOP_3_ID“>
     <SHORT-NAME>EOP_DOP_3</SHORT-NAME>
     <BASIC-STRUCTURE-REF ID-REF=”FSP_DOP_2_ID”/>
</END-OF-PDU-FIELD>

<STRUCTURE ID=”FSP_DOP_2_ID”>
     <SHORT-NAME>FSP_Struct_2</SHORT-NAME>
     <PARAMS>
        <PARAM xsi:type=”VALUE”>
             <SHORT-NAME>DTC</SHORT-NAME>
        </PARAM>
        <PARAM xsi:type=”VALUE”>
             <SHORT-NAME>DTC_State</SHORT-NAME>
        </PARAM>
        <PARAM xsi:type=”VALUE”>
             <SHORT-NAME>Temperature</SHORT-NAME>
        </PARAM>
        <PARAM xsi:type=”VALUE”>
             <SHORT-NAME>Env_Sequence</SHORT-NAME>
             <DOP-REF ID-REF="Field_DOP_ID"/>
        </PARAM>
     </PARAMS>
</STRUCTURE>

<DYNAMIC-ENDMARKER-FIELD ID="Field_DOP_ID">
    <SHORT-NAME>Field_DOP</SHORT-NAME>
    <BASIC-STRUCTURE-REF ID-REF="Env_DOP_ID"/>
</DYNAMIC-ENDMARKER_FIELD>

<STRUCTURE ID="Env_DOP_ID">
     <SHORT-NAME>Env_Struct</SHORT-NAME>
     <PARAMS>
        <PARAM xsi:type="VALUE">
             <SHORT-NAME>Temperature</SHORT-NAME>
        </PARAM>
        <PARAM xsi:type="VALUE">
             <SHORT-NAME>Speed</SHORT-NAME>
        </PARAM>
     </PARAMS>
</STRUCTURE>
```

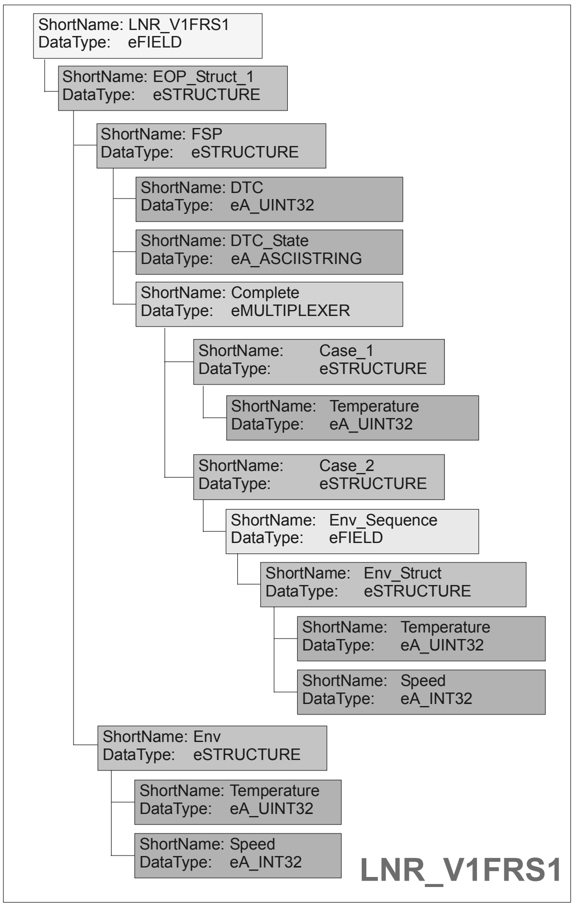


Figure 119 — Result template for Local Negative Response LNR_V1FRS1

ODX data “LNR_V1FRS1” (extract)

```
<NEG-RESPONSE ID=”NegRsp_ID”>
     <SHORT-NAME>NegRsp</SHORT-NAME>
     <PARAM xsi:type="VALUE">
        <SHORT-NAME>LNR_FRV1</SHORT-NAME>
        <DOP-REF ID-REF=“EOP_DOP_2_ID“/>
     </PARAM>
</NEG-RESPONSE>

<END-OF-PDU-FIELD ID=“EOP_DOP_2_ID“>
     <SHORT-NAME>EOP_DOP_2</SHORT-NAME>
     <BASIC-STRUCTURE-REF ID-REF=”EOP_STRUCT_1_DOP_ID”/>
</END-OF-PDU-FIELD>

<STRUCTURE ID=”EOP_STRUCT_1_DOP_ID”>
     <SHORT-NAME>EOP_Struct_1</SHORT-NAME>
     <PARAMS>
        <PARAM xsi:type=”VALUE”>
             <SHORT-NAME>FSP</SHORT-NAME>
             <STRUCTURE-REF ID-REF="FSP_DOP_1_ID"/>
        </PARAM>
        <PARAM xsi:type=”VALUE”>
             <SHORT-NAME>Env</SHORT-NAME>
             <STRUCTURE-REF ID-REF="Env_DOP_ID"/>
                </PARAM>
             </PARAMS>
        </STRUCTURE>
        <STRUCTURE ID=”FSP_DOP_1_ID”>
             <SHORT-NAME>FSP_Struct_1</SHORT-NAME>
             <PARAMS>
                <PARAM xsi:type=”VALUE”>
                    <SHORT-NAME>DTC</SHORT-NAME>
                </PARAM>
                <PARAM xsi:type=”VALUE”>
                    <SHORT-NAME>DTC_State</SHORT-NAME>
                </PARAM>
                <PARAM xsi:type=”VALUE”>
                    <SHORT-NAME>Complete</SHORT-NAME>
                    <DOP-REF ID-REF="MUX_DOP_ID"/>
                </PARAM>
             </PARAMS>
        </STRUCTURE>

        <MUX ID="MUX_DOP_ID">
             <SHORT-NAME>Mux_DOP</SHORT-NAME>
             <SWITCH-KEY>
                <DATA-OBJECT-PROP-REF ID-REF="SwitchKey_DOP_ID"/>
             </SWITCH-KEY>
             <CASES>
                <CASE>
                    <SHORT-NAME>Case_1</SHORT-NAME>
                    <STRUCTURE-REF ID-REF="Case1_DOP_ID"/>
                </CASE>
                <CASE>
                    <SHORT-NAME>Case_2</SHORT-NAME>
                    <STRUCTURE-REF ID-REF="Case2_DOP_ID"/>
                </CASE>
             </CASES>
        </MUX>

        <STRUCTURE ID="Case1_DOP_ID">
             <SHORT-NAME>StructCase_1</SHORT-NAME>
             <PARAMS>
                <PARAM xsi:type="VALUE">
                    <SHORT-NAME>Temperature</SHORT-NAME>
                </PARAM>
             </PARAMS>
        </STRUCTURE>
        <STRUCTURE ID="Case2_DOP_ID">
             <SHORT-NAME>StructCase_2</SHORT-NAME>
             <PARAMS>
                <PARAM xsi:type="VALUE">
                    <SHORT-NAME>Env_Sequence</SHORT-NAME>
                    <DOP-REF ID-REF="Field_DOP_ID"/>
                </PARAM>
             </PARAMS>
        </STRUCTURE>
        <STRUCTURE ID="Env_DOP_ID">
             <SHORT-NAME>Env_Struct</SHORT-NAME>
             <PARAMS>
                <PARAM xsi:type="VALUE">
```

```
                <SHORT-NAME>Temperature</SHORT-NAME>
            </PARAM>
            <PARAM xsi:type="VALUE">
                <SHORT-NAME>Speed</SHORT-NAME>
            </PARAM>
        </PARAMS>
</STRUCTURE>
```

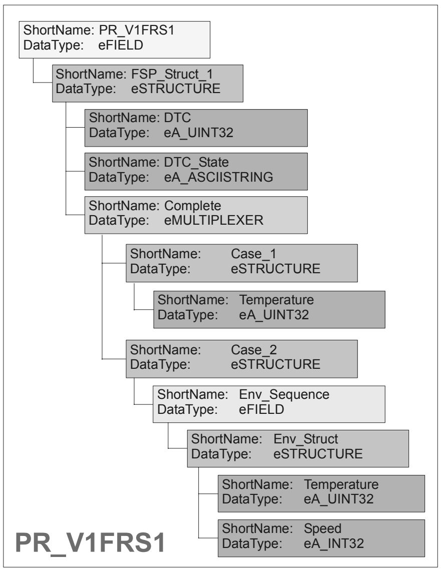


Figure 120 — Result template for Positive Response PR_V1FRS1

ODX data “PR_V1FRS1” (extract)

```
<POS-RESPONSE ID=”PosRsp_ID”>
        <SHORT-NAME>PosRsp</SHORT-NAME>
        <PARAM xsi:type="VALUE">
            <SHORT-NAME>PR_FRV1</SHORT-NAME>
            <DOP-REF ID-REF=“EOP_DOP_1_ID“/>
        </PARAM>
</POS-RESPONSE>

<END-OF-PDU-FIELD ID=“EOP_DOP_1_ID“>
        <SHORT-NAME>EOP_DOP_1</SHORT-NAME>
        <BASIC-STRUCTURE-REF ID-REF=”FSP_DOP_1_ID”/>
</END-OF-PDU-FIELD>

<STRUCTURE ID=”FSP_DOP_1_ID”>
        <SHORT-NAME>FSP_Struct_1</SHORT-NAME>
        <PARAMS>
            <PARAM xsi:type=”VALUE”>
                <SHORT-NAME>DTC</SHORT-NAME>
            </PARAM>
            <PARAM xsi:type=”VALUE”>
                <SHORT-NAME>DTC_State</SHORT-NAME>
            </PARAM>
            <PARAM xsi:type=”VALUE”>
                <SHORT-NAME>Complete</SHORT-NAME>
                <DOP-REF ID-REF="MUX_DOP_ID"/>
                   </PARAM>
                </PARAMS>
            </STRUCTURE>

            <MUX ID="MUX_DOP_ID">
                <SHORT-NAME>Mux_DOP</SHORT-NAME>
                <SWITCH-KEY>
                   <DATA-OBJECT-PROP-REF ID-REF="SwitchKey_DOP_ID"/>
                </SWITCH-KEY>
                <CASES>
                   <CASE>
                        <SHORT-NAME>Case_1</SHORT-NAME>
                        <STRUCTURE-REF ID-REF="Case1_DOP_ID"/>
                   </CASE>
                   <CASE>
                        <SHORT-NAME>Case_2</SHORT-NAME>
                        <STRUCTURE-REF ID-REF="Case2_DOP_ID"/>
                   </CASE>
                </CASES>
            </MUX>

            <STRUCTURE ID="Case1_DOP_ID">
                <SHORT-NAME>StructCase_1</SHORT-NAME>
                <PARAMS>
                   <PARAM xsi:type="VALUE">
                        <SHORT-NAME>Temperature</SHORT-NAME>
                   </PARAM>
                </PARAMS>
            </STRUCTURE>

            <STRUCTURE ID="Case2_DOP_ID">
                <SHORT-NAME>StructCase_2</SHORT-NAME>
                <PARAMS>
                   <PARAM xsi:type="VALUE">
                        <SHORT-NAME>Env_Sequence</SHORT-NAME>
                        <DOP-REF ID-REF="Field_DOP_ID"/>
                   </PARAM>
                </PARAMS>
            </STRUCTURE>

            <DYNAMIC-ENDMARKER-FIELD ID="Field_DOP_ID">
                <SHORT-NAME>Field_DOP</SHORT-NAME>
                <BASIC-STRUCTURE-REF ID-REF="Env_DOP_ID"/>
            </DYNAMIC-ENDMARKER_FIELD>

            <STRUCTURE ID="Env_DOP_ID">
                <SHORT-NAME>Env_Struct</SHORT-NAME>
                <PARAMS>
                   <PARAM xsi:type="VALUE">
                        <SHORT-NAME>Temperature</SHORT-NAME>
                   </PARAM>
                   <PARAM xsi:type="VALUE">
                        <SHORT-NAME>Speed</SHORT-NAME>
                   </PARAM>
                </PARAMS>
            </STRUCTURE>
```

In case an ECU responds with a PDU which cannot be mapped onto one of the Response Templates defined for this ECU – negative and positive –, the execution state of this service with respect to this ECU is `eINVALID_RESPONSE` (functional addressing) or `eALL_INVALID_RESPONSE` (functional / physical addressing). If a more severe error occurs, the result might have an execution state of `eFAILED`. The result contains an empty response object (MCDResponse) with a shortname #RtGen_Response. For this MCDResponse object, the method `getResponseMessage` returns the non-matching PDU (MCDValue of type bytefield). The error `eRT_INVALID_RESPONSE` is returned by the method `MCDResponse::getError()`.

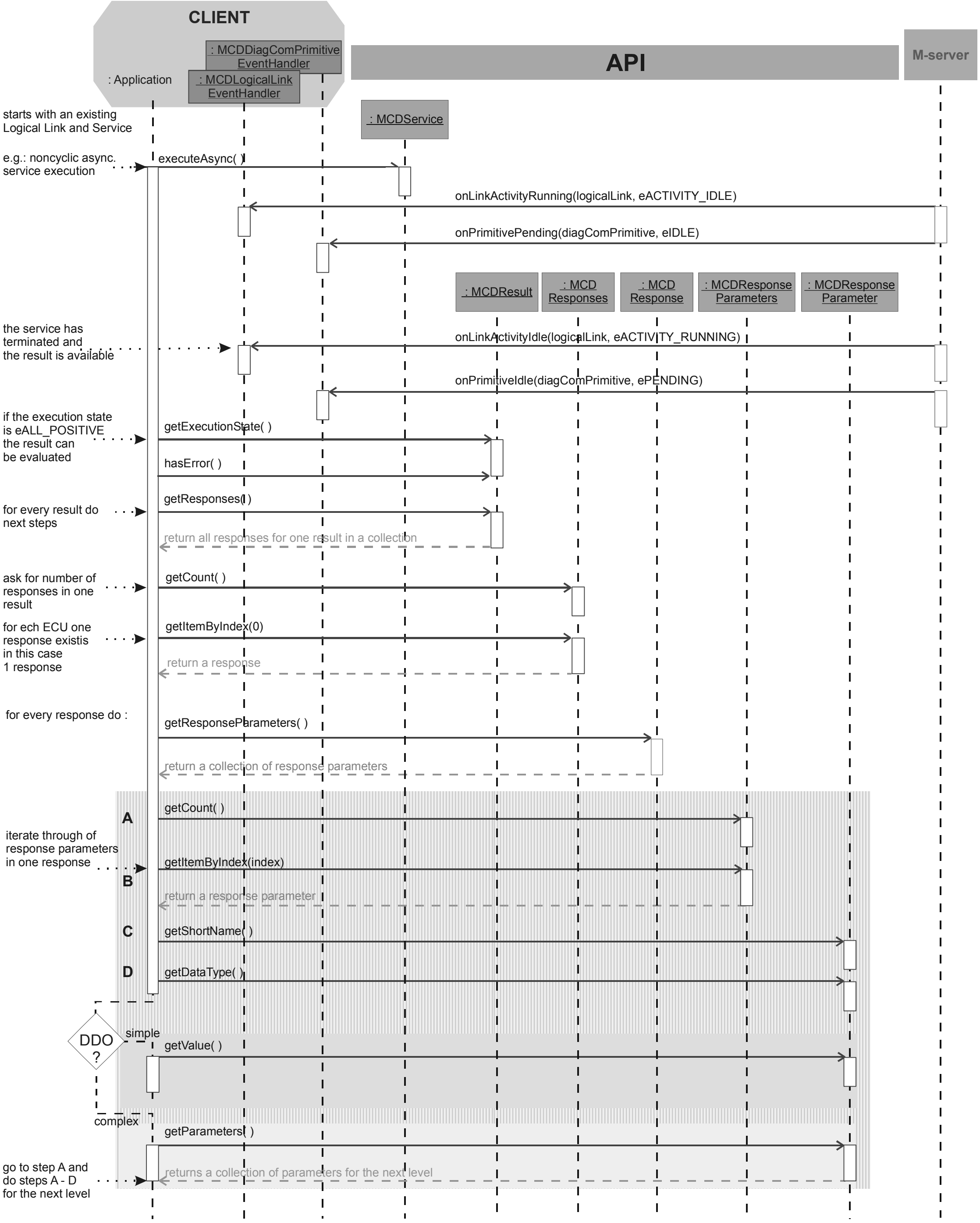

CLIENT
: MCDDiagComPrimitive EventHandler
: Application
: MCDLogicalLink EventHandler
API
M-server
: MCDService
starts with an existing Logical Link and Service
e.g.: noncyclic async. service execution
executeAsync( )
onLinkActivityRunning(logicalLink, eACTIVITY_IDLE)
onPrimitivePending(diagComPrimitive, eIDLE)
: MCDResult
: MCD Responses
: MCD Response
: MCDResponse Parameters
: MCDResponse Parameter
the service has terminated and the result is available
onLinkActivityIdle(logicalLink, eACTIVITY_RUNNING)
onPrimitiveIdle(diagComPrimitive, ePENDING)
if the execution state is eALL_POSITIVE the result can be evaluated
getExecutionState( )
hasError( )
getResponses( )
for every result do next steps
return all responses for one result in a collection
ask for number of responses in one result
getCount( )
for ech ECU one response existis in this case 1 response
getItemByIndex(0)
return a response
for every response do :
getResponseParameters( )
return a collection of response parameters
A
getCount( )
iterate through of response parameters in one response
getItemByIndex(index)
B
return a response parameter
C
getShortName( )
D
getDataType( )
DDO ?
simple
getValue( )
complex
getParameters( )
go to step A and do steps A - D for the next level
returns a collection of parameters for the next level

Figure 121 — Result access

Generally is valid:

The availability of results of DiagServices is reported via Events. Exactly one Event is reported per complete result record. The following Events are used:

- `onPrimitiveHasResult` (for complete result data records of cyclic Diagnostic services or for complete result data records of non cyclic Diagnostic services in execution mode Repetition or Single Shot)

- `onPrimitiveHasIntermediateResult` (for single responses in case of functional services (single, cyclic and repeated) with intermediate results feature switched on, for intermediate results in case of Java Jobs)

After one of these events has occurred it is possible to poll the number of complete result data records in hand.

The complete result data records are taken from the `MCDResult` Object. This contains one `MCDResponse` Object for each ECU participating in the result. Each MCD ResponseObject is recursively composed of `MCDResponseParameter` objects in collections, because of which structured results may be decomposed up until its comprised elementary components.

For structured results the elements `eFIELD` (arrays or sequences), `eSTRUCTURE` and `eMULTIPLEXER` (union) are used.

By means of this mechanism all result descriptors (IDL) used in version 1.1 may be mapped. This is illustrated by the following example from version 1.1 (DTC).

All figures represent examples rather than normative definitions. For example, the top-level elements of type `eFIELD` and `eSTRUCTURE` are not normative but represent a possible data structure that could be read from the ODX database. The data structure visible at the D-server interface does directly correspond to the data structure read from the ODX database, i.e. it does not contain artificial structuring not present in the corresponding part of the ODX database. However, several different parameters without use of an `eSTRUCTURE` may only occur on the first level of response parameters.

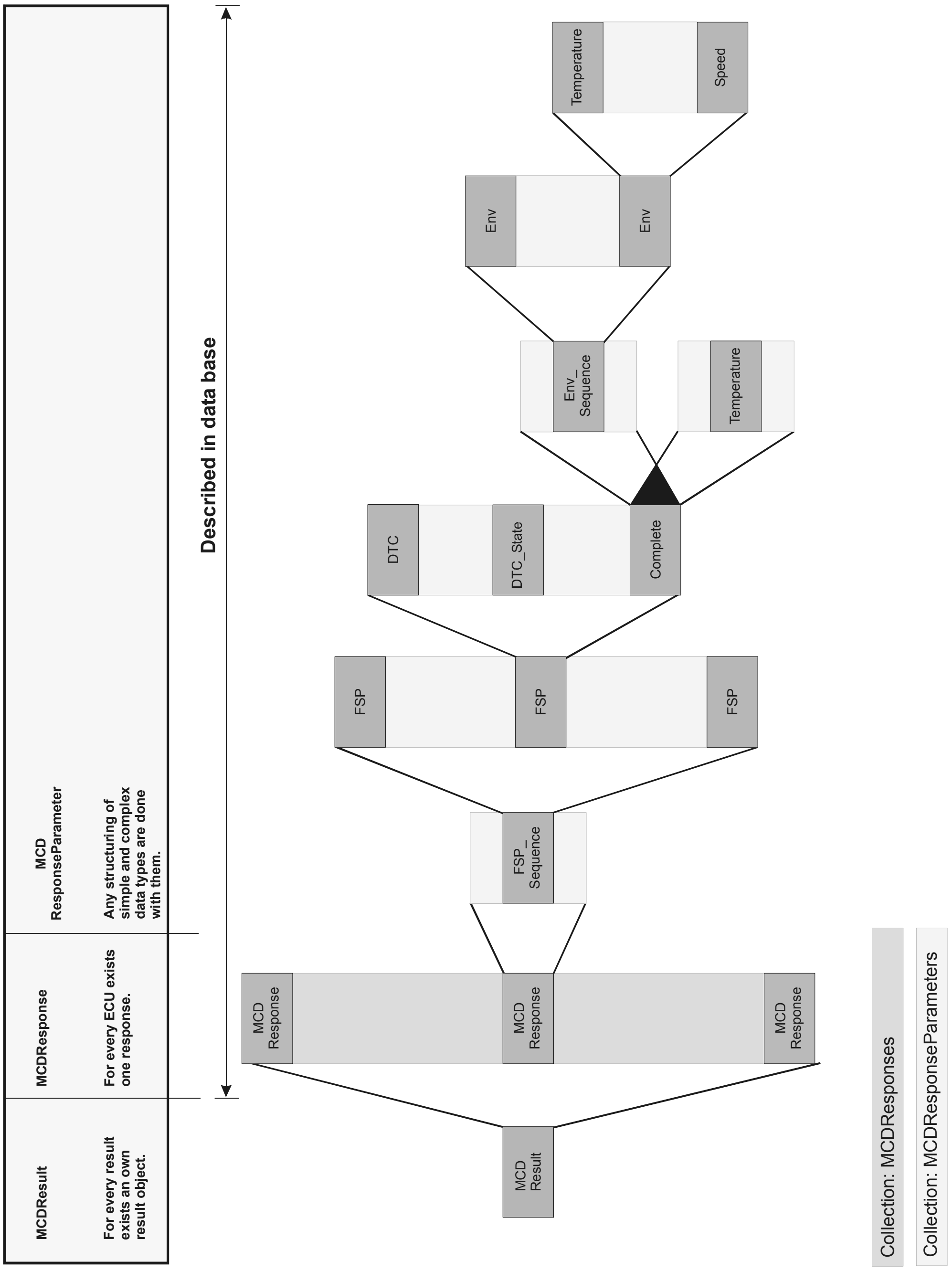
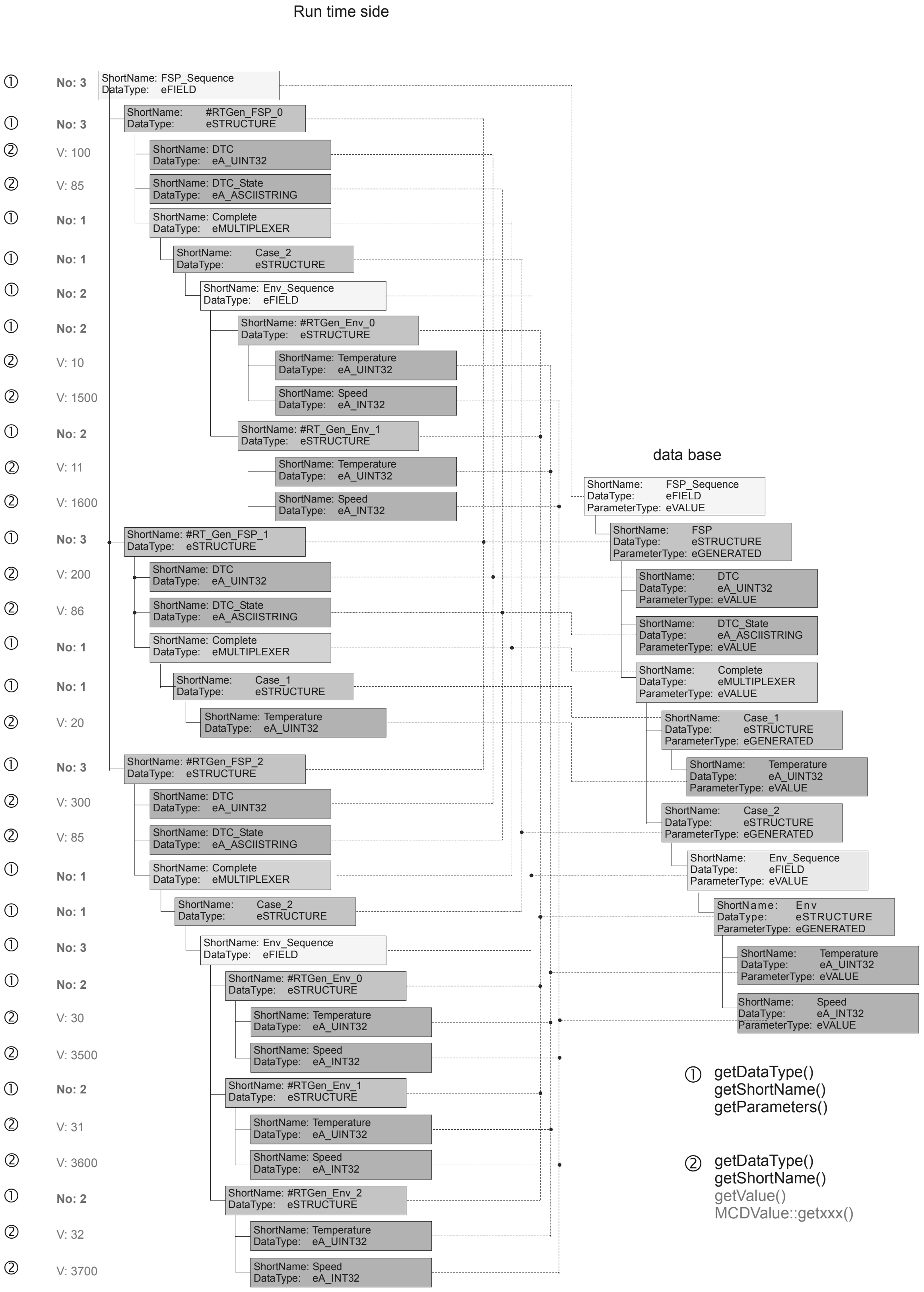


Figure 122 — Result structure DTC from example


Run time side
① No: 3 ShortName: FSP_Sequence DataType: eFIELD
① No: 3 ShortName: #RTGen_FSP_0 DataType: eSTRUCTURE
② V: 100 ShortName: DTC DataType: eA_UINT32
② V: 85 ShortName: DTC_State DataType: eA_ASCIISTRING
① No: 1 ShortName: Complete DataType: eMULTIPLEXER
① No: 1 ShortName: Case_2 DataType: eSTRUCTURE
① No: 2 ShortName: Env_Sequence DataType: eFIELD
① No: 2 ShortName: #RTGen_Env_0 DataType: eSTRUCTURE
② V: 10 ShortName: Temperature DataType: eA_UINT32
② V: 1500 ShortName: Speed DataType: eA_INT32
① No: 2 ShortName: #RT_Gen_Env_1 DataType: eSTRUCTURE
② V: 11 ShortName: Temperature DataType: eA_UINT32
② V: 1600 ShortName: Speed DataType: eA_INT32
① No: 3 ShortName: #RT_Gen_FSP_1 DataType: eSTRUCTURE
② V: 200 ShortName: DTC DataType: eA_UINT32
② V: 86 ShortName: DTC_State DataType: eA_ASCIISTRING
① No: 1 ShortName: Complete DataType: eMULTIPLEXER
① No: 1 ShortName: Case_1 DataType: eSTRUCTURE
② V: 20 ShortName: Temperature DataType: eA_UINT32
① No: 3 ShortName: #RTGen_FSP_2 DataType: eSTRUCTURE
② V: 300 ShortName: DTC DataType: eA_UINT32
② V: 85 ShortName: DTC_State DataType: eA_ASCIISTRING
① No: 1 ShortName: Complete DataType: eMULTIPLEXER
① No: 1 ShortName: Case_2 DataType: eSTRUCTURE
① No: 3 ShortName: Env_Sequence DataType: eFIELD
① No: 2 ShortName: #RTGen_Env_0 DataType: eSTRUCTURE
② V: 30 ShortName: Temperature DataType: eA_UINT32
② V: 3500 ShortName: Speed DataType: eA_INT32
① No: 2 ShortName: #RTGen_Env_1 DataType: eSTRUCTURE
② V: 31 ShortName: Temperature DataType: eA_UINT32
② V: 3600 ShortName: Speed DataType: eA_INT32
① No: 2 ShortName: #RTGen_Env_2 DataType: eSTRUCTURE
② V: 32 ShortName: Temperature DataType: eA_UINT32
② V: 3700 ShortName: Speed DataType: eA_INT32
data base
ShortName: FSP_Sequence DataType: eFIELD ParameterType: eVALUE
ShortName: FSP DataType: eSTRUCTURE ParameterType: eGENERATED
ShortName: DTC DataType: eA_UINT32 ParameterType: eVALUE
ShortName: DTC_State DataType: eA_ASCIISTRING ParameterType: eVALUE
ShortName: Complete DataType: eMULTIPLEXER ParameterType: eVALUE
ShortName: Case_1 DataType: eSTRUCTURE ParameterType: eGENERATED
ShortName: Temperature DataType: eA_UINT32 ParameterType: eVALUE
ShortName: Case_2 DataType: eSTRUCTURE ParameterType: eGENERATED
ShortName: Env_Sequence DataType: eFIELD ParameterType: eVALUE
ShortName: Env DataType: eSTRUCTURE ParameterType: eGENERATED
ShortName: Temperature DataType: eA_UINT32 ParameterType: eVALUE
ShortName: Speed DataType: eA_INT32 ParameterType: eVALUE
① getDataType() getShortName() getParameters()
② getDataType() getShortName() getValue() MCDValue::getxxx()

In case of Multiplexer the method getValue() delivers the switch type (index of the branch).

Figure 123 — Result structure DTC

ODX data for result structure DTC

```
<PARAM xsi:type="VALUE">
     <SHORT-NAME>FSP_Sequence</SHORT-NAME>
     <DOP-REF ID-REF="EOP_DOP_ID"/>
</PARAM>
<END-OF-PDU-FIELD ID=“EOP_DOP_ID“>
     <SHORT-NAME>EOP_DOP</SHORT-NAME>
     <BASIC-STRUCTURE-REF ID-REF=”FSP_ID”/>
</END-OF-PDU-FIELD>
<STRUCTURE ID=”FSP_ID”>
     <SHORT-NAME>FSP</SHORT-NAME>
     <PARAMS>
        <PARAM xsi:type=”VALUE”>
             <SHORT-NAME>DTC</SHORT-NAME>
        </PARAM>
        <PARAM xsi:type=”VALUE”>
             <SHORT-NAME>DTC_State</SHORT-NAME>
        </PARAM>
        <PARAM xsi:type=”VALUE”>
             <SHORT-NAME>Complete</SHORT-NAME>
             <DOP-REF ID-REF="MUX_DOP_ID"/>
        </PARAM>
     </PARAMS>
</STRUCTURE>
<MUX ID="MUX_DOP_ID">
     <SHORT-NAME>Mux_DOP</SHORT-NAME>
     <SWITCH-KEY>
        <DATA-OBJECT-PROP-REF ID-REF="SwitchKey_DOP_ID"/>
     </SWITCH-KEY>
     <CASES>
        <CASE>
             <SHORT-NAME>Case_1</SHORT-NAME>
             <STRUCTURE-REF ID-REF="Case1_DOP_ID"/>
        </CASE>
        <CASE>
             <SHORT-NAME>Case_2</SHORT-NAME>
             <STRUCTURE-REF ID-REF="Case2_DOP_ID"/>
        </CASE>
     </CASES>
</MUX>
<STRUCTURE ID="Case1_DOP_ID">
     <SHORT-NAME>StructCase_1</SHORT-NAME>
     <PARAMS>
        <PARAM xsi:type="VALUE">
             <SHORT-NAME>Temperature</SHORT-NAME>
        </PARAM>
     </PARAMS>
</STRUCTURE>
<STRUCTURE ID="Case2_DOP_ID">
     <SHORT-NAME>StructCase_2</SHORT-NAME>
     <PARAMS>
        <PARAM xsi:type="VALUE">
             <SHORT-NAME>Env_Sequence</SHORT-NAME>
             <DOP-REF ID-REF="Field_DOP_ID"/>
                </PARAM>
             </PARAMS>
        </STRUCTURE>
        <DYNAMIC-ENDMARKER-FIELD ID="Field_DOP_ID">
            <SHORT-NAME>Field_DOP</SHORT-NAME>
            <BASIC-STRUCTURE-REF ID-REF="Env_DOP_ID"/>
        </DYNAMIC-ENDMARKER_FIELD>
        <STRUCTURE ID="Env_DOP_ID">
             <SHORT-NAME>Env</SHORT-NAME>
             <PARAMS>
                <PARAM xsi:type="VALUE">
                     <SHORT-NAME>Temperature</SHORT-NAME>
                </PARAM>
                <PARAM xsi:type="VALUE">
                     <SHORT-NAME>Speed</SHORT-NAME>
                </PARAM>
             </PARAMS>
        </STRUCTURE>
```

DTC	DTC_State	Temperature	Speed	Temperature	Speed
100	85	10	1500	11	1600

DTC	DTC_State	Temperature
200	86	20

DTC	DTC_State	Temperature	Speed	Temperature	Speed	Temperature	Speed
300	85	30	3500	31	3600	32	3700

Figure 124 — Example of error memory (communication view)

The value of a parameter of type eDTC is the encoded trouble as read from the ODX data. This means that the value can contain an encoded P for the trouble code P130 (= 304).

All Runtime Responses and Response Parameters of the Results are based upon the respective database templates, which contain the related meta information. Thus, the name (semantic assignment) as well as the type is predefined. Concerning its structure, a Runtime result shall always be in conformance with the database template.

Firstly, the type of the Response Parameter shall always be requested. If this is a simple data type, the value and short name of the element can be polled (for the semantic interpretation of the result).

In case of a complex data type (Structure or Field) the number of the Response Parameters on the next hierarchical level is polled and following the list of these Response Parameters is read in.

This is repeated as often as no further Response Parameter is in hand.

In case of a Multiplexer the Value contains the select value (branch index beginning at 0) which has been used for the respective branch.

If an empty CASE (CASE without DOP) is used within a Multiplexer (MUX), it is not shown at the DbTemplate. The Runtime Result delivers a Multiplexer Element without a following ResponseParameter.

Additionally, there is a possibility to poll the result for the related Service.

Table 33 — Overview about Request-, Response- and Protocol parameter data types

DataType **as delivered by `getDataType()`**	**Type**	**Included in Request Parameter**	**Included in Response Parameter**	**Physical data type of the MCDValue delivered by `getValue()`**	**Content of the Value delivered by `getValue()`**
eFIELD	complex	Yes (only at Protocol Parameters)	Yes	A_UINT32	Number of entries directly contained in the field
eMULTIPLEXER	complex		Yes	A_UINT32	Branch index
eSTRUCTURE	complex	Yes	Yes	A_UINT32	Number of entries directly contained in the structure
eENVDATA	complex		Yes	A_UINT32	Value of eDTC
eENVDATADESC	complex		Yes	A_UINT32	Number of entries directly contained in the ENVDATADESC parameter
eDTC	simple	Yes	Yes	A_UINT32	Value of the corresponding element TROUBLE-CODE with respect to ODX
eEND_OF_PDU	complex	Yes		A_UINT32	Number of entries directly contained in the END_OF_PDU parameter
eTEXTTABLE	simple	Yes	Yes	A_UNICODE 2STRING	Text of current conversion
eTABLE_ROW	simple	Yes	Yes	A_UINT32	Number of entries directly contained in the table row

Complex means that the data type is structured. The next hierarchical level of parameters can contain elements. So in case of an complex (structured) data type the next collection of parameters shall be taken over with `getResponseParameters/getParameters`.

In ODX, different types of fields exist. These field can be distinguished by their type of termination. One of these fields is the END-OF-PDU field. Fields of this type can be represented by a parameter of type eFIELD in an MCDResponse in D-server as the number of elements is already known in the server when processing the response from the ECU.

The SWITCH-KEY is converted using the referenced DATAOBJECT-PROP; The LIMITs of the CASE shall be type-converted into the PHYSICAL-TYPE/BASE-DATA-TYPE of this same DATAOBJECT-PROP prior to comparison against the SWITCH-KEY Value. The DISPLAY-RADIX of the PHYSICAL-TYPE is ignored, because string-values containing numbers always default to DEC radix.

The MCDDataType eNO_TYPE can only occur at objects of type MCDValue. Here, it signals that this MCDValue has not been initialized, e.g. because it has just been created by `MCDObjectFactory::createValue().`

Table 34 — Table of parameter types for parameters generated by the MCD-server (only if no direct relation to a parameter in ODX):

DataType as delivered by getDataType()	ParameterType As delivered by getParameterType()
eSTRUCTURE	eGENERATED (in case the structure represents a MUX-CASE or is directly contained in a FIELD or an END_OF_PDU
eTABLE_ROW	eGENERATED

Table 35 — Table of parameter types for protocol parameters:

Type of Protocol Parameter with respect to ODX	ParameterType as delivered by getParameterType()	DataType as delivered by getDataType()
COMPARAM	eVALUE	depends on referenced DOP in ODX
COMPLEX-COMPARAM	eGENERATED	eSTRUCTURE

EXAMPLE Notes concerning the example:

For each ECU n DTC errors may occur. Each error is symbolized by the ResponseParameter FSP. Using the IDL Descriptor in version 1.1, the errors of one ECU at a certain instant of time have been framed in the FSP Sequence. To emphasis the mapping of the descriptor concept within the object model, the error description has been taken over qualitative identically (Element Env_Code is renamed to Temperature and element Env_State is renamed to Speed). Using the object model, the framing of errors into the FSP Sequence would not be necessary, as each error may be handed over as independent Response Parameter. Within the figure in hand, each Response contains only one Response Parameter (FSP_Sequence).

Basically, the result structure of a Service is determined by the data, with which the Runtime Result shall be in conformance. However, generally it can be applied, that for each complete result data record (of a Cyclic Diag Service or non cyclic Service in Repetition mode) one ResultObject is created, that means that the number of result data records is identical to the number of ResultObjects.

The example "read diagnostic trouble code" is also used as Job example. For this, every minute the ECU is polled, but only every two minutes an intermediate result of the Job is given out. In this case also, the number of Results is two, because every minute a result is produced by ECU.

Subsequently an example for a simple data type (e.g. temperature) is shown. In this case the Response Structure only contains the element with the basic data type.

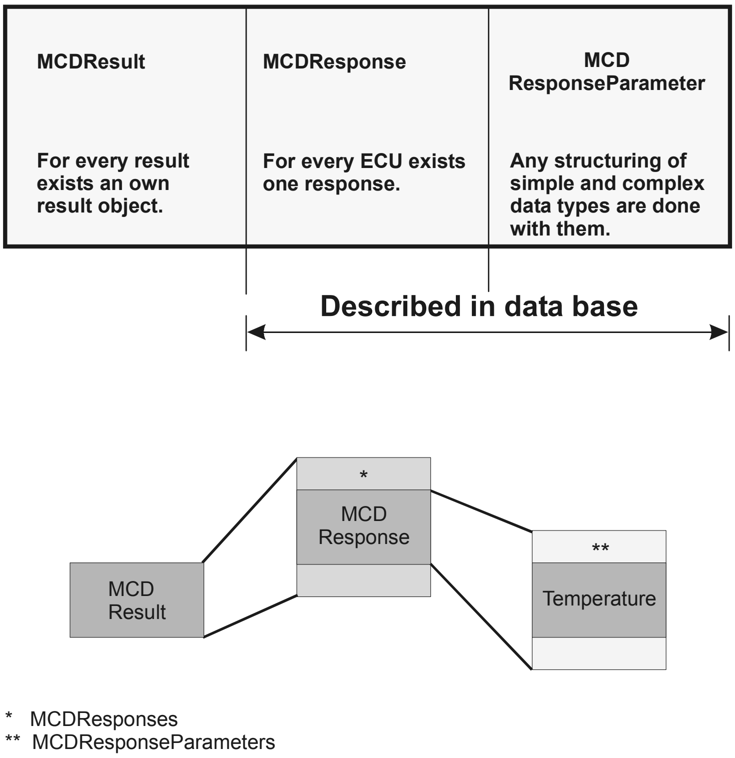

MCDResult
For every result exists an own result object.
MCDResponse
For every ECU exists one response.
MCD ResponseParameter
Any structuring of simple and complex data types are done with them.
Described in data base
*
MCD Response
MCD Result
**
Temperature
* MCDResponses
** MCDResponseParameters

Figure 125 — Response structure DTC for only one ResponseParameter

All possible response templates can be obtained at a MCDDbDiagComPrimitive by using the methods getDbResponses() or getDbResponsesByType().

Result handling in case of Functional Groups

On the functional group level, only the DbResponses (positive response, local negative responses, and global negative responses) defined on this specific layer or inherited from a protocol layer will be considered. That is, for each DiagComPrimitive accessed on a functional group layer, no responses defined on ECU base variant layer or ECU variant layer are considered.

Relevant database objects:

- 1 DbFunctional Group
- 1 DbService
- n DbResponses (positive, local negative, and global negative responses defined up to the functional group layer)

At runtime, the ECUs addressed in functional communication can respond with positive, local negative or global negative responses that conform to DbResponse on functional group, base variant, or ECU variant level.

Relevant runtime objects:

- 1 Functional Group
- 1 Service
- m Responses (one response for each ECU that has answered the functional request; responses conform to response templates on functional group, base variant, or ECU variant level)

The rule for runtime response interpretation is shown in Figure 82 — Interpretation algorithm for responses of services on different diagnostic layers. A corresponding description can also be found in 9.8.3.

In case of functional communication the access key of the responding ECU is part of its response:

— In case of response with the physical address the access key will be of base variant or variant.

— In case of response with functional address the access key will be of functional group.

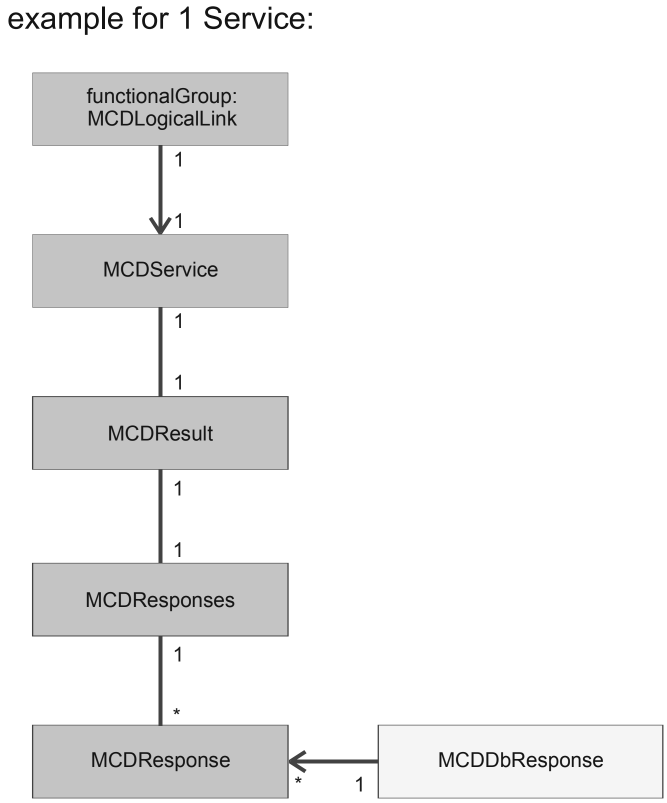


Figure 126 — Example for one service

9.12 Filtering of results

9.12.1 Principle

Results shall meet the Database Template, which means that all variants that may occur in the result shall be included in the Template. Usually, this is realized by means of multiplexers. If the application is only interested in parts of the results, this filtering shall be carried out actively by the MCD-server. For this, the application creates a new Database Template (in this case named Filter Template), which again shall be met by all results. The new Database Template may only consist of elements from the original Database Template; that means that the adding of new elements is not allowed. Filtering is done in the way, that all Response Parameter to be used are stated completely. Only the positive response of a MCDDataPrimitive can filtered.

The set of AsciiStrings used to define the filter is matched against any response returned for a certain DiagComPrimitive by the PDU API. As a result, every single AsciiString in the filter template is considered a separate mini-filter which can be applied to a response independently. For filtering a response, all mini-filters in a filter definition are applied to a runtime response. Mini-filters which do not match are discarded for that specific filter procedure – no exception is thrown. Every mini-filter, which can be applied successfully to a runtime response filters this response accordingly. After all mini-filters have been applied successfully or unsuccessfully to a runtime response, the filtering for this specific response is terminated and the remaining information of the response is passed to the client application.

This shall be demonstrated with help of the Database Template Figure 123 — Result structure DTC shown in figure Figure 127 — Database Template.

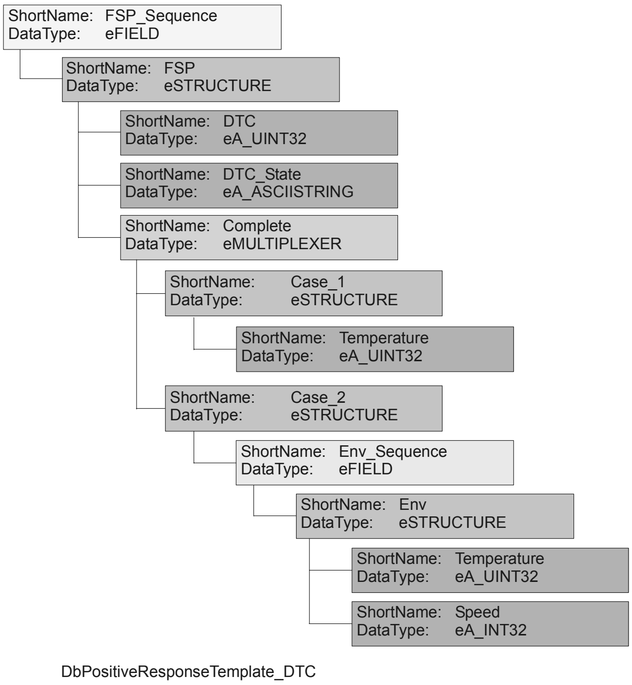


Figure 127 — Database Template

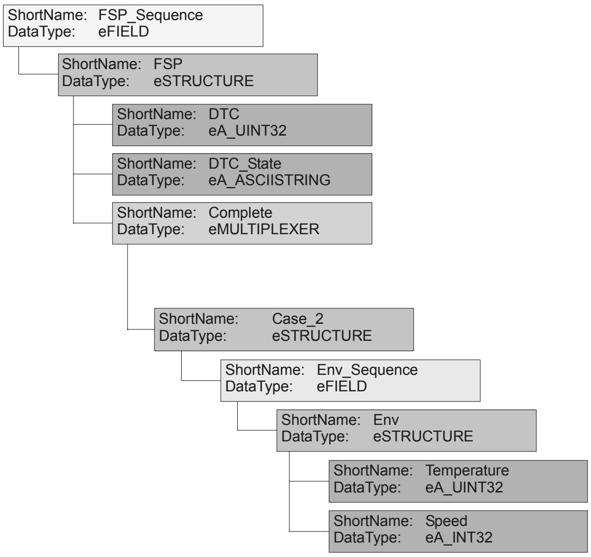
ODX data for database template

```
<PARAM xsi:type="VALUE">
     <SHORT-NAME>FSP_Sequence</SHORT-NAME>
     <DOP-REF ID-REF=“EOP_DOP_ID“/>
</PARAM>

<END-OF-PDU-FIELD ID=“EOP_DOP_ID“>
     <SHORT-NAME>EOP_DOP</SHORT-NAME>
     <BASIC-STRUCTURE-REF ID-REF=”FSP_DOP_ID”/>
</END-OF-PDU-FIELD>

<STRUCTURE ID=”FSP_DOP_ID”>
     <SHORT-NAME>FSP</SHORT-NAME>
     <PARAMS>
        <PARAM xsi:type=”VALUE”>
             <SHORT-NAME>DTC</SHORT-NAME>
        </PARAM>
        <PARAM xsi:type=”VALUE”>
             <SHORT-NAME>DTC_State</SHORT-NAME>
        </PARAM>
        <PARAM xsi:type=”VALUE”>
             <SHORT-NAME>Complete</SHORT-NAME>
             <DOP-REF ID-REF="MUX_DOP_ID"/>
        </PARAM>
     </PARAMS>
</STRUCTURE>

<MUX ID="MUX_DOP_ID">
```

```
     <SHORT-NAME>Mux_DOP</SHORT-NAME>
     <SWITCH-KEY>
        <DATA-OBJECT-PROP-REF ID-REF="SwitchKey_DOP_ID"/>
     </SWITCH-KEY>
     <CASES>
        <CASE>
             <SHORT-NAME>Case_1</SHORT-NAME>
             <STRUCTURE-REF ID-REF="Case1_DOP_ID"/>
        </CASE>
        <CASE>
             <SHORT-NAME>Case_2</SHORT-NAME>
             <STRUCTURE-REF ID-REF="Case2_DOP_ID"/>
        </CASE>
     </CASES>
</MUX>

<STRUCTURE ID="Case1_DOP_ID">
     <SHORT-NAME>StructCase_1</SHORT-NAME>
     <PARAMS>
        <PARAM xsi:type="VALUE">
             <SHORT-NAME>Temperature</SHORT-NAME>
        </PARAM>
     </PARAMS>
</STRUCTURE>

<STRUCTURE ID="Case2_DOP_ID">
     <SHORT-NAME>StructCase_2</SHORT-NAME>
     <PARAMS>
        <PARAM xsi:type="VALUE">
             <SHORT-NAME>Env_Sequence</SHORT-NAME>
             <DOP-REF ID-REF="Field_DOP_ID"/>
        </PARAM>
     </PARAMS>
</STRUCTURE>

<DYNAMIC-ENDMARKER-FIELD ID="Field_DOP_ID">
     <SHORT-NAME>Field_DOP</SHORT-NAME>
     <BASIC-STRUCTURE-REF ID-REF="Env_DOP_ID"/>
</DYNAMIC-ENDMARKER_FIELD>

<STRUCTURE ID="Env_DOP_ID">
     <SHORT-NAME>Env</SHORT-NAME>
     <PARAMS>
        <PARAM xsi:type="VALUE">
             <SHORT-NAME>Temperature</SHORT-NAME>
        </PARAM>
        <PARAM xsi:type="VALUE">
             <SHORT-NAME>Speed</SHORT-NAME>
        </PARAM>
     </PARAMS>
</STRUCTURE>
```

The Response Parameters are declared as unstructured Collection of elements. At this, one element is given as a sequence of DbShortNames each separated by a | (ASCII 124), starting from the root. Simplifications are achieved as in a positive case all elements contained below the ResponseParameter are taken over automatically. The main structure shall be retained in any case.

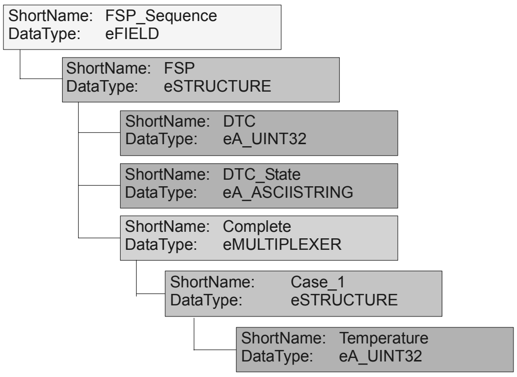
Filter Templates are created by giving the elements to be used. On the left side you see the filter template which should be used and on the right side the Collection of sequences of DbShortnames, which is used to create the filter template from the original Database template. Every picture shows a new example.



Collection:

- FSP_Sequence|FSP|DTC
- FSP_Sequence|FSP|DTC_State
- FSP_Sequence|FSP|Complete|Case_2|Env_Sequence

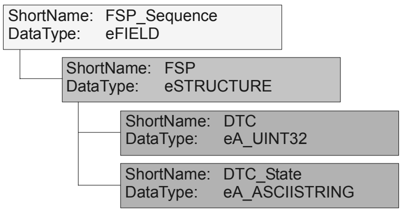
Figure 128 — Example 1: Multiplexer Env_Sequence



Collection:

- FSP_Sequence|FSP|DTC
- FSP_Sequence|FSP|DTC_State
- FSP_Sequence|FSP|Complete|Case_1|Temperature

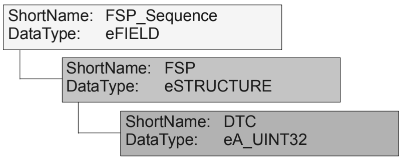
Figure 129 — Example 2: Multiplexer Temperature



Collection:

- FSP_Sequence|FSP|DTC
- FSP_Sequence|FSP|DTC_State

Figure 130 — Example 3: FSP Structure only (without Multiplexer)



Collection:

- FSP_Sequence|FSP|DTC

Figure 131 — Example 4: FSP Structure only with DTC's

9.12.2 Handling rules

The FilterTemplate is created and administrated within the D-server. Thus, the Client only presets the modification compared with the DbTemplate.

Elements occurring on RuntimeSide which do not meet the FilterTemplate shall be discarded automatically by the MCD-server. In an extreme case this may lead for some selections to the fact that no Results will be in hand for the Client.

`MCDDataPrimitive::setFilter()` sets a new filter for the positive response of this MCDDataPrimitive and overwrites any previously set filter. A FilterTemplate is generated which is used as DbTemplate for the assignment of the RuntimeResults. The method `setFilter()` is only allowed if the service is currently not executed. A filter which has been set is valid and active until

- the `MCDDataPrimitive` is removed
- the filter is overwritten with another filter using `setFilter()`
- the filter is removed using the method `removeFilter()`

Every string in the parameter `dbNames` represents a concatenation of ShortNames of response parameters such that every string is a path to a specific response parameter. Filter strings which do not match a response parameter in the current response do not result in an exception. An empty collection of filter strings is considered as an empty filter – all response parameters will pass this filter.

`MCDDataPrimitive / MCDResult :: getFilter()` returns a collection of strings representing the current filter at the `DataPrimitive`. In case no filter is set, an empty collection is returned.

`MCDDiagService::removeFilter()` removes the filter currently set at the DataPrimitive. All response parameters are returned to a client application after removing the filter. No exception will be thrown in case no filter had been set before.

`MCDDataPrimitive / MCDResult :: isFilterActive()` returns true in case a non-empty filter (non-empty collection of filter strings) is set at this DataPrimitive. False is returned otherwise.

Result filtering guarantee the performance, which had been available using the IDL Descriptor in procedural draft of MCD-3 specification.

9.13 Read DTC

Per location (DIAG-LAYER) exits different fault memories each containing a set of DTCs. In D-server all variant related DTCs should be returned for the database part. An element of type MCDDbFaultMemory (ODX: DTC-DOP) contains a collection of type MCDDbDiagTroubleCodes. The members of this collection are elements of type MCDDbDiagTroubleCode (ODX: DTC).

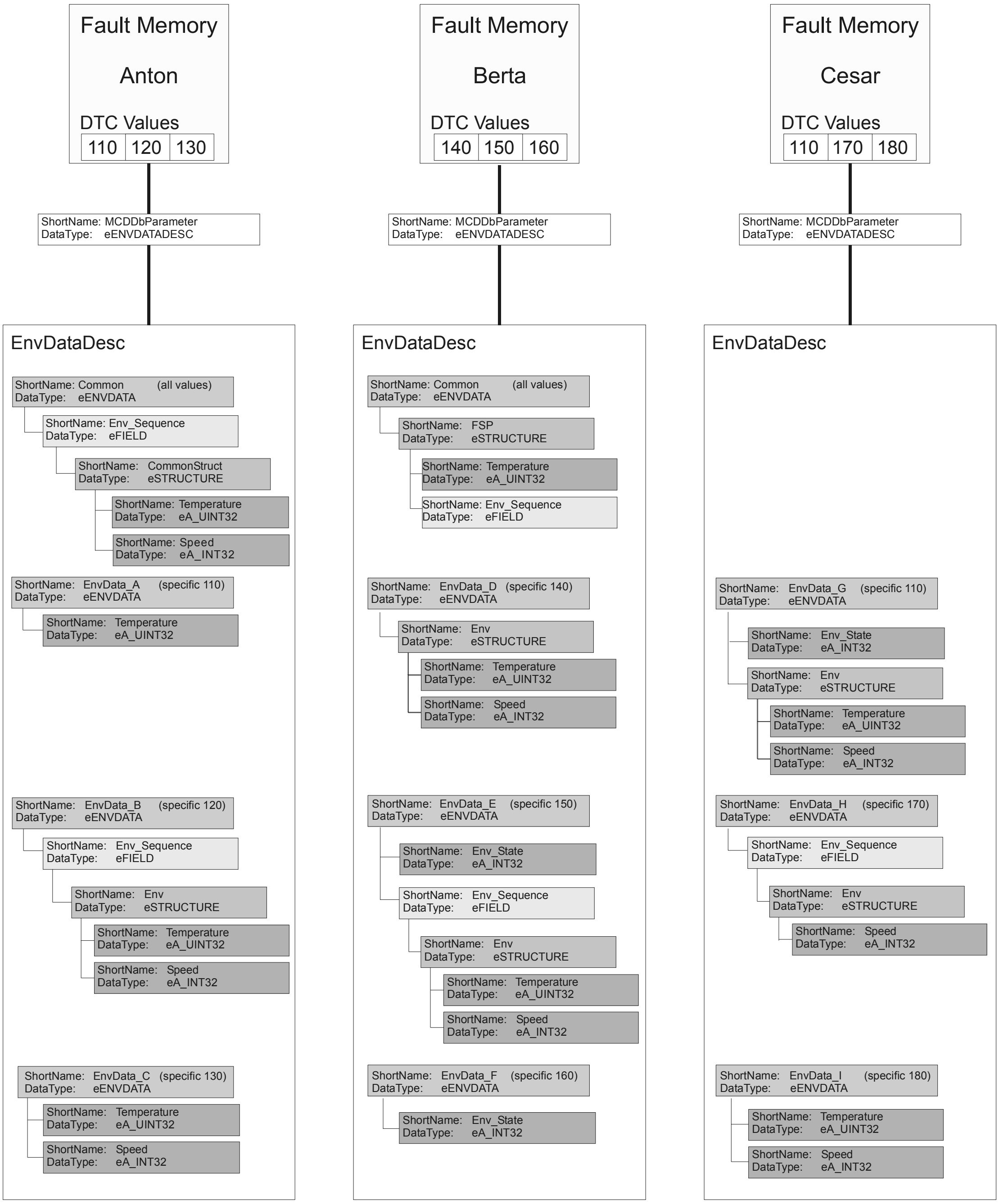


Figure 132 — Relation between FaultMemory and EnvDataDesc

In ODX, several DTC-DOPs can be referenced from one ECU. Furthermore, DTC-DOPs can be linked into another DTC-DOP. This allows to compose a new DTC-DOP from existing DTC-DOPs. At the interface of the D-server, any DTC-DOP to be presented, that is, every DTC-DOP referenced by an ECU, is already converted into a flat list of DTCs – all links have been resolved.

In addition, there can be several eENVDATADESCs in ODX. Therefore, a response parameter of type eENVDATADESC in D-server links together a eENVDATADESC from ODX with a response parameter of type eDTC. The value of this response parameter of type eDTC is then used in the server to calculate the sub-structures of the corresponding response parameter of type eENVDATADESC at runtime.

With respect to ODX, elements of type ENV-DATA are COMPLEXDOPs of type BASIC-STRUCTURE. Therefore, a response parameter of type eENVDATADESC can contain more than one complex response parameter of type eENVDATA.

Per ENV-DATA-DESC there can be at most one ENV-DATA applying to all DTCs, all others need to be DTC-specific. Every DTC can have at most one specific ENV-DATA applying to it. The order of the corresponding structures is defined as: ALL-VALUE ENVDATA comes first (named common in D-server and may be empty, which means no data in ODX), then the DTC-specific ENV-DATA.

ENVDATA's are returned as collection of type MCDDbResponseParameters.

The method `MCDbEnvDataDesc::getCompleteDbEnvDatasByDiagTroubleCode (A_UNIT32 troubleCode)` returns a collection of type MCDDbResponseParameters. This collection contains at most a MCDDbResponseParameter of type eENVDATA which contains response parameters representing common environment data and at most a MCDDbResponseParameter of type eENVDATA which contains response parameters representing environment data specific to a certain DTC value (mind the order).

The method `MCDbEnvDataDesc::getCommonDbEnvDatas()` returns a collection of type MCDDbResponseParameters. This collection contains at most a MCDDbResponseParameter of type eENVDATA which contains response parameters representing common environment data.

The method `MCDbEnvDataDesc::getSpecificDbEnvDatasByDiagTroubleCode (A_UNIT32 troubleCode)` returns a collection of type MCDDbResponseParameters. This collection contains at most a MCDDbResponseParameter of type eENVDATA which contains response parameters representing environment data specific to a certain DTC value.

In general, a DB template for environment data could contain several DTC DOPs and several ENVDATADESC DOPs. These DOPs can be structured arbitrarily. They can even reside on different hierarchical levels in the DB template.

In case of DTC there are three additional ResponseParameter types.

The type `eDTC` is a Simple DOP. For a ResponseParameter of the type `eDTC`, `getValue` returns the TroubleCode of the DTC as A_UINT32 encapsulated in MCDValue.

The type `eENVDATA` on the other hand is a Complex DOP, which represents the environment data. This type combines the advantages of a structure with the characteristic features of a Multiplexer. `eENVDATA` returns a Collection of ResponseParameters, which depending on the occurring data types (Simple or Complex DOP) may contain data or further Collections of ResponseParameters. In case of a complex type only eFIELD, eSTRUCTURE, or eMULTIPLEXER are allowed. For a ResponseParameter of the type `eENVDATA` method `getValue` returns the Switch-Param as A_UINT32. This value is equal to the value of `eDTC`.

This shall be demonstrated with help of parts of the Database Template Figure 123 — Result structure DTC shown in figure Figure 127 — Database Template (Please note the hint of page 217).

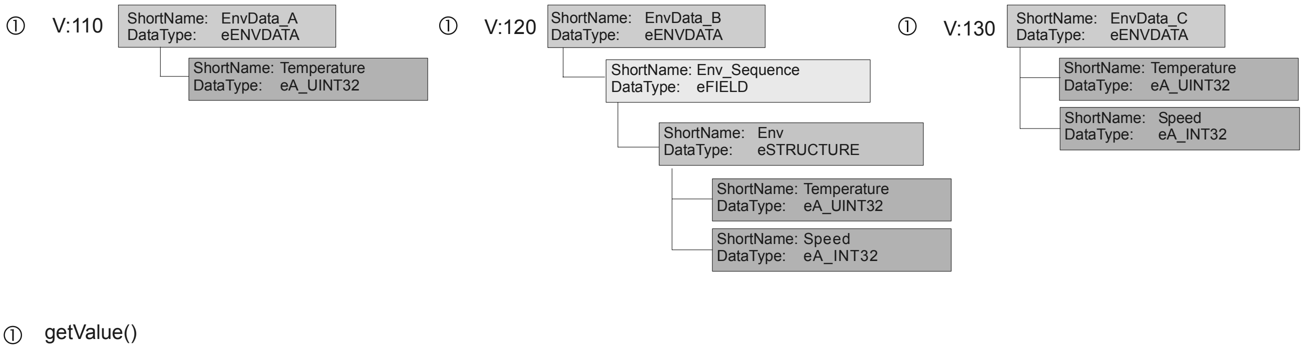


Figure 133 — Different eEnvData blocks for an eDTC element

The type `eENVDATADESC` is a complex DOP, which announces the include of an environment data (`eENVDATA`) block. Inside a Db result template `eENVDATADESC` and `eDTC` shall be on the same hierarchical level.

Every path in a response structure starting at its root element down to a leaf shall contain at most one element of type `eENVDATADESC`.

The collection of DbResponseParameters at this DOP (`eENVDATADESC`) consists of zero and the collection of ResponseParameters (run time side) consist of zero or one element of type `eENVDATA`, i.e. the collection is empty if there is no environment data available.

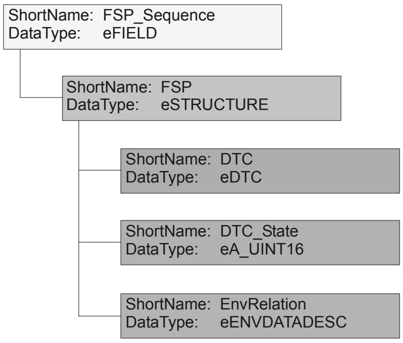
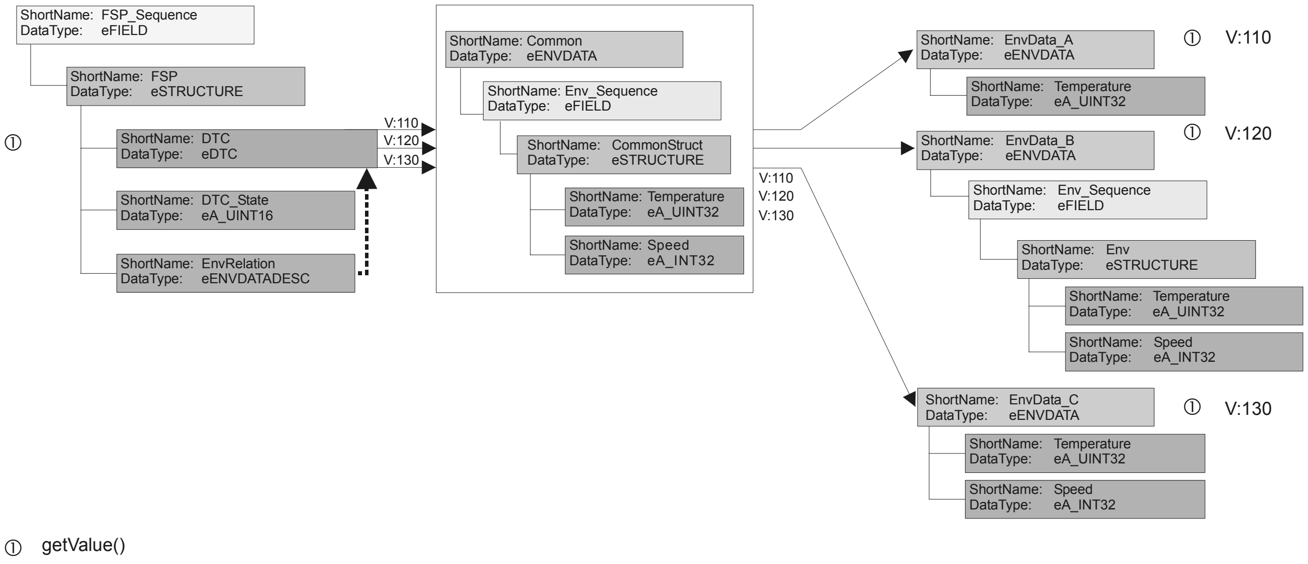


Figure 134 — Usage of Type eEnvDataDesc together with eDTC

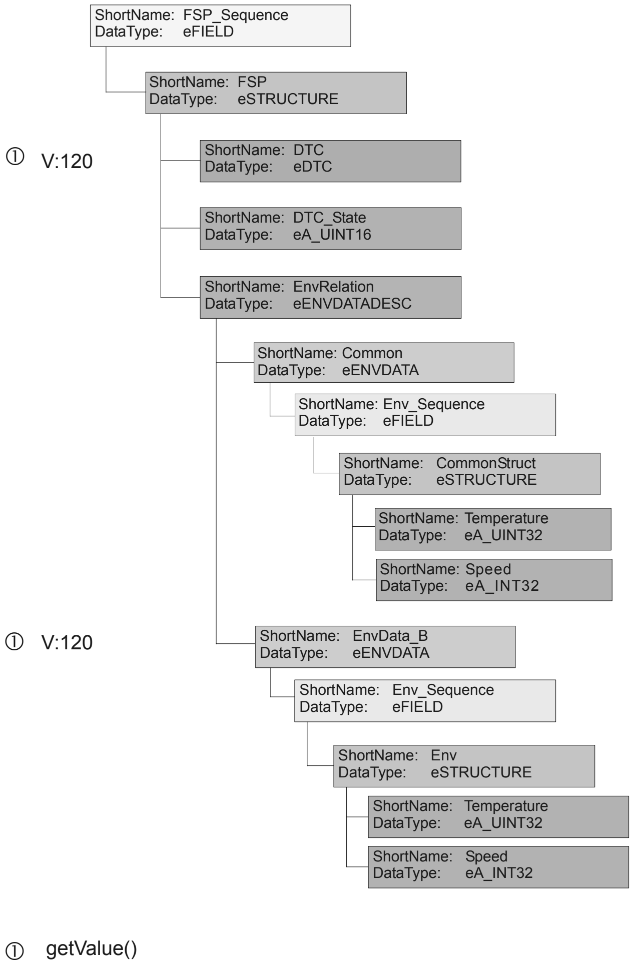
The names of all Response Parameters of the type `eENVDATA` within a database Template shall be unique. Advantage of the separation between the database template finished with `eENVDATADESC` and the different `eENVData` constructs is, that independent from the Response Parameter `eDTC` (the DTC Value), all variants of the EnvironmentData, which may occur via the selected Service for this Location, shall not be included within the Database Template.



ShortName: FSP_Sequence DataType: eFIELD
ShortName: FSP DataType: eSTRUCTURE
ShortName: DTC DataType: eDTC
ShortName: DTC_State DataType: eA_UINT16
ShortName: EnvRelation DataType: eENVDATADESC
V:110
V:120
V:130
ShortName: Common DataType: eENVDATA
ShortName: Env_Sequence DataType: eFIELD
ShortName: CommonStruct DataType: eSTRUCTURE
ShortName: Temperature DataType: eA_UINT32
ShortName: Speed DataType: eA_INT32
ShortName: EnvData_A DataType: eENVDATA
ShortName: Temperature DataType: eA_UINT32
ShortName: EnvData_B DataType: eENVDATA
ShortName: Env_Sequence DataType: eFIELD
ShortName: Env DataType: eSTRUCTURE
ShortName: Temperature DataType: eA_UINT32
ShortName: Speed DataType: eA_INT32
ShortName: EnvData_C DataType: eENVDATA
ShortName: Temperature DataType: eA_UINT32
ShortName: Speed DataType: eA_INT32
① getValue()

Figure 135 — Relation between database template and different environment data blocks

On the runtime side, the result is populated dynamically. Here, the value of the eDTC element defines which eENVDATA block is to be used in the runtime response structure.



ShortName: FSP_Sequence DataType: eFIELD
ShortName: FSP DataType: eSTRUCTURE
① V:120
ShortName: DTC DataType: eDTC
ShortName: DTC_State DataType: eA_UINT16
ShortName: EnvRelation DataType: eENVDATADESC
ShortName: Common DataType: eENVDATA
ShortName: Env_Sequence DataType: eFIELD
ShortName: CommonStruct DataType: eSTRUCTURE
ShortName: Temperature DataType: eA_UINT32
ShortName: Speed DataType: eA_INT32
① V:120
ShortName: EnvData_B DataType: eENVDATA
ShortName: Env_Sequence DataType: eFIELD
ShortName: Env DataType: eSTRUCTURE
ShortName: Temperature DataType: eA_UINT32
ShortName: Speed DataType: eA_INT32
① getValue()

Figure 136 — Runtime result for DTC example

The SEMANTIC-Attribute „ DEFAULT-FAULT-READ “, which may only occur for a single service per Location, is used for instancing the default Read DTC Service by means of the method MCDDiagComPrimitives:addBySemanticAttribute (semantic:A_ASCIISTRING): MCDDiagComPrimitive.

The method MCDDbLocation::getDbDiagComPrimitivesBySemanticAttribute (semantic A_ASCIISTRING) :MCDDbDiagComPrimitives is used to filter out DiagComPrimitives from the database on the basis of the Semantics defined within the database. To enable future extension by defining further Semantic Attributes, the Semantic Attribute is handed over as a String.

In case of the existence of more than one service for one semantic attribute, the method MCDDiagComPrimitives:addBySemanticAttribute does not create any service but returns the error eRT_NO_UNIQUE_SEMANTIC_ATTRIBUTE. Some semantic attributes are unique within a certain scope, e.g. for a DIAG-LAYER. For example, the semantic DEFAULT-FAULT-READ needs to be unique within a DIAG-LAYER.

The semantic gives additional information on the DiagComPrimitives nature, e.g. SECURITY or FAULTREAD. A number of semantics is predefined in ODX (see ASAM MCD 2 D ODX, Appendix A). Additional ones can be defined by the users of ODX.

9.14 Logical Link

9.14.1 General

In a MCD-server, a Logical Link is the combination of a physical (bus) link to (a) target ECU(s), and the protocol that is used to communicate with the target ECU(s). As such, a Logical Link unambiguously defines the access path to the ECU(s) the diagnostic application will be talking to.

In a D-server, the implementation of Logical Links is split into two parts, which will be referred to accordingly in the following subclauses. All the functionality that is common to the M, C and D function blocks of the MCD-3 standard is contained within the MCDLogicalLink and MCDDbLogicalLink classes, which are situated in the asam.mcd3 package. Inheriting from these classes are the MCDDLogicalLink and MCDDbDLogicalLink classes, respectively, which extend their base classes with functionality that is specific to the D function block of the MCD-3 standard. They are part of the asam.mcd3.d package of the API model specification.

9.14.2 Connection overview

Information about Logical Links is contained in the Logical Link Table. Elements of this table are the AccessKey (which contains the communication protocol), and the Physical Vehicle Link (because Logical Links only specify the vehicle side of a communications link). Logical Links are used to access the same ECU on different ways (for example, an ECU reachable via a CAN and a K-Line bus link), or access more than one ECU instance on the same physical bus by using different protocol-level links.

When selecting a Logical Link, selection of a Location and the respective access path within the MCD-server takes place. The application can use the short name of Logical Links to instantiate links and work with ECUs. The short name of a Logical Link is defined in the Logical Link Table.

Different Logical Links can share the same PhysicalLink to different ECUs. Each Logical Link is assigned an own instruction queue (represented by its activity state) for the execution of DiagComPrimitives.

Within the Logical Link Table, only the BaseVariant is entered for each ECU, which describes an unambiguous access path to the ECU. The true Variant of an ECU may be polled or identified using the Variant Identification mechanism of the D-server. It is also possible to directly instantiate a Variant via a Logical Link. The combination of the different access layers and a specific ECU variant or base variant is represented by a DbLocation object within a D-server. Therefore, each Logical Link has a reference to an associated Location object, which subsumes all the properties and services that result from its specific combination of ECU variant and protocol layer. For every instantiated Logical Link, only one Location can be active at one point in time. Per ECU, for each Physical Vehicle Link only one runtime Logical Link for either a Base Variant or a Variant is permitted. That means, that either a Base Variant link or a Variant link may be instanced at the same point in time.

9.14.3 State diagram of Logical Link

At the Logical Link it is distinguished between the `MCDLogicalLinkState` and the `MCDActivityState`. The state diagrams are influenced by the states of the communication primitives / services that are created on the Logical Link (see Table 29 — Events in case of single or repeated execution of DiagService and Jobs).

All state transitions are indicated by means of events. Non state changing operations, for the `MCDLogicalLinkState` as well as the `MCDActivityState`, will not produce an event. For example, the <D> method `MCDDLogicalLink:reset()`called in MCDLogicalLinkState `eCREATED` does not produce an event or an exception. This is a state transition inside the state.

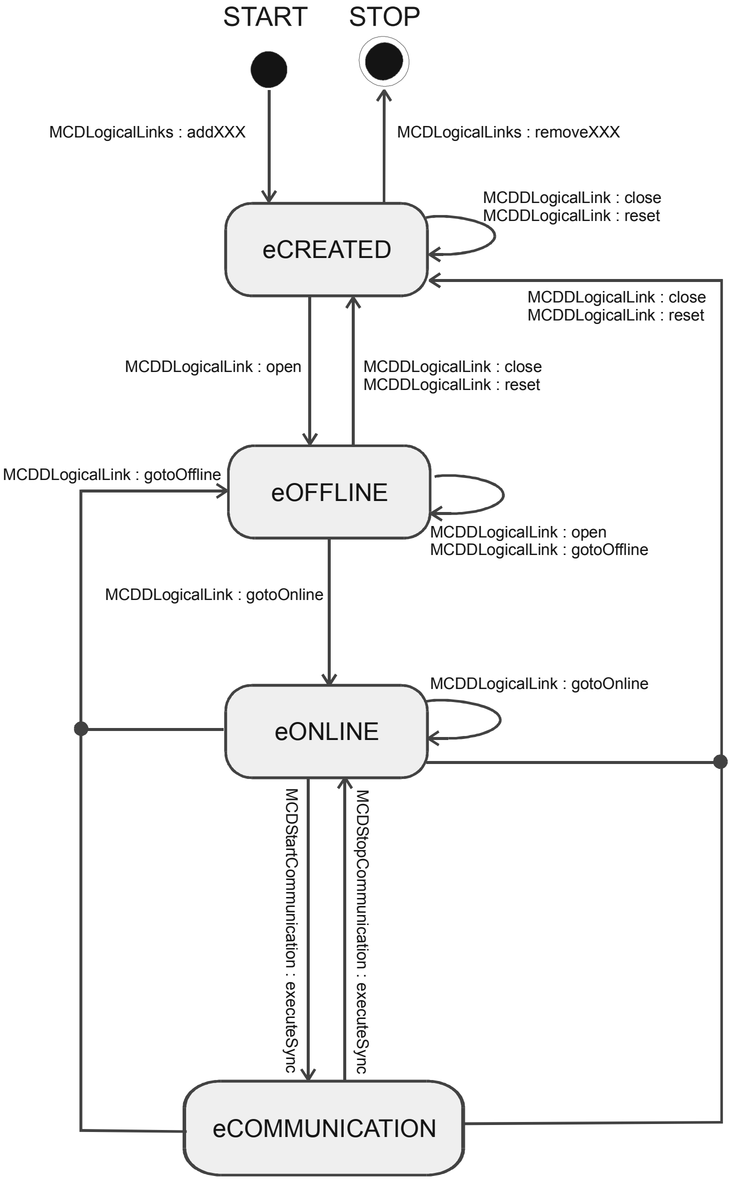


Figure 137 — State diagram MCDDLogical Link

Table 36 — Logical Link States (Function block D)

		MCDLogicalLinks			MCDDLogicalLink							MCDStart Communi cation	MCDStop Communi cation	MCDDiag ComPrimi tives
System State	LogicalLink State	removeByIndex	removeByName	removeAll	open	close reset	gotoOffline	gotoOnline	setInterfaceRessource	getMatchingInterfaceRessource	sendBreak	executeSync	executeSync	addByDbObject addByName addBySemanticAttribute addByType
eLOGICALLY_ CONNECTED	eCREATED	X	X	X	X	X			X	X				X
	eOFFLINE				X	X	X	X		X	X			X
	eONLINE					X	X	X		X	X	X		X
	eCOMMUNICATIO N					X	X			X	X		X	X

The method `MCDDLogicalLink::reset()` never throws an exception, and will always reset the links `MCDLogicalLinkState` to `eCREATED`. In case `MCDDLogicalLink::reset()` has been called at a LogicalLink which has queued DiagComPrimitives, an event of type `onPrimitiveHasResult` will be raised by the server for each of these queued DiagComPrimitives. The results carried by the event is empty and contains the execution state `eCANCELLED_FROM_QUEUE`.

NOTE 1 The method `MCDDLogicalLink::reset()` does not have any influence on the selected ECU variant. That is, by calling the `reset()` method of a LogicalLink this LogicalLink is not reset to its ECU Base Variant.

The method `MCDDLogicalLink::close()` throws an exception in case of an error, and in that case does not cause a state change.

As a general rule, if any exception occurs during a state changing operation, the associated state shall not be changed. With respect to the LogicalLinkState, state changes can result from execution of the ControlPrimitives `MCDStartCommunication` and `MCDStopCommincation`. The corresponding state changes will only be performed if the MCDExecutionState of the corresponding ControlPrimitive is `eALL_POSITIVE`. In case the `MCDExecutionState` of a `MCDStartCommunication` or `MCDStopCommincation` is `eNEGATIVE`, `eALL_NEGATIVE`, or `eFAILED`, no state change of the associated LogicalLinkState will be performed.

Table 37 — Logical Link State Description

Logical Link State	**Event** **(MCDLogicalLinkEventHandler)**	**Description**
`eCREATED`	`onLinkCreated()`	Logical Link has been created, but is not ready for operation.
`eOFFLINE`	`onLinkOffline()`	The Logical Link has opened a hardware channel. No logical connection to the ECU exists, no communication has taken place.

Table 37 (*continued*)

Logical Link State	Event **(MCDLogicalLinkEventHandler)**	Description
`eONLINE`	`onLinkOnline()`	A logical connection to the ECU exists, but no DiagComPrimitive or Service is executed. Allows communication (if supported by the logical link's resources (bus, transport protocol, diagnostic protocol) without sending a tester present.
`eCOMMUNICATION`	`onLinkCommunication()`	A logical connection to the ECU exists, at least one DiagComPrimitive or Service is executed. Allows communication and sends a tester present on this logical link.

DiagComPrimitives can be executed in state `eONLINE` and `eCOMMUNICATION`. So the D-server will not blame the executing of an ordinary DiagComPrimitive if the logical link is in state `eONLINE` (as this is senseless until the link is not connected to the ECU). It may rather cause an error returned by the PDU API stating that this is not possible with the current protocol or within the current connection state.

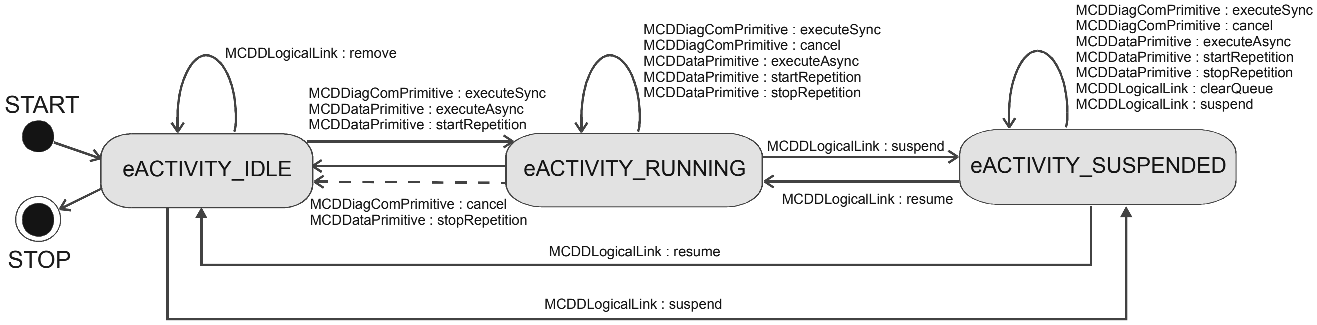

START
STOP
eACTIVITY_IDLE
eACTIVITY_RUNNING
eACTIVITY_SUSPENDED
MCDDLogicalLink : remove
MCDDiagComPrimitive : executeSync
MCDDataPrimitive : executeAsync
MCDDataPrimitive : startRepetition
MCDDiagComPrimitive : cancel
MCDDataPrimitive : stopRepetition
MCDDiagComPrimitive : executeSync
MCDDiagComPrimitive : cancel
MCDDataPrimitive : executeAsync
MCDDataPrimitive : startRepetition
MCDDataPrimitive : stopRepetition
MCDDiagComPrimitive : executeSync
MCDDiagComPrimitive : cancel
MCDDataPrimitive : executeAsync
MCDDataPrimitive : startRepetition
MCDDataPrimitive : stopRepetition
MCDDLogicalLink : clearQueue
MCDDLogicalLink : suspend
MCDDLogicalLink : suspend
MCDDLogicalLink : resume
MCDDLogicalLink : resume
MCDDLogicalLink : suspend

Figure 138 — State diagram MCDActivityState

Table 38 — Activity states

			MCDDLogicalLink			
System State	**LL State**	**Activity State**	**suspend**	**resume**	**clearQueue**	**getActivityState**
eLOGICAL_ CONNECTED	eCREATED	~~eIDLE~~				
		~~eRUNNING~~				
		~~eSUSPEND~~				
	eOFFLINE	~~eIDLE~~				
		~~eRUNNING~~				
		~~eSUSPEND~~				
	eONLINE & eCOMMUNICATION	eIDLE	X	X		X
		eRUNNING	X	X		X
		eSUSPEND	X	X	X	X

☒ Not Applicable

□ The exception ProgramViolationException eRT_NOT_ALLOWED_IN_THIS_OBJECT_STATE

The method `MCDDLogicalLink::suspend()`effects all services, also repeated services. Results coming in while the Activity State is `eACTIVITY_SUSPENDED` will be deleted and not be forwarded to the client application. Non-cyclic single diagnostic services currently running on the ECU are completed and the result is delivered to the client. The results of cyclic diagnostic services are not delivered to the application, but the cyclic service will be continued on the ECU. The result of currently running repeated services are not delivered to the application, repetition of the service will not continue until a call `MCDDLogicalLink::resume()`. The repetition timer of a repeated service is unaffected by suspension. If the repetition time is reached during suspension, no action is taken and the repetition timer is rescheduled. New services from the client and services already in repetition shall be executed intermittently by the D-server after a call to MCDDLogicalLink`::resume()`.

Repetition of services is performed by a scheduler in the D-server. Thus, `MCDDataPrimitive::stopRepetition()` does not require to send any information to the ECU. Rather, `MCDDataPrimitive::stopRepetition()` is a method that tells the D-server internal scheduler not to insert the respective service into the associated link's queue again to avoid another repetition cycle. In contrast, the method `MCDDataPrimitive::startRepetition()` inserts a service into the queue, which is then to be repeated by the D-server. For `MCDDataPrimitive::updateRepetitionParameters()` the same applies as for `MCDDataPrimitive::stopRepetition()`. As a result, `MCDDataPrimitive::stopRepetition()` and `MCDDataPrimitive::updateRepetitionParameters()` are not queued themselves, nor do these methods queue any other service.

The method `MCDDLogicalLink::clearQueue`() has no effect on running repeated and cyclic services. This method does not change the state `eACTIVITY_SUSPENDED`.

Calls of the methods `MCDDataPrimitive::executeAsync()` and `MCDDataPrimitive::startRepetition()` are allowed in state `eACTIVITY_SUSPENDED`, the execution will not be started unless `MCDDLogicalLink::resume()` has been called.

A call of `MCDDLogicalLink::resume()` changes the Logical Link's state from `eACTIVITY_SUSPENDED` to `eACTIVITY_RUNNING`. In case of an empty queue, the state `eACTIVITY_IDLE` is reached, accompanied by an event `MCDLogicalLinkEventHandler::onLinkActivityIdle()`.

Running cyclic services and repeated services are stopped by the D-server through `MCDDataPrimitive::cancel()` or `MCDDataPrimitive::stopRepetition()` calls, respectively. The state will be changed from `eACTIVITY_SUSPENDED` to `eACTIVITY_IDLE`.

Execution of VariantIdentification(AndSelection) services are also allowed in logical link state `eONLINE`. A successful execution of a variant identification and selection service does not change the link it has been executed on. However, after a variant selection has taken place on a link, the `MCDDbDLogicalLink::getDbLocation()` will return a different DbLocation object than before, which is associated with the newly identified ECU variant. In this case, the client application will have to retrieve all the Db objects below the DbLocation again using the new DbLocation object, to avoid errors and exceptions from using outdated/not applicable Db data.

Table 39 — Activity State description

Activity State	**Event** (MCDLogicalLinkEventHandler)	**Description**
eACTIVITY_IDLE	onLinkActivityIdle	A logical connection to the ECU exists but no DiagComPrimitive is executed (DiagComPrimitiveState: eIDLE for all DiagComPrimitives in the MCDDiagComPrimitives collection of the current Logical Link).

Table 39 (*continued*)

Activity State	Event (MCDLogicalLinkEventHandler)	Description
eACTIVITY_RUNNING	onLinkActivityRunning	A logical connection to the ECU exists and at least one DiagComPrimitive is executed (DiagComPrimitiveState: ePENDING or eREPEATING for at least one DiagComPrimitive in the MCDDiagComPrimitives collection of the current Logical Link). As soon as all DiagComPrimitives have returned to the DiagComPrimitiveState eIDLE, the activity state switches back to eACTIVITY_IDLE. This also applies if cancel() or stopRepetition() is called for the last DiagComPrimitive being executed repeatedly or cyclic. New DiagComPrimitives executed from the client and DataPrimitives already in repetition shall be executed intermittently by the D-server.
eACTIVITY_SUSPENDED	onLinkActivitySuspended	The execution of DiagComPrimitives in the Logical Link's activity queue has been suspended. Within this state, a DiagComPrimitive which is already in the activity queue can be cancelled or new DiagComPrimitives may be put in the activity queue by means of executeSync(), executeAsync(), or startRepetition(). All DiagComPrimitives are affected, also those executed repeatedly or cyclic. Results of repeated or cyclic DataComPrimitives will not be transferred to the application. Non-cyclic single DiagComPrimitives currently running on the ECU are completed and the result is delivered to the client. Cyclic DiagServices are continuing on the ECU and shall be cancelled separately. Repeated DataPrimitives get stopped until MCDDLogicalLink::resume() is called for a logical link. The repetition timer is not affected by the suspension. If the repetition time is reached during suspension, no action is taken and the repetition timer is re-scheduled.

Table 39 (*continued*)

Activity State	**Event** (MCDLogicalLinkEventHandler)	**Description**
		MCDDLogicalLink::clearQueue() deletes all single DiagComPrimitives from the activity queue. Afterwards, the queue size is empty. Note that repeated DataPrimitives and cyclic DiagServices are not affected. These kinds of DiagComPrimitives do not increase the queue size once they have been executed for the first time. Instead, they do live “beneath” the activity queue until they are cancelled or stopped. In case MCDDLogicalLink::resume() is called in activity state eACTIVITY_SUSPENDED, the activity state changes to - eACTIVITY_RUNNING if the activity queue contains at least one element or if there are still DataPrimitives executed repeatedly or cyclic - eACTIVITY_IDLE if the activity queue is empty and no DataPrimitives are executed repeatedly or cyclic any more.

If at least one DiagComPrimitive is executed on this link (executeSync, executeAsync, or startRepetition) MCDDLogicalLink changes state from eACTIVITY_IDLE to eACTIVITY_RUNNING.

With last DiagComPrimitive terminated on this link (executeSync, executeAsync, or startRepetition) MCDDLogicalLink changes state from eACTIVITY_RUNNING to eACTIVITY_IDLE.

With MCDDLogicalLink::suspend()MCDDLogicalLink changes state from eACTIVITY_RUNNING or eACTIVITY_IDLE to eACTIVITY_SUSPENDED.

With MCDDLogicalLink::resume()MCDDLogicalLink changes state from eACTIVITY_SUSPENDED to eACTIVITY_RUNNING (DiagComPrimitives active on logical link) or eACTIVITY_IDLE (no DiagComPrimitives active on logical link).

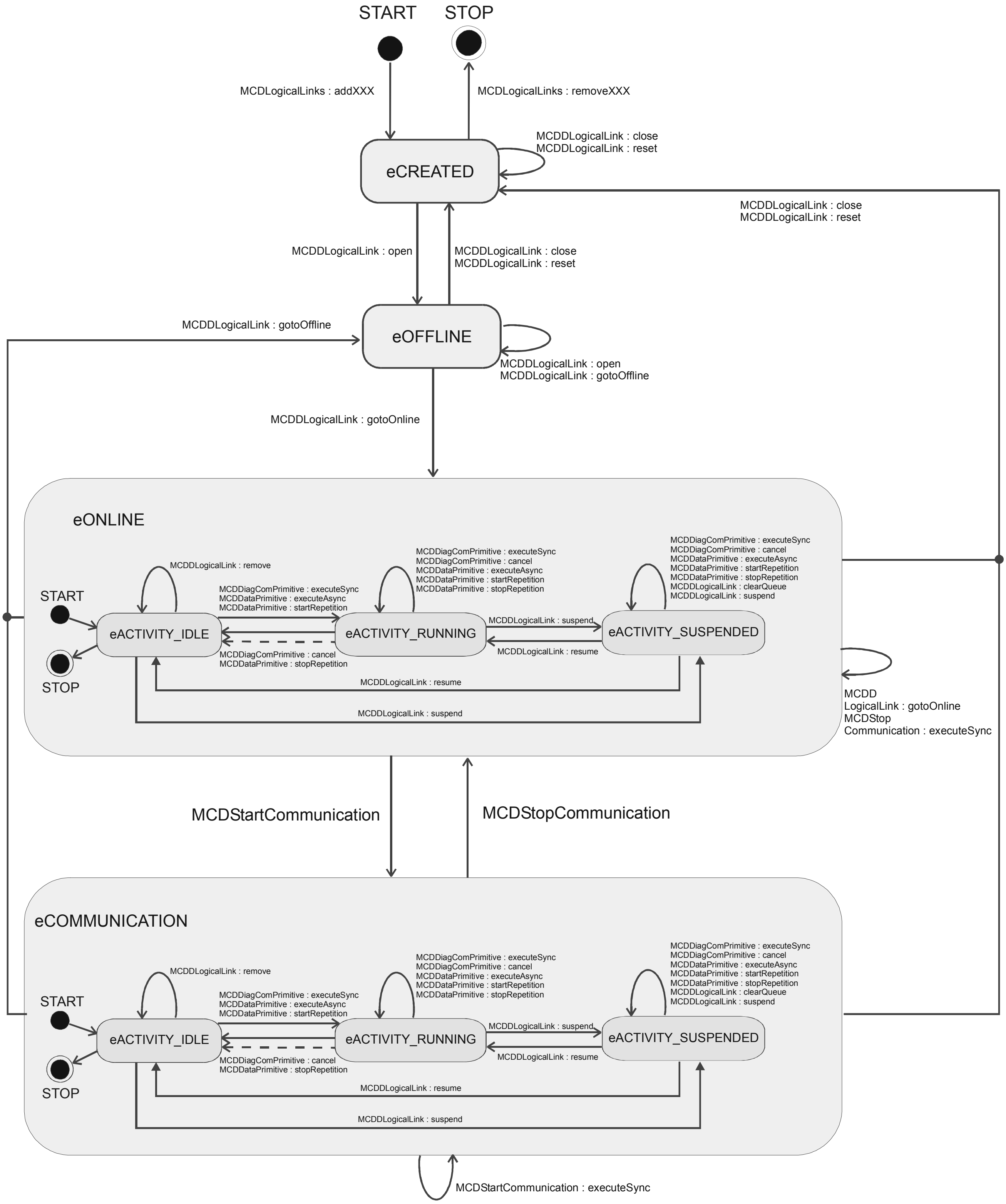

START
STOP
MCDLogicalLinks : addXXX
MCDLogicalLinks : removeXXX
MCDDLogicalLink : close
MCDDLogicalLink : reset
eCREATED
MCDDLogicalLink : close
MCDDLogicalLink : reset
MCDDLogicalLink : open
MCDDLogicalLink : close
MCDDLogicalLink : reset
MCDDLogicalLink : gotoOffline
eOFFLINE
MCDDLogicalLink : open
MCDDLogicalLink : gotoOffline
MCDDLogicalLink : gotoOnline
eONLINE
MCDDLogicalLink : remove
START
STOP
eACTIVITY_IDLE
MCDDiagComPrimitive : executeSync
MCDDataPrimitive : executeAsync
MCDDataPrimitive : startRepetition
MCDDiagComPrimitive : cancel
MCDDataPrimitive : stopRepetition
eACTIVITY_RUNNING
MCDDiagComPrimitive : executeSync
MCDDiagComPrimitive : cancel
MCDDataPrimitive : executeAsync
MCDDataPrimitive : startRepetition
MCDDataPrimitive : stopRepetition
MCDDLogicalLink : suspend
MCDDLogicalLink : resume
eACTIVITY_SUSPENDED
MCDDiagComPrimitive : executeSync
MCDDiagComPrimitive : cancel
MCDDataPrimitive : executeAsync
MCDDataPrimitive : startRepetition
MCDDataPrimitive : stopRepetition
MCDDLogicalLink : clearQueue
MCDDLogicalLink : suspend
MCDDLogicalLink : resume
MCDDLogicalLink : suspend
MCDD
LogicalLink : gotoOnline
MCDStop
Communication : executeSync
MCDStartCommunication
MCDStopCommunication
eCOMMUNICATION
MCDDLogicalLink : remove
START
STOP
eACTIVITY_IDLE
MCDDiagComPrimitive : executeSync
MCDDataPrimitive : executeAsync
MCDDataPrimitive : startRepetition
MCDDiagComPrimitive : cancel
MCDDataPrimitive : stopRepetition
eACTIVITY_RUNNING
MCDDiagComPrimitive : executeSync
MCDDiagComPrimitive : cancel
MCDDataPrimitive : executeAsync
MCDDataPrimitive : startRepetition
MCDDataPrimitive : stopRepetition
MCDDLogicalLink : suspend
MCDDLogicalLink : resume
eACTIVITY_SUSPENDED
MCDDiagComPrimitive : executeSync
MCDDiagComPrimitive : cancel
MCDDataPrimitive : executeAsync
MCDDataPrimitive : startRepetition
MCDDataPrimitive : stopRepetition
MCDDLogicalLink : clearQueue
MCDDLogicalLink : suspend
MCDDLogicalLink : resume
MCDDLogicalLink : suspend
MCDStartCommunication : executeSync

Figure 139 — Logical Link state diagram in function block D including MCDActivityState

Table 40 — Relations between states and actions

	Logical Link	**ECU**	**Database**	**Activity**
eCREATED	Logical Link Object created	Not connected	Location accessed, no DB changes	---
eOFFLINE	Channel allocated	Connected communication	Location accessed, no DB changes	---
eONLINE eACTIVITY_IDLE	Communicating	Connected communication	VI/VIS and DB changes possible	No service running
eCOMMUNICATION eACTIVITY_IDLE	Communicating	Communicating	VI/VIS and DB changes possible	No service running
eACTIVITY_ RUNNING	Communicating	---	No DB changes	Services running
eACTIVITY_ SUSPENDED	Communicating	---	No DB changes	Suspended

NOTE 2 The location will not be reset to the one originally defined for the link within the state change from eCOMMUNICATION to eONLINE or eCREATED.

In a multi-client scenario, Logical Links can be locked by a client to prevent other clients from making changes to the properties of the link. In the default case, the Logical Link is unlocked for shared use. Therefore, after creation the logical link's MCDLockState is eLS_UNLOCKED. A link can not be locked or unlocked when its activity state is eACTIVITY_RUNNING or eACTIVITY_SUSPENDED. Only the locking object may unlock the link. If unlocked, it can be locked and unlocked again by any client.

For MCDDLogicalLink::setQueueSize() the logical link state shall be in state eOFFLINE or a further state. In case the new queue size is smaller than the current queue size, the reduction becomes effective as soon as the actual activity queue is below the new threshold. The size is given as the number of DiagComPrimitives.

For MCDDLogicalLink::getQueueSize() the logical link shall not be locked by another object.

For MCDLogicalLink::setEventHandler(), MCDLogicalLink::releaseEventHandler() and MCDDiagComPrimitives::remove*(), the associated logical link's activity queue shall not be running.

At the first creation of a Logical Link, within the D-server an object is created and the reference to this object is returned to the client. If this or another client creates a further Logical Link with the same database template, only a reference to the already created Logical Link within the D-server is returned. That means for each Logical Link, there only exists one runtime instance.

Because of this, all DiagComPrimitives and Services which are executed on this Logical Link are put into the same activity queue and are executed there. Thus, no overlapping and undesired parallel executions of DiagComPrimitives may occur. As many Diagnostic Services as desired may be executed via the Activity Queue per Logical Link. This provides for the possibility to execute Diagnostic Services in parallel.

In case of removal of a Logical Link, it is distinguished between D-server and MC-server:

— In case of a D-server, MCDProject::removeLogicalLink(..) only removes the proxy object of the LogicalLink from the server. The server object of the LogicalLink is removed by the server if and only if the last client removes its proxy to this LogicalLink.

— In an MC-server, the last client leaving the MC-server needs to clean-up the system (remove all runtime objects).

The MCD-server shall make sure that the results of the DiagComPrimitive- or Service- executions get to the right reference (client) of the Logical Link.

As long as there is no DiagComPrimitive or Service within the Activity Queue, the Activity Queue is within state `eACTIVITY_IDLE`. If a DiagComPrimitive or Service is put into the Activity Queue for execution, the Activity Queue takes the state `eACTIVITY_RUNNING`.

9.14.4 Logical Link examples

Today, a single ECU Base Variant can represent more than one physical ECU. There may be several DB Logical Links pointing to the same DBLocation of an ECU Base Variant and as these DB Logical Links may have arbitrary values for the protocol parameters. There needs to be at least one DB Logical Link per physical ECU carrying the address information to access the corresponding physical ECU. However, there can be more than one DB Logical Link for the same physical ECU. As a result, these DB Logical Links carry the same address information and may thus only be opened alternatively.

The Logical Link Table consists of six columns. The first column is an index for the row. The following three columns (Shortname of Logical Link, AccessKey, Shortname of PhysicalVehicleLink) are part of a Logical Link. The column “Unique Handle from UNIQUE_ID protocol parameters” identifies address information calculated from the values of all protocol parameters of parameter class eUNIQUE_ID contained in the protocol stack attached to a Logical Link (protocol parameters with PARAM-CLASS ‘UNIQUE_ID’ in ODX or MCDProtocolParameterClass ‘eUNIQUE_ID). The calculation algorithm for the unique handles is D-server vendor specific. The same applies to the data type of the unique handle as these values will only be used D-server internally (Example for a unique handle which is human readable: [ComParamName1]<value>.[ComParamName2]<value>). The last two columns (Logical Link ShortName of Gateway and GetGatewayMode) are for information and initialization purposes.

— The unique short name of a Logical Link is used by the application/job to access the ECU.

— The D-server uses the unique short name of a Logical Link to find the related row in the Logical Link Table. From this row the D-server reads the AccessKey, the Physical Vehicle Link and the Unique Handle from UNIQUE_ID protocol parameters needed to physically address the ECU.

— Only ECU s which are referred by Logical Links in the Logical Link Table can be accessed by applications/jobs.

— The Logical Link Table is vehicle dependent.

— In case of physical communication the Logical Link a response needs to be assigned to can be determined uniquely though there may be more that one Logical Link in the Logical Link Table having the same value for its UNIQUE_ID communication parameters. The D-server knows which physical resources have been attached to a runtime Logical Link. This information shall be used in addition to the UNIQUE_ID information.

EXAMPLE 1:

in database shall exist:	- 1 reference of the ECU (also referred to as ECM hereafter)
	- a test with the instances of 4 ECMs and their different Logical Links

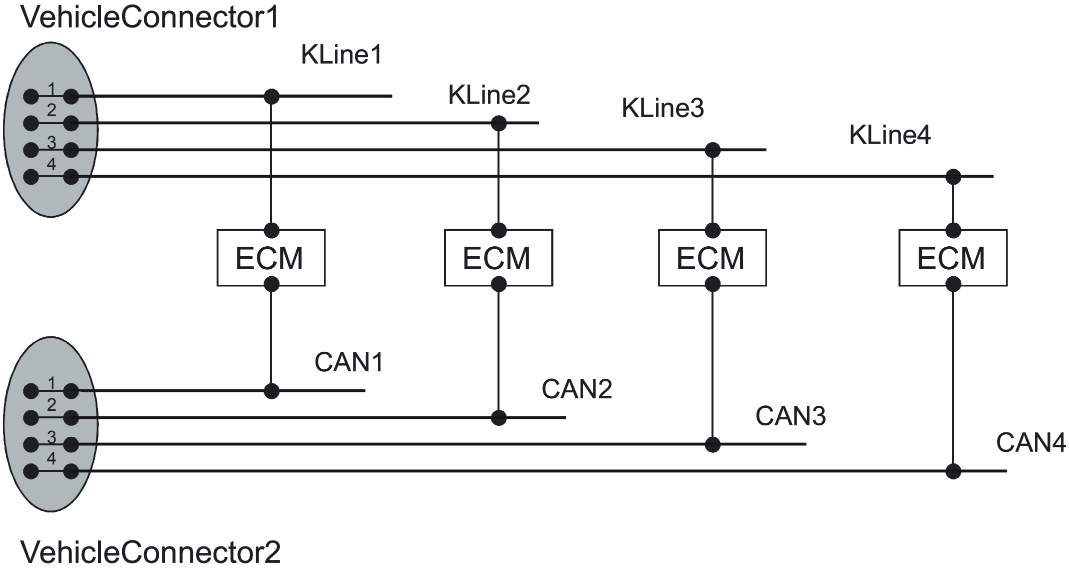


Figure 140 — Four ECM of same type in one test in one project

Table 41 — Logical Link Table for example I

Short Name	**AccessKey**	**Short Name Physical Vehicle Link**	**Unique Handle from UNIQUE_ID protocol parameters**	**Logical Link Short Name of Gateway**	**Get Gateway Mode**
ECM-KLine1	[Protocol]KWP2000.[ECUBaseVariant]ECM	KLINE1	[CP_FuncRespFormatPriorityType]0x48. [CP_FuncRespTargetAddr]0x6B. [CP_EcuRespSourceAddress]0x10		
ECM-KLine2	[Protocol]KWP2000.[ECUBaseVariant]ECM	KLINE2	[CP_FuncRespFormatPriorityType]0x49. [CP_FuncRespTargetAddr]0x6C. [CP_EcuRespSourceAddress]0x11		
ECM-KLine3	[Protocol]KWP2000.[ECUBaseVariant]ECM	KLINE2	[CP_FuncRespFormatPriorityType]0x4A. [CP_FuncRespTargetAddr]0x6D. [CP_EcuRespSourceAddress]0x12		
ECM-KLine4	[Protocol]KWP2000.[ECUBaseVariant]ECM	KLINE4	[CP_FuncRespFormatPriorityType]0x4B. [CP_FuncRespTargetAddr]0x6E. [CP_EcuRespSourceAddress]0x13		
ECM-CAN1	[Protocol]DiagOnCan.[ECU BaseVariant]ECM	CAN1	[CP_CanPhysReqExtAddr]0. [CP_CanPhysReqFormat]0x5. [CP_CanPhysReqId]0x341 [CP_CanRespUSDTExtAddr]0. [CP_CanRespUSDTFormat]0x5. [CP_CanPhysUSDTId]0x18DAF1C8 [CP_CanRespUUDTExtAddr]0. [CP_CanRespUUDTFormat]0. [CP_CanPhysUUDTId]0x141		
ECM-CAN2	[Protocol]DiagOnCan.[ECU BaseVariant]ECM	CAN2	[CP_CanPhysReqExtAddr]1. [CP_CanPhysReqFormat]0x6. [CP_CanPhysReqId]0x342 [CP_CanRespUSDTExtAddr]1. [CP_CanRespUSDTFormat]0x6. [CP_CanPhysUSDTId]0x18DAF1C9 [CP_CanRespUUDTExtAddr]1. [CP_CanRespUUDTFormat]1. [CP_CanPhysUUDTId]0x142		

Table 41 (*continued*)

Short Name	AccessKey	Short Name Physical Vehicle Link	Unique Handle from UNIQUE_ID protocol parameters	Logical Link Short Name of Gateway	Get Gateway Mode
ECM-CAN3	[Protocol]DiagOnCan.[ECU BaseVariant]ECM	CAN3	[CP_CanPhysReqExtAddr]2. [CP_CanPhysReqFormat]0x7. [CP_CanPhysReqId]0x343 [CP_CanRespUSDTExtAddr]2. [CP_CanRespUSDTFormat]0x7. [CP_CanPhysUSDTId]0x18DAF1CA [CP_CanRespUUDTExtAddr]2. [CP_CanRespUUDTFormat]2. [CP_CanPhysUUDTId]0x143		
ECM-CAN4	[Protocol]DiagOnCan.[ECU BaseVariant]ECM	CAN4	[CP_CanPhysReqExtAddr]3. [CP_CanPhysReqFormat]0x8. [CP_CanPhysReqId]0x344 [CP_CanRespUSDTExtAddr]3. [CP_CanRespUSDTFormat]0x8. [CP_CanPhysUSDTId]0x18DAF1CB [CP_CanRespUUDTExtAddr]3. [CP_CanRespUUDTFormat]3. [CP_CanPhysUUDTId]0x144		

The Vehicle Connector Information Table has five columns. The first column is a row index. The following two columns (Physical Vehicle Link and Vehicle Connector Information) describe the one to many relation (1-n) between the Physical Vehicle Links and the Pins of the VehicleConnector.

- Because one Physical Vehicle Link could be accessed by two different connectors (see row 1 and 2) the D-server could find multiple rows in this table.

- The entries for connector and pins in the column Vehicle Connector Information are unique, because two Physical Vehicle Links cannot be connected.

- One Physical Vehicle Link, e.g. CAN, could use more than one pin at a connector.

Table 42 — Vehicle Connector Information Table for example I

No	ShortName of Physical Vehicle Link	VehicleConnectorInformation	Type	LongName
1	KLine1	Connector1_Pin1	KLINE	Diagnostic Line
2	KLine2	Connector1_Pin2	KLINE	Diagnostic Line
3	KLine2	Connector1_Pin3	KLINE	Diagnostic Line
4	KLine4	Connector1_Pin4	KLINE	Diagnostic Line
5	CAN1	Connector2_Pin1	CAN	Body CAN High Speed
6	CAN2	Connector2_Pin2	CAN	Body CAN High Speed
7	CAN3	Connector2_Pin3	CAN	Body CAN High Speed
8	CAN4	Connector2_Pin4	CAN	Body CAN High Speed

EXAMPLE 2:

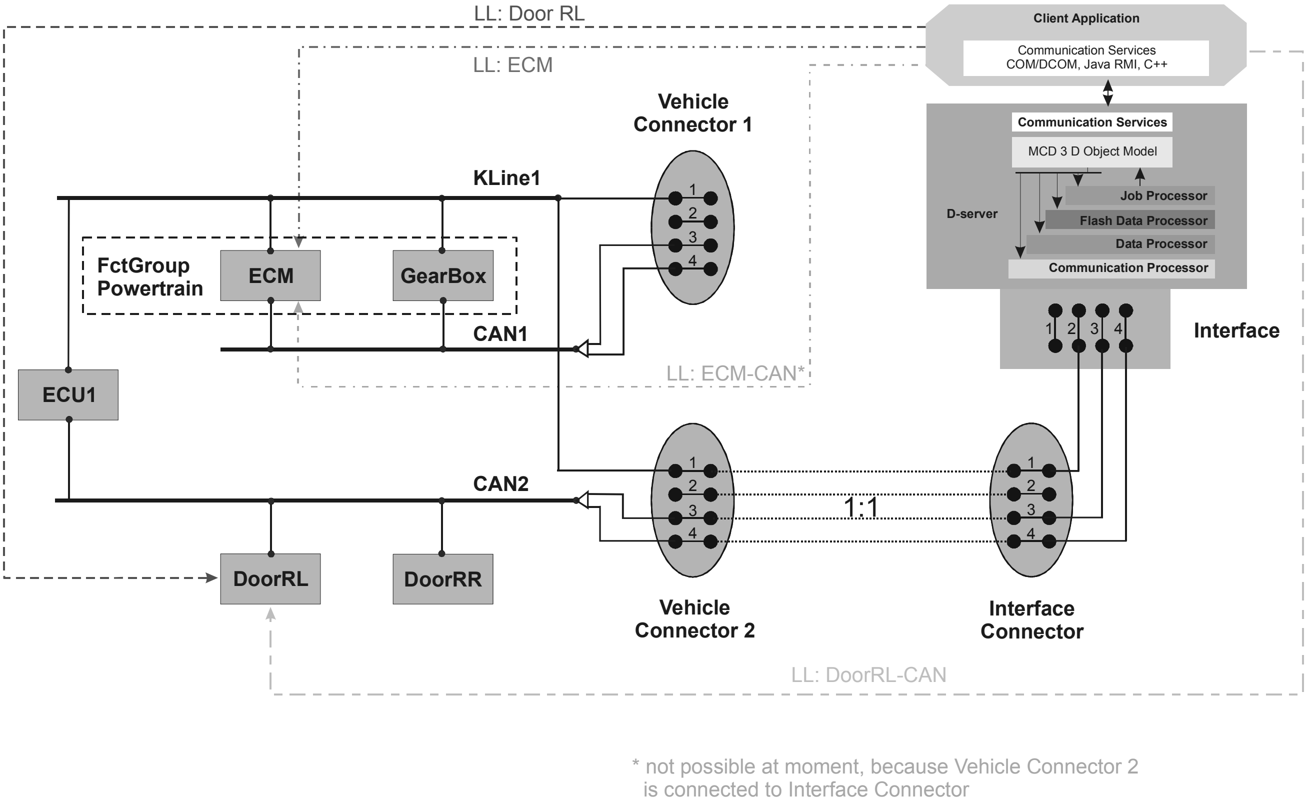


Figure 141 — Example Logical Link with functional group

Table 43 — Logical Link Table for example II

No.	Short Name of Logical Link	AccessKey	Short Name Physical Vehicle Link	Unique Handle from UNIQUE_ID protocol parameters	Logical Link Short Name of Gate way	Get Gate-way Mode
1	ECM	[Protocol]KWP2000. [ECUBaseVariant] ECM	KLINE1	[CP_FuncRespFormatPriorityType]0x48. [CP_FuncRespTargetAddr]0x6B. [CP_EcuRespSourceAddress]0x10		
2	Powertrain	[Protocol] KWP2000 [FunctionalGroup] Powertrain	KLINE1	[CP_FuncRespFormatPriorityType]0x49. [CP_FuncRespTargetAddr]0x6C. [CP_EcuRespSourceAddress]0x11		
3	GearBox	[Protocol] KWP2000. [ECUBaseVariant] GearBox	KLINE1	[CP_FuncRespFormatPriorityType]0x4A. [CP_FuncRespTargetAddr]0x6D. [CP_EcuRespSourceAddress]0x12		
4	ECU1	[Protocol] KWP2000. [ECUBaseVariant] ECU1	KLINE1	[CP_FuncRespFormatPriorityType]0x4B. [CP_FuncRespTargetAddr]0x6E. [CP_EcuRespSourceAddress]0x13		visible
5	DoorRR	[Protocol]KWP2000. [ECUBaseVariant] DoorRR	KLINE1	[CP_FuncRespFormatPriorityType]0x4C. [CP_FuncRespTargetAddr]0x6F. [CP_EcuRespSourceAddress]0x14	ECU1	
6	DoorRL	[Protocol] KWP2000. [ECUBaseVariant] DoorRL	KLINE1	[CP_FuncRespFormatPriorityType]0x4D. [CP_FuncRespTargetAddr]0x7A. [CP_EcuRespSourceAddress]0x15	ECU1	
7	ECM-CAN	[Protocol] DiagOnCAN. [ECUBaseVariant] ECM	CAN1	[CP_CanPhysReqExtAddr]0. [CP_CanPhysReqFormat]0x5. [CP_CanPhysReqId]0x341 [CP_CanRespUSDTExtAddr]0. [CP_CanRespUSDTFormat]0x5. [CP_CanPhysUSDTId]0x18DAF1C8 [CP_CanRespUUDTExtAddr]0. [CP_CanRespUUDTFormat]0. [CP_CanPhysUUDTId]0x141		
8	GearBox-CAN	[Protocol] DiagOnCAN. [ECUBaseVariant] GearBox	CAN1	[CP_CanPhysReqExtAddr]1. [CP_CanPhysReqFormat]0x6. [CP_CanPhysReqId]0x342 [CP_CanRespUSDTExtAddr]1. [CP_CanRespUSDTFormat]0x6. [CP_CanPhysUSDTId]0x18DAF1C9 [CP_CanRespUUDTExtAddr]1. [CP_CanRespUUDTFormat]1. [CP_CanPhysUUDTId]0x142		
9	ECU1-CAN	[Protocol] DiagOnCAN. [ECUBaseVariant] ECU1	CAN2	[CP_CanPhysReqExtAddr]2. [CP_CanPhysReqFormat]0x7. [CP_CanPhysReqId]0x343 [CP_CanRespUSDTExtAddr]2. [CP_CanRespUSDTFormat]0x7. [CP_CanPhysUSDTId]0x18DAF1CA [CP_CanRespUUDTExtAddr]2. [CP_CanRespUUDTFormat]2. [CP_CanPhysUUDTId]0x143		

Table 43 (*continued*)

No.	Short Name of Logical Link	AccessKey	Short Name Physical Vehicle Link	Unique Handle from UNIQUE_ID protocol parameters	Logical Link Short Name of Gate way	Get Gate-way Mode
10	DoorRR-CAN	[Protocol] DiagOnCAN. [ECUBaseVariant] DoorRR	CAN2	[CP_CanPhysReqExtAddr]3. [CP_CanPhysReqFormat]0x8. [CP_CanPhysReqId]0x344 [CP_CanRespUSDTExtAddr]3. [CP_CanRespUSDTFormat]0x8. [CP_CanPhysUSDTId]0x18DAF1CB [CP_CanRespUUDTExtAddr]3. [CP_CanRespUUDTFormat]3. [CP_CanPhysUUDTId]0x144		
11	DoorRL-CAN	[Protocol] DiagOnCAN. [ECUBaseVariant] DoorRL	CAN2	[CP_CanPhysReqExtAddr]4. [CP_CanPhysReqFormat]0x9. [CP_CanPhysReqId]0x345 [CP_CanRespUSDTExtAddr]4. [CP_CanRespUSDTFormat]0x9. [CP_CanPhysUSDTId]0x18DAF1CC [CP_CanRespUUDTExtAddr]4. [CP_CanRespUUDTFormat]4. [CP_CanPhysUUDTId]0x145		
12	CAN	[Protocol] DiagOnCAN	CAN1	[CP_CanPhysReqExtAddr]5. [CP_CanPhysReqFormat]0xA. [CP_CanPhysReqId]0x346 [CP_CanRespUSDTExtAddr]5. [CP_CanRespUSDTFormat]0xA. [CP_CanPhysUSDTId]0x18DAF1CD [CP_CanRespUUDTExtAddr]5. [CP_CanRespUUDTFormat]5. [CP_CanPhysUUDTId]0x146		
13	KWP	[Protocol] KWP2000	KLINE1	[CP_FuncRespFormatPriorityType]0x4E. [CP_FuncRespTargetAddr]0x7B. [CP_EcuRespSourceAddress]0x16		

9.14.5 Gateway handling

Gateways are ECUs that are located between the tester and a tested ECU in the bus topology. An example would be an ECU that is connected both to a diagnostic bus segment and a body bus segment, acting as a router for any messages that have to be passed from one bus segment to the other. An ECU that is accessed via a gateway is called a member ECU.

It is distinguished between visible (or non-transparent) and transparent gateways. A transparent gateway does not have to be stimulated or otherwise activated to route messages to a member ECU; if it receives a message on a bus segment which is not addressed to itself (the gateway), it automatically forwards the messages to the appropriate bus segment. So, it becomes truly transparent to the communicating instances whose messages are being routed. The member ECUs appear to reside on the same bus segment as the gateway.

A visible gateway shall be activated before it can route messages to member ECUs. For example, a K-Line gateway shall be woken up before it can accept and forward any messages.

In case of transparent gateways, logical links to ECUs behind the gateway are used in the same manner as logical links to ECUs without gateways, so that in this case the gateway is not visible to the application. Please note: Due to response timing issues (for example, if two separate bus timeout periods add up in case of communicating with an ECU behind a transparent gateway), it might be possible that a link's communication parameters will have to be adjusted accordingly.

When dealing with visible gateways (or when the vehicle topology is not known), before communicating with an ECU, it shall be checked if the ECU is accessed through a gateway. This can be done by the method `MCDDbDLogicalLink::isAccessedViaGateway()`. In case the ECU is accessed through a visible gateway, all referenced gateways of the ECU will be listed at the ECUs logical link database object (using the `MCDDbDLogicalLink::getDbLogicalLinksOfGateways()` method). If a logical link to a member ECU requires a logical link to a non-transparent gateway to be opened first, the D-server needs to determine whether all required gateway logical links are present and in state `eCOMMUNICATION`. The client application is responsible for opening all required gateway logical links prior to opening a logical link to a member ECU. If a client application tries to call `MCDDLogicalLink::gotoOnline()` for a logical link to a member ECU of a non-transparent gateway, the D-server will throw an exception in case any of the required gateway links are not available or in the wrong state (see Figure D.1 — GatewayHandling).

Generally, all the rules defined for multi-client access to objects also apply in the case of logical links to gateway ECUs. That means that the application that creates a link also owns it, and is free to either choose an appropriate cooperation level or to lock the gateway link to prevent other client applications from changing the link's parameters or even closing the gateway logical link. It shall be noted that in case a link to a gateway ECU is not protected accordingly, any client application can disrupt communication to a member ECU behind the gateway by manipulating the gateway logical link. Also, it is up to the client application to manage gateway logical links accordingly. As a logical link is a client-owned object, the respective client application shall take care to close and remove all logical links it has created before, including links to gateway ECUs. The D-server will not synchronize or in any way transfer state information (logical link state, lock state, etc.) between member logical links and gateway logical links.

As the difference between a transparent and a non-transparent (visible) gateway cannot be obtained from the ODX data directly, the following definitions describe how the gateway mode of a logical link is calculated:

If a logical link is marked to be a gateway logical link in ODX and if this logical link is not referenced from any other logical link in the same VIT, this logical link's gateway mode is `eTRANSPARENT_GATEWAY`.

If a logical link is marked to be a gateway logical link in ODX and if this logical link is referenced from at least one other logical link in the same VIT, this logical link's gateway mode is `eVISIBLE_GATEWAY`.

If a logical link is not marked to be a gateway logical link in ODX, this logical link's gateway mode is `eNO_GATEWAY`.

NOTE It might be possible that an ODX author has not marked a logical link to physically transparent gateway as a gateway logical link. As a result, this logical link would be reported with a gateway mode of `eNO_GATEWAY`.

9.14.6 Examples and Relations between Logical Links, Locations and relevant AccessKeys

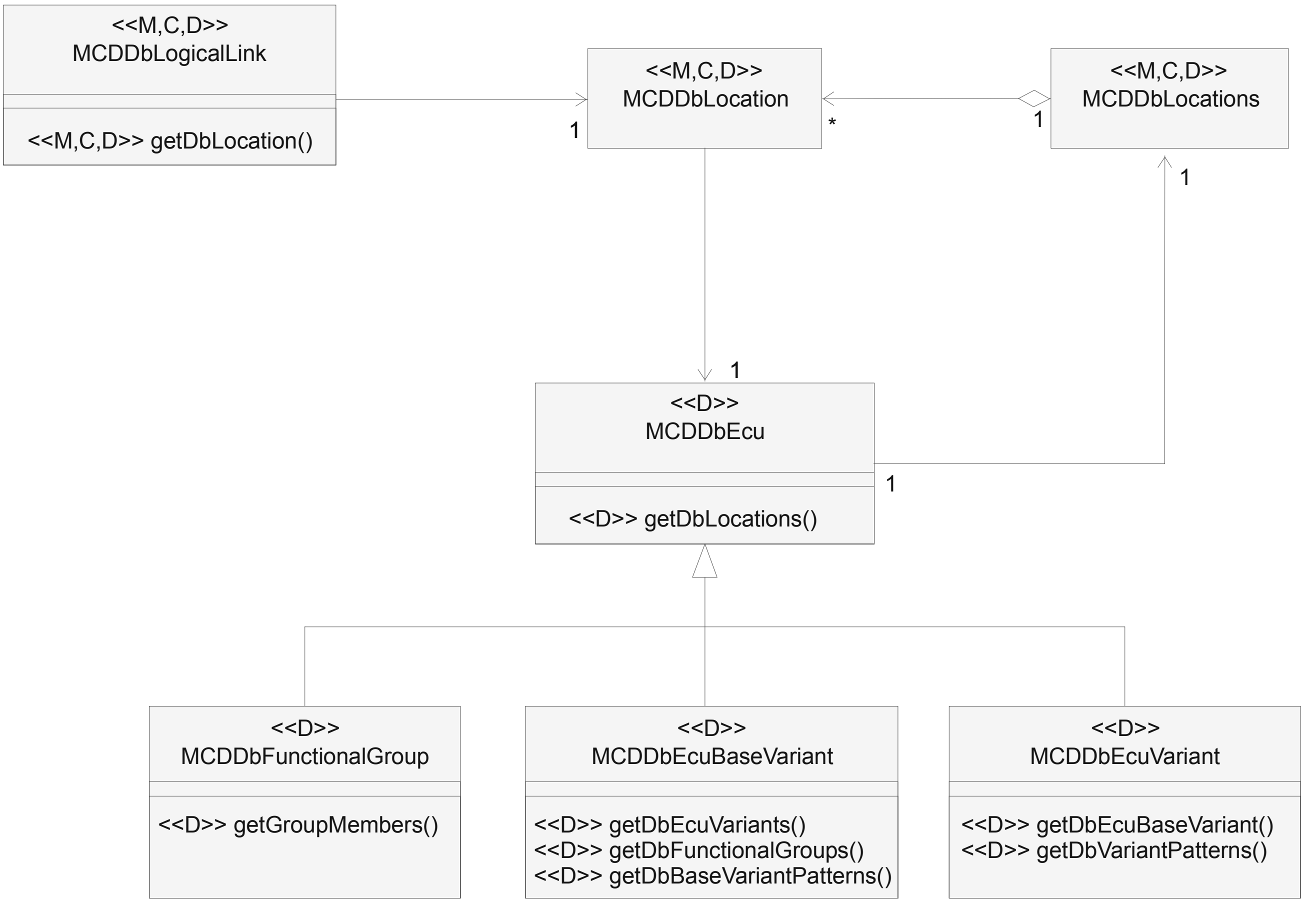
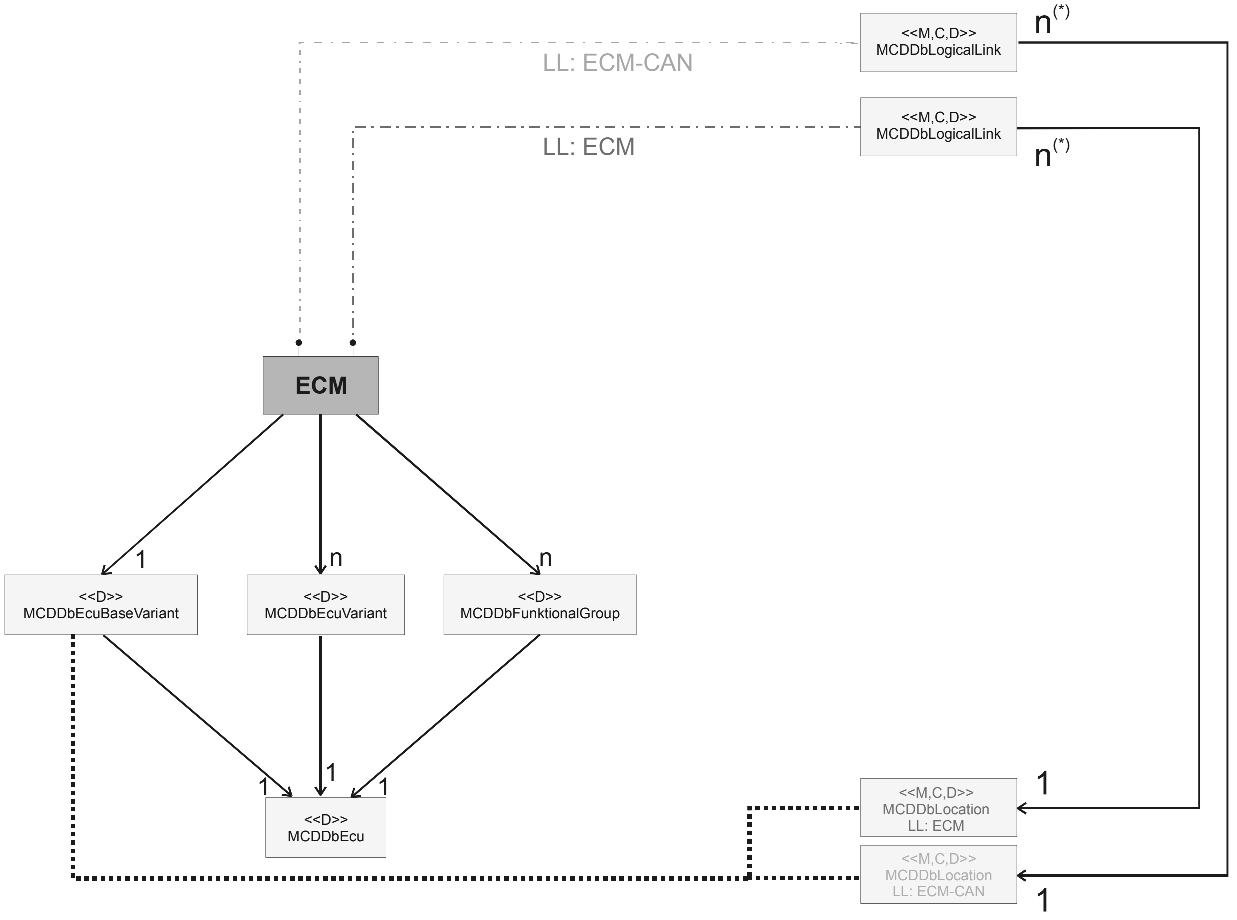


Figure 142 — DbLocation and DbECU



(*) Indeed, there can be multiple MCDDbLogicalLink's pointing to the same MCDDbLocation in a D-server.

Figure 143 — LogicalLinks and Locations in case of different protocols

`MCDDbEcu::getDbLocations()` only delivers locations that are contained in the database.

If there are two physical connections (ECM: KLINE1 and ECM-CAN: CAN1), there shall be two different LogicalLinks, which are also reflected in two different locations. That is why, if the method MCDDbEcu`::getDbLocations()` is called, always all relevant Locations are returned (two in our example). The behaviour does not depend on whether the method was called from a `MCDDbEcuBaseVariant` or a `MCDDbEcuVariant`.

If `MCDDbEcuBaseVariant::getDbECUVariants()` is called, all inherited Variants of the BaseVariant (in this example _1, _2 and _3) are returned.

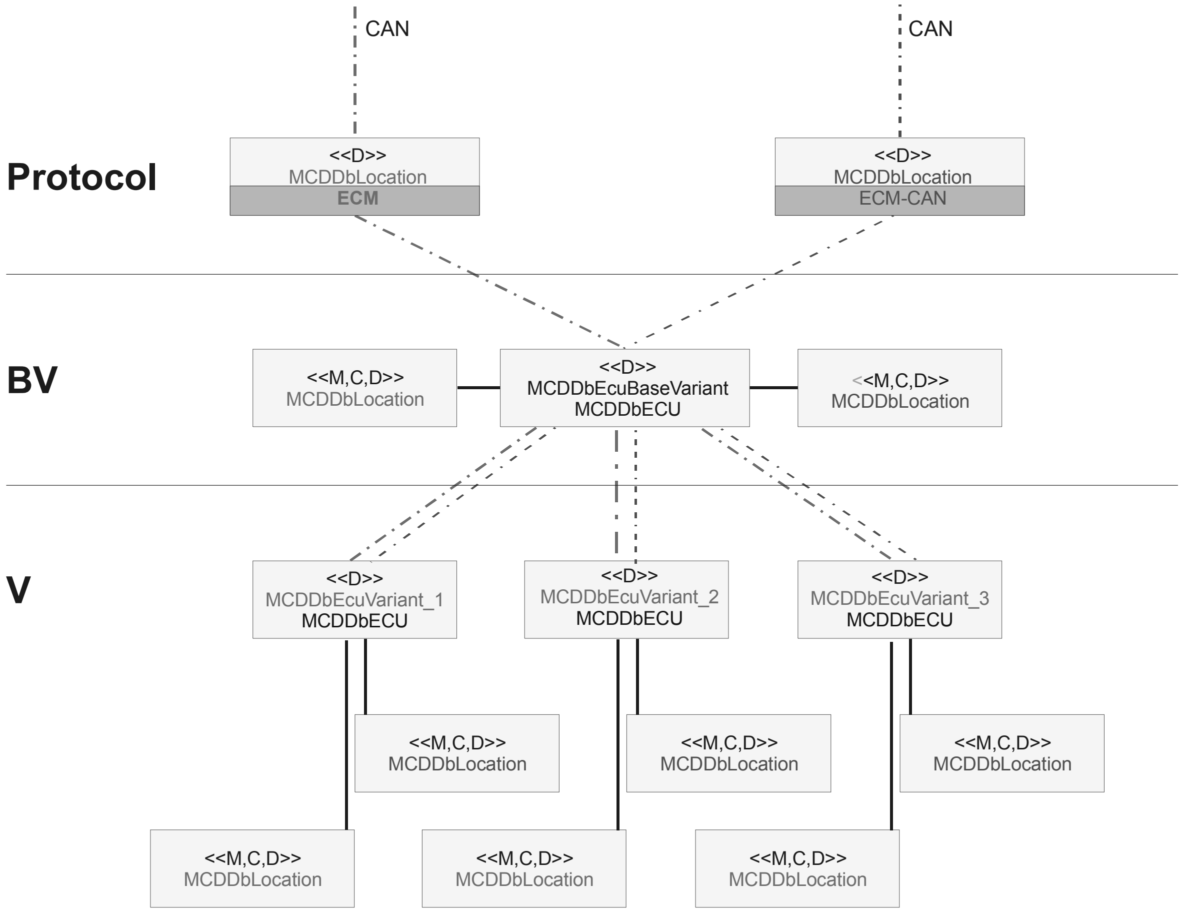


Figure 144 — One base variant for all protocols

If `MCDDbEcuBaseVariant::getDbECUVariants()` is called, it delivers all variants (in this case _4, _5 and _6) of this BaseVariant.

Resulting AccessKeys:

— [Protocol]ECM.[ECUBaseVariant]BV (PR1, BV)

— [Protocol]ECM (PR1)

— [Protocol]ECM-CAN (PR2)

— [Protocol]ECM-CAN.[ECUBaseVariant]BV (PR2, BV)

— [Protocol]ECM.[ECUBaseVariant]BV.[ECUVariant]V1 (PR1, BV, V1)

— [Protocol]ECM.[ECUBaseVariant]BV.[ECUVariant]V2 (PR1, BV, V2)

— [Protocol]ECM.[ECUBaseVariant]BV.[ECUVariant]V3 (PR1, BV, V3)

— [Protocol]ECM-CAN.[ECUBaseVariant]BV.[ECUVariant]V1 (PR2, BV, V1)

— [Protocol]ECM-CAN.[ECUBaseVariant]BV.[ECUVariant]V2 (PR2, BV, V2)

— [Protocol]ECM-CAN.[ECUBaseVariant]BV.[ECUVariant]V3 (PR2, BV, V3)

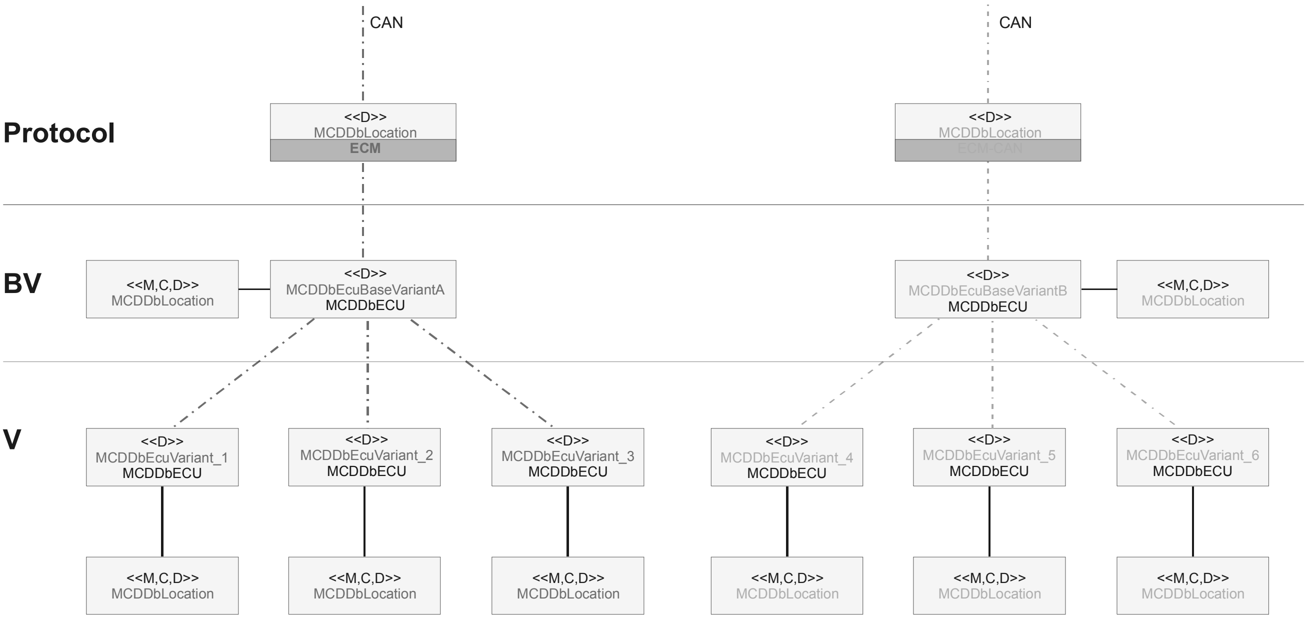


Figure 145 — For each protocol an own base variant

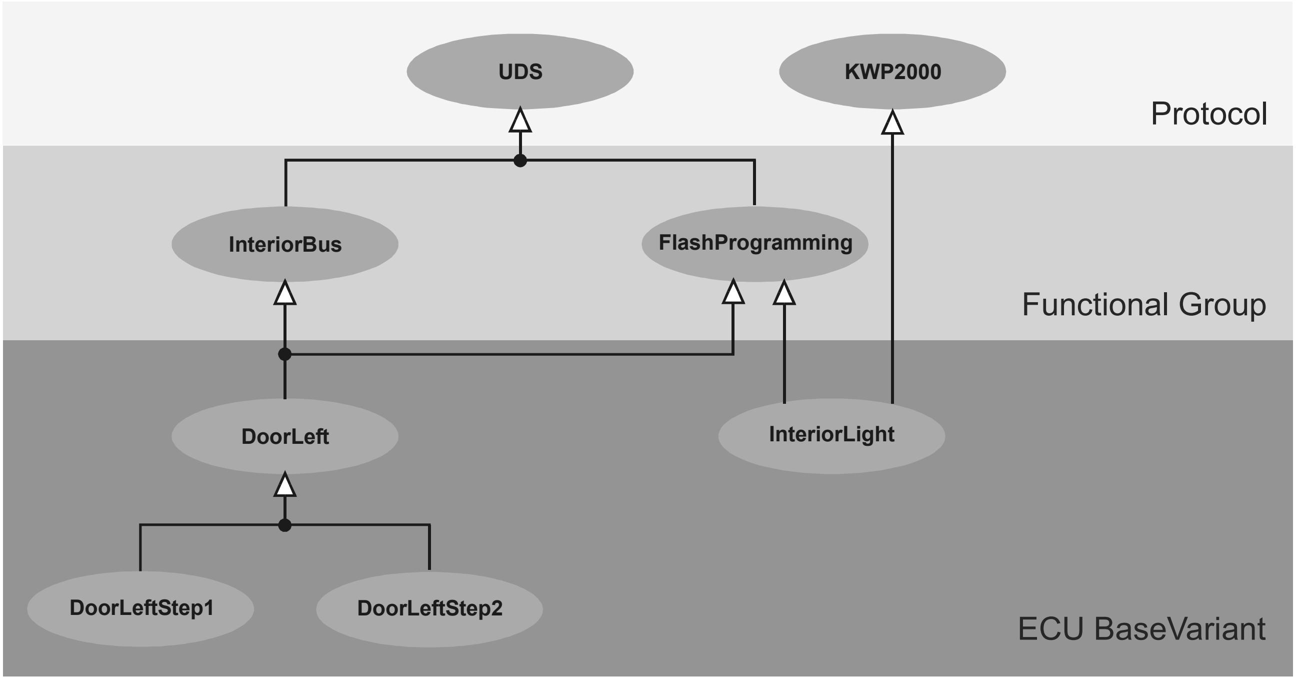


Figure 146 — AccessKey Example 1

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

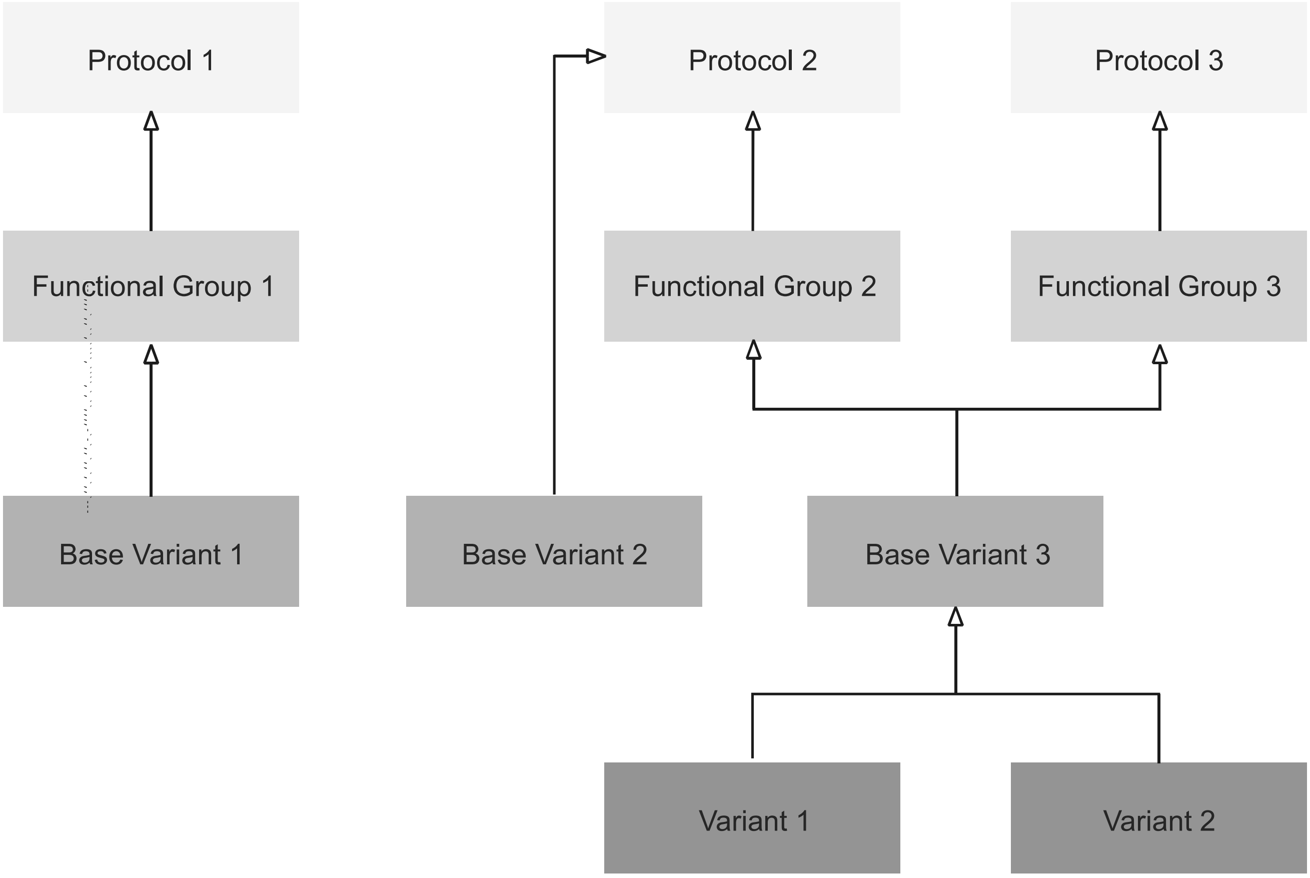


Figure 147 — AccessKey Example 2, Existing Locations

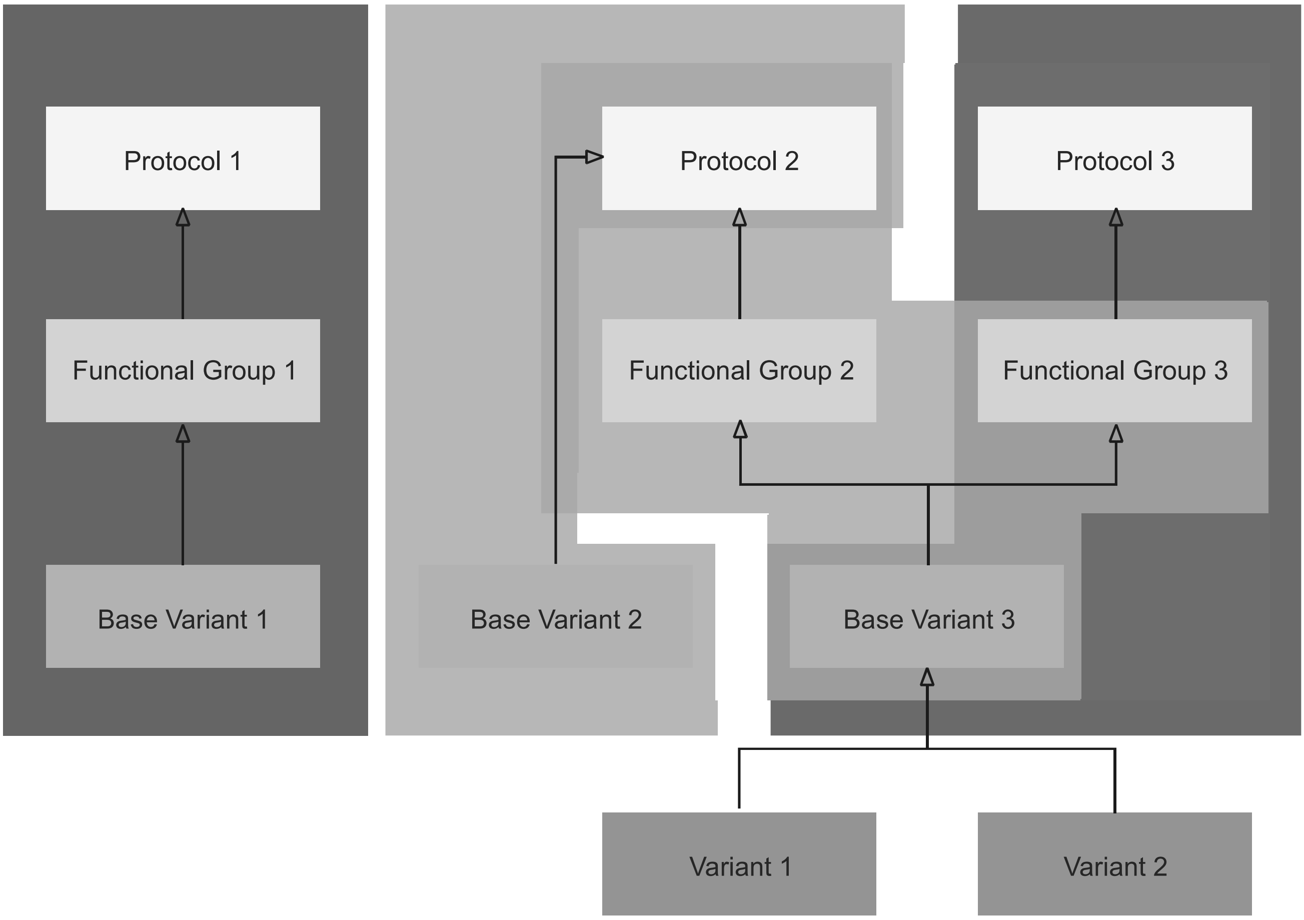


Figure 148 — AccessKey Example 2, Base Variants Locations

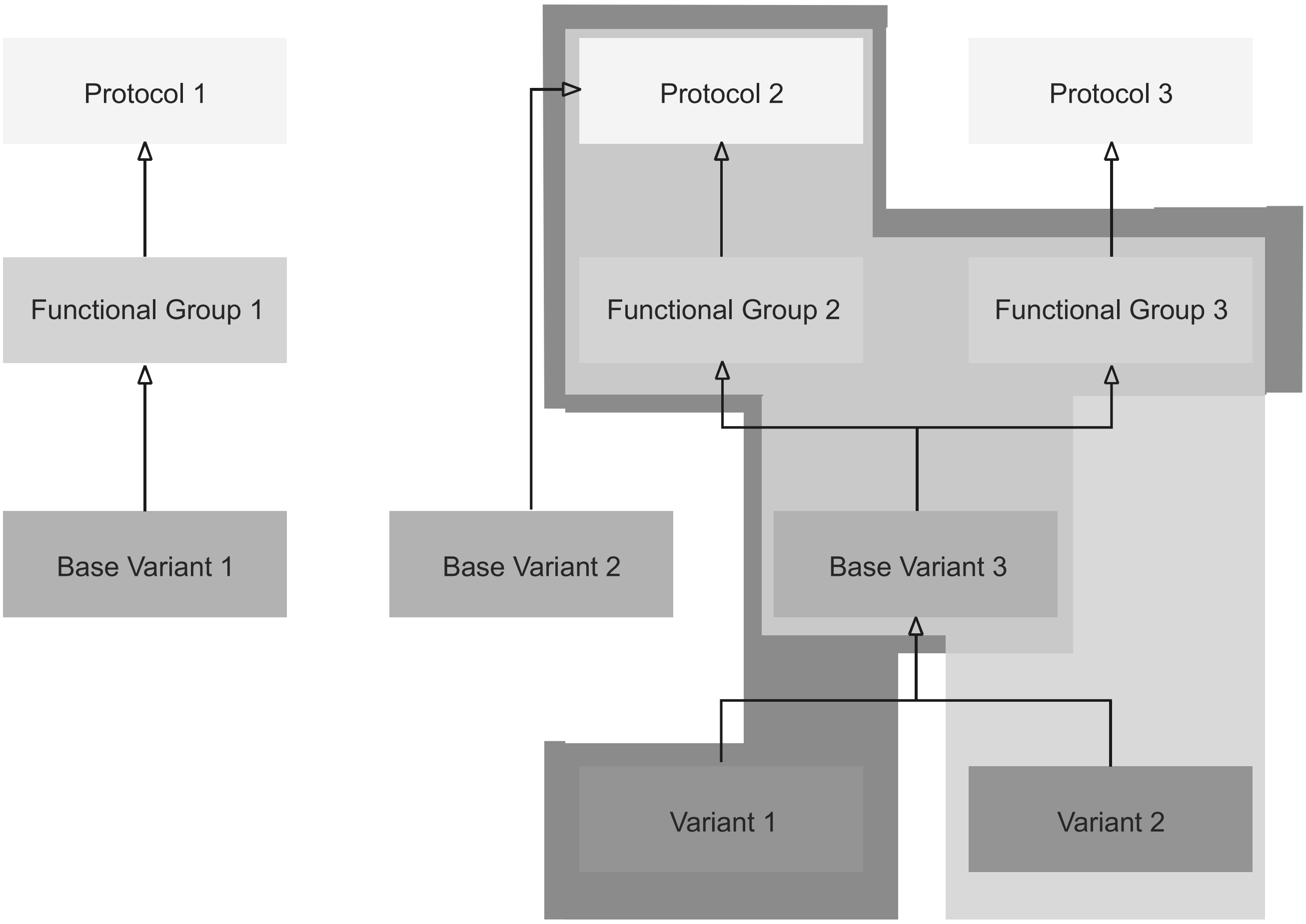


Figure 149 — AccessKey Example 2, Variant Locations

9.15 Functional Addressing

Functional addressing is a communication mode similar to a multicast in established network environments. While a physically addressed service is targeted at (and answered by) only one ECU, a functionally addressed service is usually received and answered by multiple ECUs. An example use case would be an OBD scenario where all ECUs that are emission-critical are part of one functional group and can all be addressed by a single functional service.

To use functional addressing, a logical link to a functional group shall be opened. This logical link can then be used for functional communication with the set of ECUs that are part of the functional group. It is also possible to use functional addressing alongside physical addressing. To this end, a logical link to a specific ECU and a second logical link to the functional group this ECU belongs to shall be opened. Then, the logical link of the functional group can be used for functional addressing and the logical link of the ECU can be used for physical addressing.

The following pseudo code shows an exemplary interaction between a client application and the D-server for using functional addressing [assuming the initial logical link is pointing to an ECU Base Variant]*:

a) MCDDbDLogicalLink.getDbLocation().getAccesskey().getProtocol()
=> MCDDbDatatypeShortName *protocolNameBV*

b) MCDDbDLogicalLink.getDbLocation().getDbEcu()
=> MCDDbEcu *dbEcu*

c) [IF dbEcu.getObjectType() EQUALS eMCDDBECUBASEVARIANT]*
((MCDDbEcuBaseVariant)dbEcu).getDbFunctionalGroups()
=> MCDDbEcuFunctionalGroups dbFunctionalGroups

d) FOR EACH MCDDbFunctionalGroup IN dbFunctionalGroups
MCDDbFunctionalGroup.getDbLocations() => MCDDbLocations dbLocations

e) FOR EACH MCDDbLocation IN dbLocations
MCDDbLocation.getAccessKey() => MCDAccessKey accessKey

f) accessKey.getProtocol() => MCDDatatypeShortname *protocolNameFG*

g) IF protocolNameBV EQUALS protocolNameFG

h) MCDDbDLogicalLink.getDbPhysicalVehicleLinkOrInterface.getShortname()
=> *baseVariantPhysicalVehicleLink*

i) MCDLogicalLinks.addByAccessKeyAndVehicleLink(accesskeyLL, baseVariant*PhysicalVehicleLink*, …)

When communicating functionally, the request of the functionally executed service is used as defined at the functional group. However, the response of this service could be overridden on BaseVariant or EcuVariant level. If this is the case, the responses of ECUs will be evaluated using the response patterns of the most specific diagnostic layer that is applicable (EcuVariant layer if ECU variant identification has been performed beforehand, the BaseVariant layer otherwise). If the request of a service defined on a functional level is overridden on the BaseVariant or EcuVariant level, this overridden request is only used when the service is executed physically (the service is created and executed on a logical link that points to a BaseVariant or EcuVariant location).

A functional group can contain services with and without responses. Services without responses (send-only services) can always be executed on the functional group level. For these services, no definition of physical addresses is required, that is, no information on the physical ECUs is necessary, e.g. to identify which ECUs have responded. An example for such a service would be a functional tester present message. For services with responses, response interpretation takes place on the FunctionalGroup, BaseVariant (no variant identified) or EcuVariant (variant identified) level. In case a base variant overrides the response of a functional group service, the service's response is interpreted on the base variant level for this ECU. The decision whether to use the response pattern defined on the functional group level or the one defined on the base variant level needs to be made individually for every ECU at runtime. This means that information on the physical addresses of all possibly responding ECUs is required. The physical addresses need to be defined on the base variant level, e.g. by means of corresponding communication parameter definitions. In case a functional service is overridden in an EcuVariant, a flag is set on the corresponding BaseVariant instance. This flag informs the D-server not to interpret the response of this ECU until variant identification has taken place. If in this context an ECU variant cannot be identified, the responses of that ECU are not interpreted at all. Instead, an error is put in the result for this respective response and the response's PDU is returned without interpretation. Setting the flag when overriding a functional service on an EcuVariant level is considered mandatory.

There are two ways for managing the physical response addresses of ECUs which can potentially respond to a functional request – either a list of physical response addresses on the functional group level or a communication parameter at each individual BaseVariant layer. In case of a list of physical response addresses at the functional group, no ECU base variants need to be defined in order to be able to perform functional communication (OBD use case). When interpreting a response, the response addresses on the base variant level are considered first. If none of these addresses matches, the list of response addresses stored on the functional group level is considered. Effectively, at runtime the superset of both lists is used (where duplicates have been removed).

If a response to a functional request is received which does neither match

— the response template at the EcuVariant level, if variant identification and selection has taken place,

- nor the response template defined on the base variant level,

- nor the response template defined on the functional group level (mind the order),

an interpretation error will occur. In this case, the D-server passes the erroneous response PDU to the application along with the error.

Please note: Every response to a functional request can be defined as multi-part in ODX. In this case, there can be several responses per physical ECU to a functional request.

To implement the functionality of a list of physical response addresses in the context of ODX data definitions and the D-PDU API specification, a complex communication parameter called *CP_UniqueRespIdTable* has been introduced by ODX (the equivalent concept is called URID Table in ISO 22900-2). The purpose of this communication parameter is to provide a table that allows the D-server to map the ECU responses to functional requests to their physical source ECUs. Within the *CP_UniqueRespIdTable* communication parameter, the parameter *CP_ECULayerShortName* is also of special importance. The value of the communication parameter *CP_ECULayerShortName* is used to reference the diagnostic layer a response to a functional request belongs to. Therefore, a D-server implementation shall be aware of the special semantics of these communication parameters *CP_ECULayerShortName* and *CP_ECULayerShortName* as allow for a protocol-independent implementation of functional addressing.

In principle, functional addressing works as follows:

When asked to execute a functional service, the D-server composes the unique response id table in accordance with the rules defined in 7.4.9.4 (Sequence of Events for Functional Addressing) of the ODX specification[11]. Basically, the D-server creates a table entry for each ECU base variant that is part of the functional group in question and assigns a unique identifier to each of the entries. Then, the D-server passes the unique response id table to the D-PDU API implementation alongside the request to execute the functional service in question. The D-PDU API sends the functional request and matches each ECU response to the unique response id table. If a match is found (i.e. the sender of a response to the functional request is identified to be an entry in the unique response id table), the D-PDU API tags that match with the unique identifier of the answering ECU. This then allows the D-server to relate each response to a specific ECU from the functional group and use this ECU's data for interpretation of the response PDU.

For more detail and information about dealing with deviations from the simplified procedure described above, please see the relevant clauses of ASAM MCD 2 D ODX and ISO 22900-2.

9.16 Tables

9.16.1 General

Tables have been introduced in ODX to support the concept of data identifiers and parameter identifiers. This concept describes the association of a data structure definition to a unique identifier. Thus, tables are used to describe e.g. lists of:

- Measurement values which can be read by the same (set of) DiagComPrimitive(s)

- Actuator values which can be written by the same (set of) DiagComPrimitive(s)

Tables shall be browseable in the database part of D-server independently of a specific DiagComPrimitive, that means, without having a `MCDDbDiagComPrimitive` selected. Therefore the interfaces `MCDDbTable(s)` and `MCDDbTableParameter(s)` shall be used (see Figure 152 — Usage of ODX Tables).

Each ODX table is represented by a `MCDDbTable` within a D-server. All tables defined in the database for a certain location can be obtained by calling `MCDDbLocation::getDbTables()`. A subset of this collection will be delivered by `MCDDbLocation::getDbTablesBySemanticAttribute(A_ASCIISTRING semantic)` which returns all tables of the given semantic.

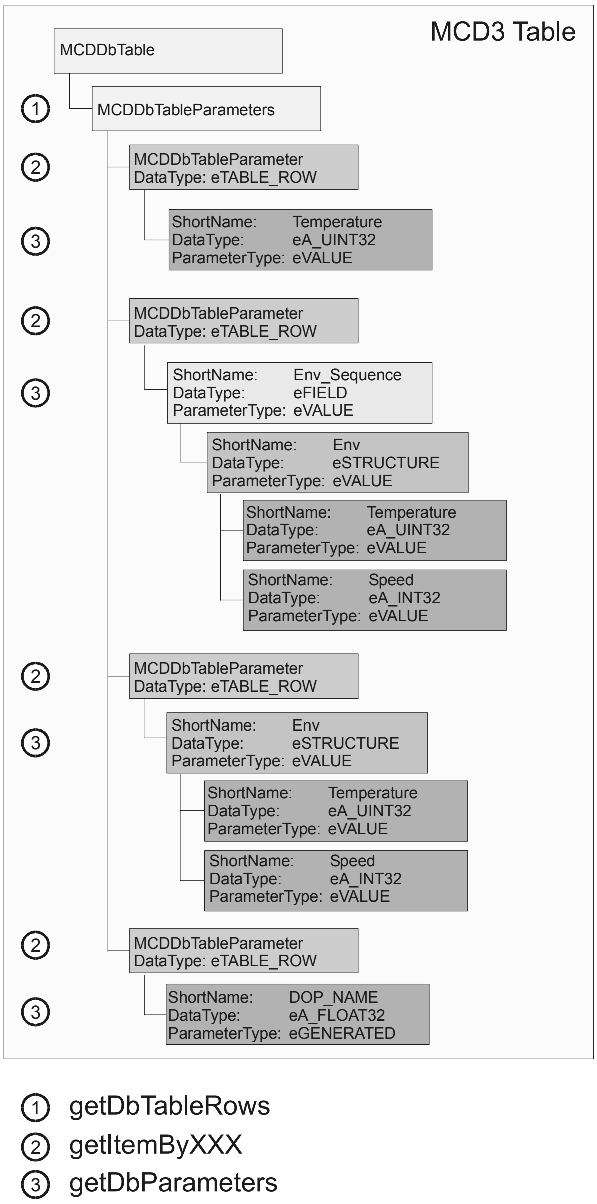


Figure 150 — Browse through a MCDDbTable

In ODX a table is composed of a non-empty set of TABLE-ROWs. Each TABLE-ROW references a STRUCTURE or a simple DOP and can be uniquely identified by a key. All table-rows of a `MCDDbTable` will be delivered by `MCDDbTable::getDbTableRows()`.

In D-server the interface `MCDDbTableParameter` is used to represent a table-row of an ODX table or a parameter in the structure referenced from a table-row, respectively. That is, it represents the ODX elements TABLE-ROW and STRUCTURE where the STRUCTURE is referenced from the TABLE-ROW.

In case of representing a TABLE-ROW, the `MCDDbTableParameter` is of data type eTABLE_ROW. Calling the method `MCDDbTableParameter::getKey()` delivers the value of the key associated with the `MCDDbTableParameter` in the corresponding `MCDDbTable`. In all other cases (representing the referenced STRUCTURE or one of its elements), the general D-server mapping of ODX-Elements to `MCDDataTypes` is applied.

9.16.2 Usage of tables within DiagComPrimitives

For the usage of tables within requests and responses of DiagComPrimitives the parameter types TABLE-STRUCT, TABLE-KEY and TABLE-ENTRY are defined in ODX.

The principle is similar to a multiplexer. The content of a TABLE-STRUCT parameter depends on a certain switch value. This switch value is given by the TABLE-KEY parameter referenced by the TABLE-STRUCT parameter[3].

More detailed, a parameter of type TABLE-KEY is the part of the PDU where the KEY (data identifier) shall be read in the case of a response or written in the case of a request. A TABLE-KEY parameter selects a TABLE-ROW of a TABLE. The selection can be done by referencing a TABLE-ROW directly (static parameter definition) or by referencing a TABLE and selecting a TABLE-ROW at runtime by using the current value of the TABLE-KEY to match it against the unique KEYs of that TABLE (dynamic parameter definition). The content of a TABLE-STRUCT parameter is the content of the STRUCTURE or simple DOP referenced by the selected TABLE-ROW.

TABLE-KEY, TABLE-STRUCT and TABLE-ENTRY parameters are represented within D-server as `MCD(Db)Parameter` objects of parameter type `eTABLE_KEY, eTABLE_STRUCT` and `eTABLE_ENTRY`.

To query all keys that can be read (`MCDResponseParameter`) or written (`MCDRequestParameter`) at runtime by a parameter of type `eTABLE_KEY` the method `MCDDbParamter::getKeys()` can be used at the corresponding `MCDDbParameter` object.

NOTE 1 In case of static parameter definition the returned collection contains exactly one key – the key of the statically referenced TABLE-ROW.

If a client needs to know which `eTABLE_STRUCT` parameters depend on a certain eTABLE_KEY parameter within one request or response, it can call `MCDDbParameter::getDbTableStructParams()` for the corresponding `eTABLE_KEY` parameter. Vice versa, calling `MCDDbParameter::getDbTableKeyParam()` on a `eTABLE_STRUCT` parameter, will deliver the referenced `eTABLE_KEY` parameter.

As stated before, the decomposition of a `eTABLE_STRUCT` parameter into further parameters represents exactly one table row of the table referenced by the corresponding `eTABLE_KEY` parameter. To get the `MCDDbTable` the parameter structure of an `eTABLE_STRUCT` parameter taken from `MCDDbParameter::getDbTable()` can be used.

NOTE 2 In case of a static parameter definition, the decomposition can already be obtained at a `eTABLE_STRUCT` parameter by calling the method `MCDDbParameter::getDbParameters()`. In case of a dynamic parameter definition, the collection returned by the method `MCDDbParameter::getDbParameters()` is empty.

If a `MCDDbTableParameter` with datatype `eTABLE_ROW` references a simple DOP in ODX, the resulting `MCDDbParameters` collection delivered by `MCDDbTableParameter::getDbParameters()` contains exactly one `MCDDbTableParameter`. The `MCDDbTableParameter's` parameter type is `eGENERATED`. Its data type and ShortName are obtained from the referenced simple DOP.

The structure of a `MCDDbParameter` with parameter type `eTABLE_ENTRY` depends on the value of the TARGET attribute of the TABLE-ENTRY parameter in ODX:

- TARGET = KEY: The `MCDDbParameter` has the datatype of the KEY-DOP of the TABLE-ROW that is referenced from the TABLE-ENTRY parameter.
- TARGET = STRUCT: The parameter has the datatype `eSTRUCTURE`. The containing `MCDDbParameters`, delivered by method `MCDDbTableParameter::getDbParameters()`, are the parameters that are contained at the referenced TABLE-ROW structure of the TABLE-ENTRY parameter.

[3] In ODX, several parameters of type TABLE-STRUCT are permitted to reference the same parameter of type TABLE-KEY. The only restriction is that these parameters need to be located on the same level in the parameter hierarchy.

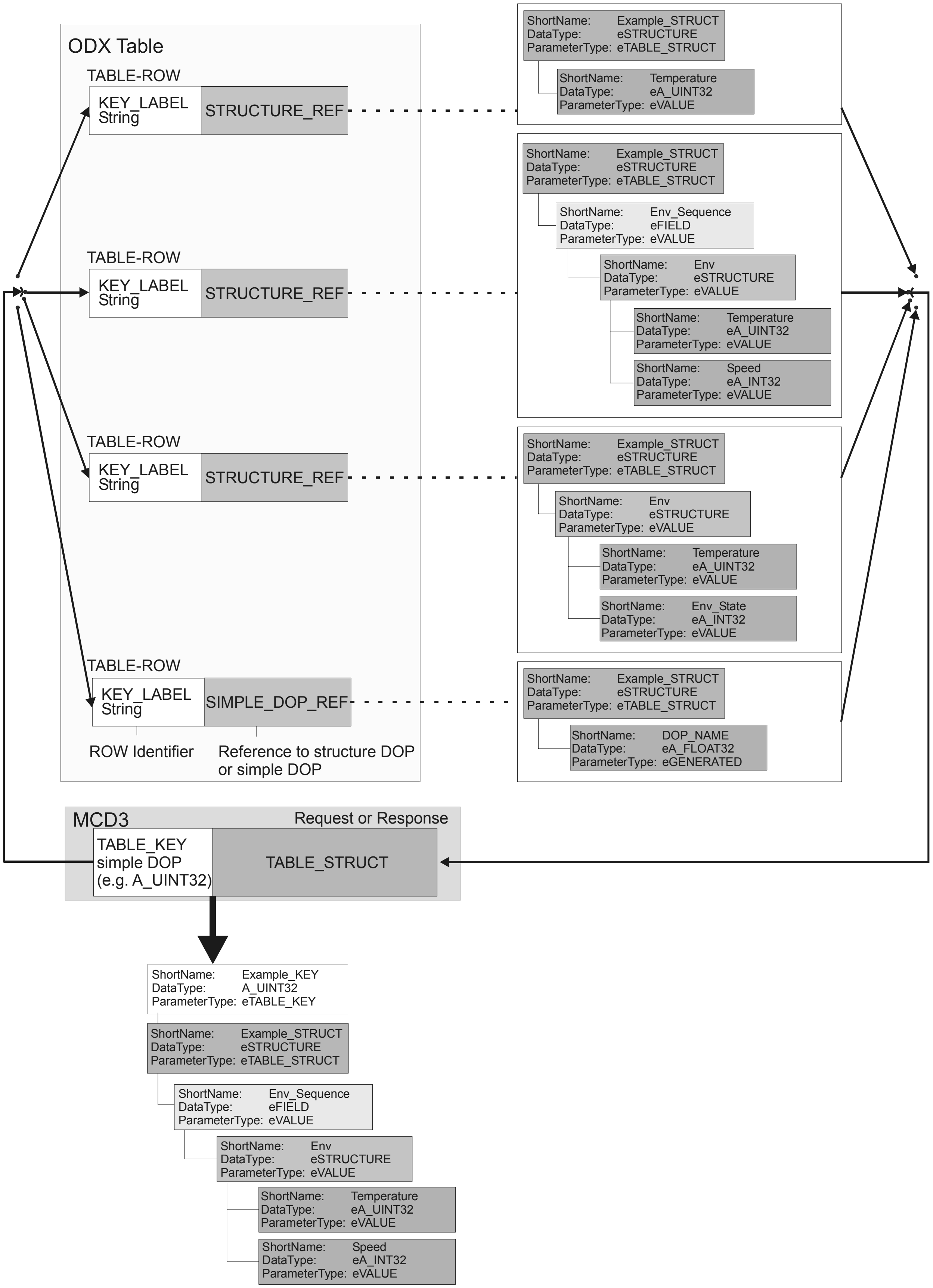

ODX Table
TABLE-ROW
KEY_LABEL String
STRUCTURE_REF
TABLE-ROW
KEY_LABEL String
STRUCTURE_REF
TABLE-ROW
KEY_LABEL String
STRUCTURE_REF
TABLE-ROW
KEY_LABEL String
SIMPLE_DOP_REF
ROW Identifier
Reference to structure DOP or simple DOP
ShortName: Example_STRUCT DataType: eSTRUCTURE ParameterType: eTABLE_STRUCT
ShortName: Temperature DataType: eA_UINT32 ParameterType: eVALUE
ShortName: Example_STRUCT DataType: eSTRUCTURE ParameterType: eTABLE_STRUCT
ShortName: Env_Sequence DataType: eFIELD ParameterType: eVALUE
ShortName: Env DataType: eSTRUCTURE ParameterType: eVALUE
ShortName: Temperature DataType: eA_UINT32 ParameterType: eVALUE
ShortName: Speed DataType: eA_INT32 ParameterType: eVALUE
ShortName: Example_STRUCT DataType: eSTRUCTURE ParameterType: eTABLE_STRUCT
ShortName: Env DataType: eSTRUCTURE ParameterType: eVALUE
ShortName: Temperature DataType: eA_UINT32 ParameterType: eVALUE
ShortName: Env_State DataType: eA_INT32 ParameterType: eVALUE
ShortName: Example_STRUCT DataType: eSTRUCTURE ParameterType: eTABLE_STRUCT
ShortName: DOP_NAME DataType: eA_FLOAT32 ParameterType: eGENERATED
MCD3
Request or Response
TABLE_KEY simple DOP (e.g. A_UINT32)
TABLE_STRUCT
ShortName: Example_KEY DataType: A_UINT32 ParameterType: eTABLE_KEY
ShortName: Example_STRUCT DataType: eSTRUCTURE ParameterType: eTABLE_STRUCT
ShortName: Env_Sequence DataType: eFIELD ParameterType: eVALUE
ShortName: Env DataType: eSTRUCTURE ParameterType: eVALUE
ShortName: Temperature DataType: eA_UINT32 ParameterType: eVALUE
ShortName: Speed DataType: eA_INT32 ParameterType: eVALUE

Figure 151 — Example of a table

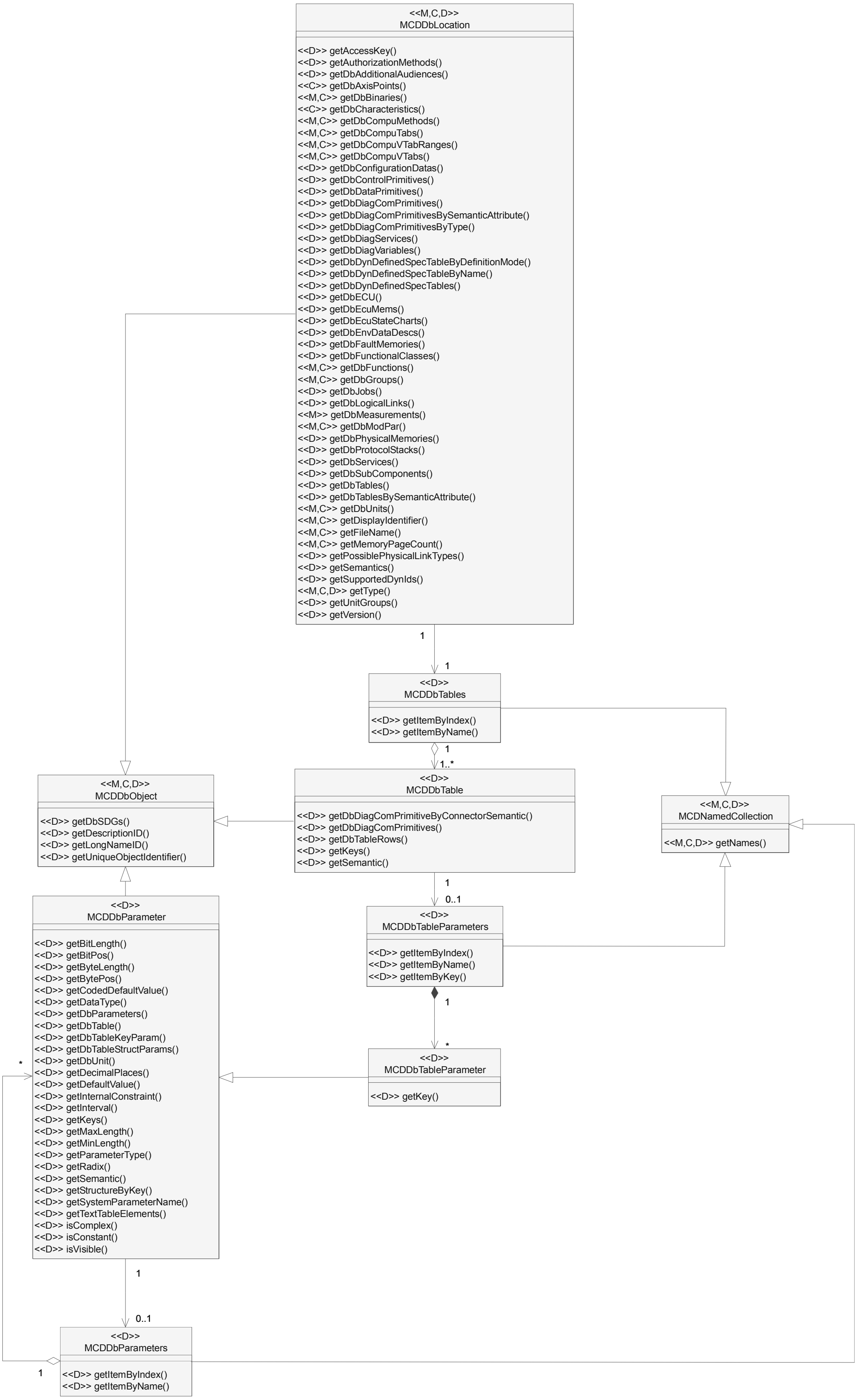


Figure 152 — Usage of ODX Tables

9.17 Dynamically Defined Local Id / Table Parameters (DDLID)

9.17.1 General

Various diagnostic application layer protocols (e.g. UDS or KWP2000) incorporate the concept of dynamically defined diagnostic services, i.e. services where the user can define the response's contents dynamically at runtime. This chapter describes how the concept of dynamically defined services is handled in the D-server API. Support for DDLID is an optional part of this part of ISO 22900.

9.17.2 DDLID principle and requirements

To use a dynamically defined diagnostic service, a diagnostic application shall first declare a dynamic service definition to an ECU. This means that a service will be sent to the ECU, which associates a certain dynamic identifier with a set of values that should be returned on future requests with that identifier. After definition of a dynamic service, the diagnostic application can then use a read-dynamically-defined-LID service to retrieve the set of values that was associated with a certain dynamic identifier. When a diagnostic application does not need a dynamically defined service anymore, a dynamic service definition can be freed/revoked by sending a third service, which tells the ECU to delete a dynamic service definition with a certain identifier. In ODX, these three services are marked by a specific diagnostic class:

- `DYN-DEF-MESSAGE` for definition of new dynamic services
- `READ-DYN-DEF-MESSAGE` for reading the contents of a previously defined dynamic service and
- `CLEAR-DYN-DEF-MESSAGE` for clearing a previously defined dynamic service definition

Although the usage principle is the same in all cases, in the context of ODX data it is distinguished between fully dynamic, semi dynamic and static DDLID. These three cases differ only in how certain parameters concerning the DYN-DEF-MESSAGE service are set up in ODX:

- For the fully dynamic case, the application is entirely free to chose the contents of a dynamic service.
- In the semi dynamic case, a part of the dynamic service's response signature is pre-defined by `CODED-CONST` parameters in ODX. However, the application is allowed to add more parameters to that dynamic service. Here, the client application can use the `READ-DYN-DEF-MESSAGE` service like a normal diagnostic service, without first having to explicitly define the contents of the specific dynamic service id. However, the client application shall still create the appropriate DynID definition service for that service id, and execute it using its default parameters.
- In the static case, all response parameters for a certain dynamic service definition are pre-set by `CODED-CONST` parameters in ODX. The application is not allowed to add more parameters to a dynamic service definition. Here, the client application uses the `READ-DYN-DEF-MESSAGE` service like a normal diagnostic service, it is not allowed to change the contents of the specific dynamic service id beforehand. However, the client application shall still create the appropriate DynID definition service for that service id, and execute it using its default parameters.

The steps a diagnostic application shall follow when using dynamically defined services are outlined below:

a) Creation of a dynamic ID:

   - Use a DiagComPrimitive for DynID definition (`DYN-DEF-MESSAGE` in ODX).
   - Get supported and available dynamic identifiers (`MCDDbLocation::getSupportedDynIds`).
   - Get a set of parameters for parameterisation of a dynamic service, set relevant parts of that parameter structure.

- Execute the DynID definition service.

b) Reading by dynamic ID:

- Use a DiagComPrimitive to read by DynID (READ-DYN-DEF-MESSAGE in ODX).
- The D-server knows through previous DynID-definition how to interpret DynID reading-service results.

c) Deletion of a dynamic ID:

- Use a DiagComPrimitive for DynID deletion (CLEAR-DYN-DEF-MESSAGE in ODX).

9.17.3 Lifecycle

9.17.3.1 General

The three steps that shall be executed during a diagnostic session that uses dynamically defined LIDs are outlined in more detail in this chapter.

9.17.3.2 Creation of Dynamically Defined Local Id

Before a dynamic service can be used by an application (`MCDDynIdReadComPrimitive`), it shall be defined first by using the `MCDDynIdDefineComPrimitive`. Depending on whether the dynamically defined service at hand is to be used dynamically, statically or semi-dynamically, the DynID definition shall be performed by the client application or will automatically be performed by the D-server.

In ODX, a specific location (EcuBaseVariant or EcuVariant) defining a `DYN-DEFINED-SPEC` can contain an arbitrary number of DiagComPrimitives with a diagnostic class of `CLEAR-DYN-DEF-MESSAGE`, `READ-DYN-DEF-MESSAGE`, or `DYN-DEF-MESSAGE`. That means that these ComPrimitives are not unique within a location. However, each location may only contain one `DYN-DEFINED-SPEC` with a certain definition mode (ODX: DEF-MODE), and for each definition mode there can only be one associated `CLEAR-DYN-DEF-MESSAGE`, `READ-DYN-DEF-MESSAGE`, or `DYN-DEF-MESSAGE` service. This means that the combination of a location and a `DYN-DEFINED-SPEC` with a certain definition mode will result in a unique set of DiagComPrimitives for defining, reading and clearing of a dynamically defined LID.

To create the default `MCDDynIdDefineComPrimitive`, `MCDDynIdReadComPrimitive` and `MCDDynIdClearComPrimitive` for a selected `DYN-DEFINED-SPEC`, the method
`MCDDiagComPrimitives::addDynIdComPrimitiveByTypeAndDefinitionMode(MCDObjectType type, A_ASCIISTRING definitionMode, MCDCooperationLevel cooperationLevel)`
should be used, using one of the unique `MCDObjectType` values.

- eMCDDYNIDDEFINECOMPRIMITIVE,
- eMCDDYNIDREADCOMPRIMITIVE or
- eMCDDYNIDCLEARCOMPRIMITIVE,

in combination with a definition mode (`DEF-MODE`) like:

- COMMON-ID,
- LOCAL-ID or
- ADDRESS.

NOTE 1 The list of valid definition modes can be extended in ODX. Only the values COMMON-ID, LOCAL-ID, and ADDRESS are predefined. The list of definition modes available for a certain DynID DiagComPrimitive can be obtained by using the method MCDDbDynIdxxxComPrimitive::getDefinitionModes().

As described above, only the combination of a definition mode and a DynID-related MCDObjectType uniquely identifies a DynID DiagComPrimitive for fully dynamic DDLID within a DYN-DEFINED-SPEC in ODX.

Also, the method MCDDiagComPrimitives::addByDbObject(MCDDbDiagComPrimitive dbDiagComPrimitve, MCDCooperationLevel cooperationLevel) can be used to create a MCDDynIdDefineComPrimitive, MCDDynIdReadComPrimitive or MCDDynIdClearComPrimitive. The list of DiagComPrimitives available at a location can be obtained by using the method MCDDbLocation::getDbDiagComPrimitivesByType(). MCDDynIdxxxComPrimitives created that way will have no initial definition mode, if multiple definition modes are valid for these MCDDynIdxxxComPrimitives. That is, MCDDynIdxxxComPrimitive()::getDefinitionMode() will throw a MCDProgramViolationException with error code eRT_ELEMENT_NOT_AVAILABLE if the collection obtained by calling getDefinitionModes() for the corresponding MCDDbDynIdxxxComPrimitive contains more than one element. The definition mode of a MCDDynIdxxxComPrimitive can be set by using method MCDDynIdxxxComPrimitive::setDefinitionMode(A_ASCIISTRING definitionMode).

NOTE 2 If methods execueSync() or executeAsync() are called for a MCDDynIdxxxComPrimitive with no definition mode or a valid DynID set (see below) a MCDParameterizationException is thrown with error code ePAR_INCOMPLETE_PARAMETERIZATION.

MCDbLocation::getSupportedDynIds(A_ASCIISTRING definitionMode) returns the list of DynIDs supported at this location for a given definition mode. This list contains all supported DynIDs (e. g. $F0 … $F9 in KWP 2000) regardless of the fact whether a DynID has already been assigned to a dynamic service or not.

NOTE 3 The supported DynIDs are dependent on location and definition mode.

For every DynID that is actually to be defined at runtime, a MCDDynIdDefineComPrimitive shall be executed. The actual DynID to be used is set at the different DiagComPrimitives (MCDDynIdDefineComPrimitive, MCDDynIdReadComPrimitive, MCDDynIdClearComPrimitive) by using the method setDynId. The setDynId-methods are used to parameterize a MCDDynId...ComPrimitive with a specific Id (e.g., $F0 … $F9 in KWP2000).

The method MCDDynIdDefineComPrimitive::setDynIdParams(MCDDatatypeAsciiStrings paramNames) is used to add a collection of parameters which should be part of the newly defined dynamic service. The parameters are selected from the respective ODX tables by using positive filter expressions (see below).

In case of a fully dynamic MCDDynIdDefineComPrimitive, all parameters shall be set by using this method. In case of a semi-dynamic MCDDynIdDefineComPrimitive, only the dynamic part of the parameter set can be defined by this method, i.e. the parameters are appended to the static part of the dynamic service. In case of a static MCDDynIdDefineComPrimitive, a MCDParameterizationException with error code ePAR_INCONSISTENT_VALUE_LIST is thrown when the client application tries to set DynID parameters. All DynID parameters used in a static, semi-dynamic or dynamic DynIdComPrimitive shall be located in the tables obtainable by MCDDbLocation::getDbDynDefinedSpecTablesByDefinitionMode(A_ASCIISTRING definitionMode).

NOTE 4 Within one DYN-DEFINED-SPEC all TABLEs referenced from one DYN-ID-DEF-MODE-INFO element shall not define or import multiple TABLE-ROWs with the same KEY. That is, the table keys used for positive filter expressions need to be unique for each definition mode.

An empty collection object of type MCDDatatypeAsciiStrings can be obtained by executing the method MCDObjectFactory::createAsciiStrings().

When a MCDDynIdDefineComPrimitive service is first created, the dynamic part of the request template is empty. It is then filled by means of the method setDynIdParams(MCDDatatypeAsciiStrings paramNames). The principle of positive filtering is used (see chapter 9.10). Every entry in the collection of filter keys has the format <KEY> (as Key of a certain TableRow| <ShortName>|…|<ShortName> (path to the element within the eSTRUCTURE or simple-DOP referenced by the corresponding TableRow). In addition, this method should throw a MCDProgramViolationException with error code eRT_NO_UNIQUE_ELEMENT if there are duplicate filter keys in the set of filter keys supplied as parameter to the method. If the MCDDynIdDefineComPrimitive has no valid definition mode set, calling method setDynIdParams(MCDDatatypeAsciiStrings paramNames) will throw a MCDParameterizationException with error code ePAR_INCOMPLETE_PARAMETERIZATION.

In ODX the DYN-DEFINED-SPEC references all Tables which can be used for creation of new DynID records. As well as the DynID DiagComPrimitives the Tables are grouped by the definition mode. For each filter key the D-server tries to find the TableRow that fits to the <KEY> given through the filter. Therefore the D-server looks up only Tables selected by the current definition mode of the MCDDynIdDefineComPrimitive. In case the D-server finds a fitting TableRow it uses the further part of the filter key to select a certain parameter within the eSTRUCTURE referenced by that TableRow. The runtime system has the protocol specific knowledge to prepare the request structure at runtime for the MCDDynIdDefineComPrimitive using the parameters selected by setDynIdParams(paramNames). This structure is also used to create the response structure at runtime for the corresponding MCDDynIdReadComPrimitive. The sequence of parameters will have the same order as the parameters added by this method. For each Logical Link, the D-server should internally maintain a list which dynamic Ids are defined and which response structure is bound to a certain dynamic service Id.

If a value (MCDValue) is passed to a setDynId method which is not in the list of supported dynamic Ids, which are obtainable by calling the method MCDDLogicalLink::getDefinableDynIds(), a MCDParameterizationException exception with error code ePAR_VALUE_OUT_OF_RANGE is thrown.

The method setDynId is used to assign a DynID to a DynIdComPrimitve, regardless of the fact whether a static, a semi-dynamic, or a dynamic DynId service is to be used. If a DynID parameter that the client application tries to change is defined as CODED-CONST in ODX, this method should throw a MCDProgramViolationException with error code eRT_INVALID_OPERATION.

When executing a MCDDynIdDefineComPrimitve, the DynId defined in the DYN-ID parameter of this DiagComPrimitve is added to the list of used DynIds at the logical link. If this DynId is already occupied, a MCDProgramViolationException with error code eRT_DYNID_ALREADY_USED is thrown.

It is not allowed to execute a service for DynID definition repeatedly, therefore if a MCDDynIdDefineComPrimitive is called with method startRepetition a MCDProgramViolationException with error code eRT_SERVICE_REPEATED_NOT_ALLOWED is thrown.

9.17.3.3 Reading by Dynamically Defined Local Id

A dynamically defined LID can be read from the ECU by executing a MCDDynIdReadComPrimitive.

The D-Server will fill the dynamic part of the response template at runtime depending on the request parameters of a previously executed MCDDynIdDefineComPrimitive with the same definition mode as the MCDDynIdReadComPrimitive.

9.17.3.4 Deletion of Dynamically Defined Local Id

Dynamic services exist until the end of a diagnostic session (execution of MCDStopCommunication, or until the ECU falls out of diagnostics by itself [Figure 137 — State diagram MCDDLogical Link)].

DynIDs can be deleted by executing a MCDDynIdClearComPrimitive.

- A DynId will be removed from the list that contains the assignable DynIds when the execution of a `MCDDynIdDefineComPrimitive` returns with a positive result. To avoid that a DynId is assigned multiple times by different `DynIdDefineComPrimitive`s executed simultaneously, the DynId used in this service needs to be locked temporarily during execution of a `MCDDynIdDefineComPrimitive`.

- A DynId will be re-added to the definable DynId list (the list that contains the assignable DynIds) when a `MCDDynIdClearComPrimitive` executed with that DynId as parameter returns with a positive response from the ECU.

- If the ECU transits from the state `eCOMMUNICATION` to `eONLINE`, all DynIDs will become available again.

If a DynID list changes, the event `onLinkDefinableDynIdListChanged(MCDValues, MCDDLogicalLink)` will be created by the logical link and sent to all registered event handlers. It transports the actual list of available (not defined) DynIds, as well as the corresponding logical link.

9.17.3.5 DB-Templates for Requests and Responses regarding DDLID

As DDLID is an intrinsically dynamic concept of diagnostics at runtime, no complete database template for read DDLID-services can be obtained from the ODX database. The part of the response that will be filled dynamically at runtime can be identified by a MCDDbParameter with parameter type eDYNAMIC. Calling method `MCDDbParameter::getDbParameters()` for this parameter will always deliver an empty collection. The client application is responsible for assembling a DynIdComPrimitive by selecting physical values from ODX tables and adding them as parameters to a DynIdComPrimitive by using the `MCDDynIdDefineComPrimitive`.

NOTE 1 The current specification of DDLID ComPrimitives as shown in the examples in the ODX specification requires protocol dependent knowledge in the D-server. The protocol dependence is, e.g. manifested in the fact that number, type, and meaning of the different elements in a DDLID item structure and DDLID content parameter is purely protocol dependent and requires protocol-specific knowledge in the D-server. To overcome this specification gap, ODX data that aims to be useable in a protocol-independent manner defines specific semantic values for all the parameters that are used for DDLID-related services. The following table defines a value for the semantic attribute of each relevant parameter that will be recognized by the D-server to allow protocol-independent handling of DDLID data.

Table 44 — DDLID parameter semantic attribute definitions

Parameter name	Semantic value	Description
DDLID	DYN-ID	The parameter that contains the DynId-value to be assigned/read/deleted.
STRUCT_DDLIDItem	DDLID-DEF-STRUCT	The structure definition that contains the parameters necessary for DDLID definition at runtime.
definitionMode	DDLID-DEF-MODE	Definition mode for a certain DDLID parameter, protocol specific.
positionInDDLID	DDLID-POS	Position of the parameter in the response returned by a DDLID read service, protocol specific.
memorySize	DDLID-MEMORY-SIZE	The size of the parameter that is to be included in a DDLID service, protocol specific.
LID	DDLID-LID	The LOCAL-ID, COMMON-ID, or ADDRESS that is the source of the DDLID parameter, protocol specific.
positionInRecordLID	DDLID-LID-POS	The position of the DDLID parameter in its source LID, protocol specific.
DDLID	DYN-ID	The parameter that contains the DynId-value to be assigned/read/deleted.

NOTE 2 The statements above do not have any implication on the D-server API. Rather, they affect the D-server internal realization of DDLID.

ODX data that is supposed to allow a D-server to support DDLID in a protocol-independent way shall follow the parameter semantic definitions outlined in Table 44 — DDLID parameter semantic attribute definitions.

9.17.3.6 Procedure description

The execution of a MCDDynIdDefineComPrimitive, a MCDDynIdReadComPrimitive or a MCDDynIDClearComPrimitive is only allowed in the logical link states eONLINE and eCOMMUNICATION.

After successful execution of the MCDDynIdDefineComPrimitive, a database template for the corresponding MCDDynIdReadComPrimitive is generated internally within the D-server. The database template is stored for one specific DDLID. If a MCDDynIdReadComPrimitive is executed on this DDLID, the stored database template is used to interpret the MCDDynIdReadComPrimitive's response.

A stored database template for a DDLID is deleted internally in the following cases:

- The ECU changes state from eCOMMUNICATION to eONLINE.
- The MCDDynIdClearComPrimitive is called (the DynID is deleted within the ECU).

Whenever the DiagComPrimitives MCDDynIdDefineComPrimitive, MCDDynIdReadComPrimitive, and MCDDynIdClearComPrimitive are called in the wrong sequence, e.g. a read is executed before the corresponding DynID has been defined, a MCDProgramViolationException with error code eRT_WRONG_SEQUENCE is thrown.

Whenever one of the ComPrimitives MCDDynIdDefineComPrimitive, MCDDynIdReadComPrimitive, and MCDDynIdClearComPrimitive is called that has not been parameterised completely, e.g. the DynID has not been set before execution, a MCDParameterizationException with error code ePAR_INCOMPLETE_PARAMETERIZATION is thrown.

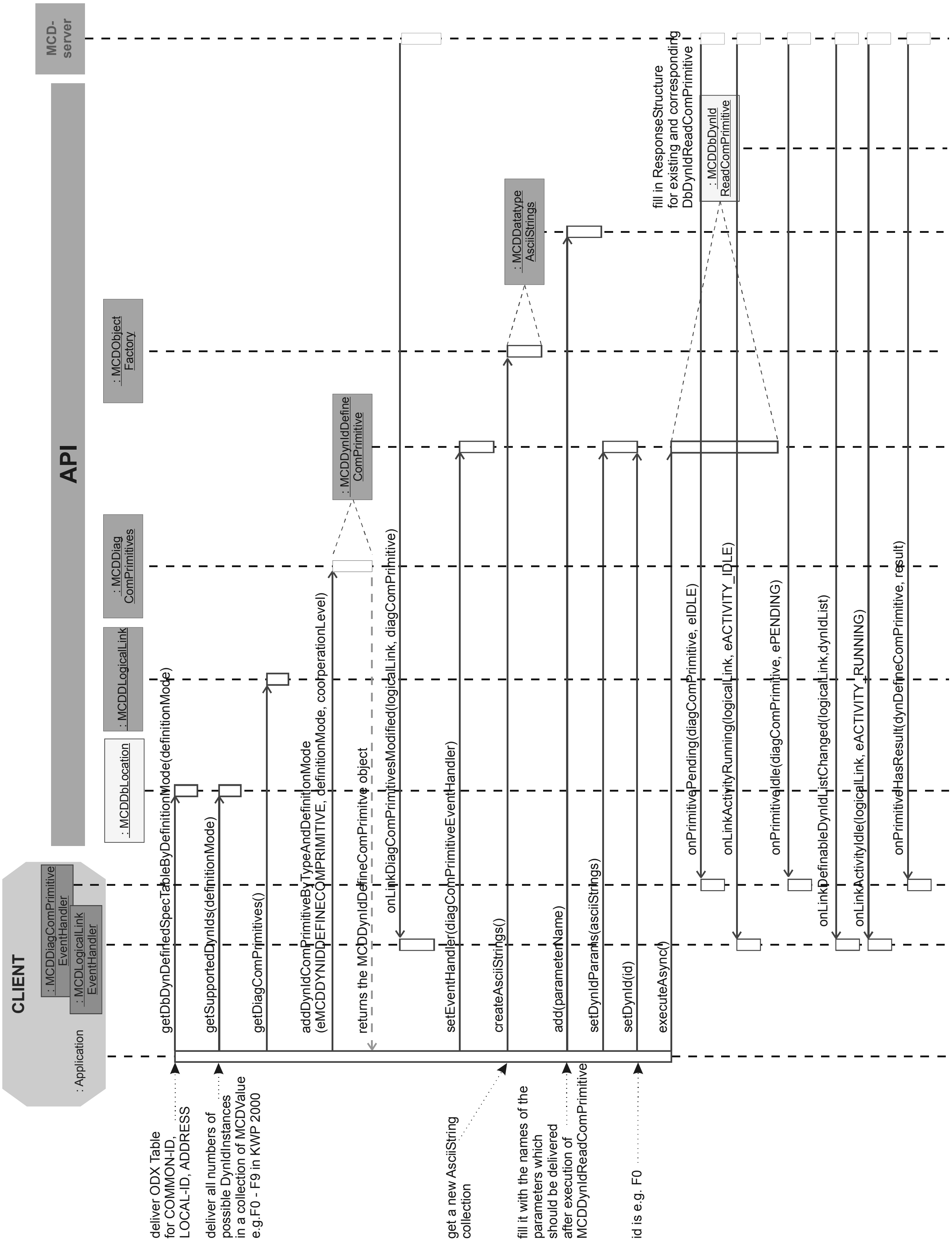

CLIENT
: Application
: MCDDiagComPrimitive EventHandler
: MCDLogicalLink EventHandler
API
MCD-server
: MCDDbLocation
: MCDDLogicalLink
: MCDDiag ComPrimitives
: MCDDynIdDefine ComPrimitive
: MCDObject Factory
: MCDDatatype AsciiStrings
: MCDDbDynId ReadComPrimitive
getDbDynDefinedSpecTableByDefinitionMode(definitionMode)
getSupportedDynIds(definitionMode)
getDiagComPrimitives()
addDynIdComPrimitiveByTypeAndDefinitionMode (eMCDDYNIDDEFINECOMPRIMITIVE, definitionMode, cooperationLevel)
returns the MCDDynIdDefineComPrimitive object
onLinkDiagComPrimitivesModified(logicalLink, diagComPrimitive)
setEventHandler(diagComPrimitiveEventHandler)
createAsciiStrings()
add(parameterName)
setDynIdParams(asciiStrings)
setDynId(id)
executeAsync()
onPrimitivePending(diagComPrimitive, eIDLE)
onLinkActivityRunning(logicalLink, eACTIVITY_IDLE)
onPrimitiveIdle(diagComPrimitive, ePENDING)
onLinkDefinableDynIdListChanged(logicalLink,dynIdList)
onLinkActivityIdle(logicalLink, eACTIVITY_RUNNING)
onPrimitiveHasResult(dynDefineComPrimitive, result)
fill in ResponseStructure for existing and corresponding DbDynIdReadComPrimitive
deliver ODX Table for COMMON-ID, LOCAL-ID, ADDRESS
deliver all numbers of possible DynIdInstances in a collection of MCDValue e.g.F0 - F9 in KWP 2000
get a new AsciiString collection
fill it with the names of the parameters which should be delivered after execution of MCDDynIdReadComPrimitive
id is e.g. F0

Figure 153 — Usage of Dynamically Defined Local Id (Part 1)

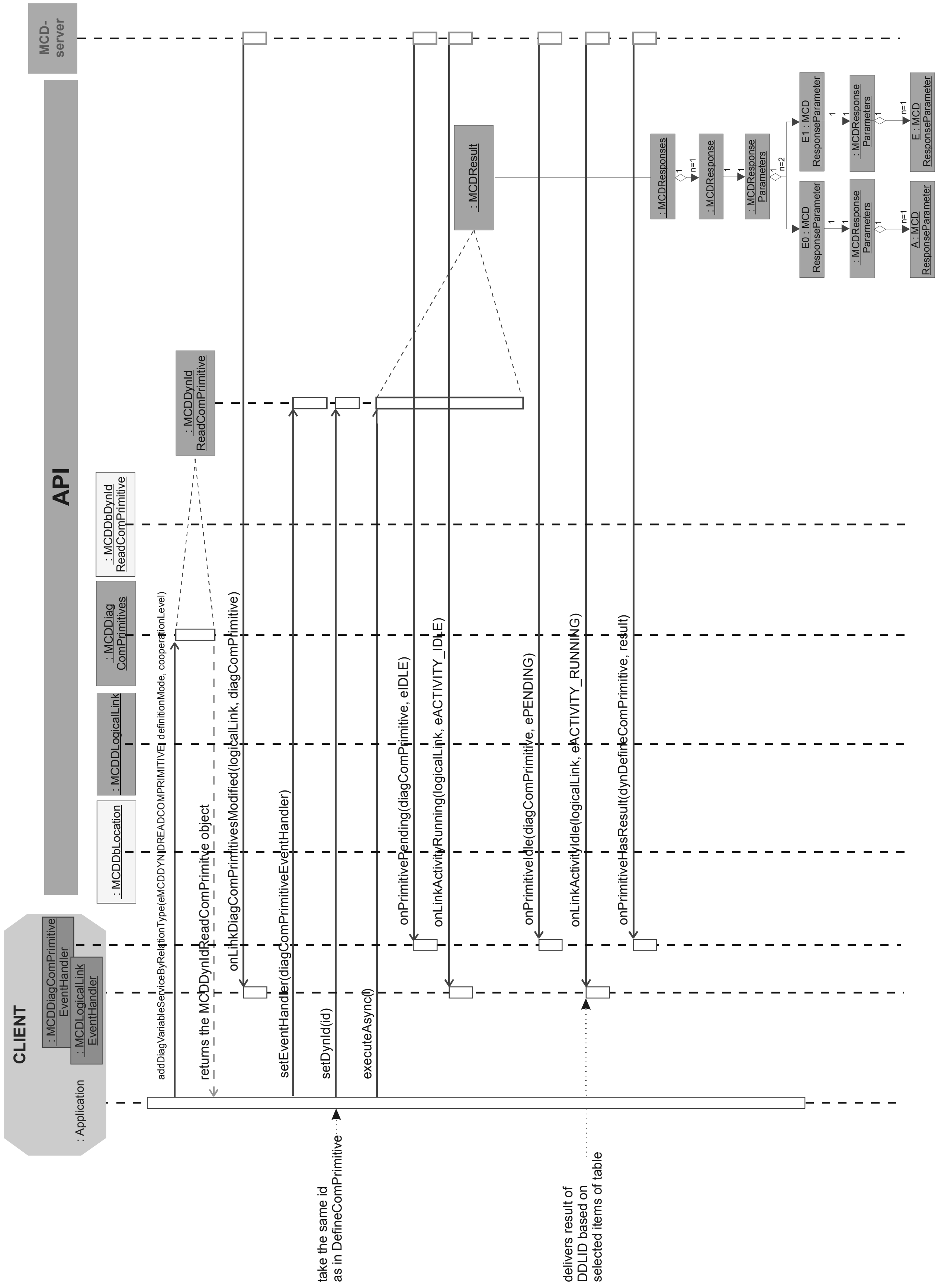

MCD-server
API
CLIENT
: Application
: MCDDiagComPrimitive EventHandler
: MCDLogicalLink EventHandler
: MCDDbLocation
: MCDDLogicalLink
: MCDDiag ComPrimitives
: MCDDbDynId ReadComPrimitive
: MCDDynId ReadComPrimitive
: MCDResult
: MCDResponses
: MCDResponse
: MCDResponse Parameters
E0 : MCD ResponseParameter
E1 : MCD ResponseParameter
: MCDResponse Parameters
: MCDResponse Parameters
A : MCD ResponseParameter
E : MCD ResponseParameter
1
n=1
n=2
addDiagVariableServiceByRelationType(eMCDDYNIDREADCOMPRIMITIVE, definitionMode, cooperationLevel)
returns the MCDDynIdReadComPrimitive object
onLinkDiagComPrimitivesModified(logicalLink, diagComPrimitive)
setEventHandler(diagComPrimitiveEventHandler)
setDynId(id)
executeAsync()
onPrimitivePending(diagComPrimitive, eIDLE)
onLinkActivityRunning(logicalLink, eACTIVITY_IDLE)
onPrimitiveIdle(diagComPrimitive, ePENDING)
onLinkActivityIdle(logicalLink, eACTIVITY_RUNNING)
onPrimitiveHasResult(dynDefineComPrimitive, result)
take the same id as in DefineComPrimitive
delivers result of DDLID based on selected items of table

Figure 154 — Usage of Dynamically Defined Local Id (Part 2)

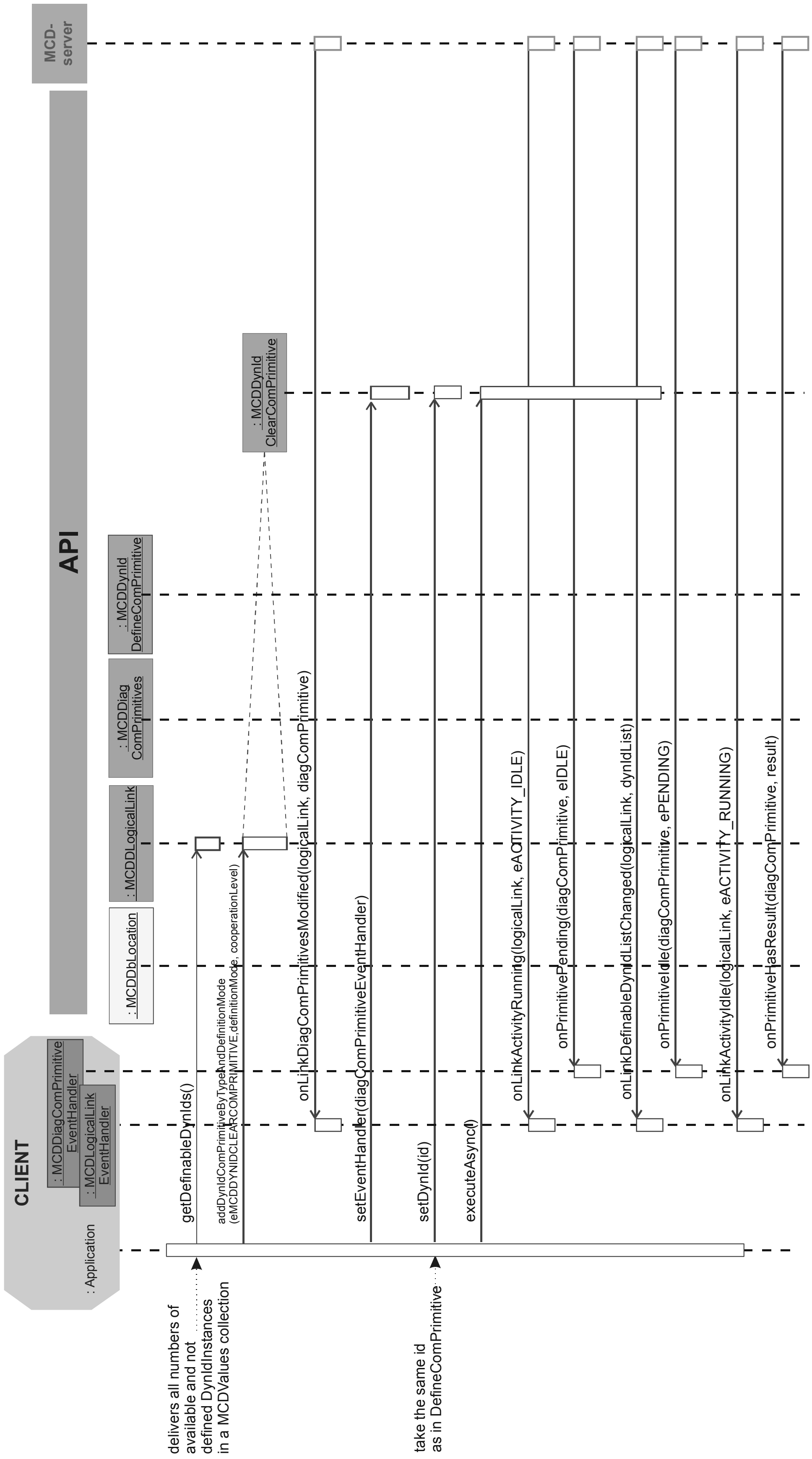

MCD-server
API
CLIENT
: Application
: MCDDiagComPrimitive EventHandler
: MCDLogicalLink EventHandler
: MCDDbLocation
: MCDDLogicalLink
: MCDDiag ComPrimitives
: MCDDynId DefineComPrimitive
: MCDDynId ClearComPrimitive
getDefinableDynIds()
addDynIdComPrimitiveByTypeAndDefinitionMode (eMCDDYNIDCLEARCOMPRIMITIVE,definitionMode, cooperationLevel)
onLinkDiagComPrimitivesModified(logicalLink, diagComPrimitive)
setEventHandler(diagComPrimitiveEventHandler)
setDynId(id)
executeAsync()
onLinkActivityRunning(logicalLink, eACTIVITY_IDLE)
onPrimitivePending(diagComPrimitive, eIDLE)
onLinkDefinableDynIdListChanged(logicalLink, dynIdList)
onPrimitiveIdle(diagComPrimitive, ePENDING)
onLinkActivityIdle(logicalLink, eACTIVITY_RUNNING)
onPrimitiveHasResult(diagComPrimitive, result)
delivers all numbers of available and not defined DynIdInstances in a MCDValues collection
take the same id as in DefineComPrimitive

Figure 155 — Usage of Dynamically Defined Local Id (Part 3)

9.18 Internationalisation

9.18.1 Multi language support

Every database object has beside of the short name additional long name and description. To support country specific settings for messages (e.g. long names) and descriptions a string ID is used. With the string ID every string can mapped in the local language by the Client. To get the string ID each database object has two additional methods: `getLongNameID()` and `getDescriptionID()`.

For `MCDParameter` and `MCDResponse` the values for translation can be received by the corresponding database objects via the method `getDbObject`.

9.18.2 Units

All units in the D-server are given in relation to SI units. But in many Client applications country specific settings are needed. Therefore the database provides units and unit groups. The unit group (of type “COUNTRY”) can be set for each Logical Link separately. (It makes no sense to set a single unit.) If the unit group “DEFAULT” is set, the D-server sets the unit group back to the original settings given by database.

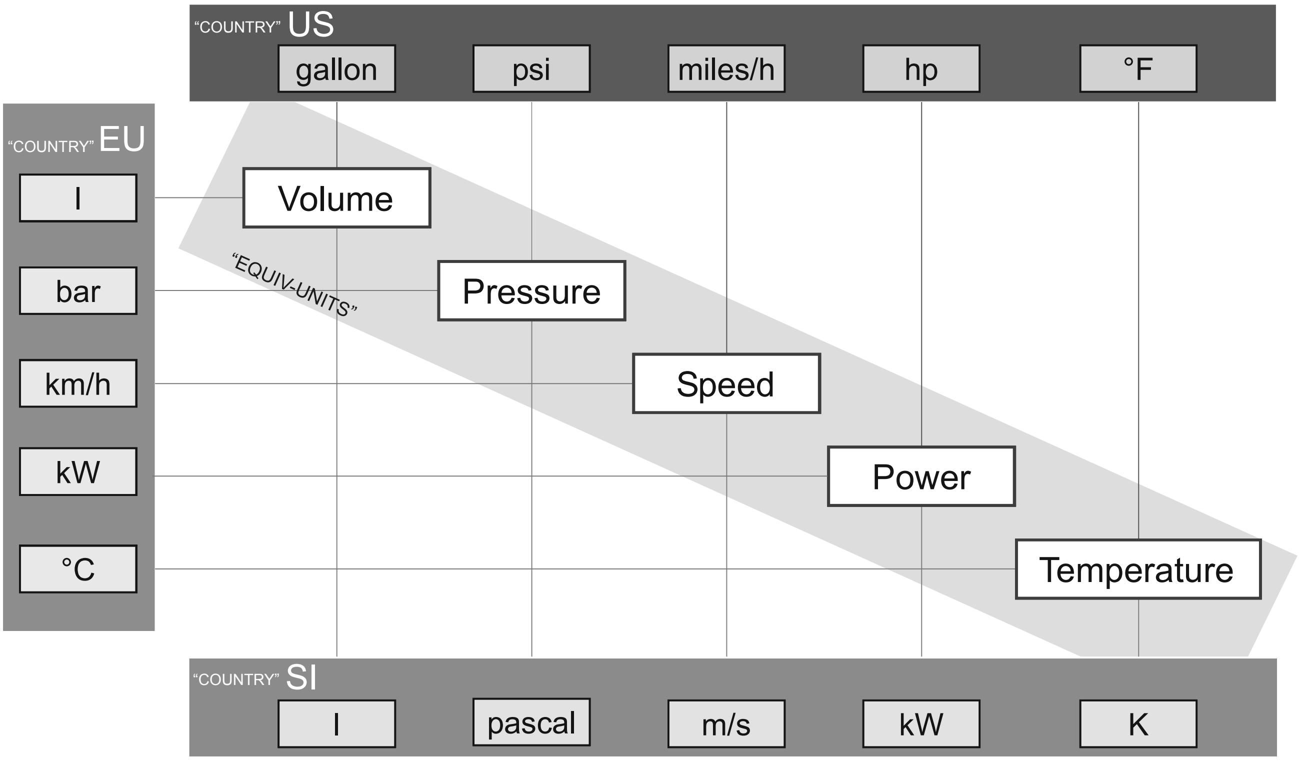


Figure 156 — Unit groups

9.19 Special Data Groups

The generic Special Data Groups (SDGs) structure was introduced to afford the definition of company specific data structures necessary for arbitrary use cases, like highly company specific purposes as ECU programming or flash processes.

Special Data Groups (SDGs) are used to add additional information to specific ODX elements, e.g. DIAG-COMM (MCDDataPrimitive), DTCs, or FlashSessions. SDGs have been introduced in ODX to be able to capture information that is not covered by general ODX mechanisms. For example, SDGs can be used to attach error set conditions to DTCs or to provide error trees. However, SDGs have been introduced to be able to provide additional information but not calculation relevant information, i.e. SDGs do not have semantics. Therefore, SDGs are not allowed in request, response, or result structures.

An SDG contains an optional MCDDbSpecialDataGroupCaption to describe the SDG content, and a list of MCDSpecialDataGroup and MCDSpecialDataElement objects that contain the special data. This list can contain an arbitrary number of MCDSpecialDataGroup and MCDSpecialDataElement, and the ordering of these MCDSpecialDataGroup and MCDSpecialDataElement is not restricted in any way.

NOTE SDGs can be nested recursively; that way, very complex data structures may be defined as SDGs. The MCDSpecialDataElement is used to add semantic information to the appropriate object.

SDGs can belong to any DbObject. An empty collection will be delivered if no DbSDGs available in the ODX data.

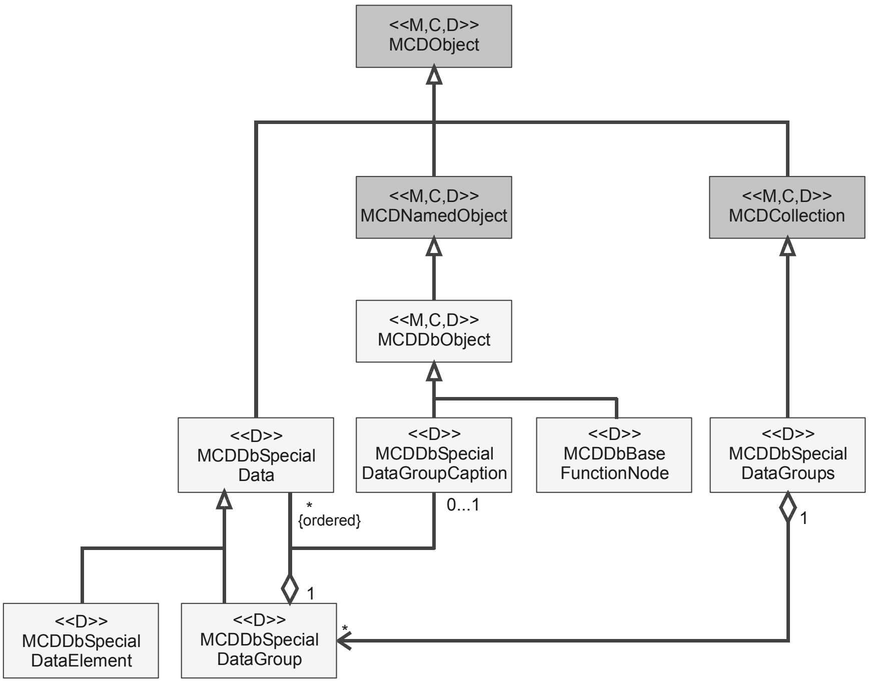


Figure 157 — Special Data Groups

9.20 ECU Flash programming

9.20.1 Goal

The design of the ECU (re-)programming is based on the following goals and requirements:

- The DB part of the object model shall provide access to all fields within ECU-MEMs and associated elements in the ODX data.
- The D-server shall be able to list ECU MEMs in dependence of the selected MCDDbLocation.
- It shall be possible to write a generic flash job that can be programmed independently of the flash programming data, e.g. a concrete flash session. For this purpose, a flash session, which is to be programmed into or read from an ECU, can be supplied as a parameter to a flash job at runtime.
- All protocol dependent activities shall be performed in a flash job or inside a protocol processor (for ISO underneath D-PDU API).

- It shall be possible to lock any logical link to be used for flash programming during the whole flash process.

- Offsets within the flash programming data will be handled by the D-server internally to calculate start and end addresses.

9.20.2 Description of Terms for ECU-Reprogramming

9.20.2.1 General

The following subclauses describe the most important terms used in ECU-Reprogramming in greater detail.

9.20.2.2 ECU-MEM Description

ECU-MEMs are containers for flash data defined in an ODX database. ECU-MEMs are represented by objects of type `MCDDbEcuMem` in the server.

9.20.2.3 Flash Class Description

FlashClasses (FLASH-CLASS) – until ASAM MCD 3D Ver. 02.00.02 known as FlashSessionClasses – introduce a means to group FlashSessionDesc objects into categories. That is, an external structure is induced on the set of all FlashSessionDescs contained in an ECU-MEM container. A single FlashSessionDesc can be associated with multiple FlashClasses. The criteria for the definition of FlashClasses and for the association of a FlashSessionDesc with such a FlashClass are company-specific. A possible criteria is the type of the data represented by a FlashSessionDesc, e.g. FlashSessionDescs for data, FlashSessionDescs for bootloader, or FlashSessionDescs for application code. Such a criteria would result in FlashClasses 'Data', Bootloader' and 'Application Data'. As a result, FlashClasses are user-defined structure information in the ODX data. No predefined FlashClasses exist.

9.20.2.4 Flash Job Description

A FlashJob (special type of SINGLE-ECU-JOB) is a new class of Java job derived from the abstract interface `MCDJob`. FlashJobs are used to execute a flash (re-)programming session (see description of term Flash Session) within the D-server.

NOTE There can be download sessions and upload sessions. In the first case, the session (re-)programs an ECU. In the latter case, the flash data is read from the ECU.

9.20.2.5 Flash Key Description

A FlashKey (attribute PARTNUMBER at SESSION-DESC in ODX) is a unique identifier for a FlashSessionDesc. This FlashKey shall be absolutely unique within the whole D-server. This identification can be used for production purposes to add a company-specific unique ID to the content of ECU-MEMs delivered by an ECU-supplier.

9.20.2.6 Flash Session Description

If a client application uses FlashJobs (see description of term Flash Job) for (re-)programming an ECU or for obtaining flash data from an ECU, a FlashSession (SESSION) is the smallest unit of flash data which can be provided to a FlashJob for processing at the job's execution time. That is, FlashSessions are the only logical units that can be chosen for programming or upload by means of FlashJobs. The choice of a FlashSession is performed by selecting an appropriate FlashSessionDesc which references the corresponding FlashSession object and the flash job responsible for programming or upload. Flash sessions are typically decomposed into smaller units which cannot be processed by a FlashJob separately – FlashDataBlocks, FlashSegments, FlashFilters, FlashSecurities etc. Read more about these objects in the remainder of this chapter.

9.20.2.7 Flash Session Desc Description

A FlashSessionDesc (SESSION-DESC) describes how a single FlashSession (see description of term Flash Session) is to be processed by the D-server or a FlashJob, respectively. More precisely, a FlashSessionDesc references a FlashSession, a FlashJob, and a possibly empty collection of FlashClasses. Furthermore, a FlashSessionDesc defines the direction (upload or download) of data flow between a D-server and an ECU when the FlashSession referenced is processed. In case the direction is upload, flash data is transferred from an ECU to a D-server and finally to a client application. In case the direction is download, flash data is transferred from the D-server to an ECU. Every FlashSessionDesc is directly contained in an ECU-MEM container.

9.20.2.8 Description of 'Late-bound data files' Mechanism

ODX defines two different ways, how binary data referenced from FlashDataBlocks are actually to be loaded into memory by the D-server, (1) immediately when accessing the corresponding FlashDataBlock or (2) only at the time when it is requested by the client application (late-bound loading)). In the first case, the method `MCDDbFlashData::isLateBound()` returns `false`. In the latter case, the method `MCDDbFlashData::isLateBound()` returns `true`.

9.20.2.9 Priority Description

The priority (attribute PRIORITY at SESSION-DESC in ODX) can be used by a client application to decide in which order the FlashSessionDescs shall be processed during a flash programming session for a single ECU. The priority information can be obtained from a `MCDDbFlashSessionDesc` by calling its method `getPriority()`.

9.20.3 Structure of the function block flash programming

9.20.3.1 Database part

The root element of the database part of the function block flash programming is the `MCDDbLocation` object (see Figure 158 — Flash Programming – Associations Database Part). At this object, all associated `MCDDbEcuMem` objects can be listed. Which `MCDDbEcuMem` objects can be reached from which `MCDDbLocation` is described by ECU-MEM-CONNECTOR objects in ODX.

NOTE In the D-server API, the class `MCDDbEcuMem` merges the aspects of the ODX elements ECU-MEM and ECU-MEM-CONNECTOR.

From a `MCDDbEcuMem`, the `MCDDbFlashSessionDesc` objects can be obtained which are defined for the current `MCDDbLocation`. A `MCDDbEcuMem` does also provide access to a collection of `MCDDbFlashClasses`. These `MCDDbFlashClass` objects allow a client application to structure the FlashSessionDescs. The methods `MCDDbFlashSessionDesc::getLongName()`, `MCDDbFlashSessionDesc::getShortName()`, and `MCDDbFlashSessionDesc::getDescription()` return the corresponding values of a SESSION-DESC element in ODX.

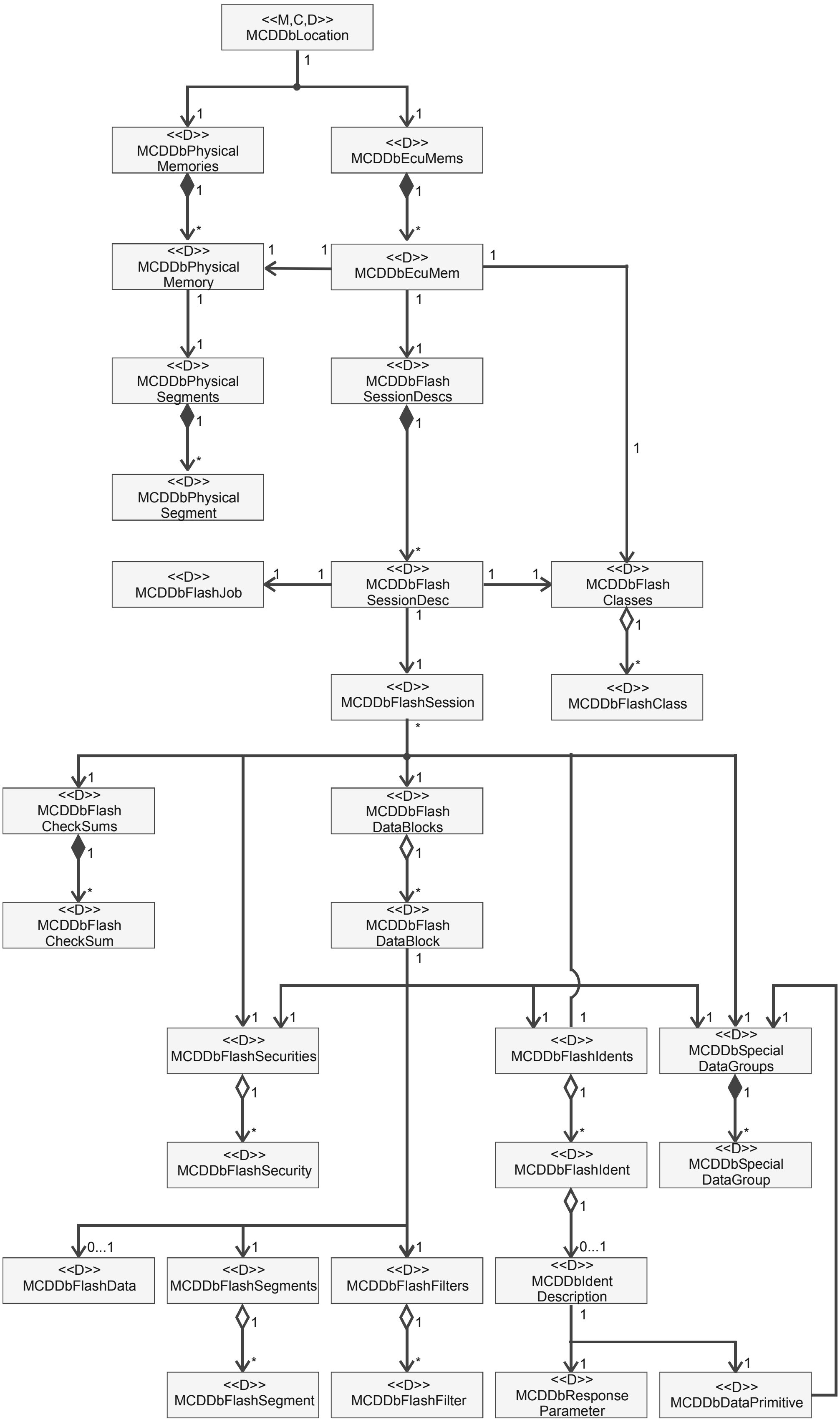


Figure 158 — Flash Programming – Associations Database Part

While the MCDDbFlashSessionDesc describes which MCDDbFlashSession can be processed by which MCDDbFlashJob in which direction (upload or download), the MCDDbFlashSession element referenced from a MCDDbFlashSessionDesc is the root element of the object hierarchy describing the structures and content of the flash data. First, a MCDDbFlashSession is decomposed into a set of MCDDbFlashDataBlocks. Each MCDDbFlashDataBlock represents an own block of possibly segmented flash data. The binary data of a MCDDbFlashDataBlock can either be part of the corresponding ODX file or it can be located in an external file. In the latter case, the binary data is accessed via a MCDDbFlashData element.

The filename of an external flash data file can either be stated completely or the filename can contain wildcards. In the latter case, the external flash data file also needs to be late-bound, that is, the concrete data file to be used for a FlashDataBlock will be determined in the runtime part of the flash module at runtime.

A MCDDbFlashDataBlock can be further decomposed into segments. These segments can be stated in the ODX data. In this case, they are represented by objects of type MCDDbFlashSegment in the database part of the flash module. However, the actual flash segments that will be used for flash programming at runtime are calculated at runtime from the address information in MCDDbFlashSegments and from the address information in the actual flash data. For calculation rules, see 9.20.3.5.

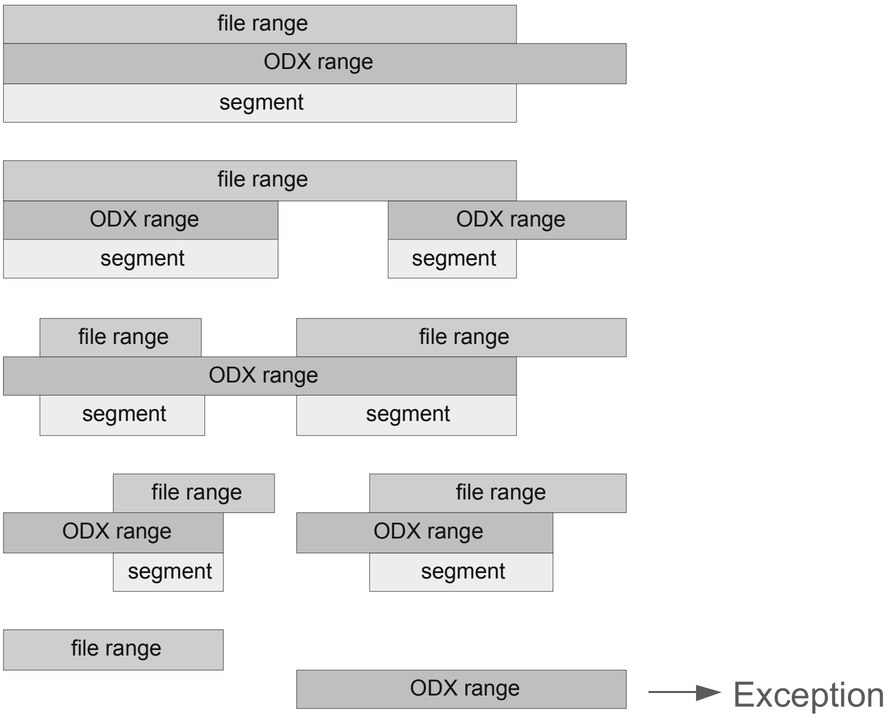


Figure 159 — Segment Handling

The MCDDbFlashFilter objects referenced from a MCDDbFlashDataBlock are used to cut-out parts of the binary data, which can be obtained from this MCDDbFlashDataBlock. More precisely, FlashFilters are applied to the binary data (including address information) calculated from the binary data referenced from a MCDDbFlashDataBlock before this binary data is decomposed into segments and before it is programmed into an ECU. Therefore, the address ranges may not overlap. As the binary data referenced from a MCDDbFlashDataBlock can be plain binary as well as in hexadecimal format with address information, the effect of FlashFilter application is different in both cases. Valid and invalid combinations of address ranges in flash data files and ODX data as well as the resulting segments are sketched in Figure 159 — Segment Handling. For more information see 9.20.3.5.

To be able to introduce a certain security in the flash programming process, a MCDDbFlashSession references further objects – a collection of type MCDDbFlashCheckSums (CHECKSUMs) and a collection of type MCDDbFlashSecurities. For one FlashSession one or more MCDDbFlashCheckSum objects may be present. Every MCDDbFlashCheckSum object defines the checksum algorithm to be used, the address range in the binary data to be considered for checksum calculation, and the expected result. The MCDDbFlashCheckSum objects are used to perform checks during the flash process. The information stored in the MCDDbFlashCheckSum objects can be used by the FlashJob to calculate the checksum of the flash data (see 9.20.6.5).

With MCDDbFlashSecurity objects it is possible to introduce security specific information (e.g. checksums, signatures) for a whole FlashSession. Security information can be used by a FlashJob to check the integrity and the authenticity of the flash data. The value returned by the method MCDDbFlashSecurity::getSecurityMethod() denotes the method chosen as security concept. The return value might be either a single information about the method used or a reference to an external access table. The flashware signature – obtainable via MCDDbFlashSecurity::getFlashwareSignature() – holds the signature which shall be sent to the target device (ECU) for verification of authenticity. The flashware checksum – obtainable via MCDDbFlashSecurity::getFlashwareChecksum() – holds the checksum (e.g. a CRC32) which shall be sent to the target device for verification of integrity. The value returned by the method MCDDbFlashSecurity::getValidity() (attribute VALIDITY-FOR in ODX) describes, which ECU belongs to the given signature. In combination with the security method, the VALIDITY-FOR information is used to influence the behaviour of the FlashJob.

It is recommended to use the FlashSecurity information to check the integrity and authenticity of the source data and to use the FlashChecksum information to calculate the checksum of the data to be programmed into or to be read from an ECU. It is also reasonable to use FlashSecurities directly in FlashDataBlocks because these define the actual flash data.

Figure 160 — Flash Programming – Interfaces Database Part shows the inheritance hierarchy of the interfaces in the database part of the flash programming modules.

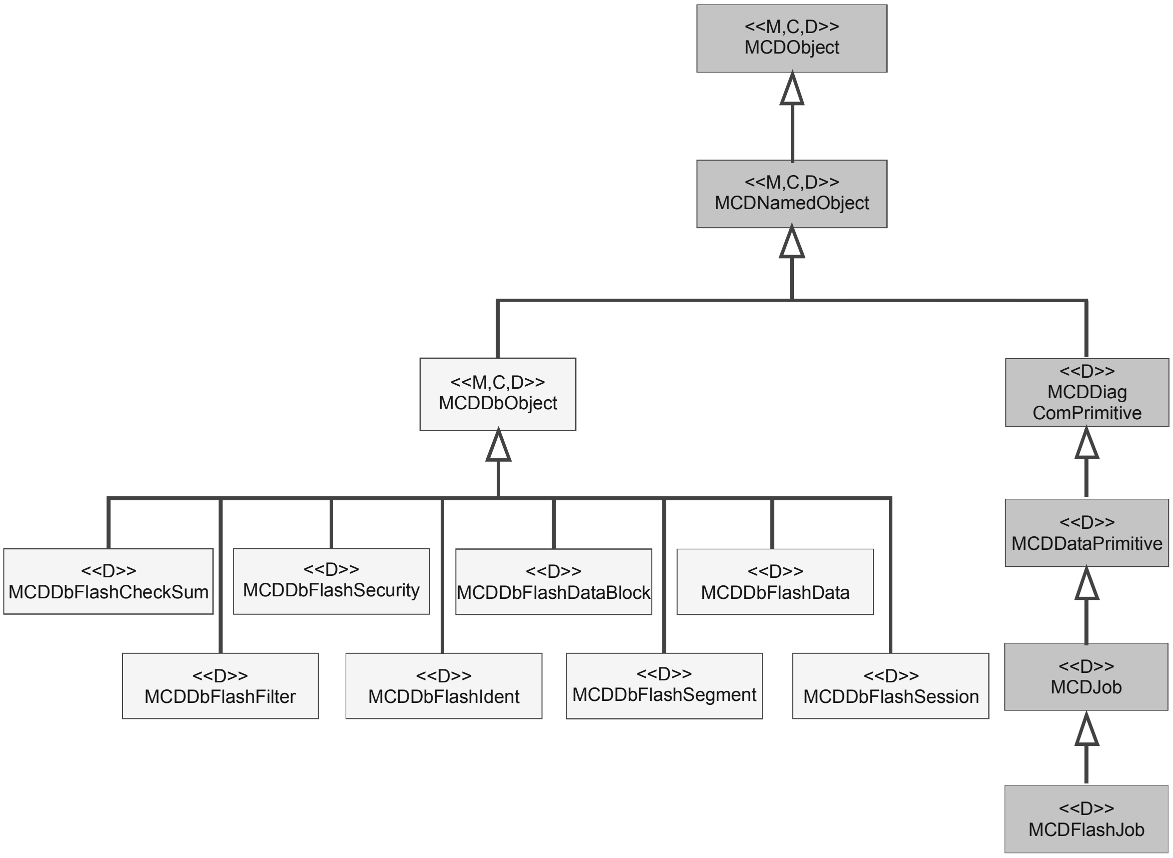


Figure 160 — Flash Programming – Interfaces Database Part

9.20.3.2 Runtime part

In order to (re-)program an ECU or to read flash data from an ECU, it is required to have runtime instances of the flash data objects and to have a runtime instance of an appropriate FlashJob. The corresponding object association model is shown in Figure 161 — Flash Programming – Associations Runtime Part.

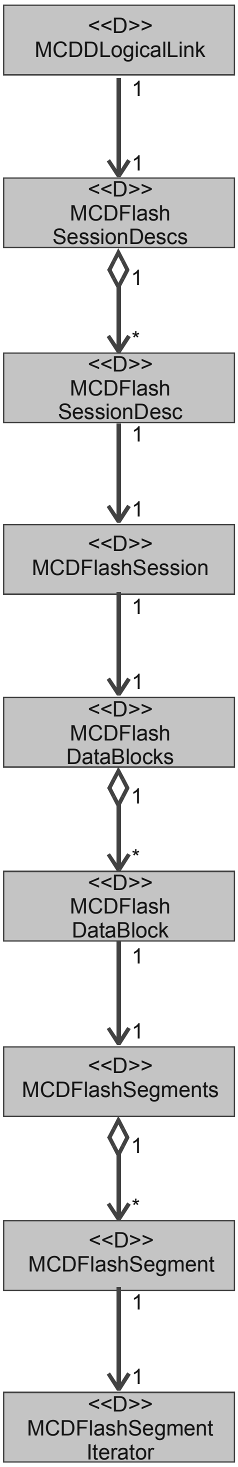


Figure 161 — Flash Programming – Associations Runtime Part

To execute ECU (re-)programming or to read the current flashware from an ECU, it is possible to create a runtime instance of a FlashSessionDesc first. Such an instance is represented by an object of type

MCDFlashSessionDesc. The reason for the requirement to create a runtime FlashSessionDesc is that the segmentation of the flash data needs to be calculated from the actually chosen binary flash data at runtime (see 9.20.3.5 for the calculation algorithm).

A runtime MCDFlashSessionDesc is structured similarly to the corresponding MCDDbFlashSessionDesc. That is, a MCDFlashSessionDesc contains a single MCDFlashSession. The MCDFlashSession contains a collection of type MCDFlashDataBlocks. Every MCDFlashDataBlock within this collection contains a collection of MCDFlashSegments.

While MCDFlashSessionDesc, MCDFlashSession, and MCDFlashDataBlock reference the corresponding database object in the database part of the flash module, the MCDFlashSegment objects contained in a MCDFlashSegments collection do not have a direct correspondence in the database part. Instead, these objects need to be calculated from the information in MCDDbFlashSegment objects and MCDDbFlashFilter objects as well as from the address information in the actually used binary flash data (see 9.20.3.5).

If a new MCDFlashSessionDesc is created, e.g. via MCDFlashSessionDescs::addByDbObject(…), all contained objects like MCDFlashSession, MCDFlashDataBlock, and MCDFlashSegment are created as far as statically defined in the corresponding database part. If any contained MCDFlashDataBlock is late-bound, no MCDFlashSegments are created for this MCDFlashDataBlock. The same applies in case of flash upload sessions. In both cases, the MCDFlashSegments are calculated on demand later.

9.20.3.3 Handling binary flash data

Late-bound data files

In the context of data files for ECU programming, the term late-bound denotes that the corresponding data file will be loaded by a D-server as late as possible. That is, a late-bound data file is loaded the latest

- when one of the methods
  MCDFlashSegment::getBinaryData(),
  MCDFlashSegmentIterator::getFirstBinaryDataChunk(),
  MCDFlashSegmentIterator::hasNextBinaryDataChunk(),
  MCDFlashSegmentIterator::getNextBinaryDataChunk()
  is called for one of the MCDFlashSegments contained in a MCDFlashDataBlock
  or

- when the method getFlashSegments() is called for the first time at an MCDFlashDataBlock in case of a Motorola-S or an Intel-Hex data file.

The binary data associated with a MCDFlashDataBlock can only be marked late-bound if this binary data is located in an external file referenced from the ODX data. That is, the binary data is not embedded into the ODX data.

If the value returned by MCDDbFlashData::isLateBound() is 'false' at a reference to an external resource file (e.g., job code, flash data, coding data), this is considered a guarantee that the content of this resource file will not change while a D-server is running. More precisely, the content of the external resource file shall be static for the time between MCDSystem::selectProjectXXX() and MCDSystem::deselectProject() for the same project.

NOTE Exchanging external resource files may lead to non-deterministic behavior of a D-server. In particular, this statement holds in case the late-bound property is set to 'true' for some ODX element.

Wildcards in data file names

The reference to external binary data from an element of type MCDDbFlashData attached to a MCDDbFlashDataBlock can either point to a specific external data file or it can be unspecific. In the first case (specific data file), the value of type A_ASCIISTRING returned by the method MCDDbFlashData::getDataFileName() does not contain any wildcards, that is, none of the characters '?' or '*' is contained in the filename. Here, the external data file can be accessed at any time, e.g. to read the binary data or to calculate MCDFlashSegments (see 9.20.3.5).

In case of an unspecific data file, the filename returned by MCDDbFlashData::getDataFileName() contains one or more wildcard characters '?' or '*'. These wildcards need to be resolved at runtime in order to identify matching data files.

NOTE Wildcards are only allowed in a filename if the flash data is also marked late-bound, that is, if the method MCDDbFlashData::isLateBound() returns true for MCDDbFlashData element. Here, the binary data file to be used for a corresponding MCDFlashDataBlock needs to be determined at runtime. For this purpose, the following steps need to take place:

- The client application calls the method MCDFlashDataBlock::getMatchingFileNames(). Now, the D-server tries to identify all files which match the pattern returned by the method getDataFileName() at the MCDDbFlashData in the database part of the current FlashDataBlock. The result of the calculation is a collection of MCDDatatypeAsciiString objects where each MCDDatatypeAsciiString represents the filename of a matching data file.

  NOTE The search scope of the D-server depends on vendor-specific definitions and settings. That is, each implementation of a D-server may have a different definition of the directories, which are searched for matching data files. Please refer to the documentation of the specific D-server implementation for more details.

- The user / client application selects one of these MCDDatatypeAsciiString objects and calls the method MCDFlashDataBlock::loadFlashSegments() with the selected MCDDatatypeAsciiString as input. Now, the D-server is able to read the data from the selected data file and to create the MCDFlashSegments for the current MCDFlashDataBlock as described in 9.20.3.5.

- The client application calls the method MCDFlashDataBlock::getFlashSegments(). If not already performed before, the D-server now creates the MCDFlashSegments for the current MCDFlashDataBlock as described in 9.20.3.5.

- In order to clear the current set of generated FlashSegments, the client application needs to call the method MCDFlashDataBlock::clearFlashSegments() at the corresponding MCDFlashDataBlock. The client application is required to explicitly clear the current set of FlashSegments (if present) before a different data file can be selected for the same MCDFlashDataBlock in Step 2.

Flash Segment Iterator

Flash data files can become quite large (several tens of megabytes). As a consequence, keeping data files in the main memory of a tester consumes huge amounts of memory. To also support thin tester applications with less main memory capacity, the concept of FlashSegmentIterators has been introduced. Such a FlashSegmentIterator allows reading the data, which represents the binary data of a FlashSegment in an external data file, in small chunks. As a result, large data files can be handled by tester applications with less memory consumption.

In this part of ISO 22900, a FlashSegmentIterator is represented by the interface MCDFlashSegmentIterator. A new iterator for a FlashSegment can be obtained by calling the method MCDFlashSegment::createFlashSegmentIterator(size : A_UINT32). The parameter size of this method defines the maximum size of the chunks that will be delivered by the newly created iterator in bytes.

NOTE The last chunk delivered by the D-server is potentially smaller than defined by size. By means of the method MCDFlashSegmentIterator::getFirstBinaryDataChunk(), the first piece of binary data is obtained from the external data file – if possible. In addition, the FlashSegmentIterator's internal pointer is set to the next piece of binary data.

By means of the method `MCDFlashSegmentIterator::hasNextBinaryDataChunk()`, a client application can check whether there are more binary data chunks available in the iterator. If this is the case, the method `MCDFlashSegmentIterator::getNextBinaryDataChunk()` can be used to obtain this next piece of binary data.

Figure 162 — Example of data chunks obtained from a FlashSegment SegA shows an example of a FlashDataBlock *BCMApplicationData* which references an external data file *bcm_app.msr* via a corresponding element of type `MCDDbFlashData`. This FlashDataBlock is decomposed into two FlashSegments *SegA* and *SegB*. In the example, the client application has created a new FlashSegmentIterator with a size of 300 bytes for FlashSegment *SegA*. When iterating through the FlashSegments binary data, the FlashSegmentIterator returns three chunks of 300 bytes and a forth chunk of 100 bytes. After the fourth chunk has been obtained by the client application, the end of FlashSegment SegA has been reached. Therefore, no further chunks are available. That is, the method `MCDFlashSegmentIterator::hasNextBinaryDataChunk()` returns false after the forth chunk has been obtained.

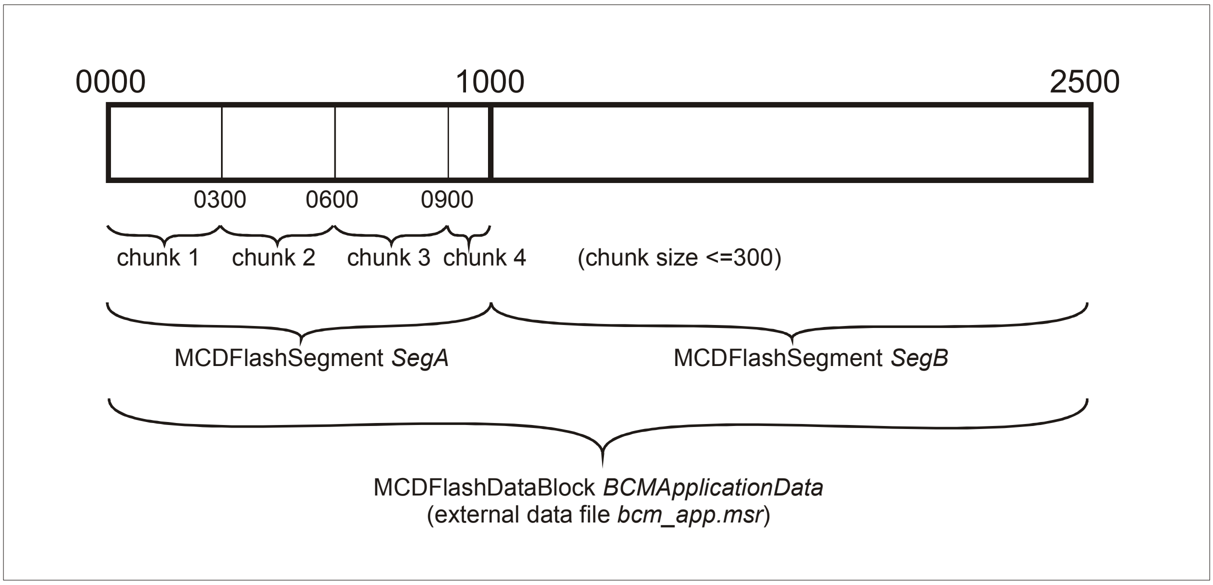


Figure 162 — Example of data chunks obtained from a FlashSegment SegA

In addition to reading data from an external data file in small chunks, the FlashSegmentIterator allows to write data to an external flash data file in small chunks. This possibility is required in case flash data is read from an ECU (flash upload). Here, the data read from the ECU can be written to an external data file in small chunks by using the method `MCDFlashSegmentIterator::setBinaryDataChunk`. Every call of this method appends the binary data supplied as parameter at the end of the current FlashSegment's binary data block in main memory as well as in the external data file. In case of an external data file, the position of the data in the file is defined by the FlashSegment the FlashSegmentIterator has been created for. Therefore, it is recommended to process FlashSegments in ascending order of their start addresses in case of flash data upload.

9.20.3.4 Identification mechanism

To prevent that the software in an ECU is accidentally overwritten with wrong flash data, a security mechanism has been introduced which is based on identification values for flash data. These identification values are attached to the corresponding flash elements in ODX and reflected here.

Two different types of identification values are distinguished – *own identification values* and *expected identification values*. Own identification values are associated with a `MCDDbFlashDataBlock`. They describe the identification values which will be written to an ECU in case this FlashDataBlock is written to the ECU.

Expected identification values are associated with a MCDDbFlashSession. These values describe which values should be obtainable from an ECU before the FlashSession has been executed. That is, the expected identification values describe a precondition of a FlashSession while the own identification values at the FlashDataBlocks describe post-conditions of the FlashSession they are contained in. Typical information which is stored in identification values is, e.g. a part number, a software version, or a supplier information. More detail on the usage of identification values is given in the following paragraphs.

Expected identification values

Expected identification values hold information on the target device (ECU). These precondition identification attributes should be fulfilled before starting a re-programming session. Expected identification values can be used to verify whether a FlashSession is compatible with the target device. This means that the current FlashSession should only be programmed into an ECU if all identification values read from the target ECU can be matched against the corresponding expected identification values. The expected identification values can be checked either by the client application or within a FlashJob.

Own identification values

Own identification values describe the post-condition identification attributes that should have been written into an ECU as part of the flash data just programmed into this ECU by means of the current FlashSession. A FlashDataBlock may contain a list of own identification values which can be used to check whether the ECU has been programmed correctly.

Representation of identification values in MCD-3 specification

Both, own identification values and expected identification values are described by objects of type MCDDbFlashIdent (IDENT-DESCs) in this part of ISO 22900. The fact that they are own identification values or expected identification values is solely determined by the method they have been obtained with from the D-server (MCDDbFlashSession::getDbExpectedIdents() for expected identification values versus MCDDbFlashDataBlock::getDbOwnIdents() for own identification values).

The information of how to read the current value of a MCDDbFlashIdent stored in an ECU from this ECU, is represented by an element of type MCDDbIdentDescription referenced from a MCDDbFlashIdent (see Figure 163 — Flash Programming – MCDDbFlashIdent and MCDDbIdentDescription). A MCDDbIdentDescription object provides information on

— the DataPrimitive which needs to be executed in order to read the corresponding identification value from an ECU and on

— the ResponseParameter (with SIMPLE-DOP) which contains the respective value after the DiagComPrimitive has been executed successfully.

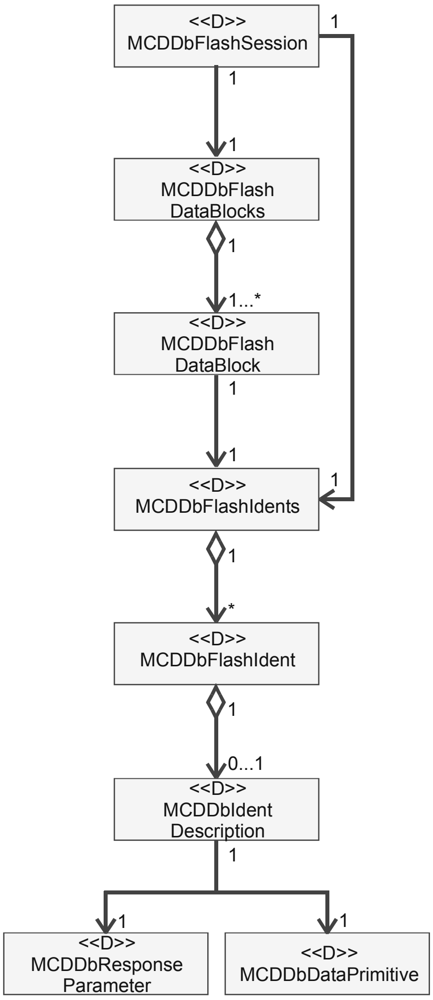


Figure 163 — Flash Programming – MCDDbFlashIdent and MCDDbIdentDescription

A typical procedure for using expected identification values is as follows: Before really transferring any flash data from the tester to the ECU, a FlashJob internally checks whether the preconditions of the FlashSessionDesc supplied as input parameter are met. That is, the FlashJob queries the identification values currently stored in the target ECU and checks these values against the values obtainable from the `MCDDbFlashIdent` objects obtained from the FlashSession referenced by the FlashSessionDesc. A possible layout of this procedure is sketched as follows:

Step1: Fetch the MCDFlashSession from the MCDFlashSessionDesc supplied as parameter to the current FlashJob (MCDFlashSessionDesc::getFlashSession()).

Step2: From the MCDFlashSession obtained in step 1, get the corresponding MCDDbFlashSession (MCDFlashSession::getDbObject()).

Step3: From the MCDDbFlashSession obtained in step 2, get the list of expected identification value objects (MCDDbFlashSession::getDbExpectedIdents()).

Step4: For each of the MCDDbFlashIdent objects obtained in step 3 (MCDDbFlashIdents::getItemByIndex()),

- get the associated MCDDbIdentDescription (MCDDbFlashIdent::getReadDbIdentDescription()),
- execute the DataPrimitive referenced from this MCDDbIdentDescription (MCDDbIdentDescription::getDbDataPrimitive(), MCDDiagComPrimitives::addByDbObject(), MCDDataPrimitive::executeSync()) and

— compare the value of the runtime ResponseParameter given by the MCDDbIdentDescription with every of the allowed values obtainable via MCDDbFlashIdent::getIdentValues().

Step5: If a matching value has been returned by the ECU for each of the MCDDbFlashIdents, the FlashJob can continue and program the data referenced from the current FlashSession into the ECU. Otherwise, the FlashJob needs to be aborted (negative response or exception plus termination) to signal that the current ECU setup shall not be modified by means of the current FlashSession.

Step6: After having executed the main programming part successfully, a FlashJob or a client application can verify that the reprogramming has been successful by checking the own identification values as follows:

Step7: Fetch the MCDFlashSession from the MCDFlashSessionDesc supplied as parameter to the current FlashJob (MCDFlashSessionDesc::getFlashSession()).

Step8: From the MCDFlashSession obtained in step 6, get the contained MCDFlashDataBlocks (MCDFlashSession::getFlashDataBlocks()).

Step9: For each of the MCDFlashDataBlock objects obtained in step 7 (MCDFlashDataBlocks::getItemByIndex()), get the corresponding MCDDbFlashDataBlock (MCDFlashDataBlock::getDbObject())

Step10: For each of the MCDDbFlashDataBlock objects obtained in step 8, get the referenced own identification values (MCDDbFlashDataBlock::getDbOwnIdents())

Step11: For each of the MCDDbFlashIdent objects obtained in step 9 (MCDbFlashIdents::getItemByIndex()),

— Get the associated MCDDbIdentDescription (MCDDbFlashIdent::getReadDbIdentDescription()),

— Execute the DataPrimitive referenced from this MCDDbIdentDescription (MCDDbIdentDescription::getDbDataPrimitive(), MCDDiagComPrimitives::addByDbObject(), MCDDataPrimitive::executeSync()) and

— Compare the value of the runtime ResponseParameter given by the MCDDbIdentDescription with every of the allowed values obtainable via MCDDbFlashIdent::getIdentValues().

Step12: If a matching value has been returned by the ECU for each of the MCDDbFlashIdents for each of the MCDDbFlashDataBlocks, the FlashJob has verified that the ECU has been reprogrammed successfully. Otherwise, the FlashJob needs to be aborted (negative response or exception plus termination) to signal that the ECU has not been programmed successfully.

9.20.3.5 Segmenting Flash Data

FlashDataBlocks are always decomposed into FlashSegments. These FlashSegments can be defined by means of `MCDDbFlashSegments`, `MCDDbFlashFilters`, and the binary data in case of HEX data files. More precisely, the FlashSegments to be used at runtime need to be calculated from all three sources of information. This subclause describes, how the runtime FlashSegments, that is, the `MCDFlashSegment` objects contained in a `MCDFlashDataBlock`, are calculated in case of plain binary data files and in case of HEX data files. Furthermore, this subclause describes how the calculated `MCDFlashSegments` are to be named and how the source addresses is determined. The calculation of FlashSegments and address information does only take place in the runtime part of the flash programming part of a D-server. The database part does only reflect the information as read from the ODX data.

Calculation of FlashSegments from plain binary data

If the flash data is represented by a plain binary file, this data file itself does not contain any address information and it does not contain any FlashSegment definitions. As a result, the corresponding information needs to be calculated as shown in Figure 164 — Calculating FlashSegments from Binary Data. Before a

MCDFlashSegment can be programmed into an ECU, the TARGET-ADDR-OFFSET defined in the corresponding FlashDataBlock shall be added to all address values. This address offset can be obtained by means of MCDFlashDataBlock::getDbObject()::getAddressOffset().

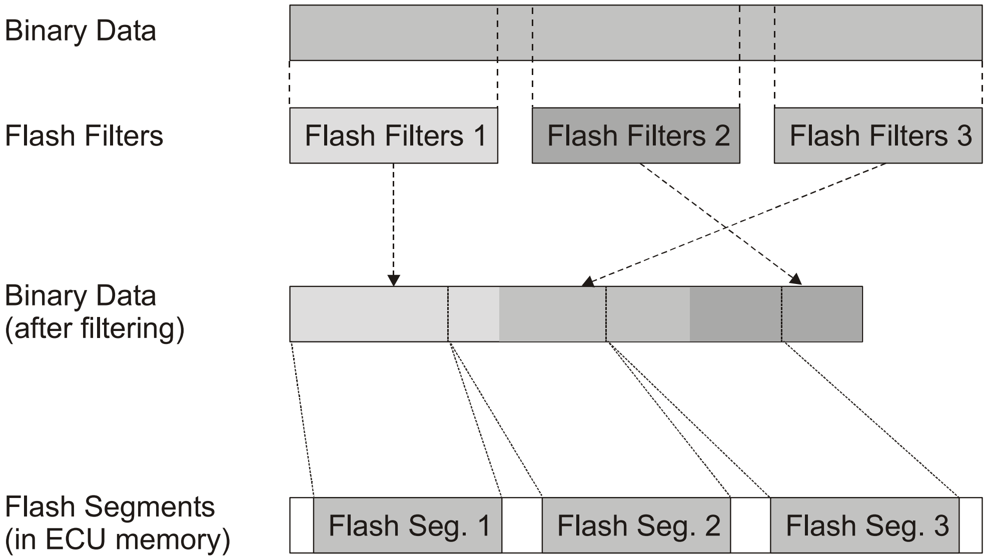


Figure 164 — Calculating FlashSegments from Binary Data

The procedure for calculating the MCDFlashSegments and their address information from a plain binary data file is defined as follows:

Step 1: Read the binary data from the MCDFlashDataBlock the MCDFlashSegments of which need to be calculated. The corresponding data can be obtained by reading the content of the file referenced from the MCDDbFlashData element attached to the MCDDbFlashDataBlock reachable via MCDFlashDataBlock::getDbObject().

Step 2: Apply all MCDDbFlashFilter objects referenced from the MCDDbFlashDataBlock in the order they are referenced to this binary data. The result of applying the flash filters to the binary data is a series of segments of binary data where the start address and size of each segment is defined as the start address and size information obtained from the corresponding FlashFilter. FlashFilters may not overlap. Their size shall be greater than zero. If no MCDDbFlashFilters are defined at the corresponding MCDDbFlashDataBlock, the whole binary data read in Step 1 is processed further.

Step 3: If any MCDDbFlashSegments are defined in the ODX data for the current MCDDbFlashDataBlock, then these FlashSegments need to be intersected with the segments calculated in Step 2 to result in the MCDFlashSegments to be attached to the current MCDFlashDataBlock. Otherwise, the MCDFlashSegments to be attached to the MCDFlashDataBlock have already been calculated in Step 2.

Step 4: In case MCDDbFlashSegments are defined in the ODX data for the current MCDDbFlashDataBlock, the address information of these MCDDbFlashDataBlocks shall be used as the target address within the target device (ECU). The first MCDFlashSegment starts at the byte position 0 in the intermediate flash data calculated in Step 2 and has the length returned by the method MCDDbFlashSegment::getUncompressedSize(). Within the D-server, the uncompressed size is calculated internally from the ODX attributes SOURCE-START-ADDR and SOURCE-END-ADDR of the corresponding SEGMENT in ODX. The next MCDFlashSegment starts always at the next byte after the last byte of the previous MCDFlashSegment and has the length calculated in the same manner as the length of the first one. If no MCDDbFlashSegment is described, the address information of the calculated MCDFlashSegments is defined as follows:

— FILTER-START as SOURCE-START-ADDR and FILTER-END as SOURCE-END-ADDR, if one or more `MCDDbFlashFilter` are defined at the corresponding `MCDDbFlashDataBlock`

— 00h as start address and the file length as UNCOMPRESSED-SIZE if there is no `MCDDbFlashFilter` defined in the corresponding `MCDDbFlashDataBlock`.

Step 5: The start address in the resulting `MCDFlashSegments` shall be interpreted as the target address in the target device (ECU). Before a `MCDFlashSegment` can be programmed into an ECU, the TARGET-ADDR-OFFSET defined in the corresponding FlashDataBlock shall be added to all address values. This address offset can be obtained by means of `MCDFlashDataBlock::getDbObject()::getAddressOffset()`.

Calculation of FlashSegments from HEX data files

If the flash data is represented by a HEX data file (Motorola-S format of Intel-Hex format), this data file does already contain address information and FlashSegment definitions. This information needs to be taken into account when calculating runtime FlashSegments. The corresponding algorithm is sketched in Figure 165 — Calculating FlashSegments from HEX Data. Again before a `MCDFlashSegment` can be programmed into an ECU, the TARGET-ADDR-OFFSET defined in the corresponding FlashDataBlock shall be added to all address values. This address offset can be obtained by means of `MCDFlashDataBlock::getDbObject()::getAddressOffset()`.

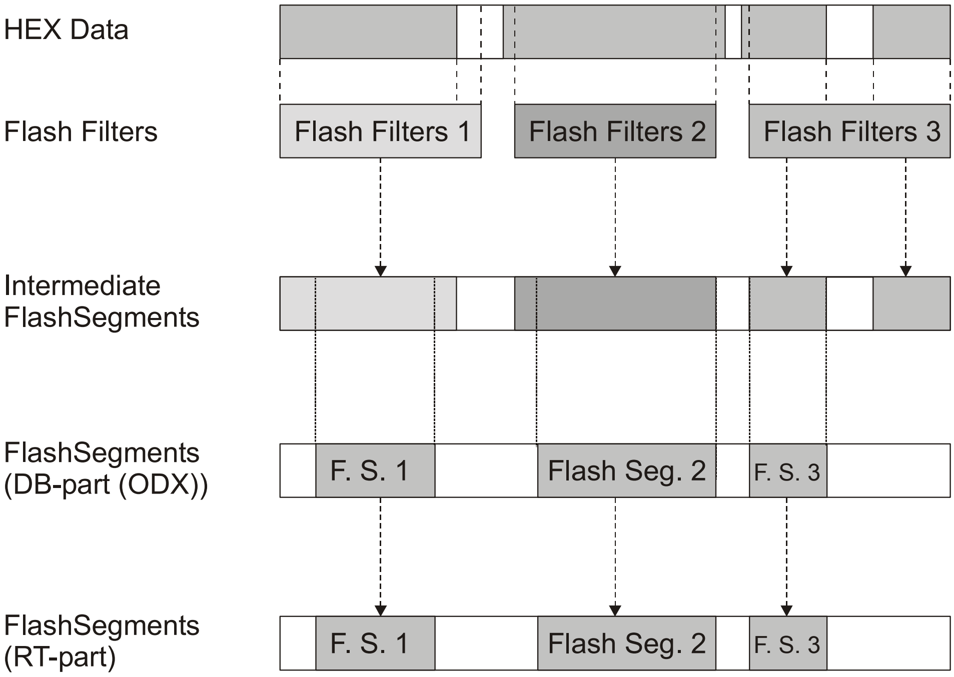


Figure 165 — Calculating FlashSegments from HEX Data

In case of a HEX data file, the data already includes the start addresses and the size information. As a result, FlashFilters potentially defined at a `MCDFlashDataBlock` currently processed need to be handled differently than in case of plain binary data. When processing HEX data, FlashFilters deliver start and end address of an area in the HEX data file, e.g. code / data / boot. The complete algorithm for calculating `MCDFlashSegments` in case of HEX data files is defined as follows:

Step 1: Read the hexadecimal data from the `MCDFlashDataBlock` the `MCDFlashSegments` of which need to be calculated. The corresponding data can be obtained by reading the content of the file referenced from the `MCDDbFlashData` element attached to the `MCDDbFlashDataBlock` reachable via `MCDFlashDataBlock::getDbObject()`.

Step 2: Convert the hexadecimal data into segments of binary data by considering the address information stored within the corresponding Intel-Hex or Motorola-S record. The result of the conversion is segments of binary data which potentially have gaps between them with respect to position in a virtual ECU memory.

Step 3: Apply all `MCDDbFlashFilter` objects referenced from the `MCDDbFlashDataBlock` in the order they are referenced to each of the temporary segments calculated in Step 2 (mind the address ranges). If a filter's address range overlaps with the address range of a temporary segment, then the current FlashFilter shall be intersected with the current temporary FlashSegment resulting in a new temporary FlashSegment. Then, the next FlashFilter is intersected with the original temporary FlashSegment if their address ranges overlap and so forth. As a result, the temporary FlashSegments calculated in Step 2 have been reduced or modified to a new set of FlashSegments. The address information of these FlashSegments corresponds to the intersected address information of temporary FlashSegment and applied FlashFilter. FlashFilters may not overlap. Their size shall be greater than zero. In addition, there may be no FlashFilters which cannot be mapped to at least one intermediate FlashSegment. If no `MCDDbFlashFilters` are defined at the corresponding `MCDDbFlashDataBlock`, the whole set of intermediate FlashSegments defined in Step 2 is processed further.

Step 4: If any `MCDDbFlashSegments` are defined in the ODX data for the current `MCDFlashDataBlock`, then these FlashSegments need to be intersected with the segments calculated in Step 3 to result in the `MCDFlashSegments` to be attached to the current `MCDFlashDataBlock`. Otherwise, the `MCDFlashSegments` to be attached to the `MCDFlashDataBlock` have already been calculated in Step 3.

Step 5: Each `MCDDbFlashSegment` used in step 4 shall reference an existing address interval with respect to the intermediate flash segments resulting from Step 3. If a certain address interval does not exist, e.g. because it did not come through the FlashFilters, the D-server should report an error message. The address information of the `MCDFlashSegments` resulting from Step 4 is given by the HEX formatted data. Before a `MCDFlashSegment` can be programmed into an ECU, the TARGET-ADDR-OFFSET defined in the corresponding FlashDataBlock shall be added to all address values. This address offset can be obtained by means of `MCDFlashDataBlock::getDbObject()::getAddressOffset()`.

Naming Calculated MCDFlashSegments

If no `MCDDbFlashSegments` have been used for the calculation of the runtime `MCDFlashSegments`, the ShortName of a `MCDFlashSegment` is constructed from the inline data of this FlashSegment in accordance with the pattern `#RTGen_Segment_<SourceAddressStart>_<SourceAddressEnd>`. Here, the address offset defined at the corresponding `MCDDbFlashDataBlock` is not considered. For example, a `MCDFlashSegment` starting at address 0x00000000 and ending at address 0x000000FF shall be named `#RtGen_Segment_0x00000000_0x000000FF`.

In case `MCDDbFlashSegments` have been used for the calculation of the runtime `MCDFlashSegments`, the ShortName of a `MCDFlashSegment` is constructed from the ShortName of the corresponding `MCDDbFlashSegment` in accordance with the pattern `#RTGen_Segment_<NameOfDbFlashSegment>`.

9.20.4 Management of ECU-MEMs

In ODX, an ECU-MEM references the ECU-VARIANTs and BASE-VARIANTs it is valid for through an ECU-MEM-CONNECTOR. The methods `MCDDbEcuMem::getLongName()`, `MCDDbEcuMem::getShortName()`, and `MCDDbEcuMem::getDescription()` return the corresponding values of an ECU-MEM-CONNECTOR in ODX.

At runtime, it shall be possible to load new FlashSessionDescs into an existing project. These new FlashSessionDescs need to be contained in a separate ECU-MEM container. Loading new ECU-MEMs into a running project, the D-server could be in any state except `eINITIALIZED`. The corresponding

MCDDLogicalLink could be in any state. If additional data is added, the system event onSystemDbEcuMemsModified (MCDDbEcuMems ecuMems) will be fired, so clients are informed about these changes. The collection MCDDbEcuMems shall be reloaded by the client at the MCDDbLocation or at the MCDDbProject.

New ECU-MEMs including their contained FlashSessionDescs can be loaded into a MCDDbProject temporarily or they can be added to a project configuration permanently. Adding an ECU-MEM temporarily to a project means that the added ECU-MEM will not be present any more after the same project gets deselected and re-selected again. That is, temporarily added ECU-MEMs need to be removed from a project when this project is deselected. Adding an ECU-MEM permanently to a project means that this ECU-MEM will still be available after this project has been deselected and re-selected again. This might include copying the corresponding ECM-MEM file to the D-server's data space for the corresponding project.

It is possible to list all ECU-MEMs within the current project by calling the method MCDDbProject::getDbEcuMems()at the corresponding MCDDbProject.

To list all ECU-MEMs which could be loaded into a project but which have not been loaded into the project yet, the method MCDDbProjectConfiguration::getAdditionalEcuMemNames(): A_ASCIISTRINGs can be used. This method issues the D-server to search for ECU-MEM containers within the boundaries defined by the server's paths. The ECU-MEM containers found are matched against those already being part of the current project. Finally, only those ECU-MEM containers are presented as result of the method, which are not part of the project yet.

NOTE Two calls of the method can deliver different results.

By means of the methods

- MCDDbProject::loadNewEcuMem(MCDDatatypeShortName ecuMemName, A_BOOLEAN permanent=false)
- MCDDbProject::loadNewEcuMemsByFilename(A_ASCIISTRING filename, A_BOOLEAN permanent=false)

an ECU-MEM, which is not part of the currently active project, can be loaded into the project. The parameter permanent controls whether this ECU-MEM is permanently added to the project or not.

The method MCDDbProject::loadNewEcuMem() throws an MCDParameterizationException with error code ePAR_ITEM_NOT_FOUND in case that a file for the MCDDatatypeShortname value supplied as parameter to this method was not found in the database.

The methods MCDDbProject::loadNewEcuMem() and MCDDbProject:: loadNewEcuMemsByFilename() throw a MCDProgramViolationException with error code eRT_ELEMENT_ALREADY_EXIST if at least one ECU-MEM with the same ShortName as the one which is to be imported already exists.

By means of the method MCDDbProject::removeEcuMemByName(MCDDatatypeShortName ecuMemName) it is possible to remove an ECU-MEM from the currently active project configuration which has not been permanently added to this project configuration before.

9.20.5 Physical Memories

The class MCDDbPhysicalMemory (PHYS-MEM) is used to describe the physical memory layout of an ECU (see Figure 166 — Flash Programming – Physical Memory). This memory description may be required by the FlashJob to check whether the logical FlashSegments – represented by a collection of MCDFlashSegment objects – do exactly fit to the physical segments defined by PhysicalSegments referenced from a PhysicalMemory. In this part of ISO 22900, the PhysicalSegments a PhysicalMemory is decomposed into are described by objects of type MCDDbPhysicalSegment (PHYS-SEGMENT).

For each PhysicalSegment the FlashJob needs the ODX members START-ADDRESS, END-ADDRESS or SIZE, BLOCKSIZE and FILLBYTE described below. A PhysicalMemory object involves the standard members SHORT-NAME, LONG-NAME, and DESC.

NOTE `MCDDbFlashDataBlocks` (described above) are independent from any `MCDDbPhysicalMemory`.

For a `MCDDbPhysicalSegment`, the attribute

— FILLBYTE describes the byte for filling empty areas to complete the physical segment, i.e. it fills gaps between logical segments in the flash data, if necessary. The value of this attribute can be obtained by means of the method `MCDDbPhysicalSegment::getFillByte()`.

— BLOCK-SIZE can be used by the FlashJob to enable parallel programming of memory sub-units to increase the performance of the flash process. The value of this attribute can be obtained by means of the method `MCDDbPhysicalSegment::getBlockSize()`.

— START-ADDRESS describes the first valid address of the segment. The value of this attribute can be obtained by means of the method `MCDDbPhysicalSegment::getStartAddress()`.

— END-ADDRESS defines the last valid address that belongs to the current segment. In ODX, the attribute SIZE can be used as alternative to END-ADDRESS. It defines the size of the segment in bytes. However, the value for the attribute END-ADDRESS can be calculated from the values of START-ADDRESS and SIZE. The value of the attribute END-ADDRESS can be obtained by means of the method `MCDDbPhysicalSegment::getEndAddress()`.

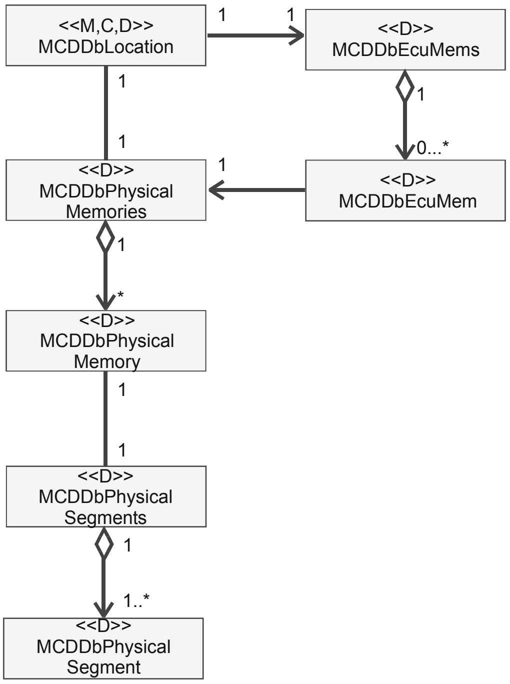


Figure 166 — Flash Programming – Physical Memory

9.20.6 Executing flash sessions

9.20.6.1 Flash session execution basics

In order to execute a FlashSession, the following prerequisites need to be fulfilled:

— An appropriate `MCDDbFlashSessionDesc` needs to be selected for the FlashSession. A runtime instance of this `MCDDbFlashSessionDesc` needs to be created at the respective LogicalLink. The resulting `MCDFlashSessionDesc` object will contain a runtime instance of the FlashSession.

— A runtime instance of the `MCDDbFlashJob` referenced from the `MCDDbFlashSessionDesc` needs to be available at the respective LogicalLink. The database template for the FlashJob to be used to process a `MCDFlashSessionDesc` can be obtained from the corresponding `MCDDbFlashSessionDesc` object (`MCDDbFlashSessionDesc::getDbFlashJob()`). This `MCDFlashJob` is not created automatically as part of the `MCDFlashSessionDesc`. The reason is that a `MCDFlashJob` can potentially by used to program different FlashSessions. That is, a single `MCDDbFlashSession` can be referenced from several `MCDDbFlashSessionDesc` objects, e.g. at the same `MCDDbLocation`, where each of these `MCDDbFlashSessionDesc` objects references the same `MCDDbFlashJob`.

— After a new `MCDFlashJob` has been created from the `MCDDbFlashJob`, e.g. by means of `MCDDiagComPrimitives::addByDbObject(MCDDbFlashJob,…)`, the `MCDFlashSessionDesc` object to be processed by the FlashJob can be passed to the corresponding `MCDFlashJob` object by means of one of its methods

  — MCDFlashJob::setFlashSessionDesc(MCDFlashSessionDesc flashSessionDesc)

  — MCDFlashJob::setFlashSessionDescByFlashKey(A_ASCIISTRING flashKeyString)

  — MCDFlashJob::setFlashSessionDescByName(MCDDbEcuMem ecuMem, MCDDataTypeShortname sessionDesc

A `MCDFlashJob` can have RequestParameters just like any DiagComPrimitive. As a result, it might be necessary to provide reasonable values to these RequestParameters prior to execution of the FlashJob. Furthermore, it is only possible to set one FlashSessionDesc at a `MCDFlashJob` at a time. That is, if multiple FlashSessionDescs are to be processed by the same FlashJob, the FlashJob needs to be parameterised and executed for each of these FlashSessionDescs separately.

To start the download of the FlashSession referenced from a FlashSessionDesc, the `executeSync()` or `executeAsync()` method of the respective `MCDFlashJob` can be called. If more than one FlashSessionDesc is to be programmed, the prerequisites described above need to be fulfilled for every FlashSessionDesc and for every FlashJob involved. Furthermore, the principle sequence of execution as described above needs to be performed for every FlashSessionDesc separately. If FlashSessionDescs need to be sorted, e.g. in accordance with priority, the sorting shall be performed by the client application. The client application can obtain the priority of every FlashSessionDesc from the project's data basis via the D-server by means of the method `MCDDbFlashSessionDesc::getPriority()`. Here, lower values mean higher priority. If two `MCDDbFlashSessionDesc` have the same priority, the order is undefined. If the priority is not defined, the default 100 is to be assumed by the client application.

Within a FlashJob, the core logic of the flash programming sequence for a specific ECU is implemented. That is, the FlashJob defines which operations need to take place for programming a specific ECU in which order. This flash programming sequence can comprise, e.g.

— security access

— session change into a diagnostic session

— download of every data block within the FlashSessionDesc provided as input to the FlashJob

— checksum calculation and comparison

— ECU reset

The FlashJob has access to all information to generate all telegrams required in order to program flash data into an ECU or to upload flash data from an ECU. This includes access to the LogicalLink in order to create the required DiagComPrimitives for communicating with the ECU and for data transfer. The binary data for every FlashSegment can be accessed by the method `getBinaryData()` at the `MCDDbFlashSegment` interface. If the data is in any other format than plain binary (e.g. Motorola-S or Intel-Hex), the D-server needs to convert the data into plain binary format prior to returning it to the caller of the method `getBinaryData()`.

For simplification and reuse, it is possible to create sub-jobs which, e.g. program a single data block or only a single segment. The MCD-server API provides all mechanisms for the FlashJob to create instances of sub-jobs in its body. For this purpose, all these sub-jobs used for ECU (re-)programming should not be directly executed by a client application. Therefore, they shall be marked as not executable in the list of DiagComPrimitives available at a LogicalLink. As a result, the database part of the MCD-server API still lists these jobs but an execution of a runtime object of these jobs outside of a `MCDFlashJob` will be denied by the D-server and an exception of type `MCDProgrammViolationException` with error code `eRT_INVALID_OPERATION` will be thrown.

9.20.6.2 Flash job basics

A FlashJob (special type of SINGLE-ECU-JOB) is a new class of Java job derived from the abstract interface `MCDJob`. FlashJobs are used to execute a flash (re)programming session (a so-called FlashSession) within the D-server (see Figure 167 — Client application view). A FlashSession can be used as a download session as well as an upload session. The direction (upload vs. download) is determined by the FlashSessionDesc referencing the FlashSession. In the case the FlashSession is used as a download session, the session (re-)programs an ECU. In case the FlashSession is used as an upload session, the flash data is read from the ECU.

The FlashSession to be processed within a FlashJob is chosen by means of a so-called `MCDFlashSessionDesc` object. A `MCDFlashSessionDesc` references the appropriate `MCDFlashSession` object and the FlashJob responsible for processing (via its corresponding `MCDDbFlashSessionDesc` object). Moreover, the `MCDDbFlashSessionDesc` an `MCDFlashSessionDesc` is based on defines whether an ECU is (re-)programmed (`MCDDbFlashSessionDesc::isDownload == true`) or whether the data described by the associated FlashSession is read from the ECU (`MCDDbFlashSessionDesc::isDownload == false`).

At runtime, a new `MCDFlashJob` instance needs to be created from the `MCDDbFlashJob` referenced from a `MCDDbFlashSessionDesc`. Then, this `MCDFlashJob` needs to be provided with the FlashSessionDesc to be processed at the Jobs execution time. For this purpose, e.g. the method `MCDFlashJob::setFlashSessionDesc()` can be used. Only a single FlashSessionDesc can be set for one FlashJob at the same time.

Please note: It is recommended that a MCDFlashSessionDesc is either locked by the client application or created with a cooperation level of eNO_COOPERATION or eREAD_ONLY prior to being filled into a MCDFlashJob. This prevents other clients from modifying the MCDFlashSessionDesc while it is being used by the flash job.

If several FlashSessions need to be processed for successfully (re-)programming an ECU, a priority information can be obtained from the corresponding `MCDDbFlashSessionDesc` objects (`getPriority()`). This information can be used by the client application to sort the FlashSessionDescs by priority before setting and processing them one after the other using the associated FlashJobs.

The java interface of flash jobs (`MCDFlashJob`) extends the java interface of normal diagnostic jobs (`MCDSingleEcuJob`) by a parameter for a `MCDFlashSessionDesc` object, i.e. the flash job's code template is defined as described in Clause C.3. A sequence diagram from the FlashJob point of view is given in Figure 168 — FlashJob view (Part 1) and Figure 169 — FlashJob view (Part 2).

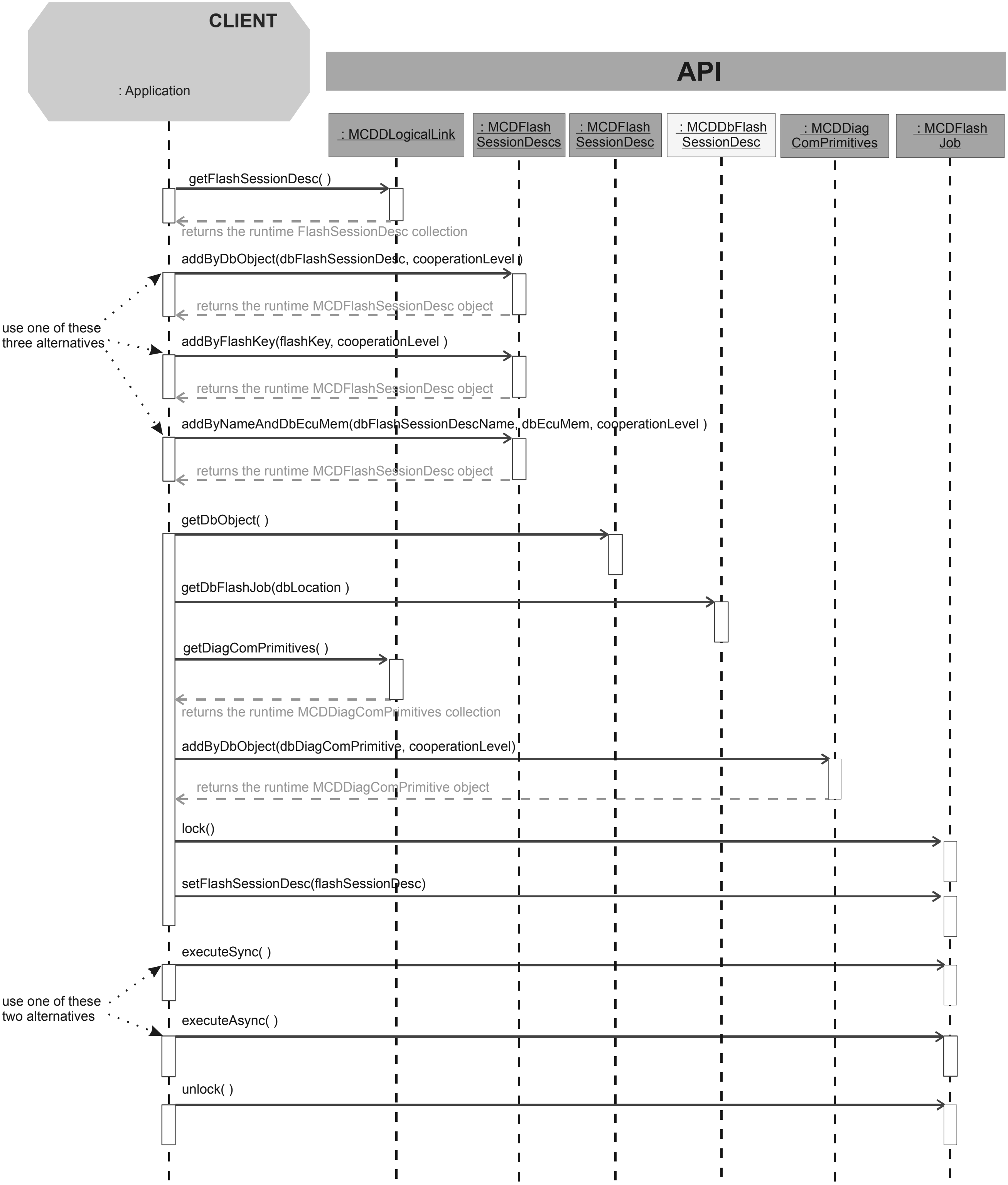

CLIENT
: Application
API
: MCDDLogicalLink
: MCDFlash SessionDescs
: MCDFlash SessionDesc
: MCDDbFlash SessionDesc
: MCDDiag ComPrimitives
: MCDFlash Job
getFlashSessionDesc( )
returns the runtime FlashSessionDesc collection
addByDbObject(dbFlashSessionDesc, cooperationLevel)
returns the runtime MCDFlashSessionDesc object
use one of these three alternatives
addByFlashKey(flashKey, cooperationLevel)
returns the runtime MCDFlashSessionDesc object
addByNameAndDbEcuMem(dbFlashSessionDescName, dbEcuMem, cooperationLevel)
returns the runtime MCDFlashSessionDesc object
getDbObject( )
getDbFlashJob(dbLocation)
getDiagComPrimitives( )
returns the runtime MCDDiagComPrimitives collection
addByDbObject(dbDiagComPrimitive, cooperationLevel)
returns the runtime MCDDiagComPrimitive object
lock()
setFlashSessionDesc(flashSessionDesc)
executeSync( )
use one of these two alternatives
executeAsync( )
unlock( )

Figure 167 — Client application view

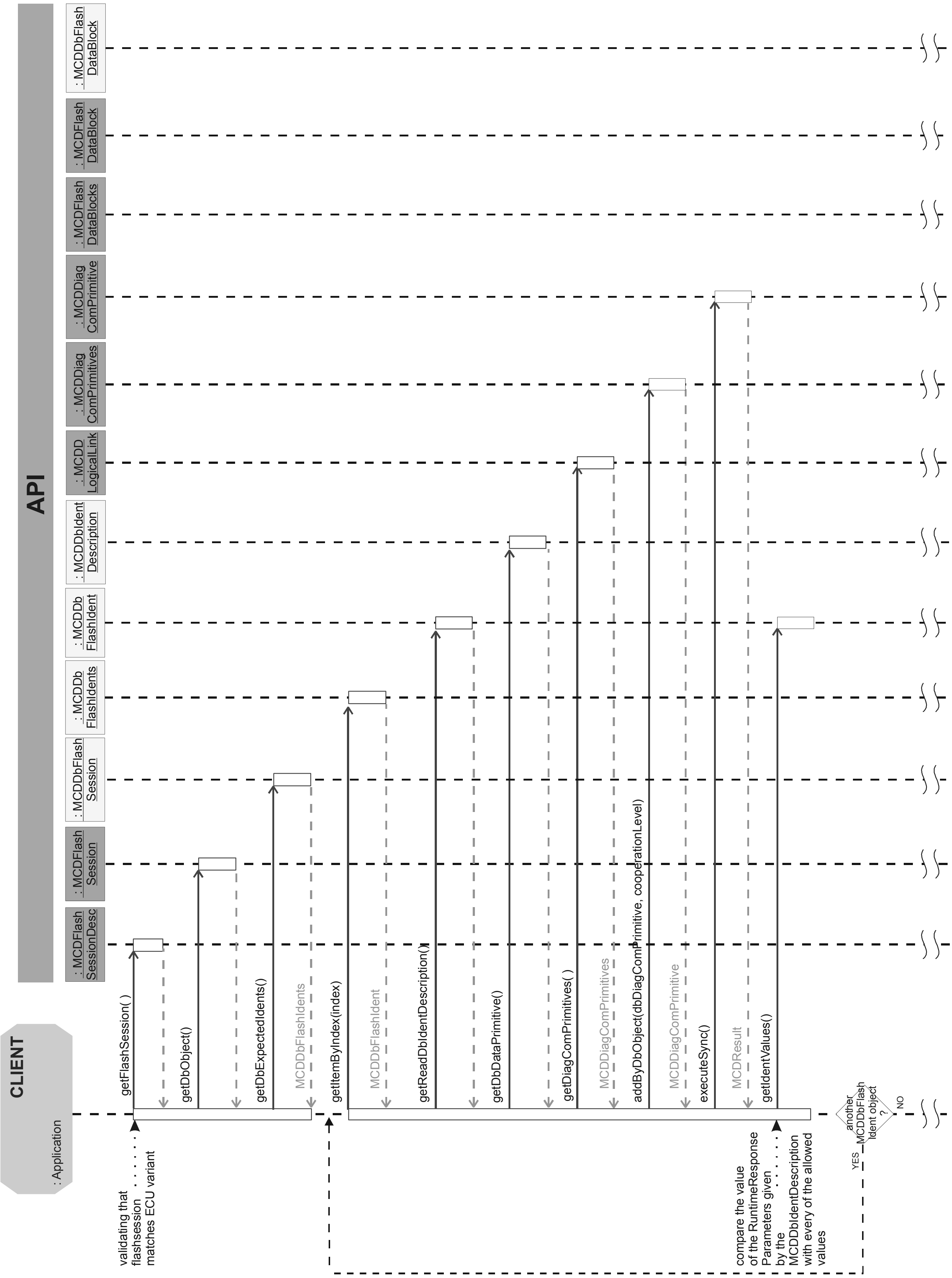


Figure 168 — FlashJob view (Part 1)

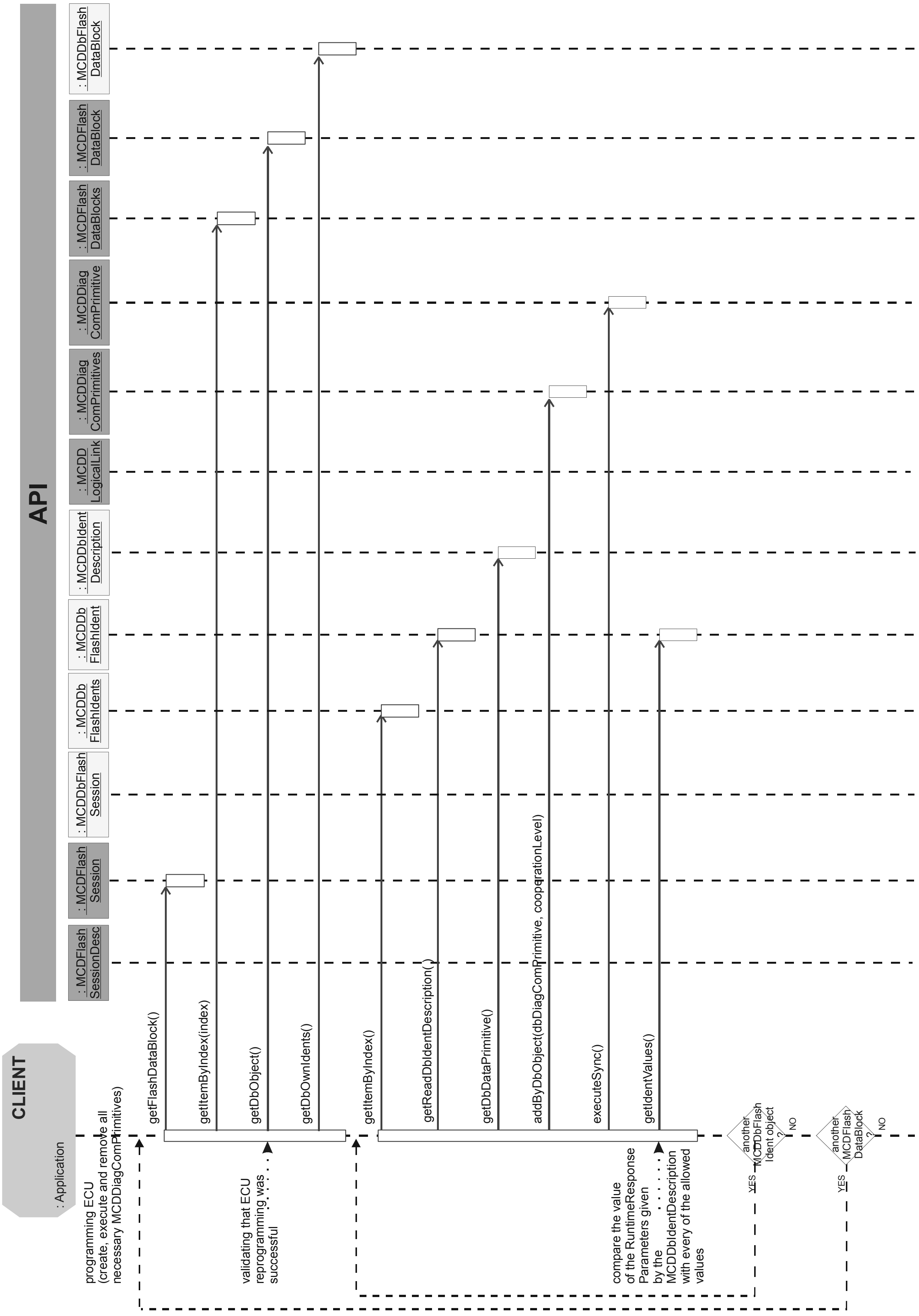


Figure 169 — FlashJob view (Part 2)

9.20.6.3 Uploading Flash Data from an ECU

For uploading the current flash data from an ECU, an appropriate MCDDbFlashSessionDesc needs to be selected in the DB-part of the flash programming module. The method MCDDbFlashSessionDesc::isDownload() of the selected FlashSessionDesc returns false if it can be used for upload purposes.

In the second step, a runtime instance of the MCDDbFlashJob referenced by the selected MCDbFlashSessionDesc needs to be created at a MCDDLogicalLink pointing to the MCDDbLocation the MCDDbFlashSessionDesc has been selected from. The instance of the FlashJob can be created, e.g. by means of the method MCDDiagComPrimitives::addByDbObject() at the collection of DiagComPrimitives obtainable from the selected MCDDLogicalLink.

Next, a runtime instance of the selected MCDDbFlashSessionDesc needs to be created, e.g. by means of the method MCDFlashSessionDescs::addByDbObject(MCDDbFlashSessionDesc). Here, it needs to be ensured that the new instance of the selected MCDDbFlashSessionDesc is created at the collection of FlashSessionDescs obtained from the same MCDDLogicalLink the instance of the FlashJob has been created at.

With the creation of the runtime instance of the selected MCDDbFlashSessionDesc, the D-server creates the complete substructure of runtime objects to be contained in the new MCDFlashSessionDesc – MCDFlashSession, MCDFlashDataBlocks, etc. This also includes the calculation of the MCDFlashSegment objects with respect to 9.20.3.5.

Please note: It is recommended that a MCDFlashSessionDesc is either locked by the client application or created with a cooperation level of eNO_COOPERATION or eREAD_ONLY prior to being filled into a MCDFlashJob. This prevents other clients from modifying the MCDFlashSessionDesc while it is being used by the flash job.

Finally, the MCDFlashSessionDesc is supplied as input to the MCDFlashJob created before (MCDFlashJob::setFlashSessionDesc(FlashSessionDesc)). Then, the MCDFlashJob can be executed, e.g. by means of MCDFlashJob::executeSync(). This triggers the execution of the Java code representing the FlashJob in the D-server's job engine. The FlashJob's Java code shall contain all statements required to

— create, execute, and remove the DiagComPrimitives to upload flash data from the target ECU, to

— process the MCDFlashSessionDesc provided as input, to

— check own identification values and expected identification values (see 9.20.3.4), and to

— fill the MCDFlashSegment objects contained in this MCDFlashSessionDesc.

Before a MCDFlashSegment can be uploaded from an ECU, the TARGET-ADDR-OFFSET defined in the corresponding FlashDataBlock shall be added to all address values. This address offset can be obtained by means of MCDFlashDataBlock::getDbObject()::getAddressOffset(). Furthermore, the flash data stored in a MCDFlashSegment by the FlashJob is not written to a file by the D-server. The file referenced from a MCDDbFlashDataBlock is only used to calculate number and size of the MCDFlashSegments. The reason is that the same MCDDbFlashSession can be used for downloading and for uploading flash data by referencing this FlashSession from two different MCDDbFlashSessionDescs – one for download and one for upload. Instead of the D-server, the client application is responsible to store the uploaded FlashSegments properly.

9.20.6.4 Downloading Flash Data to an ECU

For downloading the flash data obtained from a flash data file to an ECU, an appropriate MCDDbFlashSessionDesc needs to be selected in the DB-part of the flash programming module. The

MCDDbFlashSessionDesc::isDownload() of the selected FlashSessionDesc returns true if it can be used for download purposes.

In the second step, a runtime instance of the MCDDbFlashJob referenced by the selected MCDDbFlashSessionDesc needs to be created at a MCDDLogicalLink pointing to the MCDDbLocation the MCDDbFlashSessionDesc has been selected from. The instance of the FlashJob can be created, e.g. by means of the method MCDDiagComPrimitives::addByDbObject() at the collection of DiagComPrimitives obtainable from the selected MCDDLogicalLink.

Next, a runtime instance of the selected MCDDbFlashSessionDesc needs to be created, e.g. by means of the method MCDFlashSessionDescs::addByDbObject(MCDDbFlashSessionDesc). Here, it needs to be ensured that the new instance of the selected MCDDbFlashSessionDesc is created at the collection of FlashSessionDescs obtained from the same MCDDLogicalLink the instance of the FlashJob has been created at.

With the creation of the runtime instance of the selected MCDDbFlashSessionDesc, the D-server creates the complete substructure of runtime objects to be contained in the new MCDFlashSessionDesc – MCDFlashSession, MCDFlashDataBlocks, etc. This also includes the calculation of the MCDFlashSegment objects with respect to 9.20.3.5.

Please note: It is recommended that a MCDFlashSessionDesc is either locked by the client application or created with a cooperation level of eNO_COOPERATION or eREAD_ONLY prior to being filled into a MCDFlashJob. This prevents other clients from modifying the MCDFlashSessionDesc while it is being used by the flash job.

Finally, the MCDFlashSessionDesc is supplied as input to the MCDFlashJob created before (MCDFlashJob::setFlashSessionDesc(FlashSessionDesc)). Then, the MCDFlashJob can be executed, e.g. by means of MCDFlashJob::executeSync(). This triggers the execution of the Java code representing the FlashJob in the D-server's job engine. The FlashJob's Java code shall contain all statements required to

- create, execute, and remove the DiagComPrimitives to download flash data from the target ECU, to
- process the MCDFlashSessionDesc provided as input, to
- check own identification values and expected identification values (see 9.20.3.4), to
- read data from the MCDFlashSegment objects contained in this MCDFlashSessionDesc, to
- perform checksum calculations (see 9.20.6.5), and to
- trigger or execute integrity and/or authenticity checks of the flash data (see 9.20.3.1).

Before a MCDFlashSegment can be programmed into an ECU, the TARGET-ADDR-OFFSET defined in the corresponding FlashDataBlock shall be added to all address values. This address offset can be obtained by means of MCDFlashDataBlock::getDbObject()::getAddressOffset().

An example for downloading flash data is given in Figure 170 — Download flash data.

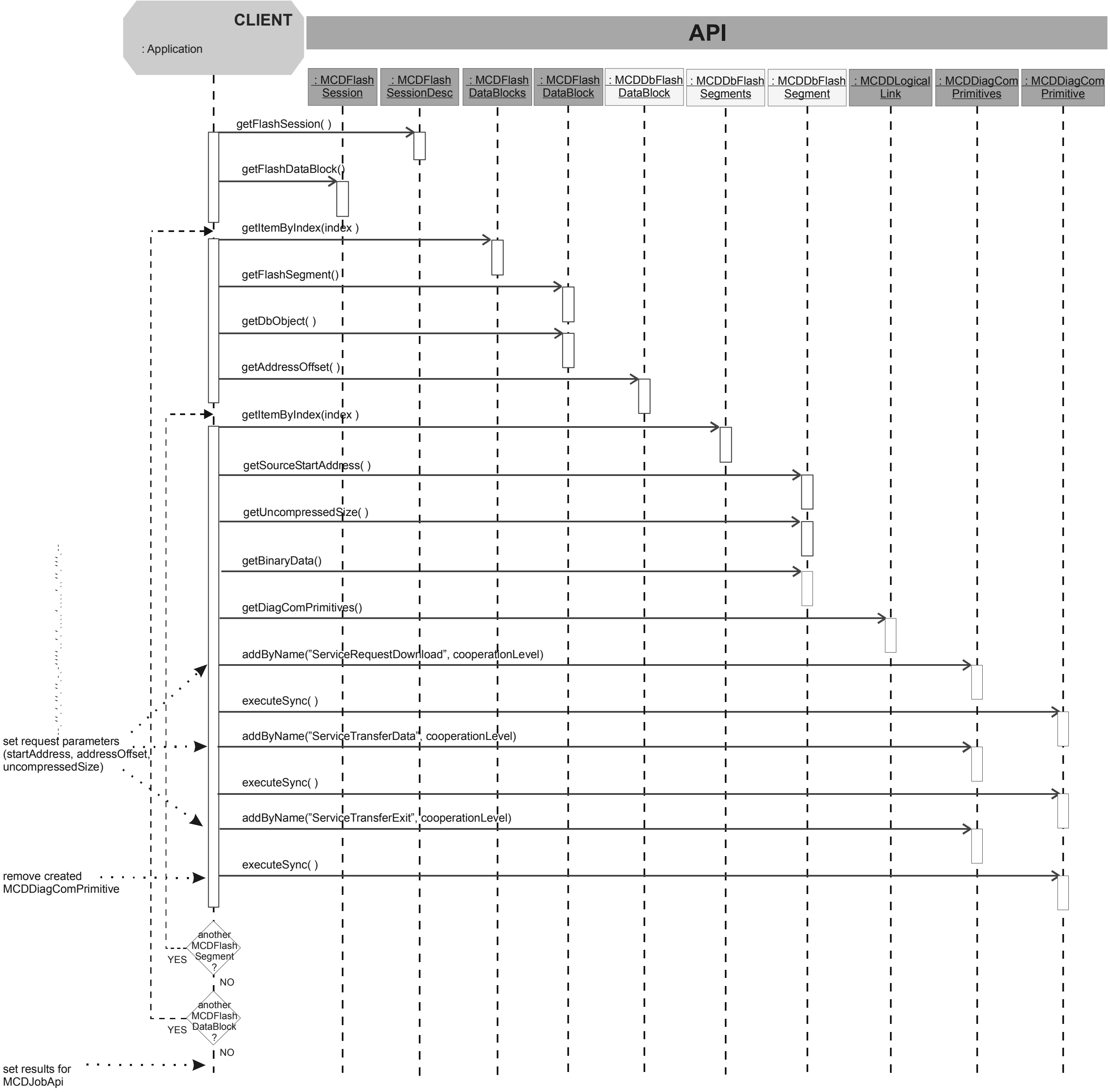


Figure 170 — Download flash data

9.20.6.5 Checksum calculation in Flash Jobs

If checksum monitoring is performed in a FlashJob, the respective algorithm is identified by the name returned by the method `getChecksumAlgorithm()` at each instance of type `MCDDbFlashCheckSum` referenced from the `MCDDbFlashSession` the `MCDFlashSession` currently processed in the FlashJob is based on. This checksum algorithm is to be used by the FlashJob to calculate the checksum which should be identical with the value returned by `getChecksumResult()` at the same `MCDDbFlashCheckSum`.

The address range to be considered for the checksum calculation is specified by the values returned by the methods `MCDDbFlashCkeckSum::getCompressedSize()`, `getUnCompressedSize()`, `getSourceStartAddress()`, and `getSourceEndAddress()`. The source start address denotes the start of an address range in the source data. If the target address inside the ECU differs from the source address of the source data, the address offset which can be obtained from every FlashDataBlock inside a FlashSession is used to describe the offset. The source start address is the first address to be included in a checksum calculation. The end of the address range in target device (ECU) is defined by source end address. The compressed size holds the size of the compressed FlashSegment. Alternatively, the uncompressed size can be used to calculate the checksum.

If the referenced area in the flash data is not contiguous, fill bytes need to be inserted to fill the gaps before the checksum is calculated for this part of the flash data. The fill byte to be used is returned by the method `MCDDbFlashCheckSum::getFillByte()`. Summarizing, fill bytes are used to fill gaps between the address ranges of the FlashSegments (SEGMENT) in the flash data for checksum calculation of the given address range.

9.21 Library

Library elements are used within ODX to specify additional program code which is used ('included' in Java terms) by the executable code referenced by the CODE-FILE attribute of a PROG-CODE instance. A data element of type PROG-CODE is used by ODX to specify Java program code which is executable by the D-server. Libraries are defined by LIBRARY elements in ODX and can be referenced by one or multiple PROG-CODE elements to extend the classpath that is associated with that PROG-CODE. A PROG-CODE shall be executed in the classpath environment defined by the program code (class file or JAR) referenced by the CODE-FILE attribute at PROG-CODE, as well as the program code referenced by CODE-FILE attribute of the referenced LIBRARY elements. This also applies if the PROG-CODE execution is embedded within another program code (Java job) execution. In such a case, the calling job and the called job need to be executed within different classloader environments if a different class context is defined by the associated ODX data.

In ODX, LIBRARY definitions are associated with a DIAG-LAYER and can be referenced by `MCDDbCodeInformation` (ODX: PROG-CODE) instances at `MCDDbSingleEcuJob` (ODX: SINGLE-ECU-JOB), `MCDDbMultipleEcuJob` (ODX: MULTIPLE-ECU-JOB) and COMPU-METHOD of category COMPUCODE. ODX also allows library elements to point to arbitrary other types of data (e.g. it is possible to reference a dynamic system library (*so* or *dll*) from a LIBRARY element. These cases cannot be covered in the scope of this part of ISO 22900, and D-server behaviour shall be defined in a customer- and project-specific way.

The following resources of program code shall be included into the Java classpath environment for a PROG-CODE (MCDDbCodeInformation and DOPs referencing Java code):

— Resources referenced by the PROG-CODE itself.

— Resources referenced by any of the LIBRARY elements that are referenced by PROG-CODE.

— Permitted packages of the standard JRE.

— MCD-3D API (except for COMPUCODE)

For performance improvements a classloader may be reused by any PROG-CODE which defines the same Java class environment.

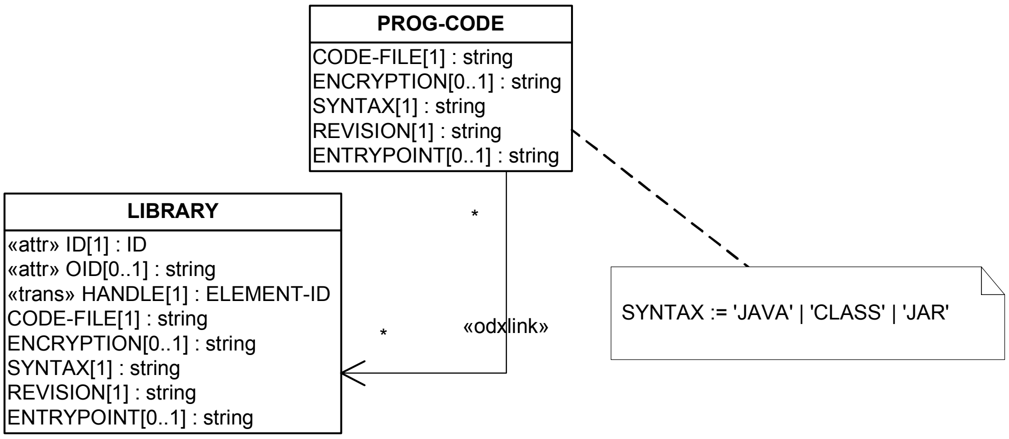

PROG-CODE
CODE-FILE[1] : string
ENCRYPTION[0..1] : string
SYNTAX[1] : string
REVISION[1] : string
ENTRYPOINT[0..1] : string
LIBRARY
«attr» ID[1] : ID
«attr» OID[0..1] : string
«trans» HANDLE[1] : ELEMENT-ID
CODE-FILE[1] : string
ENCRYPTION[0..1] : string
SYNTAX[1] : string
REVISION[1] : string
ENTRYPOINT[0..1] : string
*
*
«odxlink»
SYNTAX := 'JAVA' | 'CLASS' | 'JAR'

Figure 171 — Relation between Library and Prog-code

In contrast to “CLASS” and “JAR“, the support of Java source code is optional for a D-server. Resources referenced by PROG-CODE or LIBRARY with SYNTAX=“JAVA” would either have to be compiled at runtime, or during a pre-processing step transforming the ODX resources to a proprietary runtime format.

9.22 Java Jobs

9.22.1 General

A D-server provides the capability to execute a complex sequence of diagnostic commands. Such a sequence is called a job. Such jobs are deployed together with the ODX data used by the D-server. The implementation language for jobs is Java.

9.22.2 General information Java Jobs

Generally, jobs are handled like diagnostic services. However, jobs cannot be marked as cyclic in ODX. A job has associated meta information about input and output parameters. During its execution, it can return 0..n intermediate results, and one final result. Intermediate results can only be delivered until the final result has been sent. This mechanism allows the user to e.g. implement progress bars for a user interface, indicating the state of the work that is being done within a job.

Jobs can be executed on different logial links at the same time by the D-server, therefore job code shall be reentrant. Also, special care shall be taken when threading mechanisms are used inside job code, to ensure that a job leaves its environment in a well-defined state when it finishes execution.

Using any kind of busy-waiting techniques within a job is considered harmful and should be avoided. Polling decreases system performance and stability, and can potentially harm other system components, as well as the D-server itself. The D-server job API class provides the `MCDJobApi::sleep(A_UINT32 time_in_milliseconds)` method that allows job code to yield the D-server when it is waiting for external events to continue its work. When this method is called by a job client, the D-server will make the client wait for at least the specified time interval. Although it is not guaranteed that the time interval is met exactly, the call will always sleep for at least the specified time.

Every Job is executed in a well defined classpath environment. The classpath environment of a Job is individually defined by the Job declaration within the ODX data. Further information to classpath definition and classloader handling with Jobs can be found in 9.21.

Further information pertaining to the development practices for Java jobs can be found in 9.22.4.2 "Development of Java Jobs”.

9.22.3 Types of Java Jobs

9.22.3.1 General

There are different classes of jobs for performing different tasks. These job classes can be differentiated in ODX either by their describing entities (SINGLE-ECU-JOB, MULTIPLE-ECU-JOB), or by the value of the semantic attribute that a SINGLE-ECU-JOB inherits from the DIAG-COMM element. This subclause introduces the available job classes and their usage.

9.22.3.2 Single ECU Job

Single-ECU Jobs are used to implement a complex diagnostic sequence that involves only one ECU. Therefore, Single-ECU Jobs shall be assigned to one location at runtime. For this reason, it is not possible to use functional addressing with a Single-ECU Job. As a result, this kind of job can only be executed on the *ECU Base Variant* or *ECU Variant* level. This restriction results from the ODX definition of Single ECU Jobs, which states that a Single ECU Job "[…] contains calls to services of one and only one dedicated ECU […]" (see ASAM MCD 2 D ODX, 7.3.5.7). For the parameter definitions of Single-ECU Jobs, the mechanisms of inheritance, overwriting and elimination are supported in ODX (see ASAM MCD 2 D ODX).

However, instances of Single-ECU Jobs may still be contained in the list of available DiagComPrimitives of a DbLocation for inheritance reasons. In this case, the Single-ECU Job is, e.g. defined in a Functional Group for inheritance to all ECU Base Variants contained in this Functional Group. The Single-ECU Job is visible in the list of DiagComPrimtives of this Functional Group in the DB-part of the D-server. But, the D-server does not allow to create an instance of the Single-ECU Job on a Logical Link to the Functional Group in the runtime part.

Within a Single-ECU Job, no logical links can be created. The location on which the job is executed is an input parameter for its execute-method. Because of this, Single-ECU Jobs can be executed on different locations which have the same service names (and where the services have the same request and result structures), for example on *Base-Variant* and *Variant* locations.

9.22.3.3 Flash Job

A Flash Job (special type of `SINGLE-ECU-JOB`) is a class of Java job derived from the abstract interface `MCDJob`. Flash Jobs are used to execute a flash (re-)programming session (a so-called FlashSession) within the D-server. There can be download sessions and upload sessions (see 9.20.2.4). In the first case, the session (re-)programs an ECU. In the latter case, the flash data is read from the ECU. The FlashSession to be processed within a FlashJob is chosen by means of a so-called `MCDFlashSessionDesc` object. A `MCDFlashSessionDesc` references the appropriate `MCDFlashSession` object and the Flash Job responsible for processing (via its corresponding `MCDDbFlashSessionDesc` object). Moreover, the `MCDDbFlashSessionDesc`, on which a `MCDFlashSessionDesc` is based, defines whether an ECU is (re-)programmed (`MCDDbFlashSessionDesc::isDownload == true`) or whether the data described by the associated FlashSession is read from the ECU (`MCDDbFlashSessionDesc::isDownload == false`).

At runtime, a `MCDFlashJob` instance needs to be created from the `MCDDbFlashJob` referenced from an `MCDDbFlashSessionDesc`. Then, this `MCDFlashJob` needs to be provided with the FlashSessionDesc to be processed at the job's execution time. For this purpose, e.g. the method `MCDFlashJob::setFlashSessionDesc(…)` can be used. Only a single FlashSessionDesc can be set for one FlashJob at the same time.

If several FlashSessions need to be processed for successfully (re-)programming an ECU, a priority information can be obtained from the corresponding `MCDDbFlashSessionDesc` objects (`getPriority()`). This information can be used by the client application to sort the FlashSessionDescs by priority before setting and processing them one after the other using the associated Flash Jobs.

9.22.3.4 Security access job

A SecurityAccessJob (special type of SINGLE-ECU-JOB) is a new class of Java job derived from the abstract interface MCDJob.

ECUs often restrict access to functionality like ECU reprogramming or variant coding. To gain access to restricted functionality, a tester shall switch a diagnostic session to a special access level, which is usually obtained by correctly implementing some kind of challenge-response algorithm (commonly called seed & key authentication). Security Access Jobs can be used to alter the access level of an ECU. By bundling this kind of functionality inside a Java job and possibly assorted native libraries, security access handling can be distributed among different business units in a unified way.

Security Access Jobs are allowed to use the JNI-mechanism of the JRE in order to access external libraries. These external libraries can provide the necessary functionality to execute algorithms (e.g. seed & key) for authentication purposes. For each JNI-class (Java Native Interface for a system library) referenced by a job, there should be exactly one library conforming to this interface (1-to-1 relation). Furthermore, the library should be located in the same directory as the JNI-class. This results in a controlled and defined environment for the Java Runtime Engine and avoids search problems. Meta information on the referenced libraries can be obtained from the ODX data via the `MCDDbJob::getDbCodeInformations()` call.

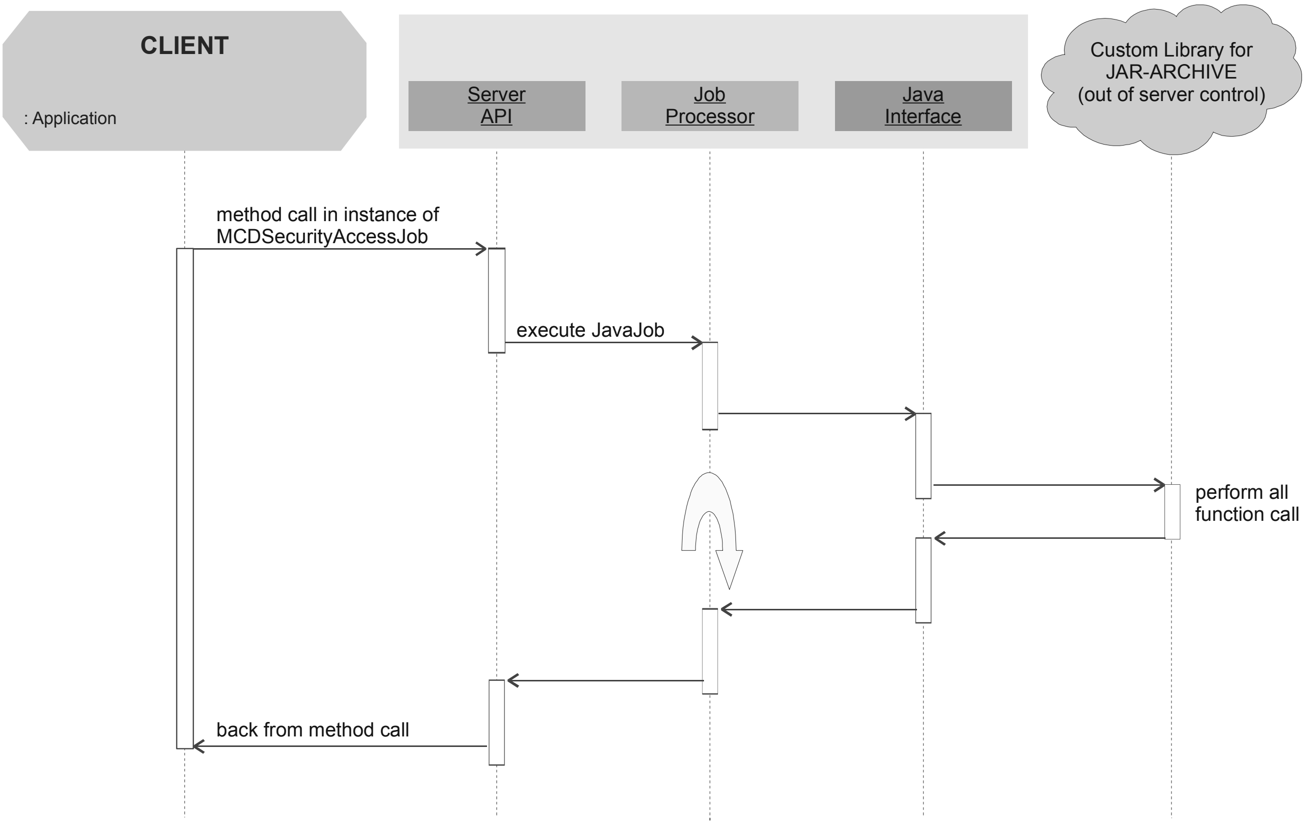

CLIENT
: Application
Server API
Job Processor
Java Interface
Custom Library for JAR-ARCHIVE (out of server control)
method call in instance of MCDSecurityAccessJob
execute JavaJob
perform all function call
back from method call

Figure 172 — Custom libraries inside Security Access Jobs (Part 1)

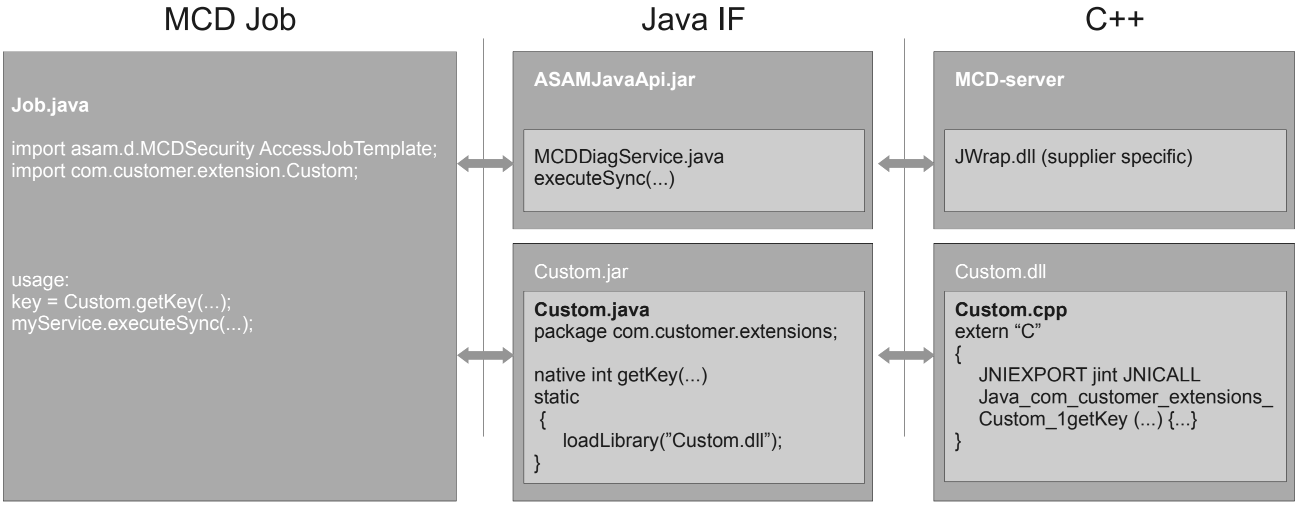

MCD Job
Java IF
C++
Job.java
import asam.d.MCDSecurity AccessJobTemplate;
import com.customer.extension.Custom;
usage:
key = Custom.getKey(...);
myService.executeSync(...);
ASAMJavaApi.jar
MCDDiagService.java
executeSync(...)
Custom.jar
Custom.java
package com.customer.extensions;
native int getKey(...)
static
{
loadLibrary(”Custom.dll”);
}
MCD-server
JWrap.dll (supplier specific)
Custom.dll
Custom.cpp
extern “C”
{
JNIEXPORT jint JNICALL
Java_com_customer_extensions_
Custom_1getKey (...) {...}
}

Figure 173 — Custom libraries inside Security Access Jobs (Part 2)

9.22.3.5 Multiple ECU Job

Multiple-ECU Jobs are used to implement a complex diagnostic sequence that can involve more than one ECU. An application domain for the Multiple-ECU Job is the identification of a vehicle, where the job communicates with multiple ECU base variants to identify the vehicle model, make and production year.

For this purpose, a Multiple-ECU Job may create logical links, allowing for the associated locations to be used by the job. If available, the D-server will provide a mechanism to list the locations (Protocols, Functional Groups, ECU Base Variants and ECU Variants) which are referenced by a Multiple-ECU Job. These locations can be accessed by the method `MCDDbMultipleEcuJob::getDbLocations()`. They are listed in the ODX element DIAG-LAYER-REFS of a MULTIPLE-ECU-JOB in ODX. For the execution of the MULTIPLE-ECU-JOB, these locations have not direct relevance. They are optional elements in ODX. However, an application may want to verify, if the locations envisaged by the author of the ODX Database and/or the author of the MULTIPLE-ECU-JOB are available prior to the execution of the job.

A Multiple-ECU Job returns one runtime response object structured in accordance with the corresponding definition in the database. The following paragraphs explain how to access the database element of a Multiple-ECU Job, and how to create and execute Multiple-ECU Jobs.

To access the database element, the method `MCDDbProject::getDbMultipleEcuJobLocation()` returns a DbLocation for all Multiple-ECU Jobs defined within that project. Within this location, only the methods `MCDDbLocation::getDbJobs()`, `MCDDbLocation::getDbDataPrimitives()`, `MCDDbLocation::getDbDiagComPrimitives()` return non-empty collections which contain the Multiple-ECU Jobs defined within the project. This location will not offer any other database objects, and that the D-server will not provide any MCD(Db)ControlPrimitives for this location.

At runtime, a logical link is required to allow the instantiation and execution of Multiple-ECU Jobs. To this end, a runtime-generated logical link shall be available at the `MCDDbVehicleInformation` which has the shortname "#RtGen_MultipleEcuJob_LogicalLink". This runtime-generated MCDDbDLogicalLink can then be used to add a runtime logical link at the `MCDLogicalLinks` collection (accessible through `MCDProject::getLogicalLinks()`), which subsequently can be used to create and execute Multiple-ECU Jobs.

Another way to create a logical link for execution of a Multiple-ECU Job is using the method `MCDLogicalLinks::addByAccessKeyAndInterfaceResource(…)`. In this case the parameter for the `MCDInterfaceResource` will be ignored by the D-server. The same applies to the parameter `MCDDbPhysicalVehicleLink` in method `MCDLogicalLinks::addByAccessKeyAndVehicleLink(…)`. The reason for this is that the Multiple-ECU Job is free to create and use whichever links it needs within the job code, and is not bound to any resources that were used to create the logical link the job is executed on.

For this reason, and because the logical link that is used to execute Multiple-ECU Jobs is a runtime-generated construct, it has no association with any physical/interface resources.

9.22.4 Handling of Java Jobs

9.22.4.1 General

The following subclauses provide a more detailed description on various aspects of job implementation and job handling mechanisms as provided by the MCD D-server.

9.22.4.2 Development of Java Jobs

A Java job usually is executed within the runtime environment of the D-server. Therefore, any lock-ups, memory leaks, exception conditions or performance issues caused by job code can potentially degrade the D-server performance or even render it completely useless. For this reason, it is strongly suggested that a Java job (including the associated libraries defined in the ODX data) only uses the external libraries listed in Table 45 — Java libraries that are allowed to be used within Java Job code (based on JDK 1.4.2).

Table 45 — Java libraries that are allowed to be used within Java Job code (based on JDK 1.4.2)

Package	Allowed Classes	Description
Java.lang	Boolean, Byte, Character, Character.Subset, Character.UnicodeBlock, Class, Double, Float, Integer, Long, Math, Number, Object, Short, StrictMath, String, StringBuffer, Throwable, Void, ArithmeticException, ArrayIndexOutOfBoundsException, ArrayStoreException, ClassCastException, ClassNotFoundException, CloneNotSupportedException, Exception, IllegalAccessException, IllegalArgumentException, IllegalMonitorStateException, IllegalStateException, IllegalThreadStateException, IndexOutOfBoundsException, InstantiationException, InterruptedException, NegativeArraySizeException, NoSuchFieldException, NoSuchMethodException, NullPointerException, NumberFormatException, RuntimeException, SecurityException, StringIndexOutOfBoundsException, UnsupportedOperationException, AbstractMethodError, AssertionError, ClassCircularityError, ClassFormatError, Error, ExceptionInInitializerError, IllegalAccessError, IncompatibleClassChangeError, InstantiationError, InternalError, LinkageError, NoClassDefFoundError, NoSuchFieldError, NoSuchMethodError, OutOfMemoryError, StackOverflowError, ThreadDeath, UnknownError, UnsatisfiedLinkError, UnsupportedClassVersionError, VerifyError, VirtualMachineError	Provides classes that are fundamental to the Java programming language.

Table 45 (*continued*)

Package	Allowed Classes	Description
Java.math	All classes allowed	Provides classes for performing arbitrary-precision integer arithmetic (BigInteger) and arbitrary-precision decimal arithmetic (BigDecimal).
Java.text	All classes allowed	Provides Classes and interfaces for handling text, dates, numbers, and messages in a manner independent of natural languages.
Java.util	ArrayList, Arrays, BitSet, Collections, Date (no deprecated methods), GregorianCalendar, HashMap, HashSet, Hashtable, LinkedList, Random, SimpleTimeZone, Stack, StringTokenizer, TimeZone, TreeMap, TreeSet, Vector, WeakHashMap; ConcurrentModificationException, EmptyStackException, MissingResourceException, NoSuchElementException, TooManyListenersException	Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalisation, and miscellaneous utility classes (a string tokeniser, a random-number generator, and a bit array).
asam.mcd	All classes allowed	Provides interfaces for interacting with the D-server
asam.d	All classes allowed	Provides interfaces for interacting with the D-server
asam.job	All classes allowed	Provides interfaces for interacting with the D-server

NOTE The usage of external libraries is potentially harmful, as the D-server is not able to control or restrict what is being done by job code. Therefore, server vendors are not to be held liable for any damage caused by a Java job using external libraries.

Java jobs are allowed to use standard Java class inheritance mechanisms. The general rules and practices of Java programming and class loading apply, e.g. it is recommended to use packages to avoid naming conflicts. For further information on Java programming guidelines and relevant style guides, please refer to the Sun Java documentation pages.

In addition, the following conventions should be observed when doing Java job programming: Job files should use the name: `MCD3_`*`jobname`*`.<extension>` - the filename is built using an MCD3 prefix and the job's name. The major version, minor version and revision numbers should be placed in the `@version` attribute of the Java source code, so they could be retrieved by using the Java programming environment. The short name of the job object at the D-server API and the filename of the job may differ. Usually exactly one PROG-CODE is associated with a Java job. If there are multiple PROG-CODEs associated with a Java job, these shall have disjoint values for their syntax attribute. The syntax JAVA, CLASS and JAR exclude each other. A standard-compliant D-server is only required to support the execution of Java jobs which have a syntax of CLASS or JAR.

In the D-server, the base lookup directory for Java jobs should be configurable to be able to extract the correct package structure from a Java class's include path description. To be able to use packages with Java jobs, class names shall be given fully qualified in the source code, Java code shall be given including subdirectories (path to packages), and for jar-files the entry point shall be given fully qualified (jar-file including path).

9.22.4.3 Deployment of Java Jobs

It is highly recommended to use Java Version 1.4.2 with the ASAM MCD 3 interface and for Java Jobs to be executed with a D-server. The reason for this decision is that there is no compatibility of the byte code between versions equal or lower than 1.4.2 and versions higher than 1.4.2. As a result, operation of a D-server based tester application as well as exchangeability of D-servers and ODX data can only be supported if this recommended Java Version is used. Furthermore, Java Virtual Machines for versions higher than 1.4.2 are currently not available for all anticipated platforms.

Migrating to a higher Java version will most probably involve re-compilation and re-testing of existing Java Jobs as well as re-compilation and re-testing of any Java based tester application.

Job code will usually be deployed along with ODX data. ODX describes the name and the format of the job data it references. There are two required and one optional job deployment formats: Every standard compliant server needs to be able to execute job code from Java class files and Java archives (jar files). These code types correspond to the CLASS and JAR designators in ODX. Optionally, a D-server might have the capability to execute jobs deployed as java source code. As compiling jobs at execution time of the D-server bears many risks and requires a kernel to ship with a full JDK, this is to be avoided. However, if source code compilation is required in a specific use case, it may be performed by the D-server. A D-server not offering this capability is still considered standard compliant.

It needs to be guaranteed that the content of external resource files for Java jobs and libraries does not change while a D-server is running. More precisely, the content of the external resource file shall be static for the time between `MCDSystem::selectProjectXXX()` and `MCDSystem::deselectProject()` for the same project. Exchanging external resource files may lead to non-deterministic behavior of a D-server.

9.22.4.4 Job Execution Modes

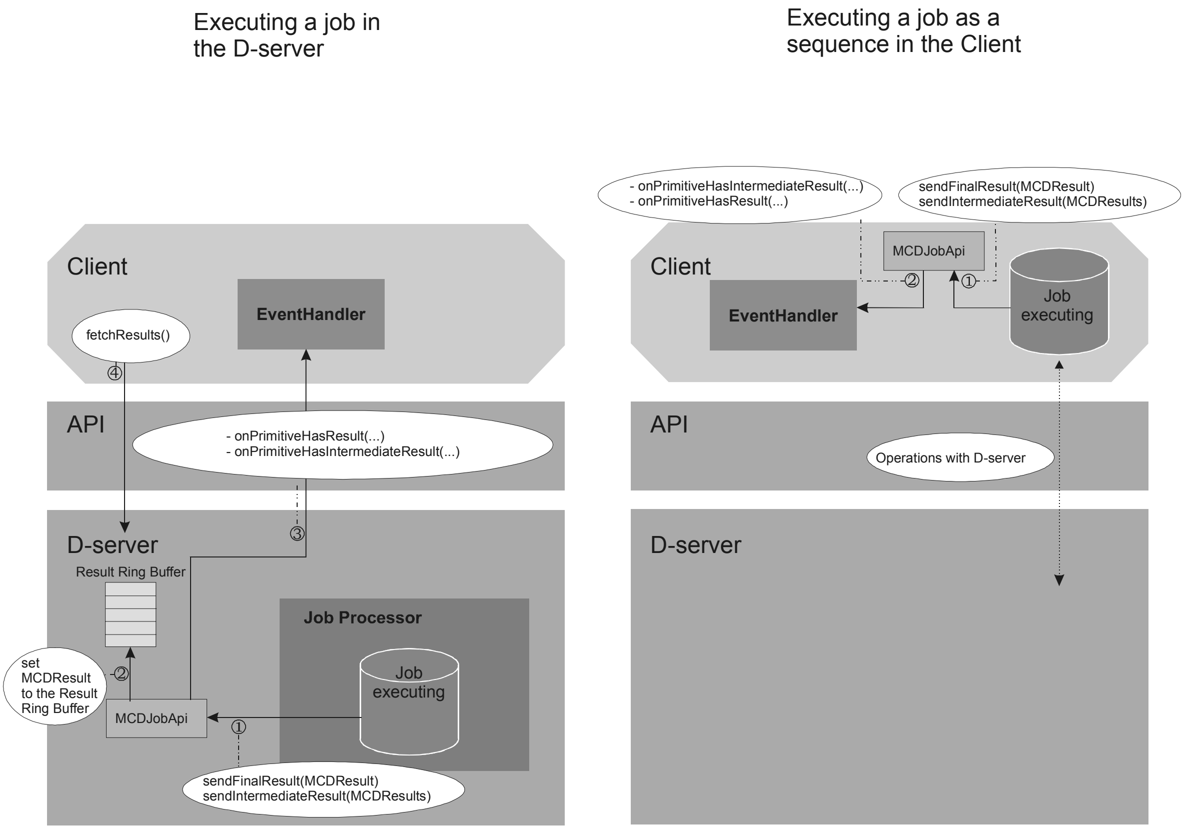

Executing a job in the D-server
Client
EventHandler
fetchResults()
④
API
- onPrimitiveHasResult(...)
- onPrimitiveHasIntermediateResult(...)
③
D-server
Result Ring Buffer
Job Processor
Job executing
set MCDResult to the Result Ring Buffer
②
MCDJobApi
①
sendFinalResult(MCDResult)
sendIntermediateResult(MCDResults)
Executing a job as a sequence in the Client
- onPrimitiveHasIntermediateResult(...)
- onPrimitiveHasResult(...)
sendFinalResult(MCDResult)
sendIntermediateResult(MCDResults)
Client
MCDJobApi
②
①
EventHandler
Job executing
API
Operations with D-server
D-server

Figure 174 — Principle of single execution of jobs

There are two conceivable ways a job can be executed, which are illustrated in Figure 174 — Principle of single execution of jobs:

- A job can be executed within the D-server's runtime environment. This is shown on the left side of the illustration.
- In the second case, the job is executed within the client. This is shown on the right side of the illustration.

The first case is the normal mode of operation, and is described in detail in this specification. The job code has access to all the D-server functionalities available to and reachable from the job API, as well as its runtime context. It uses system mechanisms to pass back results to the client application.

The second case requires the client to either re-implement or wrap all the D-server functionalities that need to be available to a job during its execution. This would allow the job to run entirely within the client's context, and under complete control of the client application. A conceivable use case for this scenario would be a job development environment, where a job development environment wraps D-server functions to gain greater control over job execution.

9.22.4.5 Hash signatures for Jobs

To increase process safety, a mechanism is required for verifying whether the Java job file delivered with a set of ODX data has been exchanged or corrupted since the original ODX data has been created. This mechanism utilizes the HASH-ALGORITHM and HASH-VALUE elements that can be added to the ODX PROG-CODE entity. These elements contain the identifier of a hash algorithm (in the HASH-ALGORITHM field, can be one of MD2, MD5, SHA-1, SHA-256, SHA-384, SHA-512) and a hash value that has been computed for the referenced Java job source or byte code using the specified algorithm (HASH-VALUE element). The evaluation of this data is handled implicitly by the runtime system and can be influenced by setting a D-server property. The property is named "ExecuteJobsUnchecked", is of type Boolean, and has a default value of false. In case the property is set to false, the server evaluates the hash value for jobs to be executed, and in case the computed value does not match the value specified in ODX, refuses to add the DiagComPrimitive representing the job to any MCDDiagComPrimitives collection. In this case, an error of type eDB_INCONSISTENT_DATABASE is raised, and the add operation does not do anything. In case the property is set to true, mismatching hash values are ignored.

In case a PROG-CODE element at a SINGLE-ECU-JOB or a MULTIPLE-ECU-JOB contains a non-empty HASH-ALGORITHM element, the D-server can perform one of the following actions:

- Server does not support execution of signed Java jobs.

  The server rejects the execution (executeSync(), executeAsync(), and startRepetition()) of the Java job by throwing an exception of type MCDProgramViolationException with error code eRT_UNSUPPORTED_HASH_ALGORITHM.

- Server does support the execution of signed Java-Jobs but does not support the hash algorithm stated in the ODX data.

  The server rejects the execution (executeSync(), executeAsync(), and startRepetition()) of the Java job by throwing an exception of type MCDProgramViolationException with error code eRT_UNSUPPORTED_HASH_ALGORITHM.

- Server supports the execution of signed Java jobs and supports the hash algorithm stated in the ODX data. However, the calculated hash value and the hash value in the ODX data differ.

  The server rejects the execution (executeSync(), executeAsync(), and startRepetition()) of the Java job by throwing an exception of type MCDProgramViolationException with error code eRT_WRONG_HASH_VALUE.

- Server supports the execution of signed Java jobs, supports the hash algorithm stated in the ODX data, and the calculated hash value and the hash value in the ODX data are identical.

  The server executes the Java job (executeSync(), executeAsync(), and startRepetition()).

- Server does not care about signatures at Java jobs.

- The server executes the Java job without checking the values of the attributes HASH-VALUE and HASH-ALGORITHM and without calculating or comparing any hash codes.

In case of using an ODX version which does not implement the HASH-ALGORITHM and HASH-VALUE elements of the PROG-CODE entity, an alternative approach is proposed: the ENCRYPTION element of PROG-CODE could be used for keeping hash value data. The hash algorithm could be specified within brackets at the beginning of the ENCRYPTION string, followed by the hash value. An example for an MD5 hash value would look like: [MD5]18a44fe98bffccabca0413e69235be1c

In addition to mandatory and optional properties defined in Annex I, each vendor or OEM is allowed to define proprietary properties. With the presence of properties, it is recommended to document any deviation of a server's behaviour from the standard by means of a meaningful property.

Behaviour of the D-server in case a Java job is encrypted (optional element ENCRYPTION has a value):

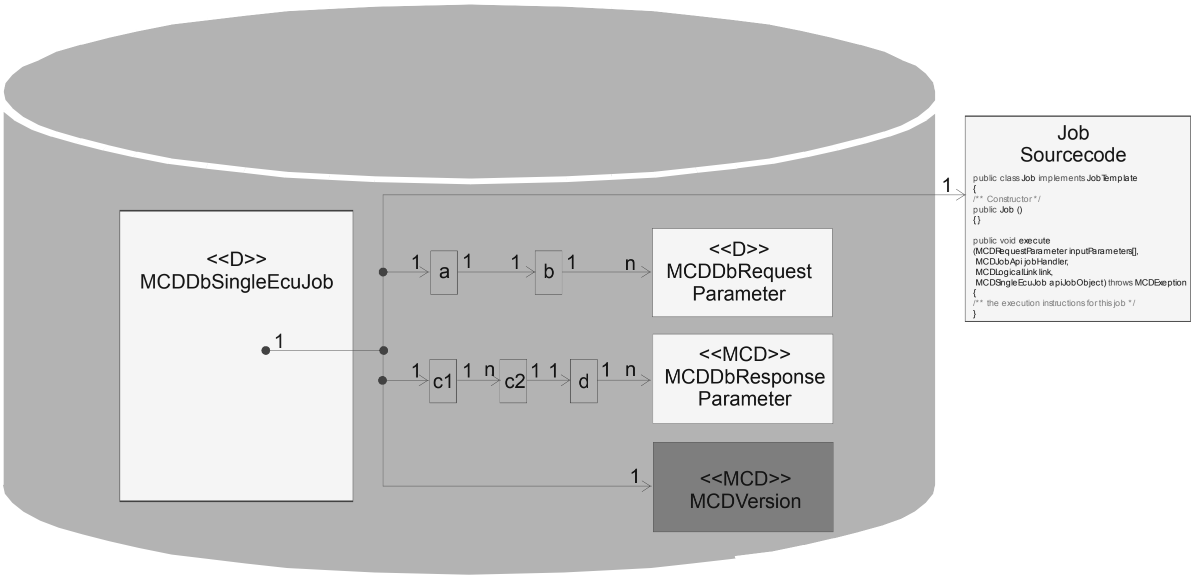
- The server supports the execution of encrypted Java jobs and supports the encryption method stated in the ODX data. In this case, the server decrypts and executes the Java job (executeSync(), executeAsync(), and startRepetition()).

- The server supports the execution of encrypted Java jobs and does not support the encryption method stated in the ODX data. In this case, the server rejects the execution of the Java job (executeSync(), executeAsync(), and startRepetition()) by throwing an exception of type MCDProgramViolationException with error code eRT_UNSUPPORTED_ENCRYPTION_ALGORITHM.

- The server does not support the execution of encrypted Java jobs. In this case, the server rejects the execution of the Java job (executeSync(), executeAsync(), and startRepetition()) by throwing an exception of type MCDProgramViolationException with error code eRT_UNSUPPORTED_ENCRYPTION_ALGORITHM.

9.22.4.6 Job Parameter handling

Job input and output parameters (including results) can use complex and simple DOPs. There is no difference between results of jobs and results of diagnostic services. For a diagnostic job, the definition of input and output parameters is always static. That means that the number of elements of a field might be increased at runtime, whereas the type of the field cannot be changed. With respect to ODX, a Single-ECU Job or a Multiple-ECU Job (hereafter, both kinds are referred to as "jobs"), both can have a possibly empty set of input parameters, a possibly empty set of positive output parameters, and a possibly empty set of negative output parameters. These three sets of parameters are represented by three database objects (a request, a positive response, and a negative response) that can be obtained from the database object of a job by using the MCDDbJob::getDbResponsesByType(MCDResponseType type) method. The D-server shall generate the required request and response templates (DbRequest and DbResponse) in accordance with these rules. The Shortname of the generated request object should be "#RtGen_Request". The Shortname of the generated positive response should be "#RtGen_Positive_Response". The Shortname of the generated negative response should be "#RtGen_Local_Neg_Response". The type of a response – positive or negative – can be obtained by the MCDDbResponse::getResponseType() method. For a positive response, the value is ePOSITIVE_RESPONSE. For a negative response, the value is eLOCAL_NEG_RESPONSE.



```
public class Job implements JobTemplate
{
/** Constructor */
public Job ()
{}

public void execute
(MCDRequestParameter inputParameters[],
 MCDJobApi jobHandler,
 MCDLogicalLink link,
 MCDSingleEcuJob apiJobObject) throws MCDExeption
{
/** the execution instructions for this job */
}
```

a MCDDbRequest
b MCDDbRequestParameters
c1 MCDDbResponses
c2 MCDDbResponse
d MCDDbResponseParameters

Figure 175 — Separation between database and job source code

As all sets of parameters of a job (input parameters, positive and negative output parameters) are optional (cardinality 0..n) in ODX[11] and can therefore be empty, the request and response templates of a job can be empty as well. That is, they would not contain any request or response parameters, respectively. Nevertheless, the database objects for request, positive and negative responses of a job shall be generated by the D server. At D-server API level, input and output parameters can always be access using the Db-methods, the same way as is the case with any other diagnostic services.

The prerequisite for handling parameters in job results are as follows:

— The method `MCDJob::createResult(…)` returns a reference to an `MCDResult` object which is maintained by the D-server. Otherwise, the D-server would not be able to support a Java job in creating its response. The new `MCDResult` can then be filled by the D-server as a result to method calls from within the Java job. Though objects of type `MCDResult` are considered to be client-controlled objects, this specific procedure of filling a `MCDResult` from a Java job is under the control of the D-server. That is, in case a Java job has created a new `MCDResult` via `MCDJob::createResult(…)`, the D-server is responsible for destroying the corresponding object. The same applies to `MCDResponse` (`MCDResponses::add(…)` and `MCDResponseParameter` objects (`MCDResponseParameters::addXXX(…)`) added to an `MCDResult`. All these objects are returned to the calling object (Java job) by reference.

— If a Java job calls one of the methods `MCDJobApi::sendIntermediateResult(…)` or `MCDJobApi::sendFinalResult(…)`, the server returns a copy of the result to the client or the calling Java job, respectively. This decouples the objects used by the Java job from the objects processed by the D-server.

— `MCDJob::createResultCollection()` creates a new and empty `MCDResults` collection. This `MCDResults` collection is returned to the calling object (client or Java job) by copy. It can then be filled by the Java job. This specific collection of type `MCDResults` is considered a client-controlled object. That is, the Java job is responsible for destroying the corresponding object (the user space object the life cycle of which is bound to the Java job). When an `MCDResults` collection is passed to the D-server by means of

MCDJobApi::sendIntermediateResults(…), it is copied by the D-server for further processing. This decouples the objects used by the Java job from the objects processed by the D-server.

9.22.4.7 Job Communication Parameter handling

If a Java job needs to alter the currently valid communication parameters, it should use and execute a MCDProtocolParameterSet primitive within its code. All changes to communication parameters caused by a MCDProtocolParameterSet executed within a Java job will be persistent after the job has terminated – just as if the application would have issued the same change. However, the usage of MCDProtocolParameterSets in a Java job's code is considered harmful as this can cause undocumented and therefore unexpected changes to the communication parameters of a logical link at runtime.

In contrast to DIAG-COMMs, SINGLE-ECU-JOBs and MULTIPLE-ECU-JOBs cannot have local communication parameters. As a result, the D-server is not required to handle local and overwritten communication parameters for Java jobs.

9.22.4.8 Job Result handling

The result structure of a job can be freely chosen by the diagnostic data author. Every result (intermediate or final) shall correspond to the output parameters defined in the job's database definition. A job is always required to fill the entire result structure before it sends the result to the D-server. It is not allowed to omit any parts of the predefined result structure.

The job's result structure can be completely independent from the result sets of diagnostic services used within the job. If necessary, the job shall select and transform results of internally called services into the (intermediate) result(s) format of the job.

To give a client application the possibility to access job status data while the job is still executing, a job may return 0..n intermediate results. A job can only send one final result, after which no further immediate results can be delivered. It is up to the job writer to decide whether elements of the intermediate results should be included in the final result as well. Intermediate and final results use the same database template.

The D-server generates events for intermediate results generated by the diagnostic job via the MCDJobApi::sendIntermediateResults(MCDResult result) method. These events are delivered to the client application by the D-server. Whether the application actually uses the intermediate results generated by the job is of no relevance - the application may chose to ignore the event signals and leave the result data in the ring buffer until the final result shows up. Depending on the ring buffer's size (maximum element count), the application will then have access to zero, several or all intermediate results, in addition to the final result.

If the sending of intermediate result events has to be activated depending on the application user's needs, the job shall provide an appropriate input parameter for switching intermediate result generation on or off. Also, if the contents of intermediate and final results are of different types, the differentiation shall be done within a single, static result structure (template), e.g. it can be handled by using a multiplexer entry as the template's root element.

In case no final result is sent before the job terminates, the job's execution is considered unsuccessful. This fact will be reported to the client by means of a server-generated MCDResult which solely contains an error of type eJOB_CRITICAL_ABORT.

9.22.4.9 Job Result Generation

The construction of a job's result object structure is achieved by a set of methods provided by the MCDJobApi class. These methods and their usage are described in this subclause. For an example of the usage of these methods, please refer to the "Job example" subclause below.

Because a job has one set of positive response parameters and one set of negative response parameters, it shall be decided within the job's source code whether a positive or a negative response is to be created before calling MCDResponses::add(MCDDbLocation dbLocation, boolean isPositive = true). By setting the isPositive flag to either true or false, it is possible to create a negative as well as a positive response within job source code.

The starting point for a job result is its database template. Based upon this, a corresponding result structure is created the same way as for ‘normal’ diagnostic services. In case of dynamic result elements (fields/arrays, multiplexers and environment data), the job code has additional possibilities when creating the result structure:

- Fields let the job create an arbitrary number of elements of one given sub element.
- A multiplexer (MUX) lets the job choose between one of many branches.
- The same applies for env-data elements, which are a part of DTC handling.

These dynamic elements can occur at any level of the result structure.

If a dynamic element has to be created, the job shall choose which and how many of the selectable sub elements are to be included in the response structure. In case of an field (array), an element is added using the `MCDResponseParameters::addElement()` and `MCDResponseParameters::addElementWithContent(…)` method. In case of a dynamic element of the type multiplexer, the respective branch is added using the `MCDResponseParameters::addMuxBranch*(…)` method. On the level of an element of type `eENVDATADESC`, the method `MCDResponseParameters::addEnvDataByDTC(A_UNIT32)` is used to insert an `eENVDATA` block.

Other than that, complex (structured) elements of type `eENVDATADESC` and `eSTRUCTURE` are handled like any other simple element and are inserted into the response structure in the same way.

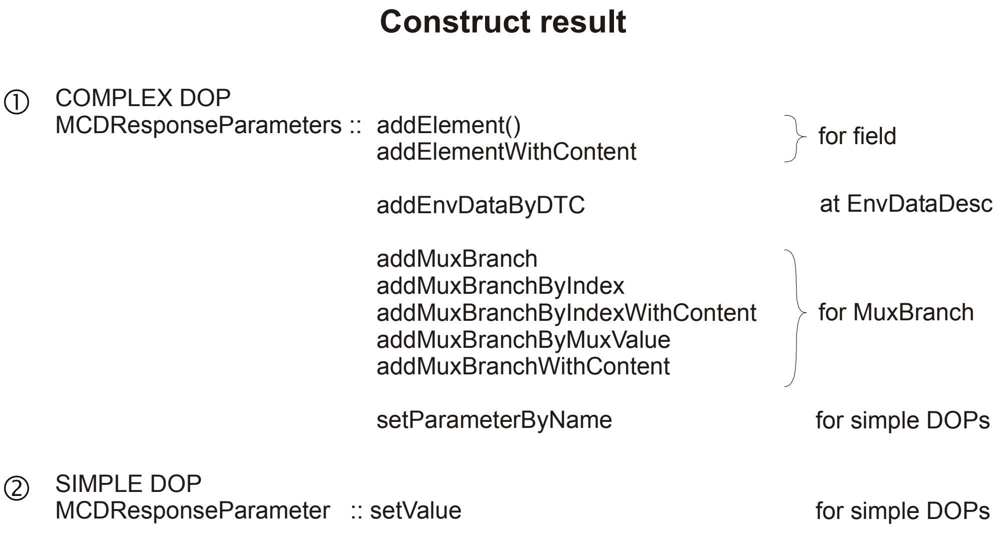


Figure 176 — Methods for result construction in Jobs

A job’s result structure shall be constructed anew for each job run. However, it is possible to reuse result objects for intermediate results by temporarily storing and copying the result structure(s) within the job code. When creating, adding or selecting a multiplexer branch to a result structure, the D-server verifies the relevant data using the job’s database template. That way result structure integrity is tested at each step of response construction, and it is guaranteed that the output format corresponds to the database template.
In case of some of the complex DOPs (`eFIELD`, `eMULTIPLEXER`, `eSTRUCTURE`, `eENVDATA`), the internal values of these elements (which the client application retrieves by calling the `MCDResponseParameter.getValue()` method of the elements that correspond to these complex DOPs) are updated internally by the D-server’s job processor; They are not to be updated within the job source code.

An MCDResponseParameter used as a parameter for an addXXX(…) method needs to comply to the corresponding MCDDbResponseParameter definition of the database response object (structure, types, ranges). Otherwise, the addXXX(…) method throws a MCDParameterizationException with error code ePAR_RESPONSEPARAMETER_MISMATCH. If the passed parameter value does not comply with the database template of the parameter, no information is copied.

Rules for MCDResponseParameters::addElementWithContent(…):

- Content is copied for:
- MCDError and Error Availability
- MCDResponseParameters
- MCDValue

Rules for MCDResponseParameters::addMuxBranchByIndexWithContent(…):

- The target MCDResponseParameter is of type eMULTIPLEXER. Otherwise an exception of type MCDProgramViolationException with error code eRT_METHOD_NOT_SUPPORTED_FOR_PARAMETER_TYPE will be thrown.
- The parameter *index* is the index of the target MUX-branch. It selects the MUX-branch in the DB-template of the target MCDResponseParameter which is to be used for validity checks.
- The elements of the MCDResponseParameters collection used as the content parameter represent the (complex) values of the top-level elements of the target MUX branch, in the order (by index) they are placed in the collection. That is, the values of the first element inside the target MUX branch are copied from the first element in the source collection.
- For copying the content of every element in the source collection to the corresponding element in the target MUX branch, the same rules apply as for MCDResponseParameters::addElementWithContent(…).

Rules for MCDResponseParameters::addMuxBranchWithContent(…):

- The target MCDResponseParameter is of type eMULTIPLEXER.
- The *branch* parameter identifies the branch of the target MUX-branch. That is, it selects the MUX-branch in the DB-template of the target MCDResponseParameter which is to be used for validity checks.
- The elements of the MCDResponseParameters collection used as the content parameter represent the (complex) values of the top-level elements of the target MUX branch, in the order (by index) they are placed in the collection. That is, the values of the first element inside the target MUX branch are copied from the first element in the source collection.
- For copying the content of every element in the source collection to the corresponding element in the target MUX branch, the same rules apply as for MCDResponseParameters::addElementWithContent(…).

Rules for MCDResponseParameters::addEnvDataByDTC(…):

- Adds a single structured response parameter of type eENVDATA to the collection of response parameters.
- Parent element of this response parameter needs to be of type eENVDATADESC. Otherwise, an exception of type MCDProgramViolationException with error code eRT_METHOD_NOT_SUPPORTED_FOR_PARAMETER_TYPE will be thrown
- Similarly to addMuxBranch(), an 'empty' response parameter structure is added up to the first dynamic element. Empty means that no parameter values have been filled in.

— Returns the response parameter of type eENVDATA which was added to the collection of response parameters.

Parameters with parameter type eTABLE_KEY are handled like other simple parameters. For parameters of type eTABLE_STRUCT, it is distinguished between the following two cases:

a) The corresponding eTABLE_KEY parameter has a valid MCDValue: In this case, an eTABLE_STRUCT parameter is handled like a static complex parameter (data type eSTRUCTURE). Thus, all elements will be automatically inserted in the created result structure up to the first dynamic result element.

b) The corresponding eTABLE_KEY parameter has no valid MCDValue: In this case, an eTABLE_STRUCT parameter is handled like a dynamic complex parameter. That means, its sub-elements will not be automatically inserted in the result structure. Thus, a call to getParameters() for this eTABLE_STRUCT parameter will deliver an empty collection. As soon as a valid MCDValue is set for the corresponding eTABLE_KEY parameter, the sub-elements are filled in and the result structure and can be retrieved by calling the getParameters() method.

   — A parameter of type eTABLE_KEY will have a valid MCDValue in case of static parameter definition or in case of dynamic parameter definition where either a default value is defined in the database or a value is already set through the job at runtime.

   — An eTABLE_KEY parameter will have a not initialized MCDValue in case of dynamic parameter definition and neither a default value is defined in the database nor a value was set by the job at runtime.

9.22.4.10 Progress information

A D-server provides a generic mechanism for passing progress information from a Java job to client applications that are interested in monitoring job execution. A Java job can send progress information to a D-server in form of a number representing the percentage of completeness of the job's operation (allowed are values between 0 and 100). Additionally, it is possible to pass generic information in form of a string to the D-server, which can be used to send e.g. log data from within job code.

Completeness information is passed from a Java job to the D-server by calling the method MCDJobApi::setProgress(…) from within the job code. The textual job info is passed to an D-server by calling the method MCDJobApi::setJobInfo(…) from within the job's code. The job developer is responsible for providing this kind of information to D-server and client application.

If job is executed synchronously, asynchronously or repeatedly, the client application can obtain progress information and job information by actively calling the MCDJob::getProgress() and MCDJob::getJobInfo() methods, respectively - the client application actively polls for the job's information.

Alternatively, a client application can be notified of available job information by registering a MCDDiagComPrimitiveEventHandler at the job (by using the MCDDiagComPrimitive::setEventHandler method) or at the MCDDiagComPrimitives collection of the corresponding logical link (by using the MCDDiagComPrimitives::setPrimitiveEventHandler method). The D-server generates an event of type onPrimitiveProgressInfo whenever the job has called the method setProgress(…) at its MCDJobApi object. Whenever a job has called the method setJobInfo(…)at its MCDJobApi object, the D-server generates an event of type onPrimitiveJobInfo.

A job is allowed to use the setProgress and setJobInfo methods at any time during its execution, even after it has sent a final result to the D-server.

9.22.5 Job execution

9.22.5.1 General

Jobs can be executed either once by using the `MCDDiagComPrimitive::executeSync()` or `MCDDataPrimitive::executeAsync()` calls, or repeatedly by using the `MCDDataPrimitive::startRepetition()` method. This subclause provides detailed information on these methods of job execution.

9.22.5.2 Single execution of a Job

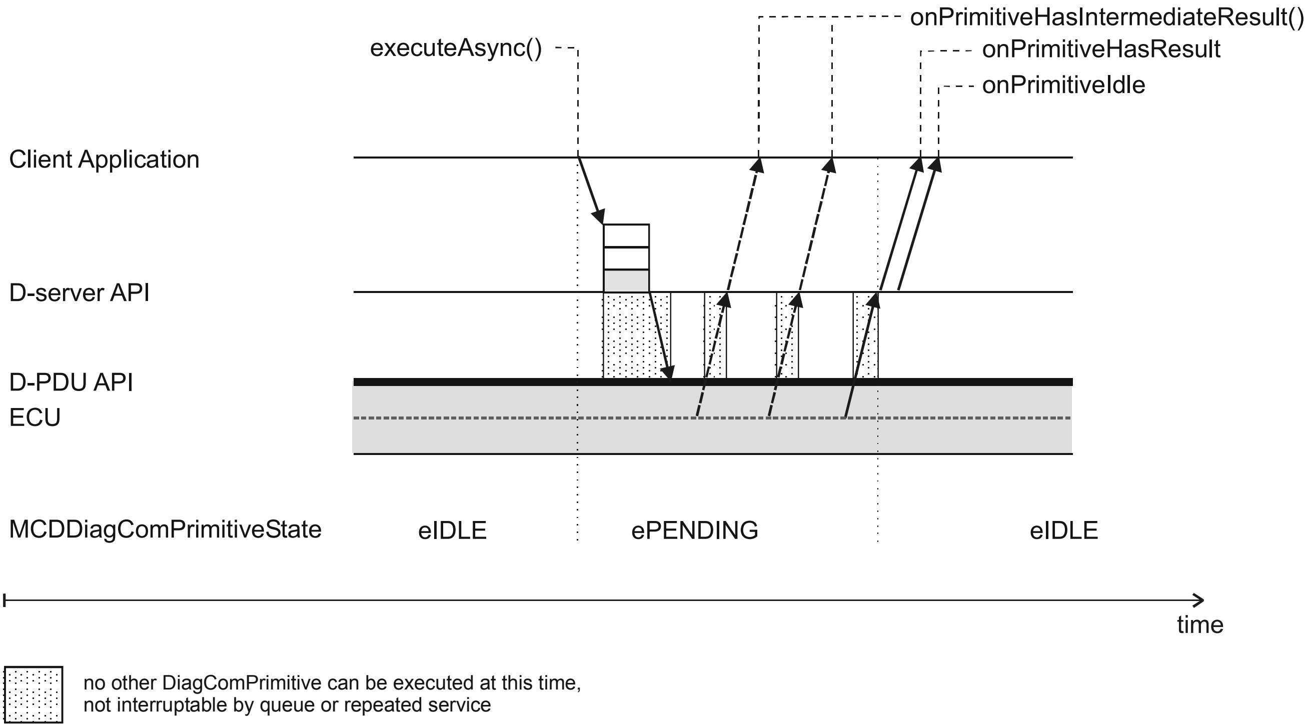


Figure 177 — Job execution asynchronous
S=J; EM=A; T=RT; AM=P; IR=N; SPR

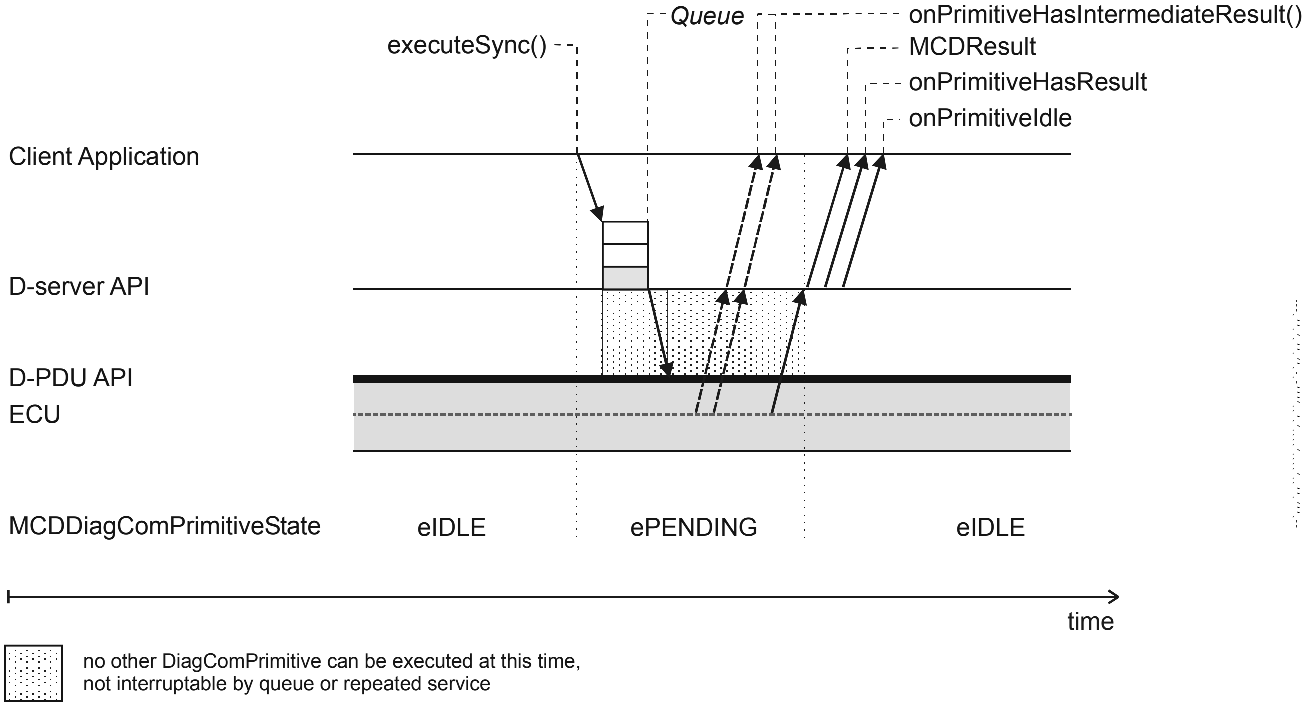


Figure 178 — Job execution synchronous S=J; EM=S; T=RT; AM=P; IR=M; SPR

EXAMPLE Normal execution of a Single-ECU Job

Method description:

`MCDDataPrimitive::executeAsync()`	Asynchronous start of job execution
`MCDDiagComPrimitive::executeSync()`	Synchronous start of job execution
`MCDDiagComPrimitive::cancel()`	Abort job execution as quickly as possible or remove the job from execution queue.

The following rules apply to the execution of jobs:

— When a job is executed synchronously or asynchronously, intermediate results are delivered to the application in the form of events at the registered event handlers. It is up to the client application to handle these events in a meaningful way. For example, a single-threaded client application will not be able to synchronously execute a job and receive intermediate result events at the same time.

— All diagnostic services or jobs which are executed inside a job shall be started synchronously (`executeSync()`). If job code tries to use the `MCDDataPrimitive::executeAsync()` method, an `MCDProgramViolationException` (`eRT_SERVICE_ASYNC_NOT_ALLOWED`) will be thrown.

— As all diagnostic services or jobs which are executed inside a job shall be started synchronously, and as cyclic `MCDDiagComPrimitives` (IS-CYCLIC is set to true in ODX) can only be executed asynchronously, the execution of cyclic services is also forbidden within a job.

— All diagnostic services which are executed inside a job are not allowed to be started repeatedly (`startRepetition()`). If job code tries to use the `MCDDataPrimitive::startRepetition()` method, an `MCDProgramViolationException` (`eRT_SERVICE_REPEATED_NOT_ALLOWED`) will be thrown.

States:

The job's MCDDiagComPrimitiveState changes from eIDLE (initial state before start of job execution) to ePENDING (state while executing), and back to eIDLE (state after execution). The states of the MCDDiagComPrimitive are managed by the D-server.

Results:

In case of successful termination, every job needs to provide a final result. The required MCDResult object for construction of the final result can be obtained via the MCDJob::createResult(…) method and shall be sent by calling MCDJobApi::sendFinalResult(…).

In case no final result is sent before the job terminates, the job's execution is considered unsuccessful. This fact will be reported to the client by means of a server-generated MCDResult which solely contains an error of type eJOB_CRITICAL_ABORT. This server-generated MCDResult is returned to the client as a return value in case of executeSync(), and as the parameter of the event of type MCDDiagComPrimitiveEventHandler::onPrimitiveHasResult(…) delivered to all registered event handlers.

With the first call of sendFinalResult(…), the server passes this result to the client. No further result will be generated by the server. If the job calls sendFinalResult(…) more than once, an exception of type MCDProgramViolationException with error code eRT_WRONG_SEQUENCE is thrown.

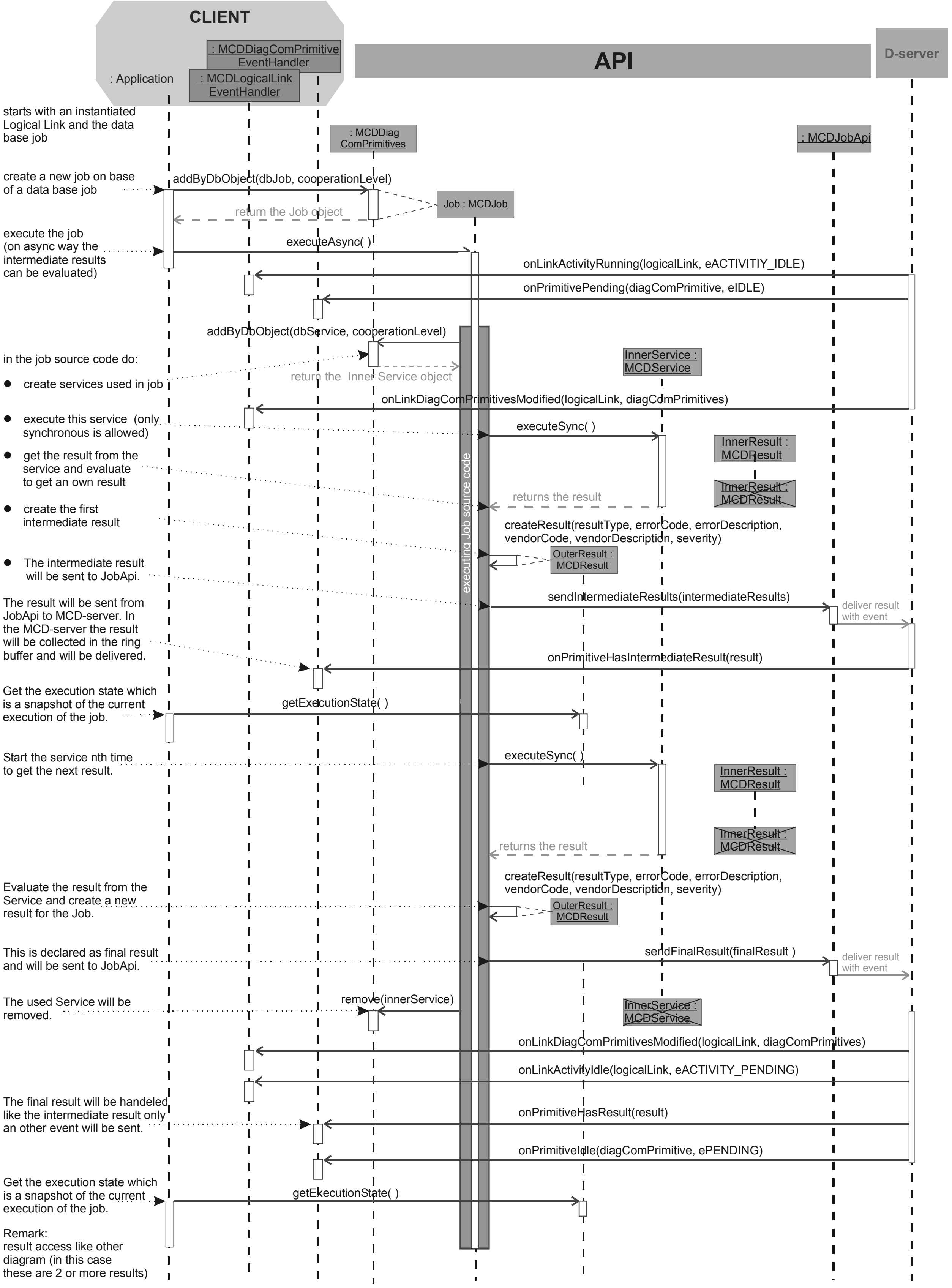
Along with the definitions above, not sending a final result corresponds to sending an empty final result. An empty result is defined as a MCDResult which does not contain any responses – the object's MCDResponses collection is empty. The result may contain an error, however.

In case the execution of a job results in a MCDException being thrown from within the job's source code without being correctly caught within this code, this exception is caught by the server. Then the error object from the exception is copied into an MCDResult, and this result is sent to all registered EventHandlers in form of an MCDDiagComPrimitiveEventHandler::onPrimitiveHasResult(…) event. Additionally, in case synchronous job execution, the result is directly returned to the calling client application.

In case the execution of a job results in an uncaught Java-Exception being thrown from within the job's source code, this exception is also caught by the server. Then a new MCDProgramViolationException is created by the server which is filled with an error of type eRT_MALFORMED_JOB_CODE and which is assigned the severity eERROR. The original exception information may be written into the VendorCode and VendorCodeDescription fields. The further processing is equivalent to that of a MCDException, as described above.

For all MCDResult objects created by a Java job, the server shall add information on execution state, logical link state, and logical link lock state before sending the result to the client.

All results (intermediate and final) are stored in a ring buffer. Intermediate results can be complete result sets of the diagnostic services used by the job. They are independent from the final job result. After receiving the event MCDDiagComPrimitiveEventHandler::onPrimitiveHasIntermediateResult(…) the execution state and the number of results can requested.

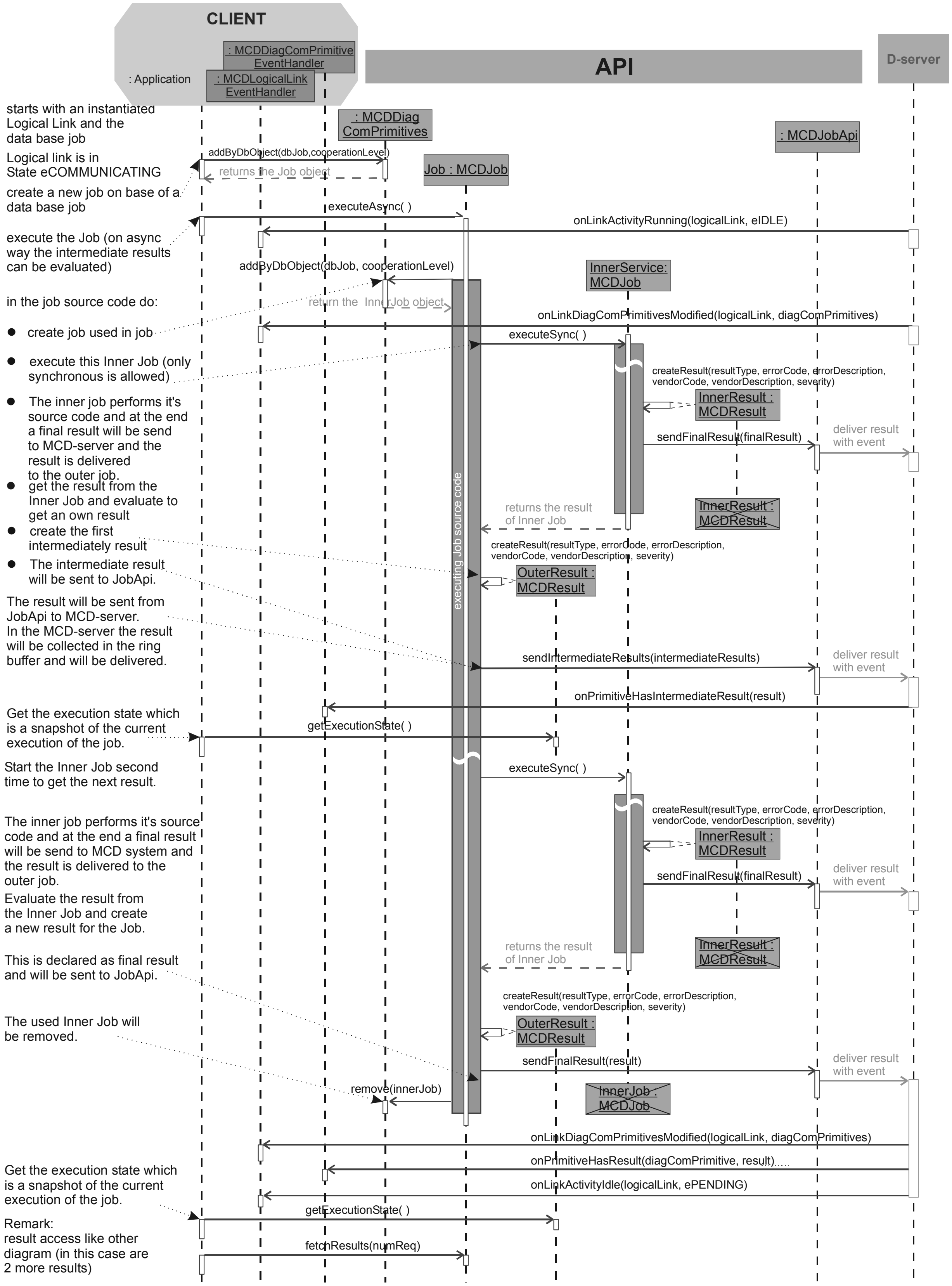

CLIENT
API
D-server
: Application
: MCDDiagComPrimitive EventHandler
: MCDLogicalLink EventHandler
: MCDDiag ComPrimitives
: MCDJobApi
Job : MCDJob
starts with an instantiated Logical Link and the data base job
create a new job on base of a data base job
addByDbObject(dbJob, cooperationLevel)
return the Job object
execute the job (on async way the intermediate results can be evaluated)
executeAsync( )
onLinkActivityRunning(logicalLink, eACTIVITIY_IDLE)
onPrimitivePending(diagComPrimitive, eIDLE)
addByDbObject(dbService, cooperationLevel)
in the job source code do:
● create services used in job
return the Inner Service object
InnerService : MCDService
onLinkDiagComPrimitivesModified(logicalLink, diagComPrimitives)
● execute this service (only synchronous is allowed)
executeSync( )
InnerResult : MCDResult
● get the result from the service and evaluate to get an own result
executing Job source code
returns the result
InnerResult : MCDResult
● create the first intermediate result
createResult(resultType, errorCode, errorDescription, vendorCode, vendorDescription, severity)
OuterResult : MCDResult
● The intermediate result will be sent to JobApi.
sendIntermediateResults(intermediateResults)
deliver result with event
The result will be sent from JobApi to MCD-server. In the MCD-server the result will be collected in the ring buffer and will be delivered.
onPrimitiveHasIntermediateResult(result)
Get the execution state which is a snapshot of the current execution of the job.
getExecutionState( )
Start the service nth time to get the next result.
executeSync( )
InnerResult : MCDResult
InnerResult : MCDResult
returns the result
createResult(resultType, errorCode, errorDescription, vendorCode, vendorDescription, severity)
Evaluate the result from the Service and create a new result for the Job.
OuterResult : MCDResult
This is declared as final result and will be sent to JobApi.
sendFinalResult(finalResult)
deliver result with event
The used Service will be removed.
remove(innerService)
InnerService : MCDService
onLinkDiagComPrimitivesModified(logicalLink, diagComPrimitives)
onLinkActivityIdle(logicalLink, eACTIVITY_PENDING)
The final result will be handeled like the intermediate result only an other event will be sent.
onPrimitiveHasResult(result)
onPrimitiveIdle(diagComPrimitive, ePENDING)
Get the execution state which is a snapshot of the current execution of the job.
getExecutionState( )
Remark:
result access like other diagram (in this case these are 2 or more results)

Remark:
Every job can deliver intermediate as mutch results as necessary and only one final result.
The results need not tu be equidistant or at the end of a service/job execution, but they have to use the data base structure for the response.

Figure 179 — Service execution inside job

Every service or job within a job shall be executed synchronously, because it is not allowed for job code to register and use an event handler (jobs are not supposed to use threading mechanisms), which would be needed to retrieve the events related to the execution of the embedded service or job. Synchronous execution returns the result as the return value of the `executeSync()` call, so it can be accessed and evaluated by the calling job code. The results of the executed services can for example be used to create an intermediate or final job result.

The following sequence diagram shows the execution of a job within a job. The job started by the client will be called *OuterJob*, while the job started by the *OuterJob* will be called *InnerJob*. The *InnerJob* is handled by the *OuterJob* like a service. The execution of the source code of the *InnerJob* is like any job execution, but no intermediate results can be evaluated by the *OuterJob*, because the *InnerJob* is executed synchronously.


CLIENT
: MCDDiagComPrimitive EventHandler
: Application
: MCDLogicalLink EventHandler
API
D-server
: MCDDiag ComPrimitives
: MCDJobApi
Job : MCDJob
starts with an instantiated Logical Link and the data base job
Logical link is in State eCOMMUNICATING
addByDbObject(dbJob,cooperationLevel)
returns the Job object
create a new job on base of a data base job
executeAsync( )
execute the Job (on async way the intermediate results can be evaluated)
onLinkActivityRunning(logicalLink, eIDLE)
addByDbObject(dbJob, cooperationLevel)
InnerService: MCDJob
in the job source code do:
return the InnerJob object
onLinkDiagComPrimitivesModified(logicalLink, diagComPrimitives)
create job used in job
executeSync( )
execute this Inner Job (only synchronous is allowed)
createResult(resultType, errorCode, errorDescription, vendorCode, vendorDescription, severity)
The inner job performs it's source code and at the end a final result will be send to MCD-server and the result is delivered to the outer job.
InnerResult : MCDResult
sendFinalResult(finalResult)
deliver result with event
get the result from the Inner Job and evaluate to get an own result
executing Job source code
returns the result of Inner Job
InnerResult : MCDResult
create the first intermediately result
createResult(resultType, errorCode, errorDescription, vendorCode, vendorDescription, severity)
The intermediate result will be sent to JobApi.
OuterResult : MCDResult
The result will be sent from JobApi to MCD-server. In the MCD-server the result will be collected in the ring buffer and will be delivered.
sendIntermediateResults(intermediateResults)
deliver result with event
onPrimitiveHasIntermediateResult(result)
Get the execution state which is a snapshot of the current execution of the job.
getExecutionState( )
Start the Inner Job second time to get the next result.
executeSync( )
createResult(resultType, errorCode, errorDescription, vendorCode, vendorDescription, severity)
The inner job performs it's source code and at the end a final result will be send to MCD system and the result is delivered to the outer job.
InnerResult : MCDResult
sendFinalResult(finalResult)
deliver result with event
Evaluate the result from the Inner Job and create a new result for the Job.
returns the result of Inner Job
InnerResult : MCDResult
This is declared as final result and will be sent to JobApi.
createResult(resultType, errorCode, errorDescription, vendorCode, vendorDescription, severity)
OuterResult : MCDResult
The used Inner Job will be removed.
sendFinalResult(result)
deliver result with event
remove(innerJob)
InnerJob : MCDJob
onLinkDiagComPrimitivesModified(logicalLink, diagComPrimitives)
onPrimitiveHasResult(diagComPrimitive, result)
onLinkActivityIdle(logicalLink, ePENDING)
Get the execution state which is a snapshot of the current execution of the job.
getExecutionState( )
Remark: result access like other diagram (in this case are 2 more results)
fetchResults(numReq)

Remark:
Every job can deliver intermediate results as much as necessary
and only one final result. They need not to be equidistant or at the end of a service/job execution, but they have to use the data base structure for the response.

Figure 180 — Job execution inside job

9.22.5.3 Repeated execution of Job

Jobs can be executed repeatedly, so that the job is repeated multiple times without having any kind of loop statement in the job. That is, the job does not “know” that it is being executed multiple times in a loop. As a result, every execution delivers one single result (as a final result). Jobs and Services act identically with regard to repeated execution.

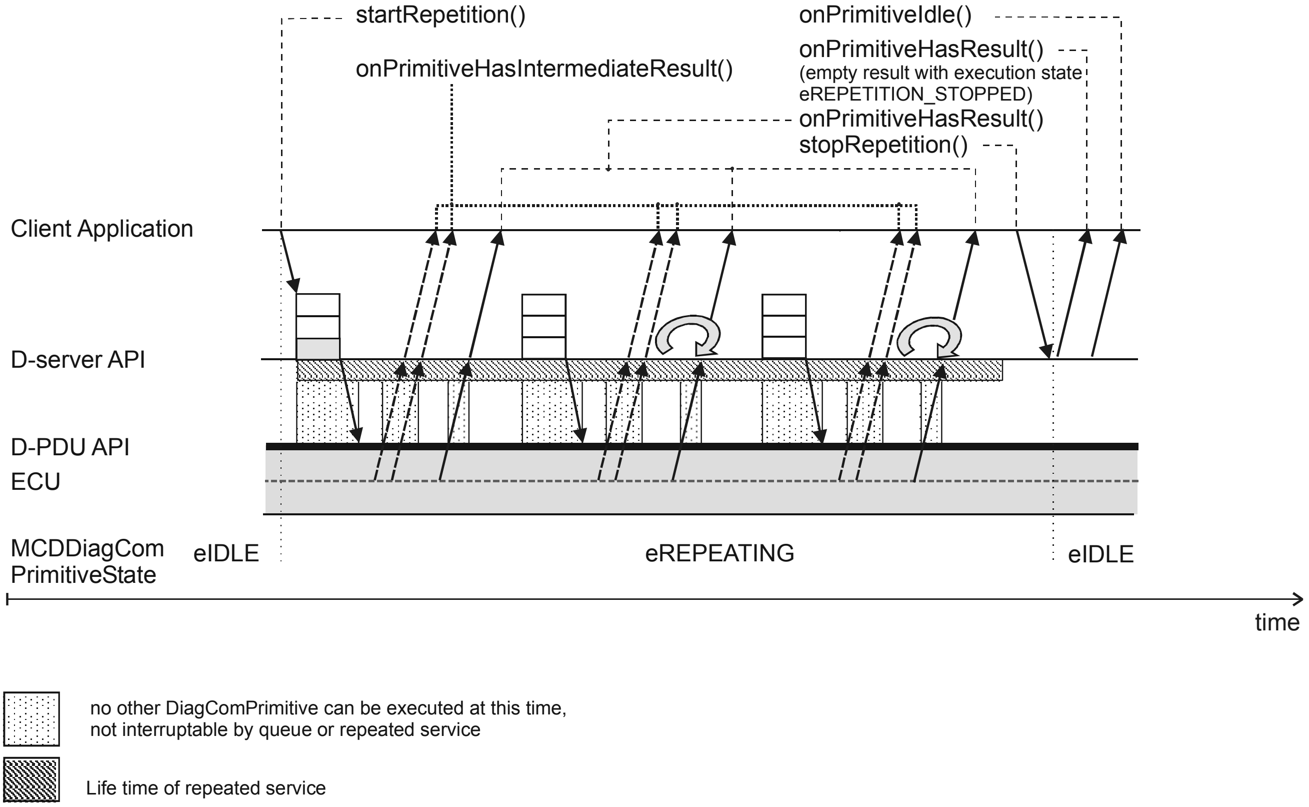

startRepetition()
onPrimitiveHasIntermediateResult()
onPrimitiveIdle()
onPrimitiveHasResult()
(empty result with execution state eREPETITION_STOPPED)
onPrimitiveHasResult()
stopRepetition()
Client Application
D-server API
D-PDU API
ECU
MCDDiagCom PrimitiveState
eIDLE
eREPEATING
eIDLE
time
no other DiagComPrimitive can be executed at this time, not interruptable by queue or repeated service
Life time of repeated service

Figure 181 — Job execution repeated
S=J; E=A; T=S; AM=P; IR=N; SPR

Description: Repeated execution of a job

The time between two repeated executions is set by the client application, and is not stored in the database.

DiagComPrimitive method description:

`MCDDataPrimitive::startRepetition()`	Start service execution – after passing the queue, the job will live in a loop and be executed repetitively (will not pass through the queue again).
`MCDDataPrimitive::stopRepetition()`	Stop repeated job execution.
`MCDDataPrimitive::updateRepetitionParameter ()`	In the state `eIDLE,` the jobs execution parameters can be changed, as this method does not have to go through the execution queue.
`MCDDiagComPrimitive::cancel()`	Terminate job execution as quickly as possible.

States:

The job's MCDDiagComPrimitiveState changes from eIDLE to eREPEATING (after calling startRepetition() and back to eIDLE (after a call to stopRepetition() or cancel()).
The MCDDiagComPrimitiveState state change from eIDLE to ePENDING is made every time the client application starts a service (by calling executeSync() or executeAsync()) that goes through the execution queue.

Results:

There can be one or more results stored in the ring buffer. For each completed job execution, exactly one result is returned by the job processor. The result of a job is based on the job's database template, which can be either a positive or a negative response template. In case of repeated execution, every execution cycle can generate a result with different parameter values, because each execution of a job is independent of previous executions. After the result has been put into the result ring buffer by the D-server, the event MCDDiagComPrimitiveEventHandler::onPrimitiveHasResult(…) is sent to the client. Additionally, a job might produce intermediate results during execution. After being put into the result buffer, these results will be indicated to the application by the event MCDDiagComPrimitiveEventHandler::onPrimitiveHasIntermediateResult(…). The sending of intermediate results is job specific. All results (intermediate and final) shall correspond to the response database template of the job.

Each repetition cycle of a job which is executed repeatedly can result in a different amount of intermediate results and potentially one final result depending on the execution path inside the job. However, all results are indicated by the appropriate events as mentioned above.

When a repeated job execution is stopped, an empty result is returned to the client application containing the execution state eREPETITION_STOPPED. The end of a repeated job execution is indicated by an event onPrimitiveHasResult (containing the empty result just mentioned), and an event onPrimitiveIdle which points to the MCDDiagComPrimitive that corresponds to the job that had been executing repeatedly.

9.22.5.4 Cancellation of a Job

A job that is already running can either be cancelled implicitly by the D-server or explicitly by the client application. An implicit cancellation occurs e.g. in case a client application tries to reset a logical link by calling the MCDDLogicalLink::reset() method, in which case the D-server will try to gracefully cancel all DiagComPrimitives (including jobs) that are being executed on that link at the time of reset request. An explicit cancellation is caused by a client application calling MCDDiagComPrimitive::cancel() on a DiagComPrimitive (Job) that is currently being executed.

In case a job is cancelled from the execution queue, its associated MCDResult will have the MCDExecutionState eCANCELED_FROM_QUEUE. In case the job quit gracefully after a call to *JobTemplate::cancel(), its associated MCDResult will have the MCDExecutionState eCANCELED_DURING_EXECUTION. In case the job had to be terminated by the D-server, its associated MCDResult will have a MCDError code of eRT_JOB_CANCELLATION_TIMEOUT_REACHED.

9.22.6 Job example

This subclause gives an example of how to use the job feature both in ODX and in a D-server. Specifically, it is shown how a dynamic result is constructed from a database template.

The following figure shows the database template of the example:

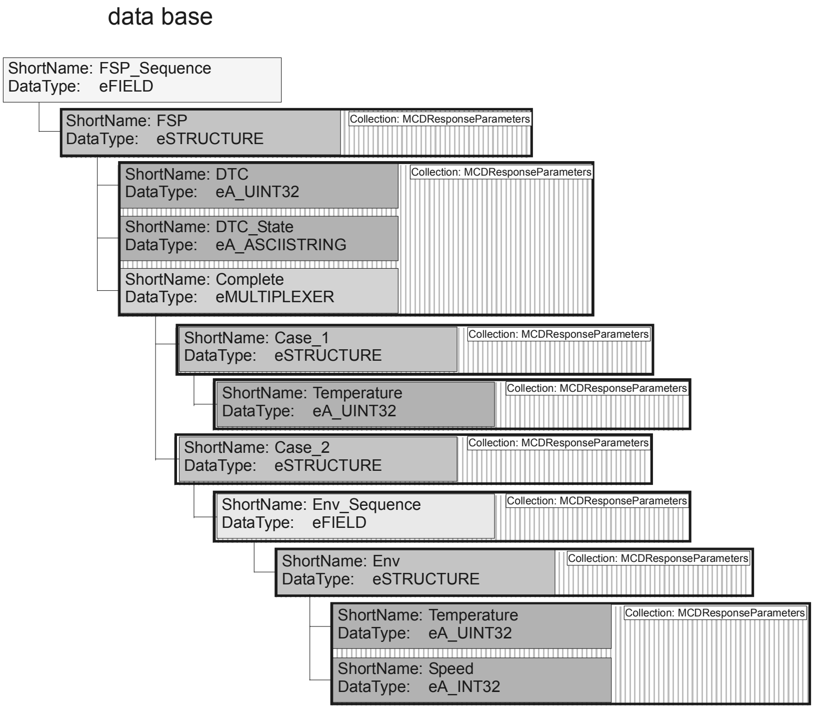


Figure 182 — Database template example DTC (positive case)

Based upon this database template for the DTC, the result structure is constructed up to the first dynamic element by means of MCDResponses::add(…). Within the example DTC, the sequence FSP_Sequence is the element on top of the structure.

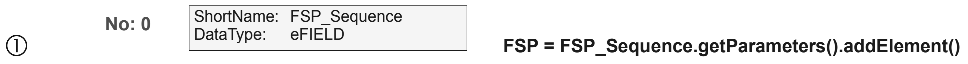


Figure 183 — Step 1 - Add to the ResponseParameterCollection "FSP_Sequence" one element "FSP"

`MCDResponseParameters::addElement(…)` adds the first element to the sequence. The added element (in our case, the first FSP structure) is constructed up to its first dynamic element, which in our example DTC is the multiplexer named *Complete*. After constructing the structure, all simple DOP´s which already exist are filled with values: *DTC* and *DTC_State*. The multiplexer branch to be used is selected by means of calling `MCDResponseParameters::addMuxBranch(MCDDataTypeShortName branch).`

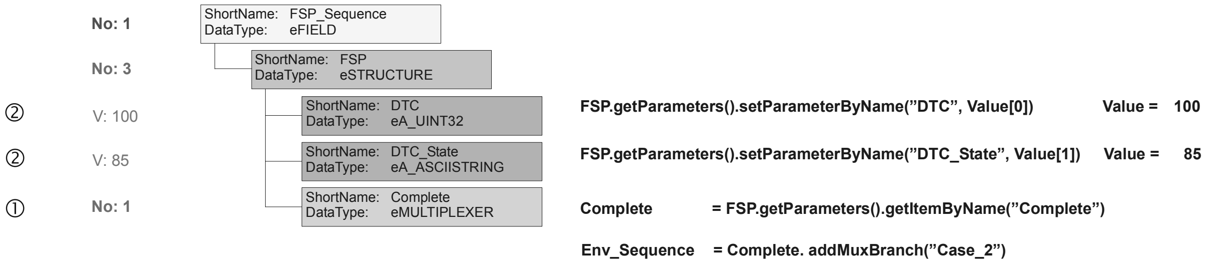

No: 1
ShortName: FSP_Sequence
DataType: eFIELD
No: 3
ShortName: FSP
DataType: eSTRUCTURE
②
V: 100
ShortName: DTC
DataType: eA_UINT32
FSP.getParameters().setParameterByName(”DTC”, Value[0]) Value = 100
②
V: 85
ShortName: DTC_State
DataType: eA_ASCIISTRING
FSP.getParameters().setParameterByName(”DTC_State”, Value[1]) Value = 85
①
No: 1
ShortName: Complete
DataType: eMULTIPLEXER
Complete = FSP.getParameters().getItemByName(”Complete”)
Env_Sequence = Complete. addMuxBranch(”Case_2”)

Figure 184 — Step 2 - addMuxBranch (Case_2)

After calling `MCDResponseParameters::addMuxBranch(MCDDataTypeShortName branch)`, the corresponding multiplexer branch is created up to the next dynamic element, in accordance with its database template. In our example *DTC*, that is the sequence *Env_Sequence*.

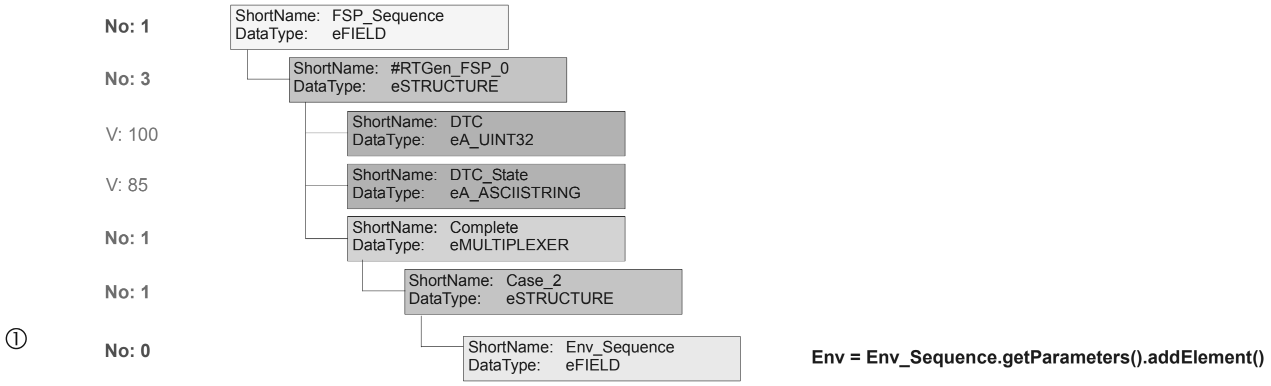


Figure 185 — Step 3 - addElement “Env”

The first element of the Sequence is created by means of the `MCDResponseParameters::addElement(…)` method.

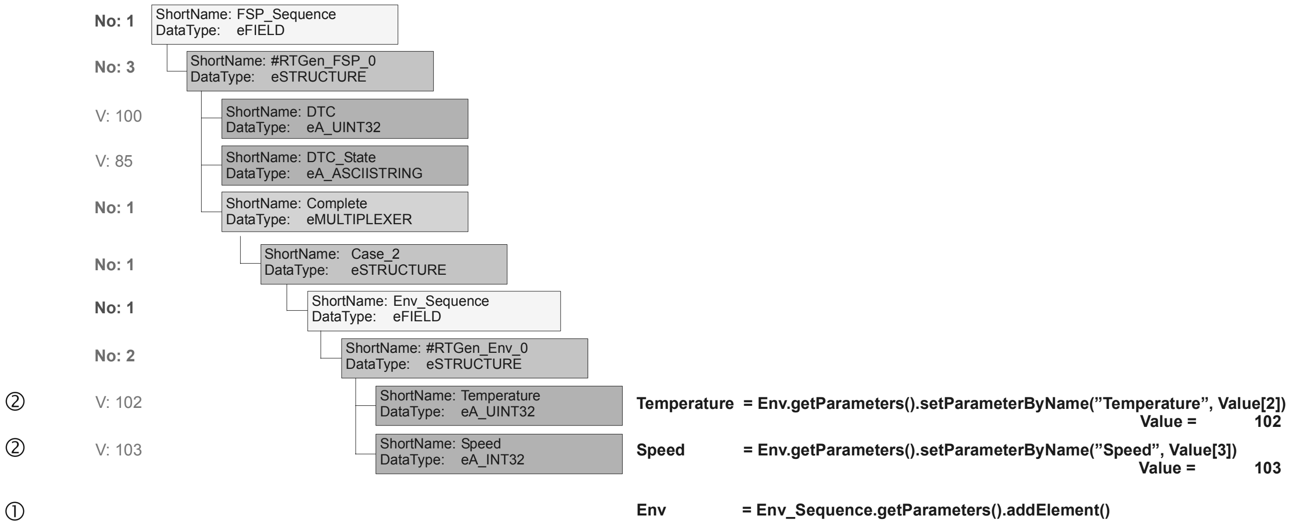


Figure 186 — Step 4 - addElement “Env”

In the example DTC, the elements of the *Env_Sequence* sequence are static, and thus are constructed completely by the `addElement(…)` method call. *Temperature* and *Speed* may now be assigned their respective values by means of `MCDResponseParameter::setValue(MCDValue value)`. By calling

`MCDResponseParameters::addElement(…)` a second time, another element of the sequence is created, and values for the elements *Temperature* and *Speed* are set by means of calling `MCDResponseParameter::setValue(MCDValue value)` again for each value.

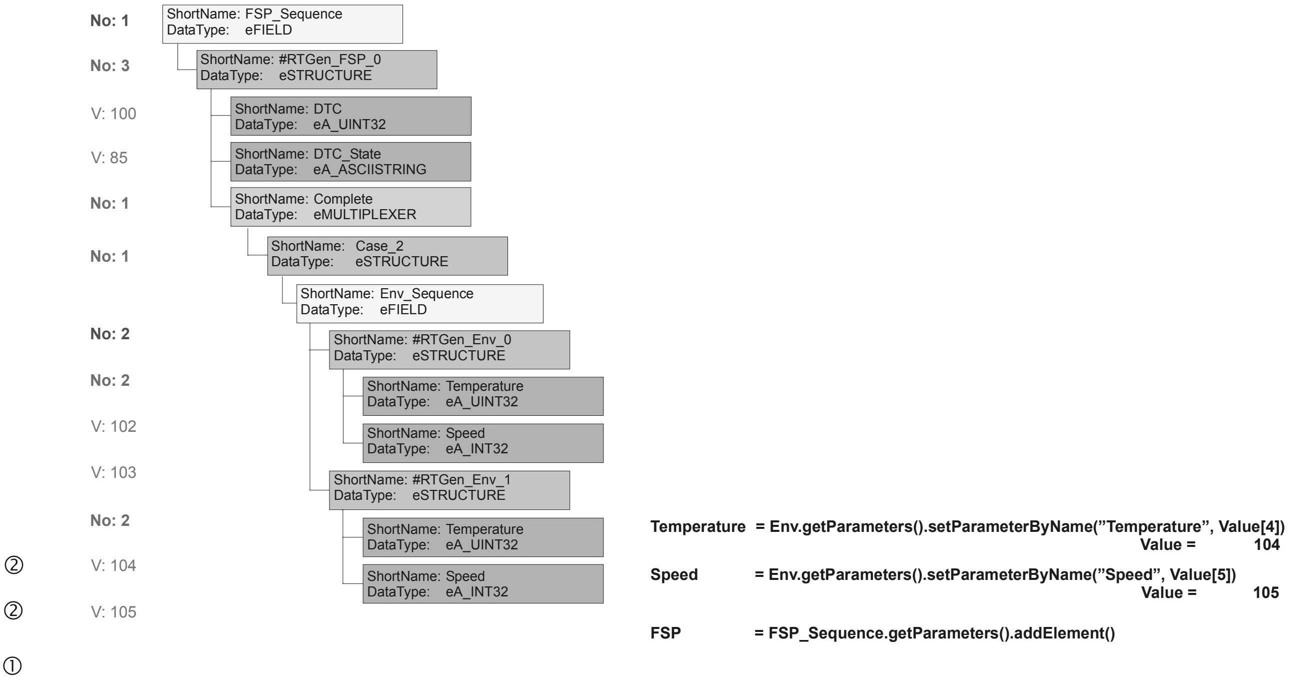


Figure 187 — Step 5 - addElement „FSP“

After the values for all the *Temperature* and *Speed* elements have been set, a further element is added to the *FSP_Sequence* (the element at the top of the result structure).

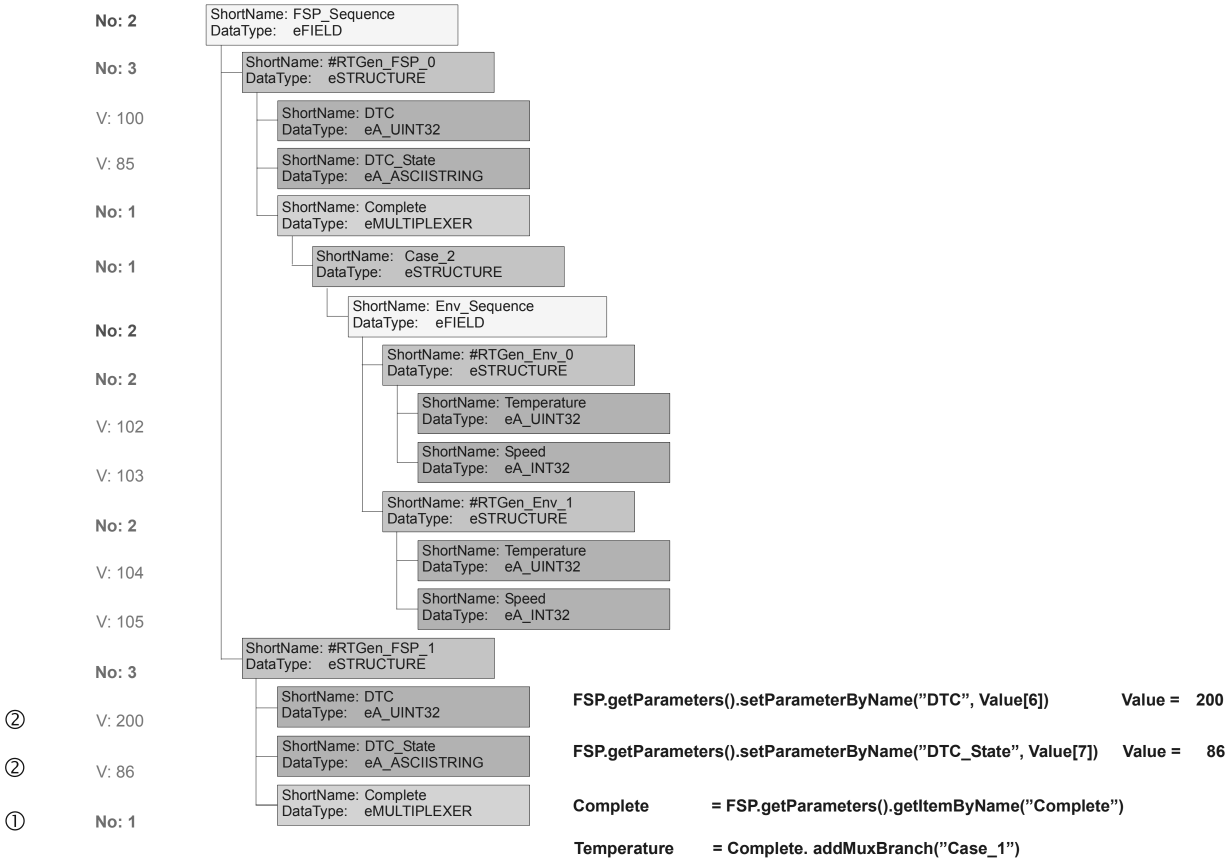


Figure 188 — Step 6 - addMuxBranch (Case_1)

Again, this element is constructed up to the first dynamic element - up to the multiplexer named *Complete*. The values for the static simple DOPs *DTC* and *DTC_State* are set by calling `MCDResponseParameter::setValue(MCDValue value)`.

The subsequent call to `MCDResponseParameters::addMuxBranch(MCDDatatypeShortName branch)` creates the corresponding multiplexer branch up to the next dynamic element in accordance with the database template.

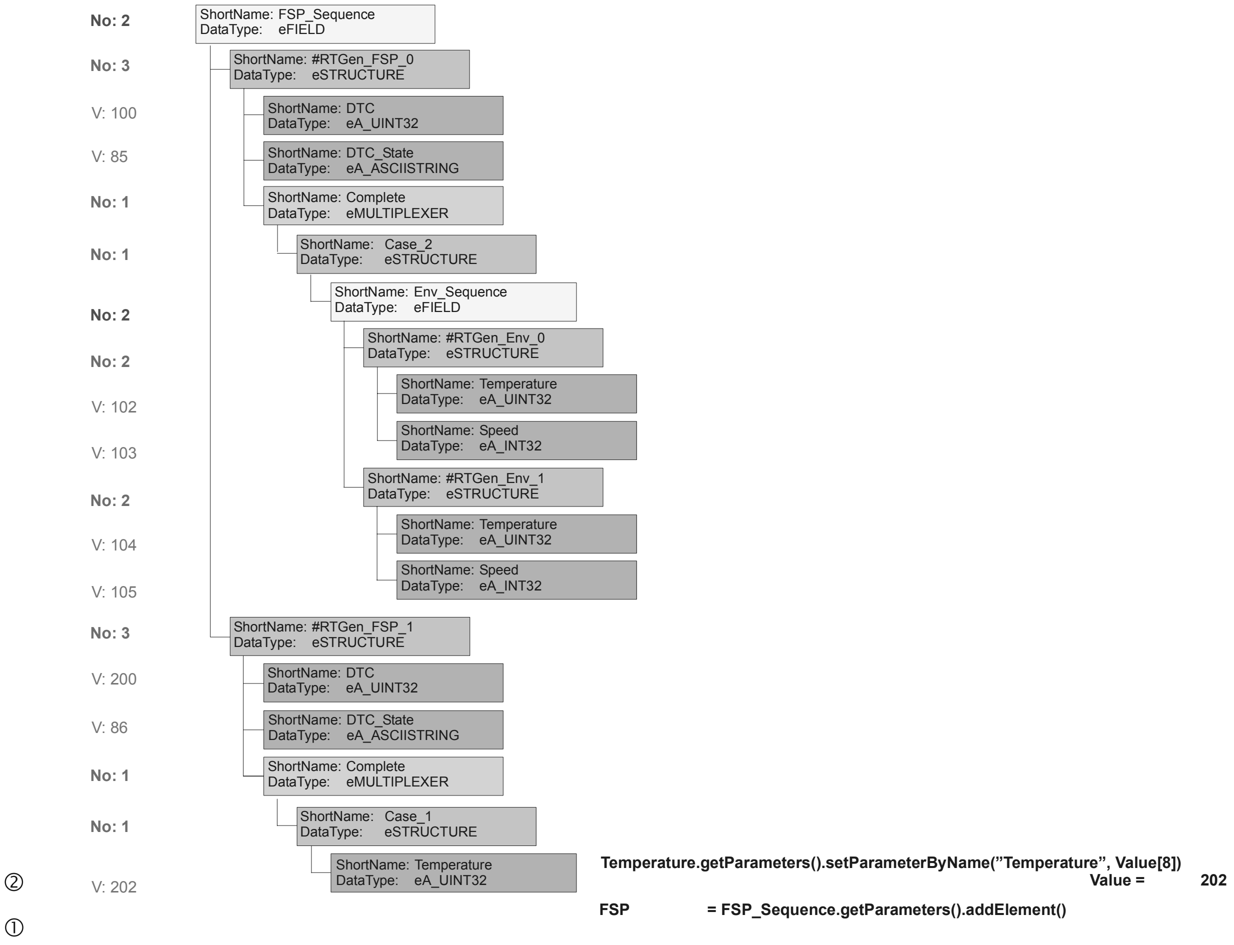
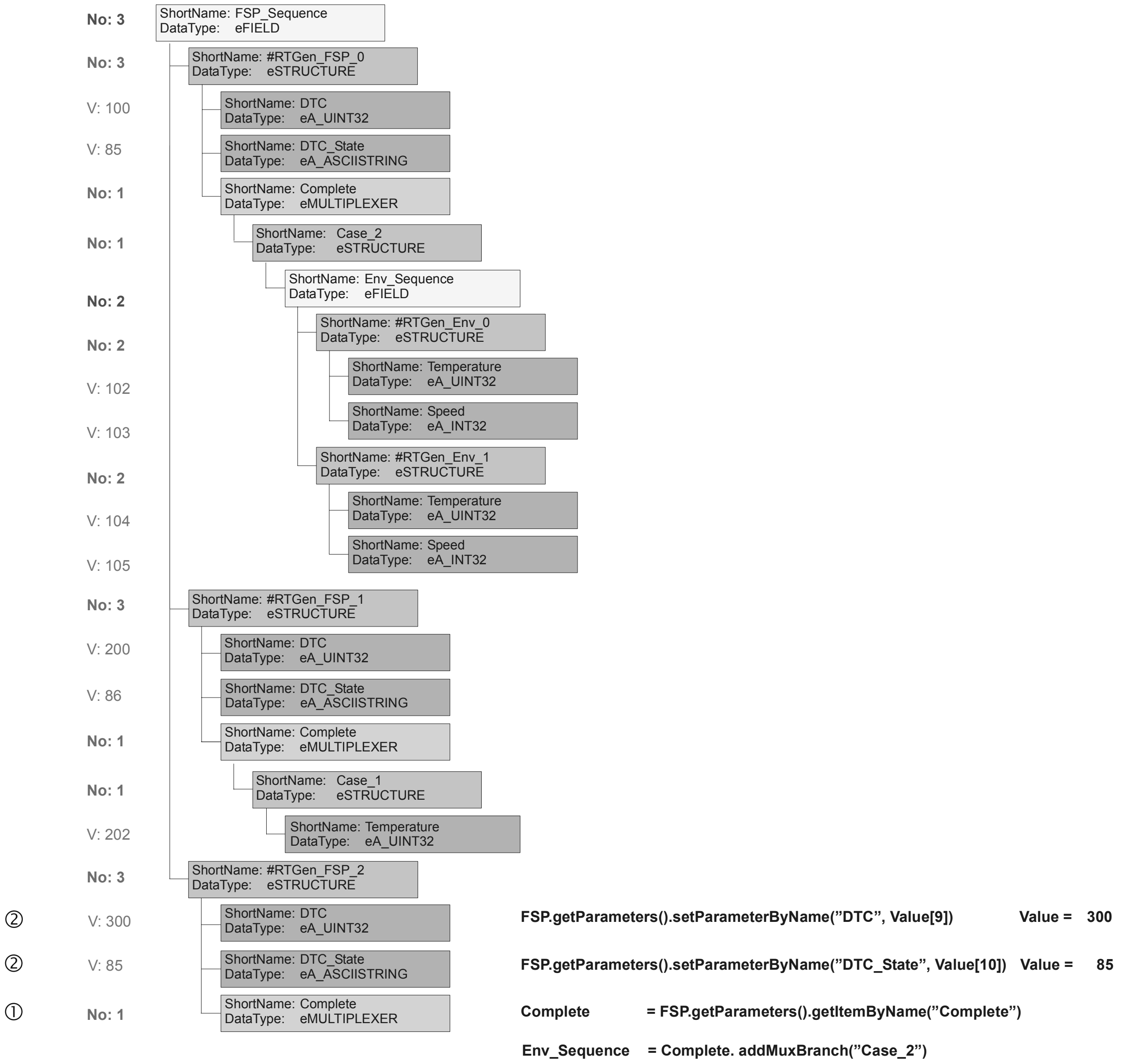


Figure 189 — Step 7 - addElement „FSP“

Within the second *FSP* in our example DTC, no dynamic elements exist, so the next step is to set the simple DOP *Temperature* with the respective value.

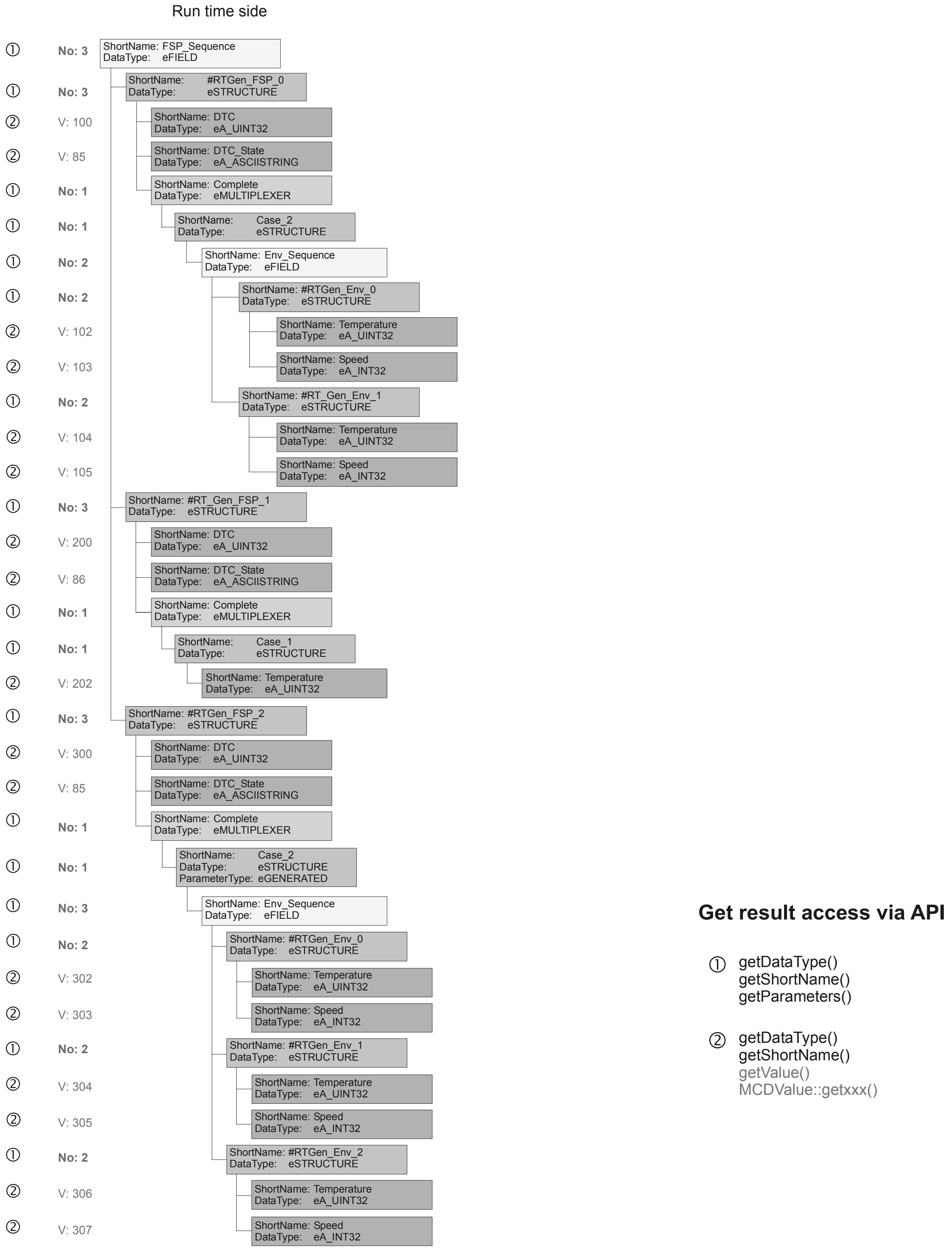
By means of calling `MCDResponseParameters::addElement(…)`, a third element is added to the FSP_Sequence.



```
FSP.getParameters().setParameterByName(”DTC”, Value[9])             Value =   300
FSP.getParameters().setParameterByName(”DTC_State”, Value[10])   Value =    85
Complete          = FSP.getParameters().getItemByName(”Complete”)
Env_Sequence   = Complete. addMuxBranch(”Case_2”)
```

Figure 190 — Step 8 - addMuxBranch (Case_2)

All further steps are carried out similarly to the previous steps, until the complete result has been constructed.


Run time side
① No: 3 ShortName: FSP_Sequence DataType: eFIELD
① No: 3 ShortName: #RTGen_FSP_0 DataType: eSTRUCTURE
② V: 100 ShortName: DTC DataType: eA_UINT32
② V: 85 ShortName: DTC_State DataType: eA_ASCIISTRING
① No: 1 ShortName: Complete DataType: eMULTIPLEXER
① No: 1 ShortName: Case_2 DataType: eSTRUCTURE
① No: 2 ShortName: Env_Sequence DataType: eFIELD
① No: 2 ShortName: #RTGen_Env_0 DataType: eSTRUCTURE
② V: 102 ShortName: Temperature DataType: eA_UINT32
② V: 103 ShortName: Speed DataType: eA_INT32
① No: 2 ShortName: #RT_Gen_Env_1 DataType: eSTRUCTURE
② V: 104 ShortName: Temperature DataType: eA_UINT32
② V: 105 ShortName: Speed DataType: eA_INT32
① No: 3 ShortName: #RT_Gen_FSP_1 DataType: eSTRUCTURE
② V: 200 ShortName: DTC DataType: eA_UINT32
② V: 86 ShortName: DTC_State DataType: eA_ASCIISTRING
① No: 1 ShortName: Complete DataType: eMULTIPLEXER
① No: 1 ShortName: Case_1 DataType: eSTRUCTURE
② V: 202 ShortName: Temperature DataType: eA_UINT32
① No: 3 ShortName: #RTGen_FSP_2 DataType: eSTRUCTURE
② V: 300 ShortName: DTC DataType: eA_UINT32
② V: 85 ShortName: DTC_State DataType: eA_ASCIISTRING
① No: 1 ShortName: Complete DataType: eMULTIPLEXER
① No: 1 ShortName: Case_2 DataType: eSTRUCTURE ParameterType: eGENERATED
① No: 3 ShortName: Env_Sequence DataType: eFIELD
① No: 2 ShortName: #RTGen_Env_0 DataType: eSTRUCTURE
② V: 302 ShortName: Temperature DataType: eA_UINT32
② V: 303 ShortName: Speed DataType: eA_INT32
① No: 2 ShortName: #RTGen_Env_1 DataType: eSTRUCTURE
② V: 304 ShortName: Temperature DataType: eA_UINT32
② V: 305 ShortName: Speed DataType: eA_INT32
① No: 2 ShortName: #RTGen_Env_2 DataType: eSTRUCTURE
② V: 306 ShortName: Temperature DataType: eA_UINT32
② V: 307 ShortName: Speed DataType: eA_INT32
Get result access via API
① getDataType() getShortName() getParameters()
② getDataType() getShortName() getValue() MCDValue::getxxx()

In case of Multiplexer the method getValue() delivers the index of the branch.

Figure 191 — Constructing the job result (example DTC)

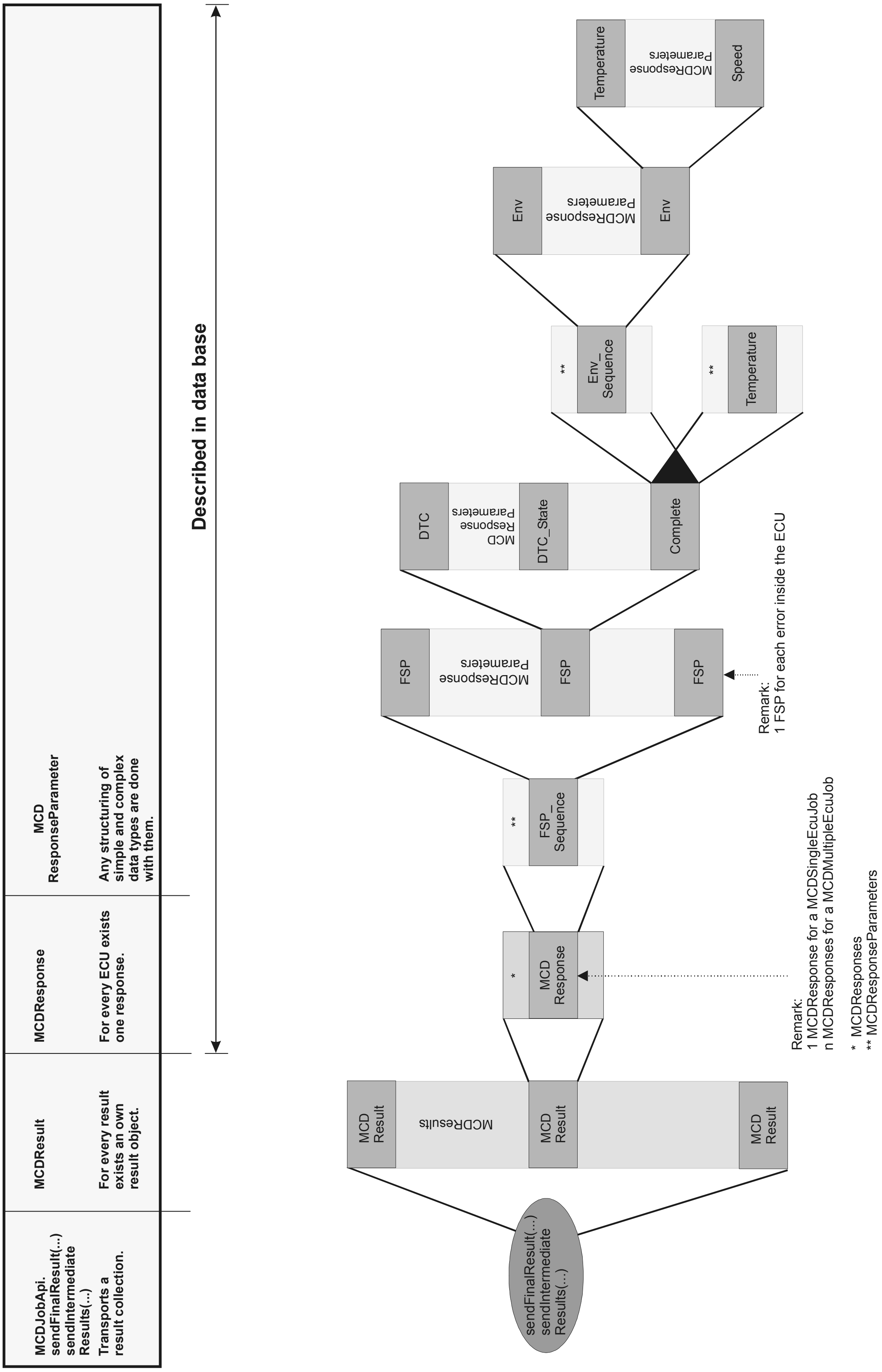
The result structure of a job is handed over to the D-server by calling the method `MCDJobAPI::sendFinalResult(MCDResult result) / sendIntermediateResults(MCDResults results)` from within the job code.

Our example job will be retrieving the contents of an ECU's error memory in one-minute cycles. To this end, the job will cyclically execute three (not further specified) diagnostic services. The results of these diagnostic services are returned unchanged as intermediate results of the job. As no functional groups are used, only one ECU will return results, which means we will have one `MCDResponse` object per service call. But, one call might result in more than one error.

As the job only generates an intermediate result every second minute, the intermediate results contains two independent result elements. Using sendIntermediateResults(…), the `MCDJobApi` hands over both results to the D-server. Subsequently, the D-server stores each result within the corresponding ring buffer and generates an event for each result `(MCDDiagComPrimitiveEventHandler::onPrimitiveHasIntermediateResult(…))`.

The final result of a job again comprises all intermediate results that were generated by the job. As only one final result can be delivered to the client application, the D-server generates an event of type `MCDDiagComPrimitiveEventHandler::onPrimitiveHasResult(…)` only once for each job execution. Because of this, all errors of one ECU are united within a combined field (*FSP_Sequence*; as only one error occurred in the example, we only have one MCDResponse object). If it was necessary to discern which errors have occurred at which point in time within the final result, it would be necessary to map every error into one FSP_Sequence element at the time of its occurrence. In this case the above example would end up creating ten *FSP_Sequence* elements which would contain the ten `MCDResponse` objects.

NOTE The output format of results is up to the diagnostic database definition of the job, what is described above is only an example. However, both intermediate and final results have to correspond to the job's database output parameter structure.



MCDJobApi.
sendFinalResult(...)
sendIntermediate
Results(...)
	MCDResult	MCDResponse	MCD
			ResponseParameter

Transports a result collection.	For every result exists an own result object.	For every ECU exists one response.	Any structuring of simple and complex data types are done with them.

Figure 192 — Job result structure (example DTC)

In the following, an alternative and more generalized variant is presented. The request includes several ECUs. The unification of errors into a *FSP_Sequence* element has been retained, to visualize the time relations of the errors.

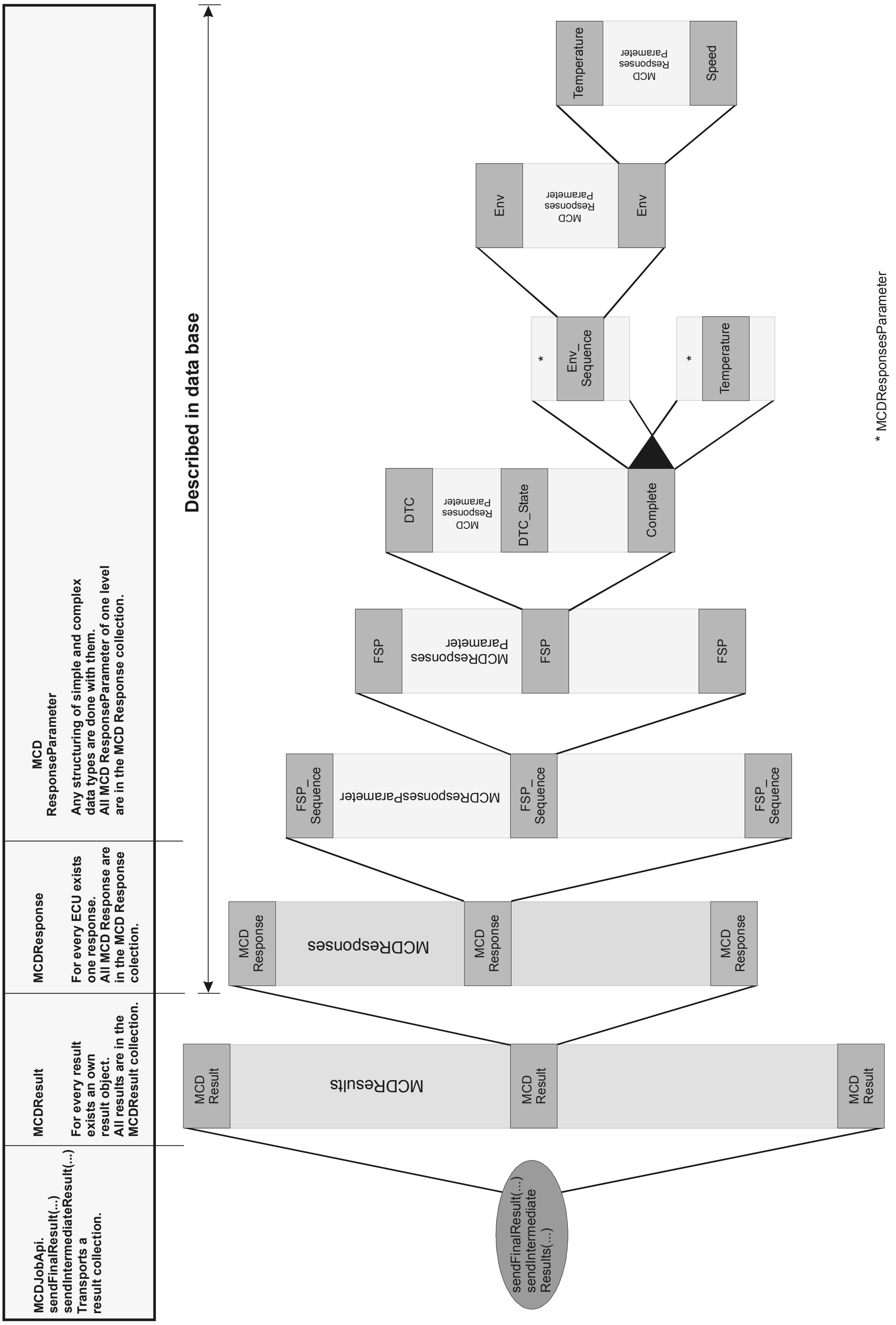


Figure 193 — Job result structure

9.23 ECU configuration

9.23.1 General

ECU configuration, also known as variant coding, describes the data elements and the process of configuring an ECU – either in the vehicle or in a test bench. The feature of ECU configuration is optional in a D-server. That means that a D-server implementation may not support ECU configuration and still be considered standard conformant.

In ECU configuration, an ECU is integrated into its vehicle environment (functional environment as well as electrical environment). For this purpose, two different types of configuration information can be distinguished – functional configuration and non-functional configuration information. By means of functional configuration information, setup information on the electrical and digital network within a vehicle is written to ECUs. For example, the presence of a GPS antenna system can be notified to an infotainment ECU. Furthermore, software features can be enabled and disabled by means of functional configuration information. In contrast, non-functional configuration information comprises all kinds of additional information which does not change a vehicles behaviour or functionality. For example, the vehicle colour or the interior colour can be written to an ECU by means of non-functional configuration information. This non-functional information can be used to, e.g. identify the correct replacement part in the service workshops.

Both, functional and non-functional configuration information are passed to or read from an ECU by means of configuration strings where each configuration string typically contains more than one piece of configuration information. That is, a single configuration string contains information on a set of configuration items where each configuration item either represents a functional or a non-functional configuration information. Indeed, the difference between functional and non-functional configuration information is not visible in a configuration string any more. Technically, a configuration string is represented by a byte array.

9.23.2 ECU Configuration Database Part

In this subclause, the database part of the ECU configuration functionality of the D-server API is described. As this database part is closely related to ODX structures, the ODX elements which correspond to a certain type of MCD-3 object are given in parentheses. The central terms and structures of the ECU configuration database part are illustrated in Figure 194 — Terms and Structure ECU Configuration Database Part. For more detail on the ODX representation of ECU configuration data, see ODX specification[11].

Similarly to the flash programming part of the D-server API, configuration information is grouped into container objects of type `MCDDbConfigurationData` (CONFIG-DATA). Several of these containers can be contained in the current project, that is, several `MCDDbConfigurationData` objects can be referenced from a `MCDDbProject`. A collection of the referenced containers can be obtained by means of the method `MCDDbProject::getDbConfigurationDatas()`.

For describing the structure and the content of a certain type of configuration string, each configuration string is associated with a database pattern. Such a database pattern is represented by an object of type `MCDDbConfigurationRecord` (CONFIG-RECORD). Multiple `MCDDbConfigurationRecord` objects can be contained in a configuration data container of type `MCDDbConfigurationData`. A `MCDDbConfigurationRecord` can have a unique identifier associated, the so-called configuration identifier (CONFIG-ID). The configuration ID allows to directly address a specific `MCDDbConfigurationRecord` in the scope of a `MCDDbConfigurationData`.

Every `MCDDbConfigurationRecord` can be composed of objects of type `MCDDbConfigurationItem` (CONFIG-ITEM). A `MCDDbConfigurationItem` represents a single piece of configuration information.

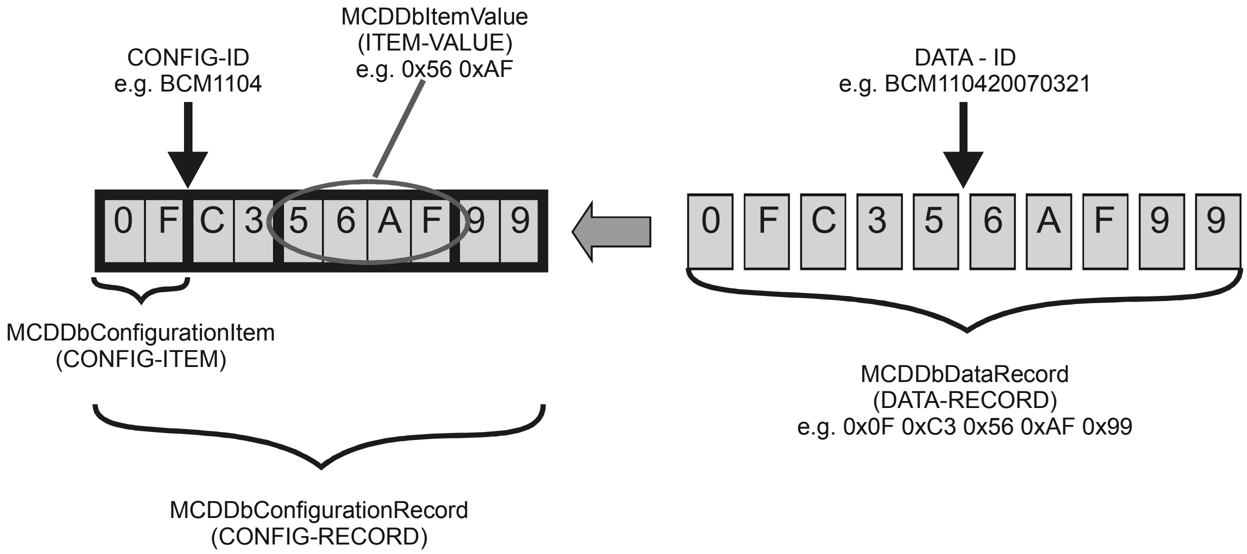


Figure 194 — Terms and Structure ECU Configuration Database Part

The following subclasses of `MCDDbConfigurationItem` exist, to be able to distinguish different types of configuration items:

- `MCDDbConfigurationIdItem` (CONFIG-ID-ITEM) – defines the position in a configuration string and the bytes that will be occupied by the configuration identifier of a `MCDDbConfigurationRecord` at runtime.
- `MCDDbDataIdItem` (DATA-ID-ITEM) – defines the position in a configuration string and the bytes that will be occupied by the data identifier of a `MCDDbDataRecord` that has been used to define a configuration string at runtime.
- `MCDDbSystemItem` (SYSTEM-ITEM) – defines the position and the bytes in a configuration string that will be filled with the value of the system parameter referenced by this element.
- `MCDDbOptionItem` (OPTION-ITEM) – defines the position and the bytes in a configuration string that represent a certain configuration option. Option items represent functional or non-functional configuration information which can be altered by the user. Similarly to request or response parameters, a value domain is associated with an option item.

In contrast to any other kind of `MCDDbConfigurationItem`, `MCDDbOptionItems` provide a physical default value and optionally a set of `MCDDbItemValue` objects. Similarly to a text table in case of a request or response parameter, this set of type `MCDDbItemValues` defines an enumeration of possible values of a `MCDDbOptionItem`. Every `MCDDbItemValue` has a constant physical value and optionally a meaning and a description. The physical value is the value to be placed in a configuration string at runtime, if the corresponding item value is selected. The meaning provides a human readable phrase which illustrates this option item's value. For example, the meaning 'available' could be assigned to a `MCDDbItemValue` which is associated with a `MCDDbOptionItem` named 'FogLights'.

The description of a `MCDDbItemValue` gives more elaborate information on the value's meaning and can be used to describe the effect caused in the ECU when setting this value. Both, meaning and description can be internationalised as already known from LONGNAMEs and DESCRIPTIONs in case of other MCD-3 objects. That is, an ID can be defined in the ODX data which is to be resolved against an external data space for the internationalised string.

Furthermore, a `MCDDbItemValue` can optionally have a key and a rule assigned. The key is a unique identifier of a `MCDDbItemValue` within its superior option item. It can be used to obtain a certain `MCDDbItemValue` without knowing its ShortName.

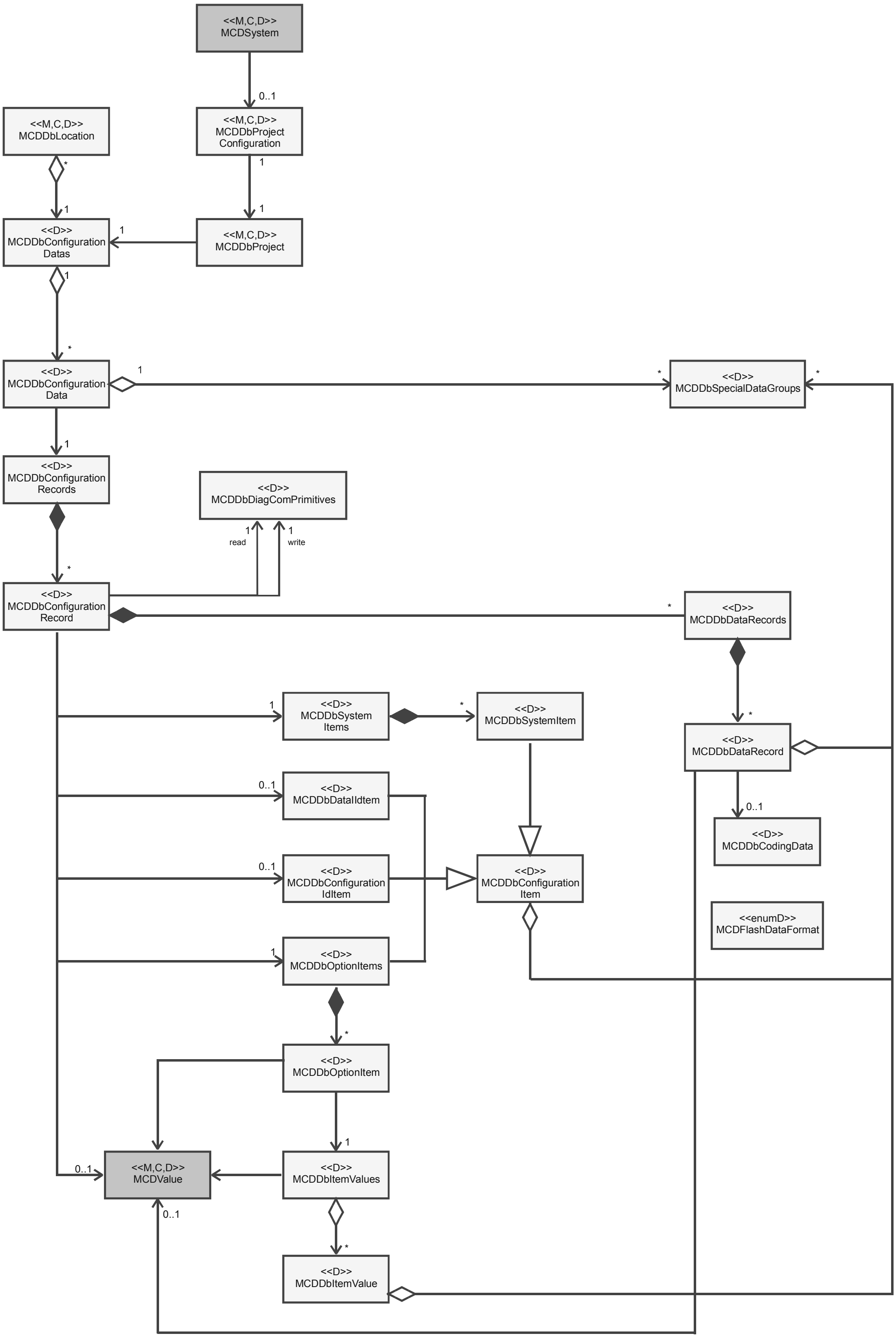


Figure 195 — ECU Configuration Model – Database Part

While Figure 195 — ECU Configuration Model – Database Part shows the classes that provide the database part of the ECU Configuration feature, Figure 196 — MCDAudience indicates audience-related meta-data as regards these elements.

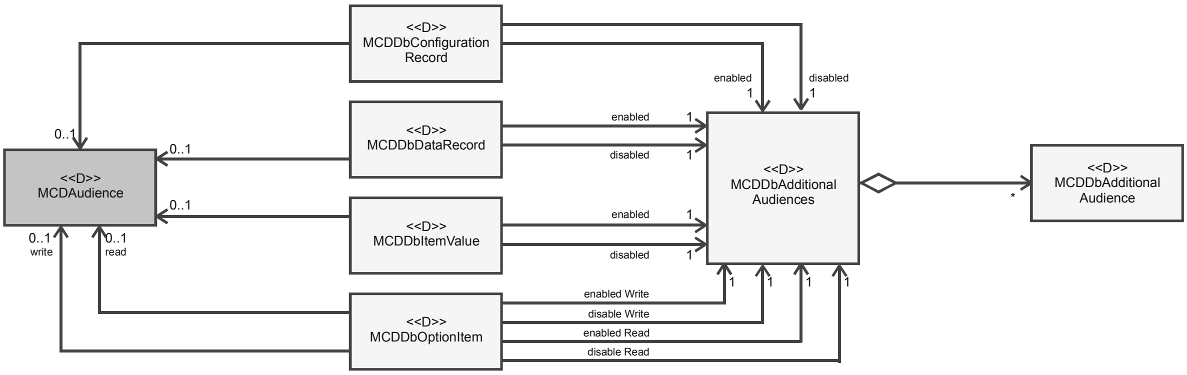


Figure 196 — MCDAudience

Default configuration strings which comply to a MCDDbConfigurationRecord are represented by objects of type MCDDbDataRecord (DATA-RECORD). That is, for every CONFIG-RECORD, the ODX data can contain zero or more default values, each represented by its own DATA-RECORD. To be able to uniquely identify and address a certain default configuration string, a MCDDbDataRecord is associated with a so-called data identifier (DATA-ID). Alternatively, the key assigned to a MCDDbDataRecord can be used.

Inside a MCDDbDataRecord, the configuration string is stored as binary data. This binary data can either be directly contained in the ODX data or it can be placed in an external file. In case of an external file, which is represented by an element of type MCDDbCodingData, the filename can be marked as 'late-bound'. Similarly to flash data, a late-bound file, is loaded by a D-server as late as possible. That is, it needs to be loaded latest when the binary data is accessed for the first time. Furthermore, the filename of a late-bound data file can contain wildcards. The wildcards which can be used in late-bound data files should conform to basic regular expressions (BRE) as supported by IEEE 1003.1-2001. In this case, the actual filename needs to be resolved at runtime. The filenames for data files in ODX should be given as Uniform Resource Identifiers (URIs). See [RFC 3305] and related documents for more details. In this case, the method MCDDbDataRecord::getBinaryData() does not return any data. For late-bound data files with wildcards in the filename, only the runtime part of ECU configuration is able to resolve the name.

The binary data contained in a MCDDbDataRecord or referenced by a MCDDbCodingData, respectively, can be formatted. The following data formats are supported (see flash programming also):

- Intel-Hex Records – formatted binary data including segment (address) information.

- Motorola-S Records – formatted binary data including segment (address) information.

- Plain binary data – binary large object without any segment or address information.

Segmented binary data referenced from a MCDDbDataRecord (intern or external) shall define values for at least all addresses from zero to size of configuration record minus 1. For each of these addresses, exactly one value shall be given, i.e. no gaps and no overlapping segments are allowed.

The size of a configuration record (configuration string) in bytes is the maximum of the sums of the SOURCE-START-ADDRESS and the UNCOMPRESSED-SIZE from its DIAG-COMM-DATA-CONNECTORs given by the ODX data.

If the value returned by MCDDbCodingData::isLateBound() is 'false' at a reference to an external resource file (e.g., job code, flash data, coding data) is considered a guarantee that the content of this resource file will not change while a D-server is running. More precisely, the content of the external resource file shall be static for the time between MCDSystem::selectProjectXXX() and MCDSystem::deselectProject() for the same project.

NOTE Exchanging external resource files may lead to non-deterministic behavior of a D-server. In particular, this statement holds in case of the LATEBOUND-DATAFILE attribute is set to 'true'.

9.23.3 ECU Configuration Runtime Part

The database part of ECU configuration merely describes the structure and the possible values of configuration strings or configuration items associated with an ECU. The ECU configuration runtime part provides the classes to create, modify, read, and write configuration strings. The corresponding classes are shown in Figure 198 — ECU Configuration Model – Runtime Part. An example illustrating the basics is shown in Figure 197 — Terms and Structure ECU Configuration Runtime Part.

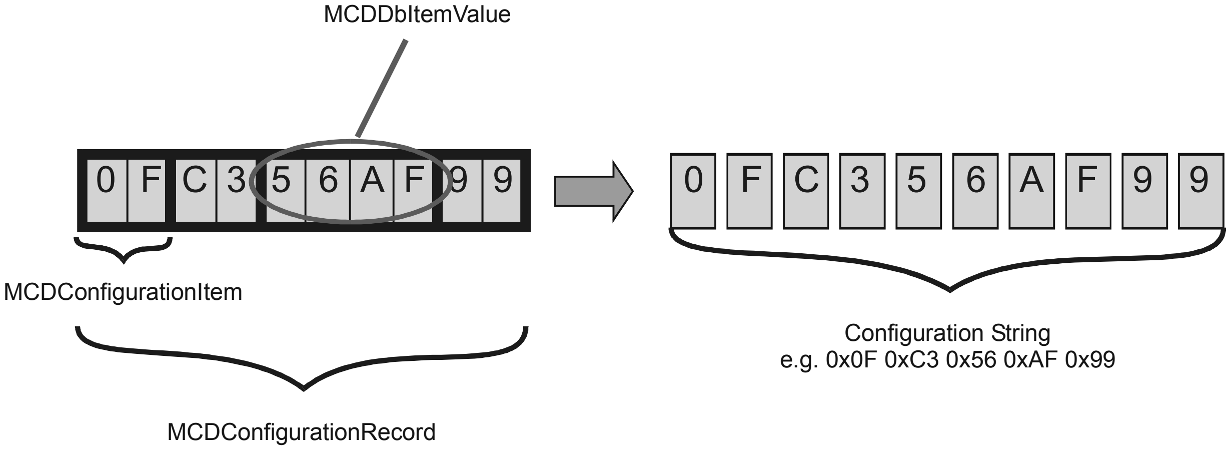


Figure 197 — Terms and Structure ECU Configuration Runtime Part

At runtime, a single configuration string is represented by the value of an object of type `MCDConfigurationRecord`. As a configuration string is specific to a `MCDDbLocation` of a certain ECU Base Variant or ECU Variant, every `MCDConfigurationRecord` is stored in a collection of type `MCDConfigurationRecords` which is a member of a `MCDDLogicalLink`. New configuration strings are created by adding a new `MCDConfigurationRecord` to a `MCDConfigurationRecords` collection. Existing configuration strings can be discarded by removing the corresponding `MCDConfigurationRecord` from the collection which contains this record. On removing a `MCDConfigurationRecord` from the collection, containing DiagComPrimitives will also be removed.

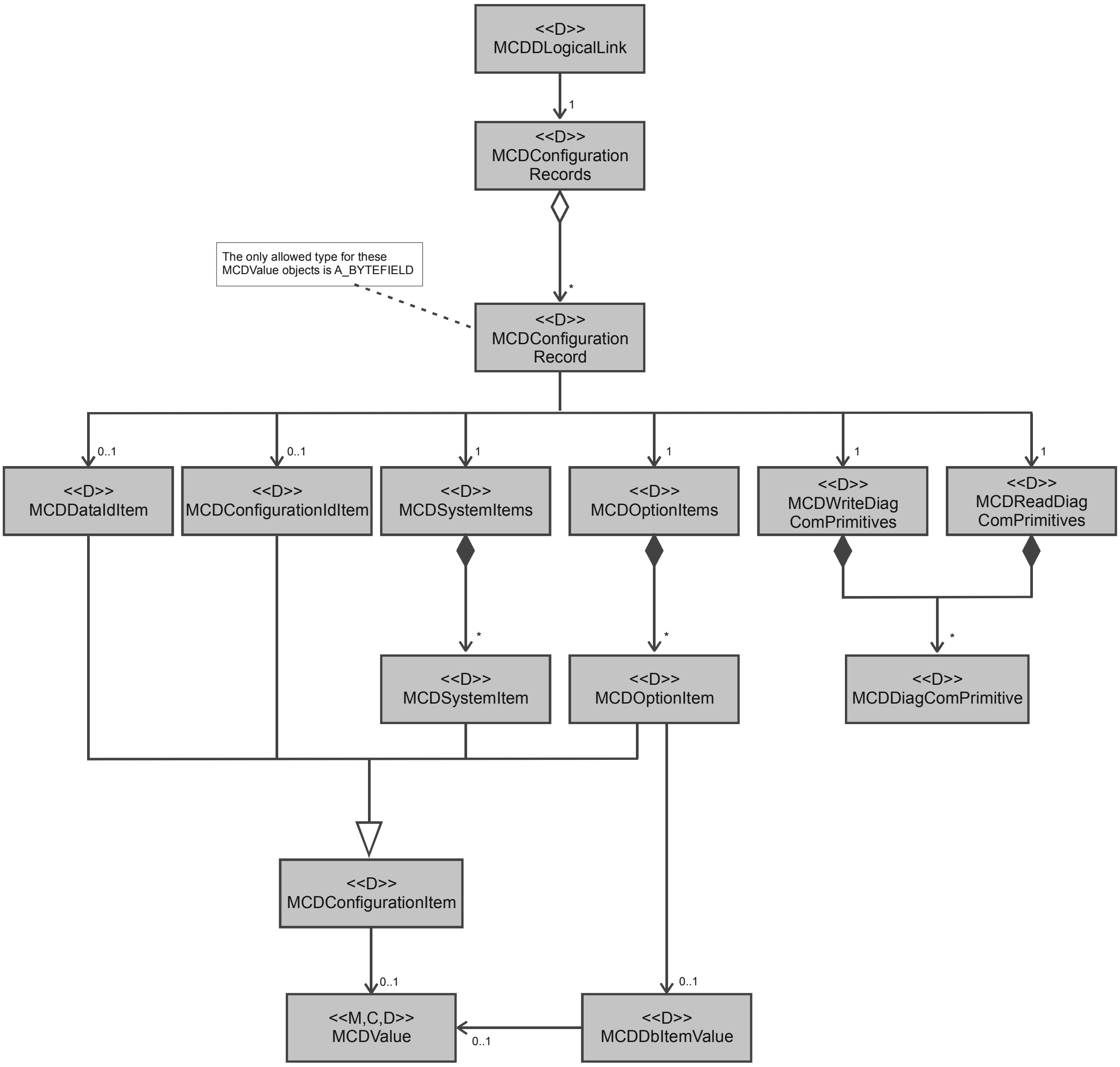

<<D>> MCDDLogicalLink
1
<<D>> MCDConfiguration Records
The only allowed type for these MCDValue objects is A_BYTEFIELD
*
<<D>> MCDConfiguration Record
0..1
0..1
1
1
1
1
<<D>> MCDDataIdItem
<<D>> MCDConfigurationIdItem
<<D>> MCDSystemItems
<<D>> MCDOptionItems
<<D>> MCDWriteDiag ComPrimitives
<<D>> MCDReadDiag ComPrimitives
*
*
*
<<D>> MCDSystemItem
<<D>> MCDOptionItem
<<D>> MCDDiagComPrimitive
<<D>> MCDConfigurationItem
0..1
0..1
<<M,C,D>> MCDValue
0..1
<<D>> MCDDbItemValue

Figure 198 — ECU Configuration Model – Runtime Part

Similarly to the database part, a `MCDConfigurationRecord` can be composed of objects of type `MCDConfigurationItem`. Again, different types of `MCDConfigurationRecords` exist:

- `MCDConfigurationIdItem` – defines the position in a configuration string and the bytes that will be occupied by the configuration identifier of the `MCDDbConfigurationRecord` a MCDConfigurationRecord is based on. The value of this element cannot be altered by a client application. Instead, the value is defined in the ODX data and will be automatically inserted into a configuration string by the D-server at runtime. In case of an upload of configuration information from an ECU, the value of a configuration ID item is read from the ECU.

- `MCDDataIdItem` – defines the position in a configuration string and the bytes that will be occupied by the data identifier of a `MCDDbDataRecord` that has been used to define a configuration string at runtime. The value of this element can only be indirectly altered by a client application. The value is defined in the ODX data and will be automatically inserted into a configuration string by the D-server at runtime whenever a `MCDDbDataRecord` is used to set the value of a `MCDConfigurationRecord` (see below). In case of an upload of configuration information from an ECU, the value of a data ID item is read from the ECU.

- MCDSystemItem – defines the position and the bytes in a configuration string that will be filled with the value of the system parameter referenced by this element. The value of this element cannot be altered by a client application. Instead, the value is calculated and will be inserted into a configuration string by the D-server at runtime. In case of an upload of configuration information from an ECU, the value of a system item is read from the ECU.

- MCDOptionItem – defines the position and the bytes in a configuration string that represent a certain configuration option. Option items represent functional or non-functional configuration information which can be altered by a client application. Similarly to request or response parameters of a diagnostic service, a value domain is associated with an option item. In case of an upload of configuration information from an ECU, the value of an option item is read from the ECU.

The value of a MCDConfigurationRecord can be set in different ways:

- by setting the value free-form (setConfigurationRecordValue(MCDValue configRecordValue));

- by setting the value by means of a selected MCDDbDataRecord (setConfigurationRecordValueByDbObject(MCDDbDataRecord dbDataRecord));

- by setting appropriate values at the MCDOptionItem objects referenced from a MCDConfigurationRecord.

In case of the former two options – entering a value free-form or overwriting the value with an MCDDbDataRecord –, the D-server needs to re-calculate the values of all MCDConfigurationItem objects contained in the current MCDConfigurationRecord, by decomposing and translating the value of the MCDConfigurationRecord appropriately. Whenever the value of a MCDOptionItem is changed (see below), the value of the containing MCDConfigurationRecord needs to be re-calculated by the D-server from the values of all contained MCDConfigurationItem objects.

The value of a MCDOptionItem can be changed either by entering a new value free-form (MCDOptionItem::setItemValue(MCDValue newValue)) or by assigning the value of a MCDDbItemValue (MCDOptionItem::setItemValueByDbObject(MCDDbItemValue dbItemValue)). In case a new MCDValue object is written to a MCDOptionItem, the value represented by this MCDValue object needs to match the value range of the MCDOptionItem. That is, the method MCDOptionItem::setItemValue(MCDValue newValue) can fail because of the following reasons:

- The data type of the value represented by the MCDValue object does not match the data type of the MCDOptionItem.

- The value represented by the MCDValue object is not within the value range (interval) defined for the MCDOptionItem. This value range can be obtained via MCDOptionItem::getDbObject()::getInterval().

- The value represented by the MCDValue object cannot be converted into a corresponding coded value. That is, the conversion associated with the corresponding MCDDbOptionItem in ODX failed.

- The value represented by the MCDValue object cannot be translated into a corresponding MCDDbItemValue. This translation does only take place in case the collection MCDDbItemValues is not empty at the corresponding MCDDbOptionItem. In this case, the method MCDOptionItem::getMatchingDbItemValue() throws an exception (MCDProgramViolationException, eRT_ELEMENT_NOT_AVAILABLE).

In case the value of a MCDOptionItem is changed by assigning a MCDDbItemValue, the value of the MCDOptionItem is defined as the physical default value of this MCDDbItemValue. The method MCDOptionItem::setItemValueByDbObject(MCDDbItemValue dbItemValue) fails, if the

MCDDbItemValue supplied as parameter is not defined for the MCDDbOptionItem which corresponds to the current MCDOptionItem. However, if an MCDOptionItem's value can be set by using an MCDDbItemValue, the method MCDOptionItem::getMatchingDbItemValue() does deliver the MCDDbItemValue which corresponds to the current value of the MCDOptionItem.

9.23.4 Error Handling

On initialization or when setting a ConfigurationRecord, various errors can occur. Regarding to the MCDResponse, the method MCDConfigurationRecord::hasErrors() returns 'true', if the ConfigurationRecord is faulty. The containing MCDErrors can be requested via method MCDConfigurationRecord::getErrors().

In case of a ConfigurationRecord initialization or uploading a configuration string from an ECU to a ConfigurationRecord that contains ConfigurationItems, e.g. OptionItems, internal to physical value conversion errors can occur. Such an error will be added to the corresponding ConfigurationItem. Regarding to the MCDResponseParameter, the MCDConfigurationItem offers the methods hasError() and getError().

If one of the ConfigurationItems of the ConfigurationRecord has an error or the ConfigurationRecord could not be initialized with its DataRecord, at least one error with the error code eRT_CONFIGRECORD_INVALID is contained in the ConfigurationRecord. The error will be resetted when the configuration string becomes valid, regarding to the ODX template.

If the execution of a ConfigurationRecord's Write-/ReadDiagComPrimitive contains errors, these errors are also adopted to the ConfigurationRecord. The Write-/ReadDiagComPrimitives of the ConfigurationRecord need to be executed in the order, given from ODX data. If the Write-/ReadDiagComPrimitives are executed in the wrong order, an error with the error code eRT_WRONG_SERVICE_EXECUTION_ORDER is added to the ConfigurationRecord. If one but not all ReadDiagComPrimitives are executed, the error with the error code eRT_CONFIGRECORD_INCOMPLETE is added to the ConfigurationRecord. The incomplete execution of WriteDiagComPrimitives does not modify the ConfigRecord. In this case, the error code eRT_CONFIGRECORD_INCOMPLETE is not added to the ConfigRecord.

The errors of a ConfigurationRecord and its ConfigurationItems will be resetted when one of the following methods from a MCDConfigurationRecord object is called: setConfigurationRecordValue (MCDValue configRecordValue), loadCodingData(A_ASCIISTRING fileName), setConfigurationRecordValueByDbObject(MCDDbDataRecord dbDataRecord), getReadDiagComPrimitives(), getWriteDiagComPrimitives().

An error of a ConfigurationItem will be resetted when the ConfigurationItem is set with a new value. This is the case when the configuration string of a ConfigurationRecord is set, e.g. via MCDConfigurationRecord::setConfigurationRecordValue(MCDValue configRecordValue). In case of an OptionItem, the new value can also be set via the MCDOptionItem methods setItemValue(MCDValue value) and setItemValueByDbObject(MCDDbItemValue dbItemValue).

9.23.5 Initialising an MCDConfigurationRecord

Whenever a new MCDConfigurationRecord is created at a Logical Link, the D-server needs to initialise the value of the configuration record such that at first valid configuration string is created within this configuration record. The initialisation of such a MCDConfigurationRecord shall be performed as defined by the following rules (only one rule can apply to a MCDConfigurationRecord):

— If no MCDDbDataRecord objects are defined for the MCDDbConfigurationRecord the MCDConfigurationRecord is based on and if no MCDDbOptionItems and no MCDDbItemValues are given in the ODX data for this MCDConfigurationRecord, the MCDConfigurationRecord's value (MCDValue of type A_BYTEFIELD) will not be initialized with a value. As a result, the method MCDValue::isValid() returns 'false' for this MCDValue.

- If no MCDDbDataRecords and no MCDDbItemValues are given for the current MCDConfigurationRecord in the ODX data but if this MCDConfigurationRecord contains MCDOptionItems, then the MCDConfigurationRecord's value will be initialized with the physical default values of all these option items. These physical default values are mandatory for every MCDDbOptionItem.

- If no MCDDbDataRecords are given in the ODX data for the MCDDbConfigurationRecord the current MCDConfigurationRecord is based on but if option items and item values are given, the MCDConfigurationRecord's value will be initialized with the physical default values of all option items. The physical default value shall match a MCDDbItemValue of the corresponding MCDDbOptionItem.

- If exactly one MCDDbDataRecord is referenced from the MCDDbConfigurationRecord the current MCDConfigurationRecord is based on and if no option items or item values are given, then this MCDDbDataRecord will be used to initialise the MCDConfigurationRecord's value. In this case it is not relevant if a DEFAULT-DATA-RECORD (MCDDbConfigurationRecord::getDbDefaultDataRecord()) references a MCDDbDataRecord or not.

- If more than one MCDDbDataRecord is referenced from the MCDDbConfigurationRecord the current MCDConfigurationRecord is based on, no DEFAULT-DATA-RECORD MCDDbConfigurationRecord::getDbDefaultDataRecord() references a MCDDbDataRecord and if no option items or item values are given, then this MCDConfigurationRecord's value (MCDValue of type A_BYTEFIELD) will not be initialised with a value. As a result, the method MCDValue::isValid() returns false for this MCDValue.

- If more than one MCDDbDataRecord is referenced from the MCDDbConfigurationRecord the current MCDConfigurationRecord is based on, a DEFAULT-DATA-RECORD MCDDbConfigurationRecord::getDbDefaultDataRecord() references a MCDDbDataRecord and if no option items or item values are given, then this referenced MCDDbDataRecord (DEFAULT-DATA-RECORD) will be used to initialise the MCDConfigurationRecord's value.

- If exactly one MCDDbDataRecord, MCDDbOptionItems, and MCDDbItemValues are given for the MCDDbConfigurationRecord the current MCDConfigurationRecord is based on, then the MCDConfigurationRecord's value will be initialised with the MCDDbDataRecord. In this case it is not relevant if a DEFAULT-DATA-RECORD MCDDbConfigurationRecord::getDbDefaultDataRecord() references a MCDDbDataRecord or not.

- If more than one MCDDbDataRecord, MCDDbOptionItems, and MCDDbItemValues are given for the MCDDbConfigurationRecord the current MCDConfigurationRecord is based on, then the MCDConfigurationRecord's value will be initialised with the physical default values of all option items. In this case it is not relevant if a DEFAULT-DATA-RECORD MCDDbConfigurationRecord::getDbDefaultDataRecord() references a MCDDbDataRecord or not.

9.23.6 Offline versus Online Configuration

Two different use cases have been defined in the context of ECU configuration – offline configuration and online configuration. For offline configuration, it is required to be able to define new configuration strings or to modify existing configuration strings without communicating with the corresponding ECU, e.g. because it is currently not available in a vehicle or a test bench. In terms of MCD-3, offline configuration means that it shall be possible to create or modify a MCDConfigurationRecord, e.g. by assigning a new value to this MCDConfigurationRecord or one of its MCDConfigurationItem objects. To support offline configuration, it is possible to add new MCDConfigurationRecord objects to the MCDConfigurationRecords collection associated with a LogicalLink while this link is in state eCREATED. In the same state, it is also possible to modify a MCDConfigurationRecord contained in the collection. In logical link state eCREATED, no communication to an ECU is running.

For online configuration, it is required to be able to initialise the value of a MCDConfigurationRecord object with the current configuration string as stored in the corresponding ECU. Online configuration is supported in a D-server by being able to use the same MCDConfigurationRecord to read a configuration string from an ECU (upload, see 9.23.7.3) and to write this configuration string back to the ECU after modification (download, see 9.23.7.2). Furthermore, it is possible to initialise a new MCDConfigurationRecord with the value of a default configuration string, that is, with the value of a MCDDbDataRecord (see 9.23.5). As a result, the following scenarios in online configuration can be realised in a client application:

- upload of current configuration data from an ECU;
- upload of current configuration data from an ECU and modification of this configuration data;
- modification of a configuration record before download;
- upload, modification and download of a configuration record;
- initialisation of a configuration record with a selected data record;
- initialisation of a configuration record with a selected data record followed by a download;
- …

9.23.7 Uploading and Downloading Configuration Strings

Configuration strings can be written to an ECU (download) and can be read from an ECU (upload). This subclause describes how the upload and download of configuration strings can be performed with a D-server.

In general, a configuration string can exceed the maximum size of a D-PDU which can be transferred to and from an ECU via a LogicalLink and the diagnostic protocol the communication via this link is based on. Based on the ODX schema, a configuration string can be transferred to and from an ECU in several fragments.

9.23.7.1 Decomposing a Configuration String for Transfer

Every configuration string is represented by a MCDConfigurationRecord in the D-server. A configuration string is transferred to and from an ECU by means of the DiagComPrimitives referenced from such a MCDConfigurationRecord. For each of the directions upload and download, a separate set of DiagComPrimitives is referenced (see Figure 195 — ECU Configuration Model – Database Part and Figure 198 — ECU Configuration Model – Runtime Part) – a set of WriteDiagComPrimitives (download) and a set of ReadDiagComPrimitives (upload).

If a configuration string exceeds the maximum size that can be transferred with a single DiagComPrimitive, each of the sets of ReadDiagComPrimitives and WriteDiagComPrimitives contains one DiagComPrimitive for every piece the configuration strings needs to be decomposed into. The information on which piece is to be transferred by which DiagComPrimitive and the size of these pieces is described by means of so-called DIAG-COMM-DATA-CONNECTORs in the ODX data (see Figure 168 “UML representation of ECU configuration: DIAG-COMM-DATA-CONNECTOR” in ASAM MCD 2 D ODX). These DIAG-COMM-DATA-CONNECTORs are only used internally within a D-server, they are not exposed at the server’s API.

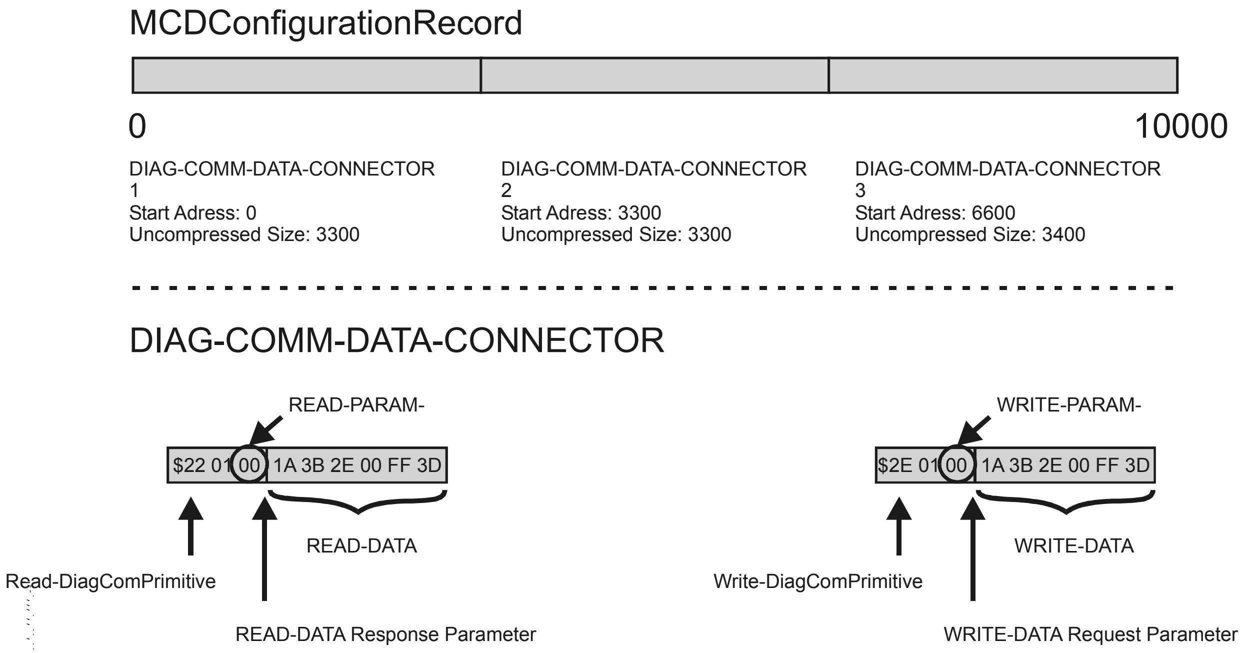


Figure 199 — Example of a configuration string which is decomposed for transfer

Every DIAG-COMM-DATA-CONNECTOR refers to a piece of configuration string, a DiagComPrimitive, request and/or response parameters, and optionally elements of type READ-PARAM-VALUE or WRITE-PARAM-VALUE. The piece of the configuration string is identified by a start address in the configuration string (starting at zero) and an uncompressed size of the piece in bytes (see Figure 199 — Example of a configuration string which is decomposed for transfer for illustration). The DiagComPrimitive (read or write) to be used as well as the request parameter to place the piece of configuration string in (WriteDiagComPrimitive) or the response parameter to read the piece of configuration string from (ReadDiagComPrimitive) are referenced directly. READ-PARAM-VALUEs can be used to obtain additional information on the current piece of configuration string, e.g. its number, from the response of a ReadDiagComPrimitive. WRITE-PARAM-VALUEs can be used to put additional information on the current piece of configuration string, e.g. its number, in the request of a WriteDiagComPrimitive.

9.23.7.2 Downloading Configuration Records to an ECU

A single configuration string is written to an ECU, i.e. downloaded to this ECU as follows. First the collection of `MCDWriteDiagComPrimitives` needs to be obtained from the corresponding `MCDConfigurationRecord` by means of the method `getWriteDiagComPrimitives()`. The content of this collection is defined as the DiagComPrimitives referenced from the corresponding WRITE-DIAG-COMM-CONNECTORs in the ODX data.

The elements in the `MCDWriteDiagComPrimitives` collection of DiagComPrimitives are ordered, regarding to the order in the ODX data. If this collection contains more than one element, this indicates that the configuration string represented by the current `MCDConfigurationRecord` is too large to be transferred to the ECU by means of a single DiagComPrimitive. That is, the size of the configuration string exceeds the maximum size of a DiagComPrimitive's request in the current diagnostic protocol. In this case, the configuration string needs to be decomposed into pieces as described in 9.23.7.1.

In the next step, all WriteDiagComPrimitives referenced by this configuration record need to be executed in the order defined in the `MCDWriteDiagComPrimitives` collection obtained in the first step. Therefore, the request parameters of every WriteDiagComPrimitive in this collection need to be filled with data as described by the corresponding DIAG-COMM-DATA-CONNECTOR. All request parameters in the WriteDiagComPrimitive which are not explicitly filled with a value from the information in the corresponding

DIAG-COMM-DATA-CONNECTOR shall set to default values by the D-server. If a WriteDiagComPrimitive cannot be executed after all request parameters have been set as described above, e.g. because at least one request parameter does not have a default value, the execution of a WriteDiagComPrimitive fails (MCDParameterizationException, ePAR_INCOMPLETE_PARAMTERIZATION) and, as a consequence, the download of the configuration string fails.

WriteDiagComPrimitives can be executed either synchronously or asynchronously. However, the D-server does not execute any WriteDiagComPrimitives automatically. Instead, the client application is responsible for executing the WriteDiagComPrimitives in the correct order. Only the preparation of the request parameters is performed by the D-server.

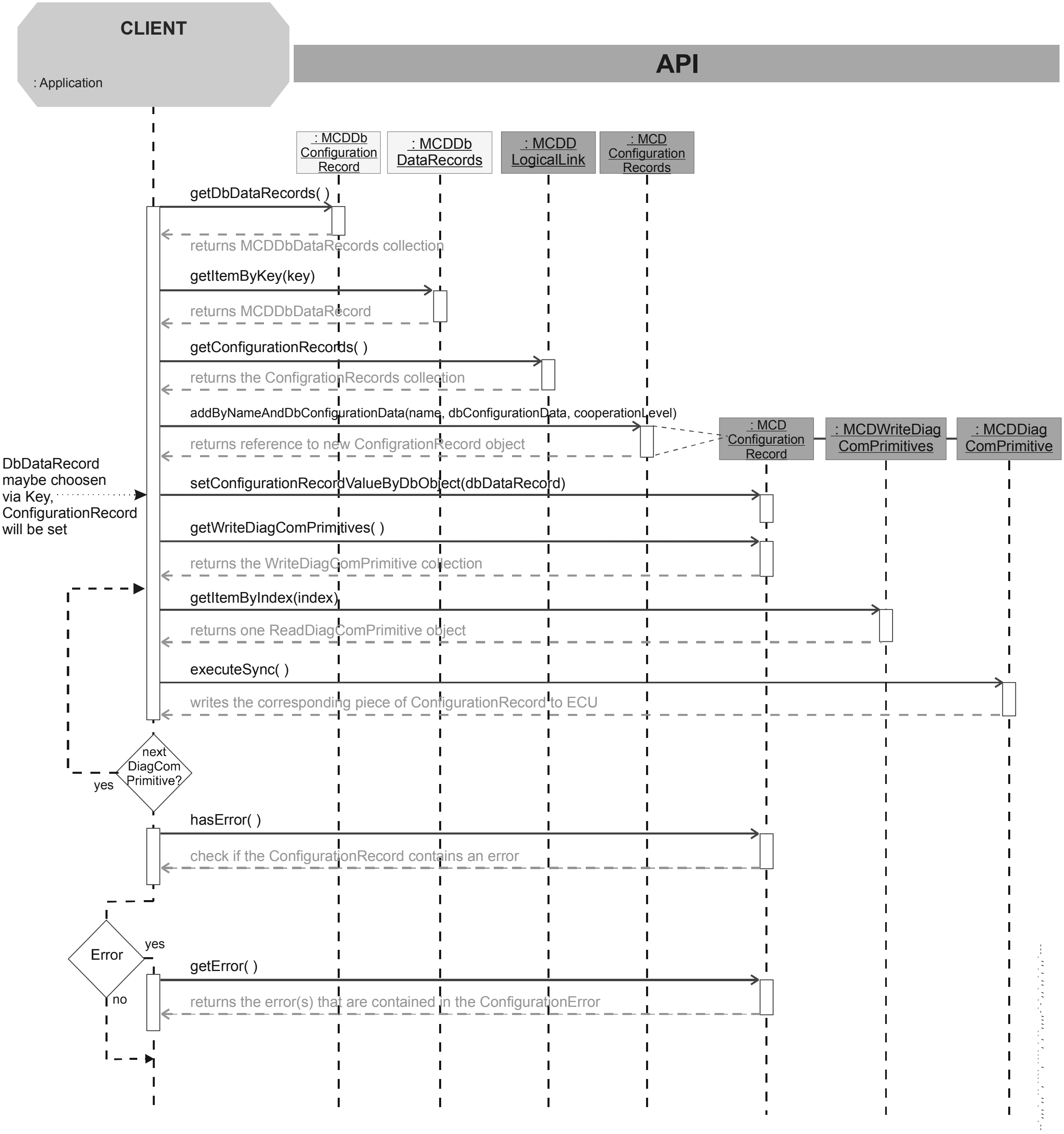

CLIENT
: Application
API
: MCDDb Configuration Record
: MCDDb DataRecords
: MCDD LogicalLink
: MCD Configuration Records
getDbDataRecords( )
returns MCDDbDataRecords collection
getItemByKey(key)
returns MCDDbDataRecord
getConfigurationRecords( )
returns the ConfigrationRecords collection
addByNameAndDbConfigurationData(name, dbConfigurationData, cooperationLevel)
returns reference to new ConfigrationRecord object
: MCD Configuration Record
: MCDWriteDiag ComPrimitives
: MCDDiag ComPrimitive
DbDataRecord maybe choosen via Key, ConfigurationRecord will be set
setConfigurationRecordValueByDbObject(dbDataRecord)
getWriteDiagComPrimitives( )
returns the WriteDiagComPrimitive collection
getItemByIndex(index)
returns one ReadDiagComPrimitive object
executeSync( )
writes the corresponding piece of ConfigurationRecord to ECU
next DiagCom Primitive?
yes
hasError( )
check if the ConfigurationRecord contains an error
Error
yes
no
getError( )
returns the error(s) that are contained in the ConfigurationError

Figure 200 — Example workflow for writing a configuration string to an ECU

A sample workflow for writing a single configuration string to an ECU is shown in Figure 200 — Example workflow for writing a configuration string to an ECU. In this example, a MCDDbDataRecord is obtained from the corresponding MCDDbConfigurationRecord first. Then, a new ConfigurationRecord is created at the runtime LogicalLink. Next, the value of this ConfigurationRecord is set by using the MCDDbDataRecord obtained in the first step. Finally, the value of the ConfigurationRecord, that is, the configuration string, is written to the ECU. For this purpose, all WriteDiagComPrimitives referenced at the ConfigurationRecord are executed in the correct order.

9.23.7.3 Uploading Configuration Records from an ECU

A single configuration string is read from an ECU, i.e. uploaded from this ECU as follows. First the collection of MCDReadDiagComPrimitives needs to be obtained from the corresponding MCDConfigurationRecord by means of the method getReadDiagComPrimitives(). The content of this collection is defined as the DiagComPrimitives referenced from the corresponding READ-DIAG-COMM-CONNECTORs in the ODX data.

The elements in the MCDReadDiagComPrimitives collection of DiagComPrimitives are ordered, regarding to the order in the ODX data. If this collection contains more than one element, this indicates that the configuration string represented by the current MCDConfigurationRecord is too large to be transferred from the ECU by means of a single DiagComPrimitive. That is, the size of the configuration string exceeds the maximum size of a DiagComPrimitive's response in the current diagnostic protocol. In this case, the configuration string needs to be composed from the different pieces of configuration information read from the ECU.

In the next step, all ReadDiagComPrimitives referenced by this configuration record need to be executed in the order defined MCDReadDiagComPrimitives collection obtained in the first step. Therefore, the request parameters of every ReadDiagComPrimitive in this collection need to be filled with data as described by the corresponding DIAG-COMM-DATA-CONNECTOR. All request parameters in the ReadDiagComPrimitive which are not explicitly filled with a value from the information in the corresponding DIAG-COMM-DATA-CONNECTOR shall set to default values by the D-server. If a ReadDiagComPrimitive cannot be executed after all request parameters have been set as described above, e.g. because at least one request parameter does not have a default value, the execution of a ReadDiagComPrimitive fails (MCDParameterizationException, ePAR_INCOMPLETE_PARAMTERIZATION) and, as a consequence, the upload of the configuration string fails.

ReadDiagComPrimitives can be executed either synchronously or asynchronously. However, the D-server does not execute any ReadDiagComPrimitive automatically. Instead, the client application is responsible for executing the ReadDiagComPrimitives in the correct order. Only the preparation of the request parameters is performed by the D-server.

In the final step, the D-server needs to write the configuration information which has been read from the ECU into the current MCDConfigurationRecord. For this purpose, the method setConfigurationRecordValue(MCDValue configRecordValue) can be used by a D-server implementation internally. Prior to setting the value of the MCDConfigurationRecord, the value needs to be created by concatenating the values of all those response parameters which are referenced from the ReadDiagComPrimitives in the MCDReadDiagComPrimitives collection. The concatenation needs to take place in the order of execution as defined by the MCDReadDiagComPrimitives collection. That is, the return value of the second ReadDiagComPrimitive is appended to the return value of the first ReadDiagComPrimitive, the third value is appended to the second, and so forth.

A sample workflow for reading a single configuration string from an ECU is shown in Figure 201 — Example workflow for reading a configuration string from an ECU. In this example, a new ConfigurationRecord is created at a LogicalLink first. Then, the value of this ConfigurationRecord is read from the ECU by means of the ReadDiagComPrimitives referenced from this ConfigurationRecord. Finally, the (byte) value of the configuration string is read from this ConfigurationRecord as well as the value and the meaning of every single OptionItem which is contained in the ConfigurationRecord.

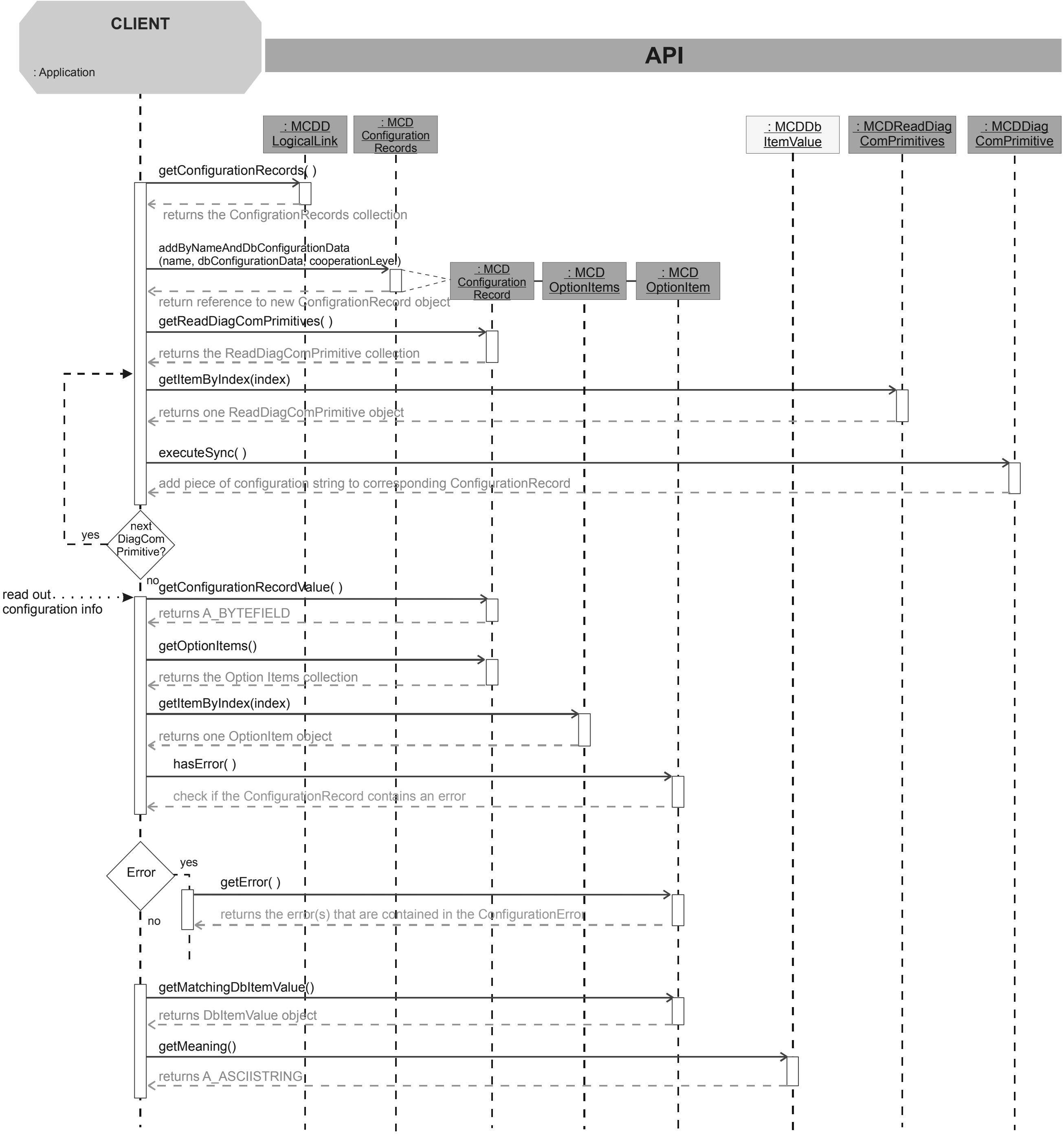

CLIENT
: Application
API
: MCDD LogicalLink
: MCD Configuration Records
: MCDDb ItemValue
: MCDReadDiag ComPrimitives
: MCDDiag ComPrimitive
getConfigurationRecords( )
returns the ConfigrationRecords collection
addByNameAndDbConfigurationData (name, dbConfigurationData, cooperationLevel)
: MCD Configuration Record
: MCD OptionItems
: MCD OptionItem
return reference to new ConfigrationRecord object
getReadDiagComPrimitives( )
returns the ReadDiagComPrimitive collection
getItemByIndex(index)
returns one ReadDiagComPrimitive object
executeSync( )
add piece of configuration string to corresponding ConfigurationRecord
yes
next DiagCom Primitive?
no
read out configuration info
getConfigurationRecordValue( )
returns A_BYTEFIELD
getOptionItems()
returns the Option Items collection
getItemByIndex(index)
returns one OptionItem object
hasError( )
check if the ConfigurationRecord contains an error
Error
yes
no
getError( )
returns the error(s) that are contained in the ConfigurationError
getMatchingDbItemValue()
returns DbItemValue object
getMeaning()
returns A_ASCIISTRING

Figure 201 — Example workflow for reading a configuration string from an ECU

9.23.7.4 Management of CONFIG-DATAs

In ODX, a CONFIG-DATA references the ECU-VARIANTs and BASE-VARIANTs it is valid for through the element VALID-BASE-VARIANT. The methods MCDDbConfigurationData::getLongName(), MCDDbConfigurationData::getShortName(), and MCDDbConfigurationData::getDescription() return the corresponding values of a CONFIG-DATA in ODX.

At runtime, it shall be possible to load new ConfigurationRecords into an existing project. These new ConfigurationRecords need to be contained in a separate CONFIG-DATA container. For loading new CONFIG-DATAs into a running project, the D-server can be in any state except eINITIALIZED. The corresponding MCDDLogicalLink can be in any state. If additional data is added, the system event onSystemDbConfigurationDatasModified (MCDDbConfigurationDatas configurationDatas) will be fired. So, the clients are informed about these changes. The collection MCDDbConfigurationDatas shall be reloaded by the client at the MCDDbLocation or at the MCDDbProject.

New CONFIG-DATAs including their contained ConfigurationRecords can be loaded into a MCDDbProject temporarily or they can be added to a project configuration permanently. Adding a CONFIG-DATA temporarily to a project means that the CONFIG-DATA is not present if the same project is deselected and re-selected again. That is, temporary CONFIG-DATAs need not to be removed from a project when this project is deselected. Adding a CONFIG-DATA permanently to a project means that this CONFIG-DATA is also available after this project has been deselected and re-selected again. This might include copying the corresponding CONFIG-DATA file to the D-server's data space for the corresponding project.

It is possible to list all CONFIG-DATAs within the current project by calling the method getDbConfigurationDatas()at the corresponding MCDDbProject.

To list all CONFIG-DATAs which could be loaded into a project but which have not been loaded into the project yet, the method MCDDbProjectConfiguration:: getAdditionalConfigurationDataNames(): MCDDatatypeAsciiStrings can be used. This method issues the D-server to search for CONFIG-DATA containers in the boundaries defined by the server's paths. The CONFIG-DATA containers found are matched against those already being part of the current project. Finally, only those CONFIG-DATA containers are presented as a result of the method, which are not part of the project yet.

NOTE Two calls of the method can deliver different results.

By means of the methods

- MCDDbProject::loadNewConfigurationData (MCDDatatypeShortName configurationDataName, A_BOOLEAN permanent=false)

- MCDDbProject::loadNewConfigurationDatasByFilename(A_ASCIISTRING filename, A_BOOLEAN permanent=false)

a CONFIG-DATA, which is not part of the currently active project, is loaded into the project. The parameter permanent controls whether this CONFIG-DATA is added to the project permanently or not.

The method MCDDbProject::loadNewConfigurationData() throws a MCDParameterizationException with error code ePAR_ITEM_NOT_FOUND in case that a file for the MCDDatatypeShortname value supplied as parameter to this method was not found in the database.

The methods MCDDbProject::loadNewConfigurationData() and MCDDbProject::loadNewConfigurationDatasByFilename() throw a MCDProgramViolationException with error code eRT_ELEMENT_ALREADY_EXIST if at least one CONFIG-DATA with the same ShortName as the one to be imported already exists.

By means of the method MCDDbProject::removeConfigurationDataByName (MCDDatatypeShortName configurationDataName) it is possible to remove a CONFIG-DATA from the currently active project configuration which has not been added to this project configuration permanently before.

9.24 Audiences and Additional Audiences

9.24.1 General

Audiences and additional audiences make it possible to supply information about access restrictions for MCD objects with respect to predefined target groups of users. For example, audience definitions can be used to declare that MCD-3D database objects should only be available to a certain class of clients (e.g. service testers). The roles that can be handled by the standard audience filter are AfterMarket, AfterSales, Development, Manufacturing and Supplier. The feature of additional audiences (see below) can be used to define additional roles to extend the standard audience definitions.

Filtering of D-server database objects in accordance with audience and additional audience settings shall always be done by the client application. The reason for this is that it is not possible for the D-server to handle all cases where audience definitions can be applied to ODX data in a meaningful way. For example, server-side filtering of database objects in accordance with their audience settings could lead to the following scenarios:

- Diagnostic services, which have been available at engineering time, might disappear at runtime (caused by the D-server filtering by audience settings). As a result, ECUs might be damaged, for example when the filtered diagnostic service is necessary to finish an ECU reprogramming session.
- With respect to related services, audience attributes at DataPrimitives are potentially harmful, because a DataPrimitive that is defined as a precondition to another DiagComPrimitive might disappear at runtime (when it gets filtered out as a result of its audience settings).
- Also, creating ODX data which contains audience information that is to be used for variant identification or base variant identification could result in undesirable runtime behaviour: In case the D-server executes one of these services, and in case these services in turn subsume DataPrimitives that have an associated audience attribute which disallows them in the current D-server instance, the (base) variant identification service will fail.
- Even more subtle complications could result by the D-server filtering out specific datablocks from a flash session, or coding fragments from an ECU configuration data set.

Out of these considerations, it has been decided that audience settings will be accessible at the D-server API, but that any kind of filtering or program logic depending on audience settings will have to be implemented by the client application.

Audience settings can be defined for the following MCD objects (ODX element is given in parentheses):

- MCDDbDataPrimitive (DIAG-COMM, MULTIPLE-ECU-JOB)
- MCDDbConfigurationRecord (CONFIG-RECORD)
- MCDDbItemValue (ITEM-VALUE)
- MCDDbOptionItem (OPTION-ITEM)
- MCDDbFlashSessionDesc (SESSION-DESC)
- MCDDbFlashDataBlock (FLASH-DATABLOCK)
- MCDDbDataRecord (DATA-RECORD)
- MCDDbFunctionNodeGroup (FUNCTION-NODE-GROUP)
- MCDDbFunctionNode (FUNCTION-NODE)
- MCDDbTableParameter (TABLE-ROW)

9.24.2 Audiences

Audiences reflect a predefined set of user groups for diagnostic data. The following user groups have been defined in ODX:

- Supplier
- Development
- Manufacturing
- AfterSales
- AfterMarket

In MCD-SERVER, the current access status with respect to these audiences is represented by an object of type MCDAudience which is returned by the method getAudienceState() available at the database objects listed above. Within the class MCDAudience, the access status with respect to each of the user groups is represented by a boolean attribute which can be queried by means of a corresponding method:

- isSupplier()
- isDevelopment()
- isManufacturing()
- isAfterSales()
- isAfterMarket()

Each of the attribute values can be set to "true" or "false" in ODX. If no information on the audience access status of an element is available in ODX, then this element's access status defaults to "true" for all five user groups. That means that the corresponding MCDAudience object is generated by the D-server, and delivers "true" for all of its status methods.

9.24.3 Additional Audiences

In addition to the predefined audiences, so-called additional audiences can be referenced from the MCD objects listed in the introduction of this subclause. These additional audiences allow to expand or to redefine the list of user groups that are subject to audience restrictions. The additional audiences in MCD-3 are represented by objects of type MCDDbAdditionalAudience. Every object of this type represents one ODX element ADDITIONAL-AUDIENCE which is contained in a DIAGLAYER. Additional audiences define individual lists of users or user groups which can be subject to accessibility constraints on the corresponding diagnostic elements. A diagnostic element (e.g. a MCDDataPrimitive) may contain references to either enabled or disabled ADDITIONAL-AUDIENCE elements. By means of the method getDbAdditionalAudiences(), the additional audiences that are associated with one of the following elements can be listed and evaluated by a client application:

- MCDDbLocation (introduced at DIAG-LAYER in ODX and valid for DIAG-COMMs and MULTIPLE-ECU-JOBs)
- MCDDbConfigurationData (introduced at ECU-CONFIG in ODX and valid for CONFIG-RECORD, DATA-RECORD, ITEM-VALUE and OPTION-ITEM)
- MCDDbEcuMem (introduced at FLASH in ODX and valid for SESSION-DESC and DATA-BLOCK)
- MCDDbFunctionDictionary (introduced at FUNCTION-DICTIONARY in ODX and valid for FUNCTION-NODE-GROUP and FUNCTION-NODE)

At a specific MCD object (e.g. a `MCDDataPrimitive`), either enabled additional audiences or disabled additional audiences can be defined. That means that the group of users with access rights is either extended or restricted by additional audience definitions. In ODX, a corresponding element provides a so-called ENABLED-AUDIENCE-REF or a DISABLED-AUDIENCE-REF:

- The ENABLED-AUDIENCE-REF – represented by the method `MCDDbDataPrimitive.getDbEnabledAdditionalAudiences()` – means that this element, e.g. a DIAG-COMM, is only suitable for the referenced ADDITIONAL-AUDIENCEs .
- The "DISABLED-AUDIENCE-REF" – represented by the method `MCDDbDataPrimitive.getDbDisabledAdditionalAudiences()` – means that this element is not appropriate for the referenced ADDITIONAL-AUDIENCEs, but available for all other listed ADDITIONAL-AUDIENCEs.

As with the predefined audiences, additional audiences shall evaluated by the client application. That is, the D-server will not perform any access control on any MCD object depending on additional audience settings.

9.25 Function Dictionary and Sub-Components

9.25.1 Terms and requirements

9.25.1.1 General

In a wide range a communication-oriented view on an ECU's diagnostic functionality is provided by 3D-server and ODX. However, this does not always meet today's way of designing vehicles, because many functions of a vehicle are distributed across several ECUs.

Function oriented diagnostics becomes more and more important since the functional point of view is much closer to the customer experienced symptoms a malfunction might cause.

This is covered with the structures described in this subclause based on the aspects of communication oriented diagnostics definitions and requirements. The following use cases were considered:

9.25.1.2 MCDDbFunctionDictionary

A MCDDbFunctionDictionary (available at MCDDbProjekt) is a definition of a hierarchical function catalog to organize service tester user interfaces:

- References to one or several ECUs and their diagnostic data content relevant for that function.
- Reference to services/jobs to make functions "executable".
- Definition of function input and output parameters with optional references to parameters of related services.

9.25.1.3 MCDDbSubComponent

A MCDDbSubComponent (available at MCDDbLocation) is a description of ECU sub functionality- or components (e.g. LIN-slaves).

9.25.2 Functions and function group in ODX

A function represents a vehicle subsystem or functionality (e.g. Indicator lights) considered from the point of view of diagnostics. A function may divide into one or more system components/sub functions that each may consist of several parts again and so on.

A function in MCDDbFunctionDictionary refers to the set of diagnostic information implemented in one or several ECUs that is related to this function including diagnostic services, DTCs, environment data and parameters. It is not intended (and therefore not possible) to (re-)define any information that is already available.

For example the function “Indicator Light Left” as a sub function of “Indicator Lights” is implemented across different ECUs and related to the according fault memories, input/output controls, and measurement values and so on. Furthermore the function “Blinking left” may be valid for several model lines and model years. From a functional point of view it is interesting to know which ECUs and diagnostic elements of an ECU are part of a function.

Finally, a function may have input and output parameters to influence the behavior of a function respectively indicating the correct function behavior.

Functions are defined in hierarchical structures (by recursion) to allow expressing different functional granularities:

- Indicator Lights
  - Indicator Light Left
  - Indicator Light Right
- Warning Indicator Lights
- …

A MCDDbFunctionNodeGroup is a container for already defined MCDDbFunctionNodes regardless to their hierarchical context. The MCDDbFunctionNodeGroup “Lights” could for example contain the MCDDbFunctionNodes “Indicator Lights” and “Head Lights”:

- Lights
  - Head Lights
  - …
- Indicator Lights
  - Indicator Light Left
  - Indicator Light Right
  - Warning Indicator Lights
  - …

A MCDDbFunctionNodeGroup may also contain other MCDDbFunctionNodeGroups. For a MCDDbFunctionNodeGroup applies the same requirements as for a MCDDbFunctionNode (relation to ECUs, services, DTCs, parameters…). In the example above it is use case dependent either to choose a MCDDbFunctionNode or a MCDDbFunctionNodeGroup.

9.25.3 Function dictionary data model description

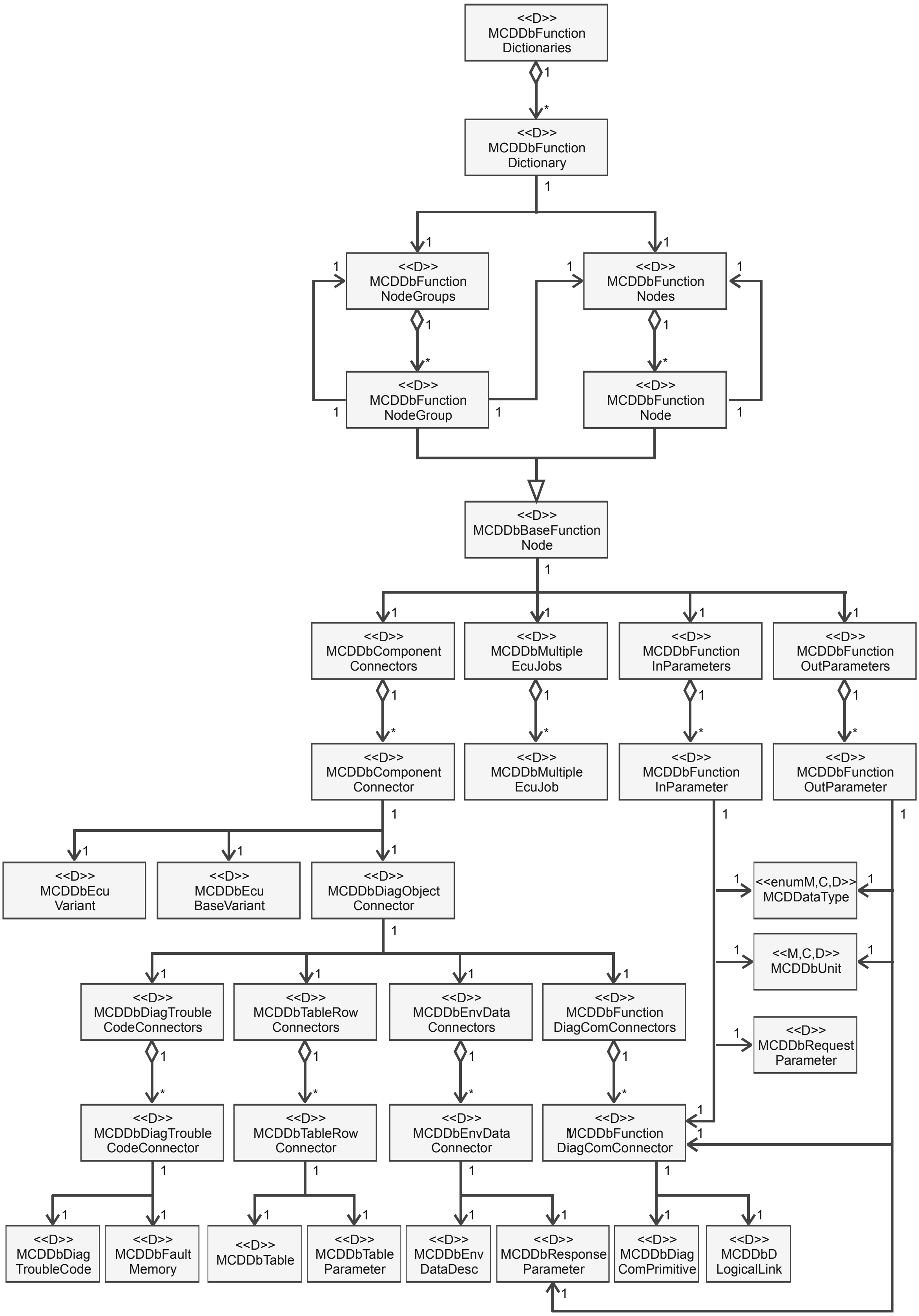

<<D>> MCDDbFunction Dictionaries
<<D>> MCDDbFunction Dictionary
<<D>> MCDDbFunction NodeGroups
<<D>> MCDDbFunction Nodes
<<D>> MCDDbFunction NodeGroup
<<D>> MCDDbFunction Node
<<D>> MCDDbBaseFunction Node
<<D>> MCDDbComponent Connectors
<<D>> MCDDbMultiple EcuJobs
<<D>> MCDDbFunction InParameters
<<D>> MCDDbFunction OutParameters
<<D>> MCDDbComponent Connector
<<D>> MCDDbMultiple EcuJob
<<D>> MCDDbFunction InParameter
<<D>> MCDDbFunction OutParameter
<<D>> MCDDbEcu Variant
<<D>> MCDDbEcu BaseVariant
<<D>> MCDDbDiagObject Connector
<<enumM,C,D>> MCDDataType
<<M,C,D>> MCDDbUnit
<<D>> MCDDbRequest Parameter
<<D>> MCDDbDiagTrouble CodeConnectors
<<D>> MCDDbTableRow Connectors
<<D>> MCDDbEnvData Connectors
<<D>> MCDDbFunction DiagComConnectors
<<D>> MCDDbDiagTrouble CodeConnector
<<D>> MCDDbTableRow Connector
<<D>> MCDDbEnvData Connector
<<D>> MCDDbFunction DiagComConnector
<<D>> MCDDbDiag TroubleCode
<<D>> MCDDbFault Memory
<<D>> MCDDbTable
<<D>> MCDDbTable Parameter
<<D>> MCDDbEnv DataDesc
<<D>> MCDDbResponse Parameter
<<D>> MCDDbDiag ComPrimitive
<<D>> MCDDbD LogicalLink
1
*

Figure 202 — MCDDbFunctionDictionaries data model

A MCDDbBaseFunctionNode aggregates all common features of a MCDDbFunctionNode/MCDDbFunctionNodeGroup (see previous subclause) and therefore is implemented by the according subclasses.

The hierarchy of MCDDbFunctionNodes / MCDDbFunctionNodeGroups can be considered as a function catalog representing the functional layout of the vehicle. It might be useful to generate the function catalog out of the same system where the vehicle's functional layout is described.

The element MCDDbFunctionNode represents a function as part of a function hierarchy. The requirement of grouping functions is implemented by a MCDDbFunctionNodeGroup. A MCDDbFunctionNode or MCDDbFunctionNodeGroup may only be relevant for or even restricted to certain departments or data customers. To reflect this, it is allowed to specify (ADDITIONAL-) AUDIENCEs, which may be used by a diagnostic application or as an data export/ conversion filter criteria by appropriate tools.

As mentioned above, a function often is distributed across ECUs and other components of the vehicle's network and therefore not only implemented in one single. The layout of the function's layout may vary from model line to model line. From that perspective it makes sense at first to describe which components (references to BASE- /ECU-VARIANTS) are contributing to a certain function and also to describe what is the right layout for a function.

A component's contribution to a function from diagnostic point of view may include DTCs (MCDDiagTroubleCodes and its corresponding MCDDbFlautMemories), ENVDATAs (MCDDbResponseParameters and its corresponding MCDDbEnvDataDescs), DIAG-COMMs (MCDDbDiagComPrimitives via MCDDbFunctionDiagComConnector) and TABLE-ROWS (MCDDbTableRows and its corresponding MCDDbTables). The MCDDbDiagObjectConnector aggregates these objects. The uniqueness is guaranteed by the MCDDbLocations that are referenced by the super ordinate MCDDbComponentConnectors.

For the MCDDbComponentConnetors controls the validity for following objects:

- BASE-FUNCTION-NODE (MCDDbFunctionNode, MCDDbFunctionNodeGroup);

- DIAG-OBJECT-CONNECTOR and its sub-objects DTC-DOPs, TABLEs and ENV-DATA-DESCs.

Case 1: Only a BASE-VARIANT is specified.

The BASE-FUNCTION-NODE/DIAG-OBJECT-CONNECTOR (DTC-DOPs, TABLEs, ENV-DATA-DESCs) applies to this BASE-VARIANT and all of its ECU-VARIANTs.

Case 2: Only one (or more) ECU-VARIANTs are specified.

All specified ECU-VARIANTs shall inherit from the same BASE-VARIANT. The BASE-FUNCTION-NODE/DIAG-OBJECT-CONNECTOR (DTC-DOPs, TABLEs, ENV-DATA-DESCs) applies to these ECU-VARIANTs, not to the common BASE-VARIANT.

Case 3: A BASE-VARIANT and one (or more) ECU-VARIANTs are specified.

All specified ECU-VARIANTs shall inherit from the same BASE-VARIANT. The BASE-FUNCTION-NODE/DIAG-OBJECT-CONNECTOR (DTC-DOPs, TABLEs, ENV-DATA-DESCs) applies to these ECU-VARIANTs and the common BASE-VARIANT.

Case 4: Neither a BASE-VARIANT nor an ECU-VARIANT is specified. This case is forbidden.

MCDDbFunctionInParameters and MCDDbFunctionOutParameters cover the following use cases:

- documentation of high level function input and output parameters without any technical relation, that means e.g. no relevance for a D-server;

- documentation of Vehicle Message Matrix (VMM) input and output signals related to a function;

- mapping of VMM-Signals to diagnostic content.

For the higher level description of a function, higher level input and output parameter descriptions can be useful, therefore the diagnostic description of the input and output parameters is optional. The higher level input and output parameter description could be useful, if a MCDDbMultipeEcuJob is used for diagnosing the function.

For example, the value of one function's output signal is described as a parameter of a measurement value, while the value of an function's input signal may be covered by a routine's service parameter. In the example the output signal and the routine's service parameter could both be accessed by diagnostic services, but for a higher level description only input "Current in V" and output "Temperature in C" is necessary.

Both MCDDbFunctionInParameters and MCDDbFunctionOutParameters have a MCDDbDataType (PHYSICAL-TYPE) and a MCDDbUnit to reflect this information without the necessity to resolve the (optionally) attached parameters. Since the optional MCDDbRequestParameters / MCDDbResponseParameters may only be referenced by SHORT-NAME, it is necessary to specify the service scope, for which the SHORT-NAME of the parameters shall be unique. This scope is given by the MCDDbDiagDomPrimitive referenced by the MCDDbFunctionInParameters and MCDDbFunctionOutParameters element via MCDDbFunctionDiagComConnector. An corresponding MCDDbDLogicalLink may be available in the database where the service can be generated and executed.

NOTE The reference to the MCDDbDiagComPrimitive is optional. In the case of having no MCDDbDiagComPrimitive referenced it is not allowed to have an MCDDbRequestParameters / MCDDbResponseParameters referenced by SHORT-NAME specified and vice versa.

9.25.4 Function dictionary usage scenario

Consider a vehicle with four doors, each having an ECU in it.

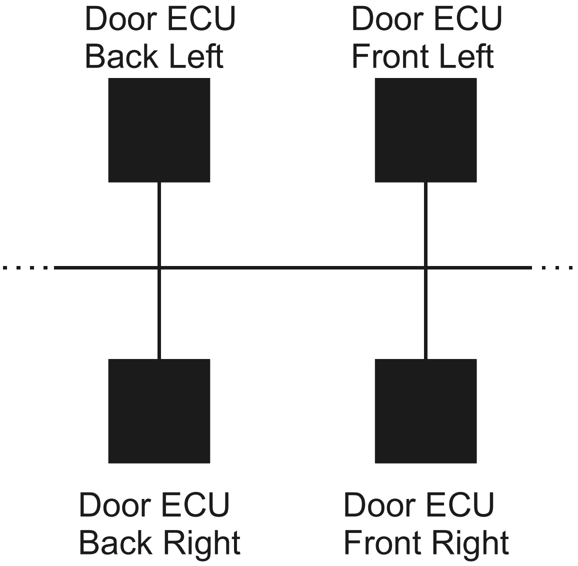


Figure 203 — Example Four Door ECUs in a vehicle network

The four ECUs each have their own functionality (e.g. window up and down, lock/ unlock door), but may also be addressed by a common functionality, e.g. central lock activating all locks or heat extraction opening all windows simultaneously.

From the diagnostics point of view, these functions may be reflected in 3 ways:

a) The activation of the window or lock actuator is performed by an IO-Control service.

b) Any problems with the actuators are expressed by error codes.

c) The current position of the window or the locker of a door is requested by a measurement service.

Table 46 — Individual and Common functionality of Door ECUs

	Door ECU Back Left	Door ECU Back Right	Door ECU Front Left	Door ECU Front Right
Individual Functionaltity	Window Back Left up / down	Window Back Right up / down	Window Front Left up / down	Window Front Right up / down
	Door Lock Back Right open / close	Door Lock Back Left open / close	Door Lock Front Right open / close	Door Lock Front Left open / close
Common Functionality	Central			
	Heat Extraction			

Additional Comments:

- In function hierarchies it is intentionally not predefined how MCDDbFunctionNodes and their subordinates relate regarding their diagnostic functionality. That means, the author (or the diagnostic application) decides whether the content related to a subordinate function automatically is relevant for its superior function and is automatically considered in its context.

- Function parameters are considered mainly to be used for documentation purposes (e.g. Vehicle Message Matrix / VMM signals). As mentioned above, if there are any I MCDDbRequestParameters / MCDDbResponseParameters referenced, their SHORT-NAMEs are to be resolved in the scope of the related services referenced by the MCDDbFunctionDiagComConnector aggregated to the MCDDbFunctionInParameters / MCDDbFunctionOutParameters.

- There is no relation defined between DTCs and ENV-DATAS referenced by a MCDDbDiagObjectConnector.

Again, the author or the diagnostic application decides about whether or not making this relation and how to define such a relation.

NOTE Multiple MCDDbFunctionDictionaries are allowed. But the SHORT-NAMEs of all MCDDbFunctionNodeGroups/MCDDbFunctionNodes shall be globally unique.

9.25.5 Sub-Component data model description

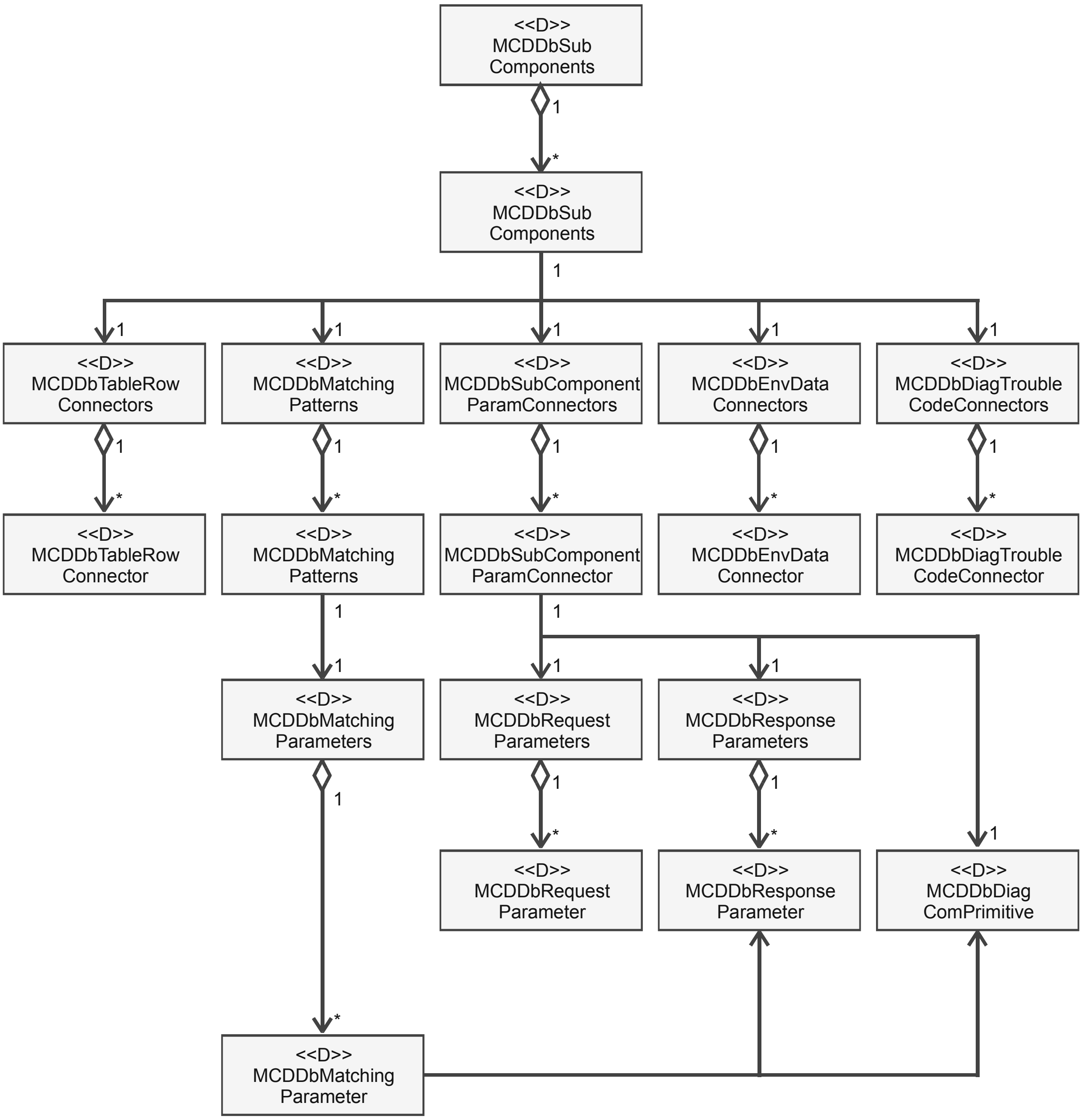

<<D>> MCDDbSub Components
1
*
<<D>> MCDDbSub Components
1
1
<<D>> MCDDbTableRow Connectors
<<D>> MCDDbMatching Patterns
<<D>> MCDDbSubComponent ParamConnectors
<<D>> MCDDbEnvData Connectors
<<D>> MCDDbDiagTrouble CodeConnectors
<<D>> MCDDbTableRow Connector
<<D>> MCDDbMatching Patterns
<<D>> MCDDbSubComponent ParamConnector
<<D>> MCDDbEnvData Connector
<<D>> MCDDbDiagTrouble CodeConnector
<<D>> MCDDbMatching Parameters
<<D>> MCDDbRequest Parameters
<<D>> MCDDbResponse Parameters
<<D>> MCDDbRequest Parameter
<<D>> MCDDbResponse Parameter
<<D>> MCDDbDiag ComPrimitive
<<D>> MCDDbMatching Parameter

Figure 204 — MCDDbSubComponents data model

A MCDDbSubComponent, defined at the MCDDbLocation, is considered to be a functional unit in- or outside of an ECU that covers certain additional diagnostics relevant functionality either physically (e.g. a LIN slave) or logically. To point out the use case the SEMANTIC attribute may be used. Two SEMANTICS are predefined:

- SLAVE, if the MCDDbSubComponent describes a physical function unit;

- FUNCTION, if the MCDDbSubComponent describes a logical function unit.

The latter is interesting in context with MCDDbFunctionDictionary, when a counterpart to a MCDDbFunctionNode/MCDDbFunctionNodeGroup shall be defined (also see additional comments).

Unlike a MCDDbFunctionNode or a MCDDbFunctionNodeGroup, a MCDDbSubComponent is always related to one explicit ECU (or even ECU-VARIANT) and can be considered as an additional layer below MCDDbLocation.

The difference is that no new data (DTC, ENV-DATA, and TABLE-ROW) is defined but only reused (referenced) from other layers. Therefore, the MCDDbDiagTroubleCodeConnector, MCDDbTableRowConnector and MCDDbEnvDataConnector elements aggregated by a MCDDbSubComponent shall always only point to elements that are part of the MCDDbLocation that the MCDDbSubComponent belongs to.

A MCDDbSubComponentParamConnector allows referring to a MCDDbDiagComPrimitive and optionally to one or more MCDDbRequestParameters and MCDDbResponseParameters. The related MCDDbDiagComPrimitive is the SHORT-NAME boundary for these parameters.

NOTE 1 The MCDDbDiagComPrimitive may be referenced without any MCDDbRequestParameter/ MCDDbResponseParameter. In this case, the whole MCDDbDiagComPrimitive is relevant for the referencing MCDDbSubComponent.

MCDDbMatchingPatterns (MCDDbSubComponent::getDbSubComponentPatterns()) may be used to specify how the presence of a MCDDbSubComponent can be determined at runtime. In opposite to an ECU-VARIANTs VARIANT-PATTERN (MCDDbEcuVariant::getDbVariantPatterns()) this is not intended to be performed automatically but only for documentation purposes. However, after a MCDDbSubComponent has been selected or "identified" by a diagnostic application, the diagnostic application may provide according functionality. For example it can filter out any content that is not relevant for this MCDDbSubComponent.

NOTE 2 There is no inheritance given for SUB-COMPONENTs. SUB-COMPONENTs should be defined for each BASE-/ECU-VARIANT if necessary.

9.25.6 Sub-Component usage scenario

Consider a multi purpose ECU with two Seat LIN slave controllers attached.

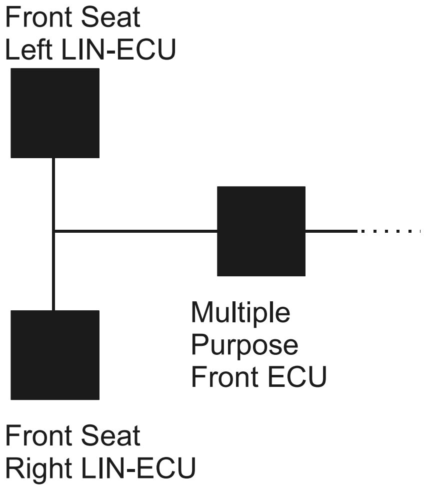


Figure 205 — Multi Purpose ECU with 2 LIN slaves

Since only the multi purpose ECU (master) is attached to the CAN network, all diagnostic messages, that have an influence on the behavior or request the status of the seat ECU, will be sent to the master. The master then decides how to handle those diagnostic messages. On the other hand, if an error occurs inside one of the LIN slaves, this is probably indicated by a DTC activated in the master's fault memory since the slaves may not have their own fault memory.

This means, the diagnostic data description of the master ECU also shall consider the diagnostic functionality of the Slaves. This relationship can be expressed by defining a MCDDbSubComponent for each LINslave that:

- reuses the content of the master ECU and therefore avoid redundancy;
- resides inside the diagnostic description of the master ECU, to keep the MCDDbSubComponent that helps to keep it self contained;
- may be checked for validity or presence by using its MCDDbSubComponentPatterns to filter out any information that is not relevant in the current MCDDbSubComponent ‘s context.

Additional Comments:

- The MCDDbSubComponent addresses primarily documentation use cases and offers another approach to the diagnostic content of the master. In other scenarios it might be more useful to have a separate DIAGLAYER also for the LIN slaves.
- There may be the use case to have a relationship between a function defined in MCDDbFunctionDictonary and a MCDDbSubComponent that e.g. covers this functionality in the ECU. This is not explicitly modeled but still can be covered by a diagnostic application mapping to SEMANTICS or naming conventions.

9.26 ECU States

ECUs in vehicles are stateful electronic components. They usually implement state machines for different purposes. The state changes in these state machines are triggered, e.g. by sensor data, by diagnostic requests or by internal functionality. Those parts of the state machines which are relevant in vehicle diagnostics can be modelled by means of state charts in ODX. For example, the possible session changes and the security states can be modelled as state charts in ODX. As a result of this focus, the session and security sub model in ODX mainly covers two aspects:

- The first is to describe possible state transitions resulting from the execution of a DiagComPrimitive.
- The second is to describe preconditions for the execution of a DiagComPrimitive.

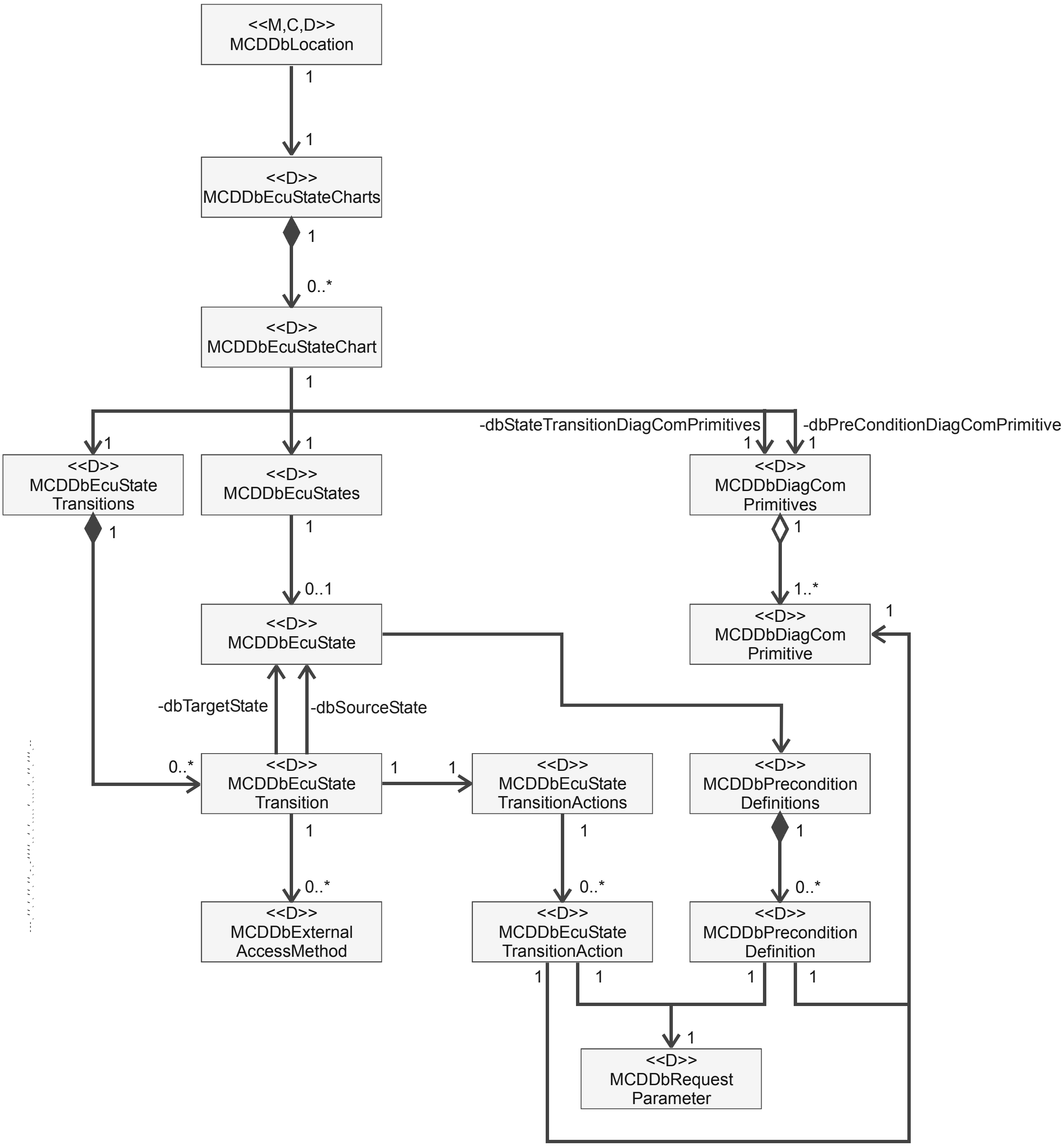


Figure 206 — UML Class diagram of the ECU state chart sub model

Both the session handling and the security handling are modelled within one generic state machine model. The element `MCDDbEcuStartChart` stores the possible states of an ECU (see Figure 206 — UML Class diagram of the ECU state chart sub model). The collection of all state charts valid at a certain DbLocation can be fetched using the method `MCDDbLocation::getDbEcuStateCharts()`. Every state within a state chart is represented by an own `MCDDbEcuState` object. The attribute 'semantic' associated with a `MCDDbEcuStateChart` defines the type of the state chart. The value of this attribute can be obtained by means of the method `MCDDbEcuStateChart::getSemantic()`. The values "SESSION" and "SECURITY" are predefined for this semantic in ODX. However, the set of values of the attribute 'semantic' is extendible by the user.

Each state chart references all ECU states belonging to this state chart. One of these ECU states is the start state. This start state of a state chart can be obtained by means of the method `MCDDbEcuStateChart::getDbStartState()`. There is exactly one start state per state chart.

A state transition within one state machine is modelled by an object of type `MCDDbStateTransition`. A state transition references exactly one source state and exactly one target state. Self transitions can be described by state transitions with identical SOURCE and TARGET states. The state transitions belonging to an ECU state chart are also referenced from the corresponding `MCDDbEcuStateChart` object.

NOTE 1 ECU state charts need to be disjoint with respect to their collections of ECU states and state transitions. That is, ECU states and state transitions cannot be shared between two different ECU state charts.

State transitions are fired as a result, e.g. to

- ECU-internal logic,
- processing a request of a DiagComPrimitive,
- reception of specific sensor signals.

In the first case, the corresponding `MCDDbEcuStateTransition` object does neither provide a so-called external access method nor does it reference any state transition event in terms of an object of type `MCDDbStateTransitionAction`. In the second case, the state transition refers to at least one `MCDDbStateTransitionAction`. In the third case, the `MCDDbEcuStateTransition` provides information on an external access method.

External access methods are represented by objects of type `MCDDbExternalAccessMethod`. An external access method allows specifying an OEM-specific method necessary to perform the corresponding state transition. The textual information returned by the method `MCDDbExternalAccessMethod::getMethod()` needs to be interpreted by an OEM-specific extension of a D-server. The method information can be used, e.g., in case of security-critical applications to link to protected functionality. Another example is to simulate a certain sensor signal in case of a test bench.

If the collection of `MCDDbEcuStateTransitionAction` objects which can be obtained using the method `getDbEcuStateTransitionActions()` at a `MCDDbEcuStateTransition` is non-empty, this has the following semantic: If the DiagComPrimitive referenced from one of the `MCDDbEcuStateTransitionActions` is successfully executed, the state of the ECU changes from the specified source state to the specified target state. In case that the `MCDDbEcuStateTransitionAction` refers to a request parameter and a value, the DiagComPrimitive needs to have this value set at the request parameter to fire the state transition after successful execution.

Sometimes, the execution of a DiagComPrimitive is only possible or successful if the target ECU is in a certain state, e.g. a certain session or a certain security state. In ODX, such preconditions can be modelled by means of PRE-CONDITION-STATE-REFs at a DIAG-COMM. PRE-CONDITION-STATE-REF can be used to define the allowed states for the execution of the DIAG-COMM in cases where the execution of the DIAG-COMM does not result in a state-transition of the ECU but its execution is bound to the condition that the ECU already is in a certain state. In the DB part of the D-server API, a PRE-CONDITION-STATE-REF is represented by an object of type `MCDDbPreconditionDefinition`. Similarly to `MCDDbStateTransitionAction`, a `MCDDbPreconditionDefinition` can refer to a request parameter and a value. If both are set, the precondition definition is bound to this configuration of the referenced DiagComPrimitive. The DiagComPrimitives which are restricted by a certain ECU state can be obtained by means the method `MCDDbEcuState::getDbRestrictedDiagComPrimitives()`. This method returns a collection of MCDDbPreconditionDefinition objects each referencing a DiagComPrimitive restricted by the current ECU state.

If a STATE-TRANSITION-REF is used with a DIAG-COMM in ODX, there optionally may be a PRE-CONDITION-STATE-REF for the source states of the referenced transitions. If STATE-TRANSITION-REFs and/or PRE-CONDITION-STATE-REFs are used, the DIAG-COMM is executable in the SOURCE states of the referenced STATE-TRANSITIONs and in the referenced preconditions' STATEs but no other states. This means that the collection of MCDDbEcuState objects returned by the methods `MCDDbDiagComPrimitive::getDbPreConditionStatesByDbObject(…)` and `MCDDbDiagComPrimitive::getDbPreConditionStatesBySemantic(…)` are the union of the source states of the state transitions referencing this DiagComPrimitive with those ECU states which reference a MCDDbPreconditionDefinition to this DiagComPrimitive - both with respect to the same ECU state chart.

If a DiagComPrimitive is neither referenced from a `MCDDbEcuStateTransition` nor from any `MCDDbPreconditionDefinition`, this DiagComPrimitive is executable in all states and the execution does not result in a state transition. In this case, the collection of precondition states at this DiagComPrimitive is empty for all ECU state charts at the same DbLocation. Otherwise, this DiagComPrimitive is only executable successfully in one of the ECU states within the collection of precondition states at this DiagComPrimitive.

NOTE 2 The entire ECU states and state charts model is only available at the DB part of the D-server API. The D-server will not provide any support for active tracking of the current ECU states. Moreover, the D-server will not prevent the client application to execute restricted DiagComPrimitives in ECU states not supported for these DiagComPrimitives. Hence, the client application is responsible for processing and interpreting the information in ECU state charts.

9.27 Monitoring vehicle bus traffic

A common use case for a diagnostic server is the need to monitor traffic on a vehicle communication bus. To provide simple bus monitoring capability, this chapter defines how bus monitoring should be implemented in the D-server. The concepts presented in this chapter are based on standardized features of the D-PDU API layer.

The monitoring link that is introduced in this chapter is an entirely passive entity regarding the monitored physical resource. That means that a monitoring link offers no way to alter any protocol parameters that are associated with the monitored bus resource. The reason for this is that the monitoring functionality is required to be available at any time in a diagnostic session – even before a database project has been selected by a client application. As all information pertaining to protocol parameters is part of the diagnostic database, this information is not available before a database is selected. Therefore, a monitoring link will always use the already existing settings of the physical resource that is being monitored – if a logical link to that resource already exists in the D-server, a monitoring link to that resource will be using the link settings that have previously been configured by the logical link. In case no logical link to a resource exists when a monitoring link to that resource is created, the monitoring link will be using that resource's default settings as implemented in the communications/protocol layer. As the D-PDU API allows any link to a physical resource to modify protocol parameters at any time, it is always possible to create a logical link to a resource that already has an associated monitoring link, and modify that resource's configuration by setting protocol parameters at the logical link. A monitoring link could be implemented by using corresponding D-PDU API protocols such as ISO_11898_RAW, which forwards all bus communication without applying any application layer logic to the monitored messages.

In the D-server API, the class `MCDMonitoringLink` is provided for performing bus monitoring. Instances of this class can be created based on `MCDInterfaceResources` to monitor a specific protocol on one of the communication channels available to the D-server. `MCDMonitoringLink` instances are created using the method `MCDMonitoringLinks::addByInterfaceResource(…)`, while the `MCDMonitoringLinks` collection can be obtained from the `MCDSystem`.

NOTE The method `MCDSystem::prepareVCIAccessLayer()` has to be called before doing bus monitoring, so that the D-PDU API layer can be set up by the D-server. In case the VCI access layer has not been prepared beforehand, a call to `MCDMonitoringLinks::addByInterfaceResource(…)` will result in a `MCDProgramViolationException` with error code `eRT_VCI_ACCESS_LAYER_NOT_PREPARED` being thrown.

The ShortName of a `MCDMonitoringLink` will be generated by the D-server and it will include the ShortName of the `MCDInterface` as well as the ShortName of the `MCDInterfaceResource` that is being monitored:

```
#RtGen_<ShortnameOfMCDInterface>_<ShortnameOfMCDInterfaceResource>
```

Through the MCDMonitoringLink object, a MCDCollector object can be retrieved, which uses the standard collector functionalities for monitoring bus traffic. The MCDCollector object can always be obtained by any client, except when the associated MCDMonitoringLink has a cooperation level of eNO_COOPERATION. After the first call to the method MCDMonitoringLink::getMonitoringCollector(), all subsequent calls always return a proxy to the object created by the first call. The MCDCollector object is invalidated when the MCDMonitoringLink that it was obtained from is removed from the server. Any calls to methods of a MCDCollector object will in this case result in a MCDProgramViolationException with error code eRT_INVALID_OBJECT being thrown.

Monitoring can be switched on or off on a per-link basis by calling the MCDCollector::start() and MCDCollector::stop() methods. Monitored data will be contained in the MCDDatatypeAsciiStrings collection returned by the MCDCollector::fetchMonitoringFrames() method. Each monitored message will be contained in one MCDDatatypeAsciiString within that collection.

To allow different client applications to monitor the same resource at the same time, but with individual per-client monitoring message handling, the MCDCollector provides the 'buffer ID' mechanism. By using the MCDCollector::createBufferIDForPolling() method, a client application receives a buffer ID that shall subsequently be used when retrieving monitoring results via the method MCDCollector::fetchMonitoringFrames. The buffer ID is also used within the MCDCollectorEventHandler, to indicate the buffer that an onCollectorMonitoringFramesReady event relates to. When a client application has finished monitoring, it can release its buffer ID by using the method MCDCollector::removeBufferIDForPolling().

As mentioned above, client applications can either directly poll a MCDCollector object for available monitoring results, or can use an event handler to be notified by the D-server when monitoring frames are available. To lessen the event load on the client application, the 'number of samples before firing event' field within a collector can be used to define how many samples should be collected before an event is raised by the D-server. This number of samples can be set by calling the method MCDCollector::setNoOfSampleToFireEvent().

In addition, the client application can use the MCDMessageFilter class, which is available through the MCDMessageFilters collection from a MCDMonitoringLink object, to specify filters for the incoming bus messages. These filter definitions are analogous to the IOCTL filter data structure specifications from the PDU-API standard. For detailed semantic descriptions please refer to the relevant chapters of the PDU-API specification[11].

Despite message filtering and other performance-enhancing features of the collector, the monitoring solution proposed here is not supposed to be used (or expected to scale) for monitoring of high-speed, high-volume bus traffic. Due to internal restrictions, e.g. in the coupling between D-server and D-PDU API implementation[4)], bus monitoring using a D-server is a very resource-intensive task, and as such probably will not perform adequately for high-load situations until a more suitable solution can be implemented based on upcoming versions of the relevant standards.

The format of the monitored messages as returned by the method MCDCollector::fetchMonitoringFrames() cannot be defined in sufficient detail and comprehensiveness to cover all bus architectures and VCI driver implementations that might have to be covered by this specification. However, an exemplary data format definition for CAN bus monitoring is provided in Annex L.

4) Currently, the D-PDU API will send an event to the D-server for every single received frame, which potentially results in the D-server being completely overloaded by D-PDU API events in high-traffic situations. It is of course up to the implementation of the D-server and the D-PDU API to circumvent these problems; however, such optimizations cannot be considered in this part of ISO 22900.

9.28 Support of VCI module selection and other VCI module features in accordance with D-PDU API Standard

9.28.1 General

For being able to physically connect to a vehicle or a set of ECUs, respectively, a D-server uses a vehicle communication interface (VCI). This VCI can either be a proprietary interface or it can be compliant with ISO 22900-2. The integration of a D-PDU API compliant MVCI access is described in this chapter. If different definitions are required for integrating a proprietary VCI, these are stated in the corresponding places in addition.

9.28.2 Definitions

The MVCI D-PDU API supports the selection of a specific VCI module to be used for diagnostic communication by a client application, e.g. a D-server. VCI module is the D-PDU API term for a vehicle communication interface. The difference to the usual usage of the term VCI is that from the D-PDU API's point of view several of these VCI modules can be managed below the D-DPU API at the same time. However, in general only one of these VCI modules is actively used by the D-server. Furthermore, the D-PDU API provides access to the properties of the VCI modules.

A physical vehicle link is the physical connection between a vehicle's diagnostic connector and the ECU. The physical vehicle links available in a vehicle and the properties of these physical vehicle links are defined in the ODX data used to describe a vehicle's electrical configuration.

A physical interface link is the physical connection between the VCI connector of a VCI and the interface connector. Technically speaking, the VCI connector and the interface connector are the connectors at the two opposite ends of the cable connecting a VCI with a vehicle. In the VCI, a VCI connector is driven by a VCI module (also called interface in the following subclauses). The VCI modules contained in a VCI and their properties are accessed by the D-server via the D-PDU API.

The interface connector (called DLC connector in the D-PDU API) at the vehicle's end of the *interface cable* is plugged into the vehicle connector. Therefore, both connectors shall match mechanically and have identical pin layouts (1:1 match). At the other end, the interface cable is plugged into the VCI module's VCI connector. The cable description file (CDF) shipped with the D-PDU API lists the supported interface cables.

The set of types of possibly available physical interface links and their properties are defined inside a so-called module description file (MDF) and the so-called cable description file (CDF). Both files are shipped together with the D-PDU API to be used by the D-server.

A physical link is the combination of a physical vehicle link connected to a physical interface link (see 9.1.6).

The following subclauses describe the behaviour of a D-server supporting VCI modules in accordance with D-PDU API Standard, and chapter 9.28.15 describes the behaviour of a D-server not supporting VCI modules in accordance with D-PDU API Standard.

9.28.3 General behaviour of D-PDU API related D-server methods

Many of the D-server methods described in the following subclauses include a call to a D-PDU API function. In these cases, the name of the called D-PDU API function is mentioned.

In general, each call of a D-PDU API function may fail returning a D-PDU API error code, which is described in the D-PDU API standard. In this case the related MCD-3D method throws a `MCDProgramViolationException` with error code `eRT_PDU_API_CALL_FAILED`, and the vendor code of the error shall deliver the specific D-PDU API error code.

9.28.4 Overview of VCI module related classes

The class diagram in Figure 207 — Class diagram of VCI module related classes and their properties shows an overview of the classes required to represent VCI modules in the D-server API and to access the corresponding D-PDU API resources.

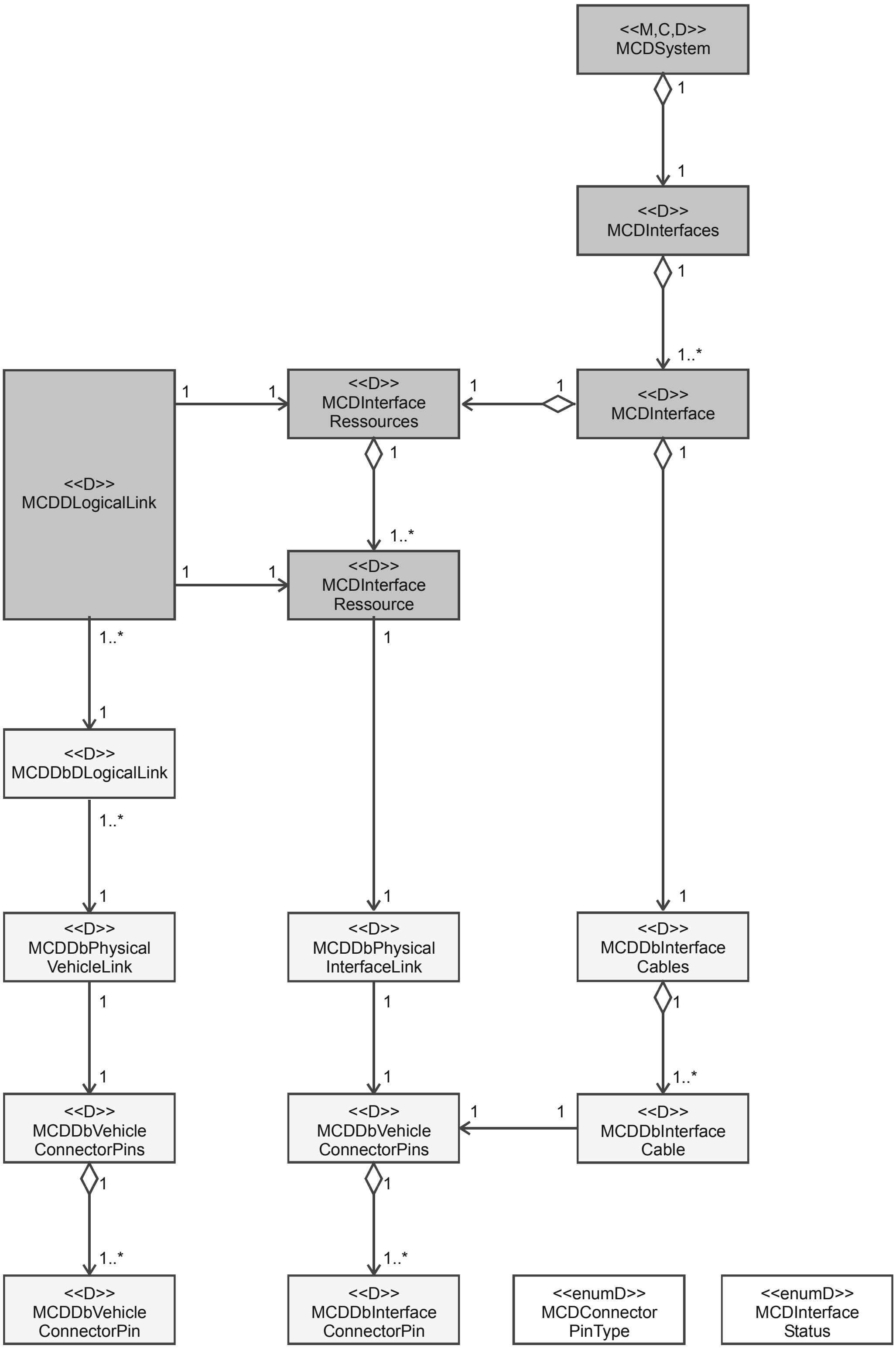


Figure 207 — Class diagram of VCI module related classes and their properties

9.28.5 VCI module selection

For some diagnostic applications based on a D-server, it may be necessary to dynamically select or re-select the VCI module to be used for vehicle diagnostics at runtime. For example, a service tester application in a workshop with several VCI modules (indirectly) hooked up to different vehicles needs to be able to connect to each of these VCI modules on demand.

The available VCI modules are detected by the MVCI D-PDU API internally. The application running on top of the D-server does not provide any connection parameters itself. The D-server provides the list of currently available VCI modules to the diagnostic application. In this list, every VCI module is identified by its unique ShortName which is provided by the D-server. Now, the diagnostic application can choose the VCI module to be used for diagnostic communication from the list, connect to this module or disconnect from this module to be able to connect to a different module in the next step.

It is also possible to connect to more than one VCI module at the same time, e.g. when there are two VCI modules connected to the same vehicle via a Y-cable with a shared interface connector.

9.28.6 MCDInterface

The class `MCDInterface` represents a single VCI module at the D-server API and provides access to the features of this VCI module. The class `MCDInterface` is derived from `MCDNamedObject`.

To retrieve the named collection of `MCDInterfaces` currently available for a D-server, the method `MCDSystem::getCurrentInterfaces():MCDInterfaces` has been introduced. This method never throws an exception, at least an empty collection is returned. Every `MCDInterface` object can be identified by its unique ShortName. This ShortName is generated by the D-server with the following pattern: #RtGen_MCDInterface_<Number>. The number in this generated ShortName is increased for every available or newly available VCI module. The generated ShortName does not contain the denoted brackets. For the first instance, the number is zero. This pattern is also used if only a single VCI module, represented by the MCDInterface object, is part of the MCDInterfaces collection.

The LongName of the MCDInterface object is composed of the structure element that is delivered by the D-PDU API via method PDUGetModuleIds. The PDU_MODULE_ITEM list contains the PDU_MODULE_DATA structures. Each PDU_MODULE_DATA structure contains an optional vendor specific module identification name <pVendorModuleName> and an optional vendor specific additional information <pVendorAdditionalInfo>. These two optional strings are composed to the LongName of the corresponding MCDInterface by the D-server separated by a space character: <pVendorModuleName> <pVendorAdditionalInfo>. The LongName does not contain the denoted brackets.

Before it is possible to obtain the list of available `MCDInterfaces` by calling the method `MCDSystem::getCurrentInterfaces():MCDInterfaces` (calls the D-PDU API function `PDUGetModulIds`), the method `MCDSystem::prepareVciAccessLayer()` shall be called (note: replaces former `MCDSystem::prepareInterface`). A call to `prepareVciAccessLayer()` triggers the initialization of the VCI access layer in the D-PDU API (calls the D-PDU API function `PDUConstruct`). This includes the identification of all available VCI modules. Once the VCI access layer has been initialized, the D-PDU API continuously tracks which VCI modules are currently available. This includes detection of new VCI modules in the visibility scope and detection of VCI modules which left the visibility scope. The method to release the D-PDU APIs VCI access layer is `MCDSystem::unprepareVciAccessLayer()` (note: replaces former `unprepareInterface,` calls the D-PDU API function `PDUDestruct`).

For the multi-client scenario, each client can independently call the methods `prepareVciAccessLayer()` and `unprepareVciAccessLayer(),` but a call to `prepareVciAccessLayer()` will fail, if the VCI access layer is already prepared (`MCDProgramViolationException, eRT_VCI_ACCESS_LAYER_ALREADY_PREPARED`). A call to `unprepareVciAccessLayer()` will fail, if the VCI access layer is not prepared (`MCDProgramViolationException, eRT_VCI_ACCESS_LAYER_NOT_PREPARED`).

For the multi-client scenario, the method `MCDInterface::lock()` allows a client to exclusively access the methods that change the behaviour of the `MCDInterface` object, i.e. the methods `connect()`, `disconnect()`, `reset()` and `setProgrammingVoltage()`. A client can release its lock by the method `MCDInterface::unlock()`. Any client can request the current lock state by `MCDInterface::getLockState()`.

The diagnostic application on top of the D-server shall decide which of the available interfaces to use for diagnostic communication to a specific vehicle. Otherwise, no diagnostic communication to an ECU or a vehicle, respectively, can be established. This decision is usually based upon the unique ShortName (delivered by the D-PDU API function `PDUGetModuleIds`) of an interface known to be physically connected to the vehicle. To select an interface for diagnostic communication, the application calls `MCDInterface::connect():void` at the corresponding interface object, which calls the D-PDU API function `PDUModuleConnect`.

NOTE Calling `connect()` for at least one interface is mandatory to run any diagnostic communication in the D-server, if the D-server supports the D-PDU API.

If the diagnostic application has finished its diagnostic communication via a selected interface, it needs to call the method `MCDInterface::disconnect():void` to disconnect from this interface, which calls the D-PDU API function `PDUModuleDisconnect`.

To find out which interfaces are currently connected, the diagnostic application can fetch the list of connected interfaces by means of the method `MCDSystem::getConnectedInterfaces(): MCDInterfaces (`calls the D-PDU API function `PDUGetStatus` for all VCI modules to find out which modules are connected). This method never throws an exception, at least an empty collection is returned.

Before connecting to an interface, the diagnostic application should check if this interface is available for connection by checking its status via `MCDInterface::getStatus(): MCDInterfaceStatus (`calls the D-PDU API function `PDUGetStatus` for the related VCI module). Connection is only possible if the status `MCDInterfaceStatus::eAVAILABLE` is returned.

As a Logical Link requires a physical link to the vehicle to be available, at least one VCI module which is connected and in state `MCDInterfaceStatus::eREADY` needs to be present, if the D-server supports the D-PDU API. Of course, this VCI module needs to provide the physical resources and protocol resources defined for the logical link. The combination of physical resource and protocol resource is called a `MCDInterfaceResource`. In principle a single interface could provide more than one `MCDInterfaceResource` matching the requirements of a Logical Link. In this case, the interface resource which will be used for a specific Logical Link is either selected automatically by the D-server or it is actively selected by the diagnostic application (see chapter 9.28.10). The same applies if multiple interfaces are connected to the same vehicle at the same time and if some of these interfaces provide matching interface resources for the Logical Link.

9.28.7 VCI module selection sequence

Table 47 — Method for VCI module selection - D-server API and D-PDU API shows the sequence of methods to call at the D-server API for VCI module selection together with the corresponding calls of D-PDU API methods from within the D-server logic.

Table 47 — Method for VCI module selection - D-server API and D-PDU API

Methods called at D-server API	**Methods called at D-PDU API**
MCDSystem::prepareVciAccessLayer()	PDUConstruct() will be executed for all available VCI Access Libraries (e.g. D-PDU API)
interfaces = MCDSystem::getCurrentInterfaces()	PDUGetModuleIds(): delivers list of handles hMod1, hMod2, … of available or connected VCI modules

Table 47 (*continued*)

Methods called at D-server API	**Methods called at D-PDU API**
Decide to which interface(s) to connect… interface1.connect() [interface2.connect() …]	 PDUModuleConnect(hMod1) [PDUModuleConnect(hMod2) …]
create and open Logical Links, execute DiagComPrimitives, …. close and remove Logical Links	…all method calls carry the selected VCI module handle hModx as input parameter…

interface1.disconnect()
[interface2.disconnect() …]
=> Only allowed in LogicalLink state eCREATED!
	PDUModuleDisconnect(hMod1)
	PDUModuleDisconnect(hMod2)

MCDSystem::unprepareVciAccessLayer()	PDUDestruct() will be executed for all available VCI Access Libraries (e.g. D-PDU API)

9.28.8 Interface status events

Status changes of a VCI module are notified by the D-server to a diagnostic application by means of different events.

The event `MCDSystemEventHandler::onInterfaceStatusChanged(MCDInterface interface, MCDInterfaceStatus status)` indicates that the status of the VCI module given as first parameter has changed to the state given as the second parameter. This event is triggered by a D-PDU API module callback with a module status event item.

The event `MCDSystemEventHandler::onInterfacesChanged(void)` indicates that the list of VCI modules available for a D-server has changed – either because a new VCI module has become available or because a VCI module is not available any more. This event is sent when the D-server or the D-PDU API, respectively, has automatically detected a new VCI module or when communication to a VCI module has been lost (see paragraph below for details). This event is triggered by a D-PDU API system callback with the information event PDU_INFO_MODULE_LIST_CHG. The diagnostic application can use the method `MCDSystem::getCurrentInterfaces()` to retrieve an updated list of currently available VCI modules in this case.

The event `MCDSystemEventHandler::onInterfaceError (MCDInterface interface, MCDError error)` indicates that the D-server or the D-PDU API, respectively, has automatically detected that communication to the VCI module given as first parameter has been lost or a hardware fault occurred indicated by the `MCDErrorCode` `eCOM_PDU_ERR_EVT_LOST_COMM_TO_VCI` or `eCOM_PDU_ERR_EVT_VCI_HARDWARE_FAULT`. These events are only possible, if the affected `MCDInterface` object has been successfully connected via the method `MCDInterface::connect()`, and has not yet been disconnected via the method `MCDInterface::disconnect()`. These events are triggered by a D-PDU API module callback with a module error event `PDU_ERR_EVT_LOST_COMM_TO_VCI` or `PDU_ERR_EVT_VCI_HARDWARE_FAULT`. In case of the event `onInterfaceError()` the affected `MCDInterface` object including all its `MCDDLogicalLink` objects are invalid. In this case the `MCDDLocalLinks` can only be closed or resetted, respectively, and removed from the `MCDLogicalLinks` collection. The `MCDDLogicalLinks` cannot be reused for further communication, i.e. the call `gotoOnline()` results in an `MCDProgramViolationException` with the error code `eRT_PDU_API_CALL_FAILED`. The MCDInterface object is also invalid and can only be removed from the `MCDInterfaces` collection. All calls to one of the MCDInterface specific object methods lead to a `MCDProgramViolationException` with the error code eRT_PDU_API_CALL_FAILED, except a final call to the method `MCDInterface::disconnet()`. This final call shall be performed by the application, after the application has received the event `MCDSystemEventHandler::onInterfaceError.`

9.28.9 MCDInterfaceResource

The class MCDInterfaceResource can be used to obtain information about the resources available at an interface (VCI module). In addition, this class allows selecting a specific interface resource to be used with a Logical Link. An interface resource is equivalent to a “Resource” object as defined in the D-PDU API. Hence, an interface resource is characterized by the attributes

— communication protocol,

— physical interface link type (called “Bustype” in D-PDU API), and

— pins on the interface connector (called “DLC” connector in D-PDU API).

To retrieve the named collection of MCDInterfaceResources which are provided by a VCI module, the method MCDInterface::getInterfaceResources():MCDInterfaceResources is used. Every MCDInterfaceResource is a named object which can be identified by its ShortName. All information needed for this method including the ShortName of a MCDInterfaceResource is provided by the D-PDU API’s MDF file.

The method MCDInterfaceResource::getInterface():MCDInterface allows to navigate back to the interface the current resource belongs to.

9.28.10 Selection of an interface resource

To open a Logical Link, it is necessary to select an interface resource which matches the requirements of this Logical Link, i.e. it shall match the communication protocol type required for the Logical Link, match the physical vehicle link type, and the pins on the vehicle connector. For a Logical Link, the requirements can be obtained from the ODX data.

This means in detail:

— The resource’s protocol type, obtainable via the method
MCDInterfaceResource::getProtocolType():A_ASCIISTRING,
shall match the Logical Link’s protocol type, obtainable via the method
MCDDbDLogicalLink::getDbProtocolStack():MCDDbProtocolStack::getDbProtocolType():A_ASCIISTRING.

— The resource's physical interface link type, obtainable via the method MCDInterfaceResource::getDbPhysicalInterfaceLink():MCDDbPhysicalInterfaceLink::getType():A_ASCIISTRING, shall match the Logical Link's physical vehicle link type, obtainable via MCDDbDLogicalLink::getDbPhysicalVehicleLink():MCDDbPhysicalVehicleLink::getType():A_ASCIISTRING.

— The resource’s interface connector pins, obtainable via the method
MCDDbPhysicalInterfaceLink::getDbInterfaceConnectorPins():
MCDDbInterfaceConnectorPins,
shall match the Logical Link’s vehicle connector pins, obtainable via the method
MCDDbPhysicalVehicleLink::getDbVehicleConnectorPins():
MCDDbVehicleConnectorPins,
where pin number and pin type MCDConnectorPinType shall match for each connector pin.
If the D-server does not support a D-PDU API, this method returns an empty collection.

A matching interface resource can be selected for a Logical Link in the following two ways:

— Automatic selection by the D-server:
When opening a Logical Link, the D-server automatically selects a matching interface resource, supported by the D-PDU API function PDUGetResourceIds. In this case, the application using the D-server does not have to care about this process, no additional method calls are necessary. During the

automatic selection, the D-server is free to choose which of the matching and available interface to use for opening the LogicalLink. This selection may differ between different D-server implementations and is even not reproducible on multiple executions of the same D-server implementation.

- Selection of the interface resource by the application:
  Before opening a Logical Link, the application actively selects a matching interface resource. The details of this process are explained below.

The class `MCDDLogicalLink` – which represents a runtime Logical Link in the D-server – provides the following methods to select an interface resource and to retrieve the selected interface resource:

- `MCDDLogicalLink::getInterfaceResource():MCDInterfaceResource`
  returns the interface resource to be used with the LogicalLink. Throws an exception of type `MCDProgramViolationException` with error code `eRT_NOT_ALLOWED_IN_LL_STATE_CREATED` if it is called before the Logical Link has been opened (`MCDDLogicalLink::open()`) and if `MCDDLogicalLink::selectInterfaceResource()` has not been called before. When the Logical Link has been opened without calling `selectInterfaceResource()`before, `getInterfaceResource()` returns the interface resource which has been automatically selected by the D-server.

- `MCDDLogicalLink::selectInterfaceResource(MCDInterfaceResource resource):void`
  allows the application to select a specific interface resource belonging to a certain interface to be used for opening the Logical Link. If not called, the D-server selects one of possibly multiple matching interface resources from possibly different interfaces, and uses this interface resource when opening the Logical Link. This method is only allowed to be called in LogicalLinkState eCREATED. Otherwise, an exception of type `MCDProgramViolationException` with error code `eRT_ONLY_ALLOWED_IN_LL_STATE_CREATED` is thrown.

- `MCDDLogicalLink::getMatchingInterfaceResources():MCDInterfaceResources`
  returns the interface resources matching the requirements of this Logical Link (protocol and physical vehicle link). This method calls the D-PDU API function `PDUGetResourceIds`. The application can call `getMatchingInterfaceResources()` to pre-select a matching resource to use for a call of `selectInterfaceResource()`.

Before calling `selectInterfaceResource()`, the application shall check, if the resource is currently available by calling `MCDInterfaceResource::isAvailable():A_BOOLEAN`.

NOTE The method `MCDInterfaceResource::isInUse():A_BOOLEAN` returns true if this resource is currently in use. Even if the resource is already in use, it may still be available - depending on the characteristics of this resource. For example, if a resource for a CAN protocol on a certain CAN bus is already in use by one Logical Link, it may also be used multiple times by other Logical Links, depending on the capabilities of the VCI module's CAN protocol driver.

The status values for `isAvailable()` and `isInUse()` are retrieved via the D-PDU API function `PDUGetResourceStatus`.

9.28.11 Send Break Signal

The method `MCDDLogicalLink::sendBreak():void` allows to send a break signal on the Logical Link (calls the D-PDU API function PDUIoCtl with command PDU_IOCTL_SEND_BREAK). A break signal is a feature of certain physical layers and can only be sent on these physical layers (e.g. SAE J1850 VPW physical links and UART physical links; for further information on PDU_IOCTL_SEND_BREAK, see ISO 22900-2). Throws an exception of type `MCDProgramViolationException` with error code `eRT_NOT_ALLOWED_IN_LL_STATE_CREATED` if the Logical Link is in state eCREATED, or with error eRT_PDU_API_CALL_FAILED if the corresponding function call of PDUIoCtl() at the D-PDU API failed.

9.28.12 MCDDbInterfaceCable

A VCI module is connected to a vehicle using an interface cable which has the interface connector (to be plugged into the vehicle connector) attached to one end of the cable and the VCI connector (to be plugged into the VCI) attached to the other end. To connect a VCI to a vehicle with a different type of vehicle connector, e.g. an OEM specific connector instead of an OBD connector, a VCI is usually shipped with several different interface cables which have different types of connectors.

The class `MCDInterface` offers methods to access information about the different interface cables available for a certain VCI module. However, this information is not required for any runtime operations. It is information only and can be used in a diagnostic application to guide the user to connect the correct cable to the VCI module.

The method `MCDInterface::getDbInterfaceCables():MCDDbInterfaceCables` returns the collection of all possible interface cables specified for the VCI module. These cables are defined in the MVCI's cable description file (CDF). The D-server shall parse the CDF file to create the result of this method.

The method `MCDInterface::getCurrentDbInterfaceCable():MCDDbInterfaceCable` returns the interface cable which is currently connected to the VCI module. A VCI module may use an automatic cable detection or internal configuration to provide this information. The method calls the D-PDU API function `PDUIoCtl` with the command PDU_IOCTL_GET_CABLE_ID, and shall parse the CDF file in addition to retrieve the properties of the cable with the cable ID delivered by `PDUIoCtl`.

A `MCDDbInterfaceCable` object provides information about an interface cable. This information is provided by the MVCI's cable description file (CDF):

The method `MCDDbInterfaceCable::getInterfaceConnectorType():A_ASCIISTRING` returns the type of the interface connector of the interface cable. The connector type is provided in the CDF.

NOTE There is no predefined list of possible return values defined in the D-server or D-DPU API standards.

The method `MCDDbInterfaceCable::getDbInterfaceConnectorPins():MCDDbInterfaceConnectorPins` returns the interface connector pins of the interface cable.

A `MCDDbInterfaceConnectorPin` object contains the mapping of the interface connector pin, returned by `getPinNumber():A_UINT32`, to a pin on the VCI connector of the interface cable, returned by `getPinNumberOnVci():A_UINT32`. The pin number on the VCI connector is only additional information which is not required for any runtime operation. Information about the VCI connector pin number is provided in the CDF.

9.28.13 Accessing VCI module features

Several features of VCI modules are accessible by the following methods of `MCDInterface:`

The method `MCDInterface::reset():void` resets the VCI module (calls the D-PDU API function PDUIoCtl with the command PDU_IOCTL_RESET). The reset command will cancel all activities currently being executed by the VCI module (without proper termination). All existing Logical Links and associated ComPrimitives will be removed. All hardware properties of the MVCI Protocol Module (e.g. programming voltage) will be reset to the default settings. After the completion of the reset command, the VCI module shall be treated as if it were a newly connected VCI module.

The method `MCDInterface::setProgrammingVoltage(pinOnInterfaceConnector:A_UINT32, voltage:A_FLOAT64):void` sets the programming voltage (in Volts) on the specified pin of the interface connector of the VCI module (calls the D-PDU API function PDUIoCtl with the command PDU_IOCTL_SET_PROG_VOLTAGE).

The method `MCDInterface::getProgrammingVoltage(pinOnInterfaceConnector:A_UINT32):A_FLOAT64` returns the programming voltage (in Volts) on the specified pin of the interface connector of the VCI module. This method is used to read the feedback of the programming voltage from the voltage source, which is set by the method `setProgrammingVoltage` (calls the D-PDU API function `PDUIoCtl` with the command PDU_IOCTL_READ_PROG_VOLTAGE).

The method `MCDInterface::getBatteryVoltage():A_FLOAT64` reads the battery voltage (in Volts) on the VCI module's VCI connector (calls the D-PDU API function `PDUIoCtl` with the command PDU_IOCTL_READ_VBATT).

The method `MCDInterface::getClampState(pinOnInterfaceConnector:A_UINT32, clampName:A_ASCIISTRING):MCDValue` returns the current state of the clamp specified by the pin number on the interface connector (first parameter). In general, the specified clamp name (second parameter) is tool manufacturer specific, and shall be supplied by an equivalent manufacturer specific IO_CTRL-command to be carried out with the D-PDU API function `PDUIoCtl`, where the clamp name is expected to match the ShortName of the IO_CTRL-command. The clamp state is returned as a `MCDValue`. The data type of this value and the interpretation of the value (e.g. resolution, unit and range) depend on the tool manufacturer specific definition for the specific clamp name.

The following clamp name is expected to be supplied by any D-server supporting a D-PDU API:

- “IgnitionClamp”:
  clamp to retrieve the ignition state of the vehicle. Data type of the returned value is A_BOOLEAN, where value “true” means “ignition on”, “false” means “ignition off”.

Tool manufacturer specific clamp names shall start with the domain name (similar to getProperty()). For a german vendor A a clamp name would look like “de.VendorA.NameOfTheClamp”.

Other `MCDInterface` methods like `getVendorName()`, `getPDUApiSoftwareName()`, `getPDUApiSoftwareVersion()`, `getHardwareSerialNumber()` provide information about the VCI vendor and several different version numbers of the VCI module. This information is retrieved from the D-PDU API by the function `PDUGetVersion`.

9.28.14 Adding Logical Links which are not found in the Vehicle Information

There are engineering use cases, where a Logical Link shall be used, which is not defined in the ODX data (DbLogicalLink is not available). For these cases, the D-server API offers the possibility to create a runtime logical link without an existing DbLogicalLink.

- Use case 1:
  There is a PHYSICAL-VEHICLE-LINK in the VEHICLE-INFO-SPEC, which can be used to create a runtime Logical Link

- Use case 2:
  No VEHICLE-INFO-SPEC is available at all, or there is no PHYSICAL-VEHICLE-LINK in the VEHICLE-INFO-SPEC, which can be used to create a runtime Logical Link

The following methods cover these use cases:

Use case 1: the method `<<D>> MCDLogicalLinks::addByAccessKeyAndVehicleLink (A_ASCIISTRING accessKeyString, MCDDbPhysicalVehicleLink physicalVehicleLink, MCDDbProtocolStack protocolStack, MCDCooperationLevel cooperationLevel): MCDDLogicalLink` allows the creation of a runtime instance of a Logical Link, which is not found in the VEHICLE-INFO-SPEC. The first parameter is the Access Key for the Logical Link as string. The second parameter is a `MCDDbPhysicalVehicleLink` object, which shall be retrieved by

MCDDbVehicleInformation::getDbPhysicalVehicleLink(). Thus, this method affords to use a MCDDbPhysicalVehicleLink object, which shall belong to an available Vehicle Information.

Use case 2: the method
<<D>> MCDLogicalLinks::addByAccessKeyAndInterfaceResource (A_ASCIISTRING accessKeyString, MCDInterfaceResource interfaceResource, MCDCooperationLevel cooperationLevel): MCDDLogicalLink allows the creation of a runtime instance of a Logical Link, where no VEHICLE-INFO-SPEC is available at all, or there is no PHYSICAL-VEHICLE-LINK in the VEHICLE-INFO-SPEC, which can be used to create a runtime Logical Link. The first parameter is the Access Key for the Logical Link as string. The second parameter is an MCDInterfaceResource object, which the application shall retrieve from a connected interface by MCDInterface::getInterfaceResources(). The application shall parse the resources of this interface to find an available interface resource, which has a protocol type and Physical Interface Link suitable for the Logical Link to be created. When the Logical Link is opened, it will use the interface resource, which has been selected by the method addByAccessKeyAndInterfaceResource(). If the D-server does not support D-PDU API, this method throws a MCDProgramViolationException with error code eRT_PDU_API_NOT_SUPPORTED.

NOTE In contrast to the methods MCDLogicalLinks::addXXX, the methods addByAccessKeyAndVehicleLink and addByAccessKeyAndInterfaceResource always create a new runtime MCDDLogicalLink object. That means if a client calls such a method twice with exactly the same input parameters two different MCDDLogicalLink objects will be created. Every MCDDLogicalLink object can be identified by its unique ShortName. This ShortName is generated by the D-server with the following pattern: #RtGen_MCDDLogicalLink_<Number>. The number in this generated ShortName is increased for each creation of a MCDDLogicalLink. The counting starts at zero.

9.28.15 Behaviour of a MCD-server not supporting VCI Modules in accordance with D-PDU API Standard

If the D-server does not support the D-PDU API standard, it is not possible to select a specific VCI module from the application, and also it is not possible to access any VCI module features as defined in the classes MCDInterface and MCDInterfaceResource.

The application can check whether the D-PDU API standard is supported by the system property SupportsPduApi.

In case D-PDU API is not supported, the methods getConnectedInterfaces() and getCurrentInterfaces() of MCDSystem always return an empty collection. The methods getInterfaceResource()and selectInterfaceResource() of MCDDLogicalLink, and MCDLogicalLinks::addByAccessKeyAndInterfaceResource() throw an MCDProgramViolationException with error code eRT_PDU_API_NOT_SUPPORTED. All these methods are not used by the application if the D-server does not support D-PDU API. In this case the application will never retrieve any object of the classes MCDInterface, MCDInterfaceResource, MCDDbPhysicalInterfaceLink, MCDDbInterfaceCable and MCDDbInterfaceConnectorPin.

In case D-PDU API is not supported, the application can open a Logical Link without knowledge about any VCI module, and without having to select a specific VCI module. Handling and selection of one or more VCI modules is done completely inside the D-server in this case. The only VCI related method to be called before opening a Logical Link is MCDSystem::prepareVciAccessLayer().

In case D-PDU API is not supported, the methods prepareVciAccessLayer() and unprepareVciAccessLayer() are not related to D-PDU API functions, but are still required to be called by the application. In this case these methods will throw a MCDProgramViolationException with error code eRT_INTERNAL_ERROR, if an error occurs.

9.29 Mapping of D-PDU API methods

9.29.1 General

The following subclauses describe the internal behaviour of an MCD-server implementation using a D-PDU interface in accordance with ISO 22900-2.

For such an MCD-server implementation, the mapping of MCD-3 methods to D-PDU API methods as described in this subclause shall be used.

9.29.2 Initialization and Selection of VCI Modules

See Table "Method for VCI module selection – D-server API and D-PDU API" from subclause "Support of VCI module selection and other VCI module features in accordance with D-PDU API Standard".

9.29.3 Communication on a Logical Link

Table 48 — Methods for Communication on a Logical Link – MCD-server API and D-PDU API shows the sequence of method calls from a client application at the MCD-3 API and the corresponding method calls from the MCD-server implementation at the D-PDU API used for communication on a Logical Link.

Table 48 — Methods for Communication on a Logical Link – MCD-server API and D-PDU API

Methods called at MCD-server API	Methods called at D-PDU API
MCDLogicalLinks::addByXXX() where XXX refers to all variants of add methods in the MCDLogicalLinks collection Initial link state is eCREATED	(No counterpart)
MCDDLogicalLink::open() Link state changes to eOFFLINE	Access to D-PDU API resources: PDUGetModuleIds PDUGetObjectId or by parsing the Module Description File (MDF) Optional: the availability of the resources can be checked by PDUGetResourceIds() PDUGetResourceStatus() PDUGetConflictingResources() Allocate the resource and create the D-PDU Logical Link: PDUCreateComLogicalLink()

Setting the communication parameters:
No counterpart. MCD-server internally reads the communication parameters from ODX.
	Get default values from the D-PDU API:
	PDUGetComParam(), PDUGetUniqueRespIdTable()
	Set values from ODX to the D-PDU API:
	PDUSetComParam(),PDUSetUniqueRespIdTable()

MCDDLogicalLink::gotoOnline() Link state changes to eONLINE	PDUConnect() Communication parameters become active during PDUConnect()
MCDDiagComPrimitives::add...() refers to all variants of add methods in the MCDDiagComPrimitives collection	(No counterpart)
MCDDLogicalLink::suspend()	PDUIoctl(PDU_IOCTL_SUSPEND_TX_QUEUE)

Table 48 (*continued*)

Methods called at MCD-server API	Methods called at D-PDU API
MCDDLogicalLink::resume()	PDUIoctl(PDU_IOCTL_RESUME_TX_ QUEUE)
MCDDLogicalLink::clearQueue()	1. PDUIoctl (PDU_IOCTL_SUSPEND_TX_QUEUE) 2. PDUIoctl (PDU_IOCTL_CLEAR_TX_ QUEUE) 3. PDUIoctl (PDU_IOCTL_RESUME_TX_ QUEUE)
MCDDLogicalLink::sendBreak()	PDUIoctl(PDU_IOCTL_SEND_BREAK)
MCDDLogicalLink::reset()	1. PDUIoctl (PDU_IOCTL_SUSPEND_TX_QUEUE) 2. PDUIoctl (PDU_IOCTL_CLEAR_TX_ QUEUE) 3. PDUIoctl (PDU_IOCTL_RESUME_TX_ QUEUE) 4. PDUDestroyComLogicalLink() / PDUDisconnect()
MCDDiagComPrimitive::executeSync()	PDUStartComPrimitive () The server shall suspend the calling thread until the interface has completely processed the passed comprimitive.
MCDDataPrimitive::executeAsync()	PDUStartComPrimitive () The server stores the comprimitive only in the related queue and returns immediacy.
MCDDataPrimitive::startRepetition()	PDUStartComPrimitive ()
MCDDiagComPrimitive::cancel()	PDUCancelComPrimitive()
MCDDLogicalLink::gotoOffline() Link state changes to eOFFLINE	PDUDisconnect()

MCDDLogicalLink::close()
Or
MCDDLogicalLink::reset()
Link state changes to eCREATED
	Only in Link State eONLINE or eCOMMUNICATION: PDUDisconnect()
	In all cases: PDUDestroyComLogicalLink()

MCDLogicalLinks::remove…() Refers to all variants of remove methods in the MCDLogicalLinks collection	(No counterpart)

Figure 208 — D-server API (MCD-3 API) and D-PDU API (prefix PDU) methods for Logical Link shows the mapping of those MCD-server API and D-PDU API methods which are used for handling a Logical Link.

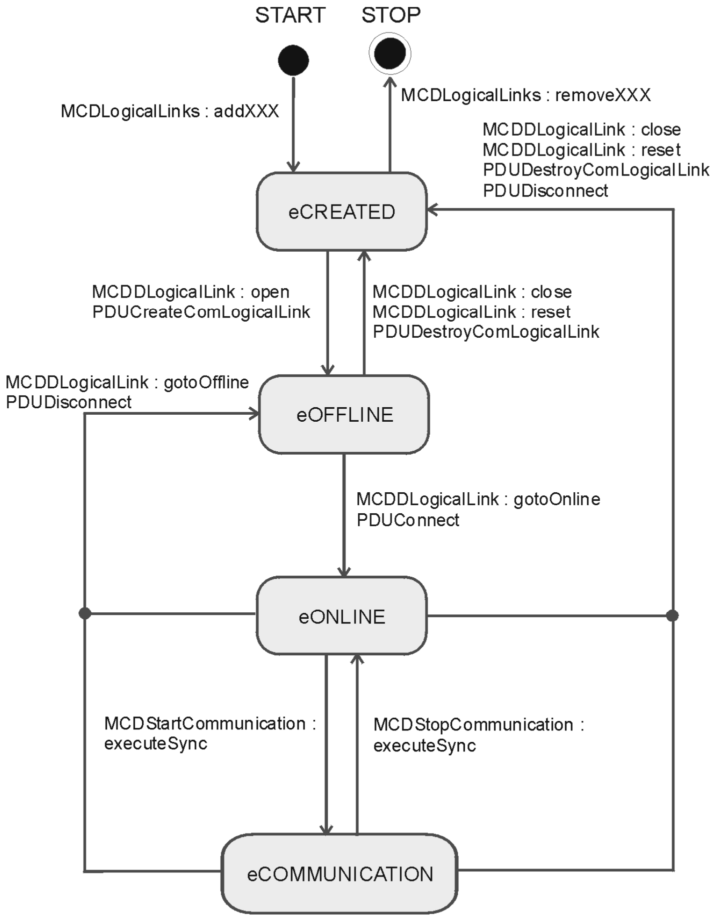


Figure 208 — D-server API (MCD-3 API) and D-PDU API (prefix PDU) methods for Logical Link

9.29.4 Handling of Communication Parameters

9.29.4.1 Changing communication parameters from the client application

A client application may want to set one or more communication parameters at runtime to values differing from the values defined in the ODX data. Table 49 — Changing communication parameters from the client application — MCD-3 API and D-PDU API shows the sequence of method calls in the MCD-3 API and the D-PDU API required for changing the communication parameters of a Logical Link.

Table 49 — Changing communication parameters from the client application — MCD-3 API and D-PDU API

Methods called at MCD-3 API	Methods called at D-PDU API
create a Control Primitive of type MCDProtocolParameterSet at the Logical Link	Get default values of all communication parameters from the DataBase: (no counterpart)
change value of one ore more Request Parameters of the MCDProtocolParameterSet Control Primitive (the Request Parameters of this Control Primitive correspond to the communication parameters of the logical link)	(no counterpart)

Table 49 (*continued*)

Methods called at MCD-3 API	Methods called at D-PDU API
MCDProtocolParameterSet::executeSync()	Set values changed by the client application to the D-PDU API: PDUSetComParam() or PDUSetUniqueRespIdTable() (depends on type of ComParam) for each modified ComParam PDUStartComPrimitive() with ComPrimitive type PDU_COPT_UPDATEPARAM: the changed ComParams become active

MCDProtocolParameterSet::fetchValueFromInterface()
MCDProtocolParameterSet::fetchValuesFromInterface()
	PDUGetComParam() shall be called to retrieve current settings from interface. (either the specified one with the given shortName or all protocolParameter defined for the related DbLoacation)

create a Control Primitive of type MCDProtocolParameterSet at the Logical Link	Get default values of all communication parameters from the DataBase: (no counterpart)

9.29.4.2 Setting temporary communication parameters for a DiagComPrimitive

Several communication parameter values may be overwritten at a DIAG-SERVICE in the ODX data. These values shall only be temporarily valid for the duration of the execution of this service.

Table 50 — Setting temporary communication parameters for a DiagComPrimitive — MCD-3 API and D-PDU API shows the sequence of method calls for this case.

Table 50 — Setting temporary communication parameters for a DiagComPrimitive — MCD-3 API and D-PDU API

Methods called at MCD-3 API	Methods called at D-PDU API
execute a DiagComPrimitive with temporary ComParams	Set values overwritten at the DIAG-SERVICE in the ODX data to the D-PDU API: PDUSetComParam() for each overwritten ComParam PDUStartComPrimitive() with flag TempParamUpdate set to „true“ (1)

9.29.4.3 Changing UNIQUE_ID Communication Parameters

The D-server allows to change the values of all communication parameters (protocol parameters) of a runtime logical link by using the control primitive `MCDProtocolParameterSet`. As there is no restriction with respect to which kinds of protocol parameters can be changed, it is also possible to change the addressing information represented by the protocol parameters of parameter class `eUNIQUE_ID`. The result of overwriting the values of the `UNIQUE_ID` protocol parameters is that the handle calculated from the `UNIQUE_ID` information can differ between runtime LogicalLink and DbLogicalLink in case the values of the corresponding protocol parameters have been changed at the runtime LogicalLink. This may lead to problems in the D-server as it might fail in resolving responses etc. Hence, changing the value of `UNIQUE_ID` protocol parameters is considered harmful.

9.29.5 MCDStartCommunication and MCDStopCommunication

The execution of Control Primitives of type MCDStartCommunication and MCDStopCommunication is mapped to D-PDU API method calls as shown in Table 51 — MCDStartCommunication and MCDStopCommunication - MCD-3 API and D-PDU API.

Table 51 — MCDStartCommunication and MCDStopCommunication - MCD-3 API and D-PDU API

Methods called at MCD-3 API	Methods called at D-PDU API
MCDStartCommunication::executeSync (There can be a definition for the StartCom service in the ODX data. In this case, this service's request and possibly overwritten communication parameters need to be passed to the D-PDU API.)	PDUStartComPrimitive() with ComPrimitive type PDU_COPT_STARTCOMM (May include request and overwritten communication parameters if supplied by the MCD-server, handling as described in Table 50 — Setting temporary communication parameters for a DiagComPrimitive — MCD-3 API and D-PDU API)
MCDStopCommunication::executeSync	PDUStartComPrimitive() with ComPrimitive type PDU_COPT_STOPCOMM

10 Error Codes

10.1 Principle

This clause describes the errors which may occur. Working with the API, errors may occur during or after a method call or generally within the server. The MCD-server returns the resulting error objects (`MCDError`) via the API to the Client using different ways.

- If an error crops up during the method execution, that means before the method returns, this error is passed on by means of an Exception. In case of serious errors e.g. for `DiagComPrimitives` this is independent from asynchronous or synchronous execution.
- If any error occurs independent from a method call, for example a server wide error, this error is returned by means of an Event. The error object will either be located directly within the Event (`onSystemError`, `onLinkError, onCollectorError`) and can be polled using `getError`, or the Event (`onPrimitiveError`) transports a `MCDResult`, within which the error object is located.

For the Diagnostic part:

- In case of a synchronous execution of a `DiagComPrimitive` and the cropping up of non serious errors, the error is handed over as return value in `MCDResult`.
- In case of asynchronous execution of `DiagComPrimitives`, the error is returned within the `MCDResult` object via an Event (`onPrimitiveError`).

Most of the methods in this object model can throw an exception, an `MCDException` or one of the from `MCDException` derived exceptions. These exceptions only transport the error object.

<u>Types of exception:</u>

- `MCDParameterizationException`
  (inadmissible or inconsistent parameterisation for the execution of a method)
- `MCDProgramViolationException`
  (Problem at program flow)
- `MCDDatabaseException`
  (Problem at database access)
- `MCDSystemException`
  (system wide problem)
- `MCDCommunicationException`
  (Problem in Communication between MCD-server and ECU)
- `MCDShareException`
  (Problem at the handling of shared objects)
- `MCDJobException`
  (Problem generated at handling of jobs)

An error object consists of :

- Code
- Code description
- Severity
- Vendor Code
- Vendor Code description.

The **Code** is represented by a A_UINT16 and is defined in the following tables as a general definition of the errors. The code 0x0000 means error free.

The **Code description** of the error is represented by a A_ASCIISTRING and is defined in the following tables as a general definition of the errors.

The **Vendor Code** is represented by a A_UINT16 and intended as manufacturer specific error supplement (Information). The code is not standardized. The vendor code is be useable in all cases of an not empty code, that means a common standardized error shall be used in code.

The **Vendor Code description** is represented by a A_ASCIISTRING and is provided by the manufacturer of the MCD-server. These are the description of the manufacturer specific vendor error code.

The **Severity** is used for the assessment of the error and can be subdivided as follows:

Table 52 — Severity

Severity	Short cut	Description
eMESSAGE	M	This is information with importance for the user. This does not change the execution path in the software.
eWARNING	W	A problem occurred and was successfully solved by the system.
eERROR	E	A problem occurred and shall be handled by the client. The object still exists and could react normally after the problem is solved by the respective clients.
eFATAL_ERROR	F	An unsolvable problem occurred at an object and the complete operation could not be performed or done work is lost. The object still exists, but can not act normally and usually is not accessible. The problem can not be completely handled or solved in the software.
eTERMINATE	T	An unsolvable problem occurred, the system is in an unsafe state and shuts down immediately (the server is not accessible / the object can not be accessed anymore).

To unify the error handling with normal runtime response, the two Severity's eMESSAGE and eWARNING are used. An eMESSAGE or an eWARNING will never be reported by an exception, but could use all logging functionality of the error handler.

The main difference between eERROR and eFATAL_ERROR is, that after an eERROR the application could continue if it handles the error. eFATAL_ERROR means that something principle is not working correct. Even if the application handles the error, some damage was done or it is not possible to resolve the problem.

The following describes the determination of error severity selection:

EXAMPLE 1 If the application could not open a database-file, this is an eERROR, because no damage was done and the application could open a different database file.

EXAMPLE 2 If an ECU is disconnected, this is an eFATAL_ERROR, because the application itself could not reconnect the hardware and the operations could not be completed.

10.2 Description of the errors

10.2.1 Error free behaviour

If no error has occurred, but an error object is handed over within the Event or `MCDResult` (this only in diagnostic part), the ErrorCode of the `MCDError` object is `eNO_ERROR (0000)`.

10.2.2 Parameterisation errors

This errors can be carried by `MCDParameterizationException`, every event or `MCDResult`.

An error occurred on the basis of a wrongly set method parameter. The parameter has e.g. the wrong type or has a wrong value.

The exception can be solved by adjusting the parameter, so that it fits the defined constraints of the method.

10.2.3 Runtime / ProgramViolation errors

This errors can be carried by `MCDCommunicationException`, `MCDShareException`, every event or `MCDResult`.

ProgramViolation Errors are errors, where the client can do something or has provoked the error itself (usage error), e.g. false LogicalLink state, VI in false LL state (`eCREATED`).

10.2.4 Database errors

Database errors occur if an access to the ASAM MCD 2 database has failed. An error occurred, while the D-server tried to access a database entry. The database access can e.g. fail, if an entry is not filled with data. The exception can be solved by adjusting the database.

This errors can be carried by `MCDDatabaseException`, every event or `MCDResult`.

10.2.5 System errors

This errors can be carried by `MCDSystemException` or `onSystemError`.

SystemErrors are critical errors, where a client can do nothing (the error can be in OS or in MCD-server), e.g memory overflow, division by zero.

10.2.6 Communication errors

This errors can be carried by `MCDCommunicationException` or `onSystemError`.

A problem occurred during the communication between the D-server and the connected ECU (hardware). (e.g. ECU was removed from the system, ECU does not respond, wrong ECU is attached.).

The exception can be solved by adjusting the hardware environment.

10.2.7 Job error

This errors can be carried by MCDJobException, Job-specific events (onPrimitiveTerminated, onPrimitiveHasIntermediateResult, onPrimitiveError) or MCDResult.

Exceptions of type MCDJobException are to be used by Jobs only.

Job Errors are errors, which are generated by the Job itself.

Annex A
(informative)

Code examples

A.1 Sample : collector result access

```
// *******************************************************************
// ***  Part of the ASAM MCD3 Specification                        ***
// ***  File        : Pseudocode for collector result access        ***
// ***  Version     : 0.2                                           ***
// ***  Date        : June, 29 2004                                 ***
// *******************************************************************

//Structure of result example 1
//
//CollectorResult        MCDResult collection
//                   (index 0 of result collection: )
//CollectorResultLine    MCDResults
//                   (index 0 of response collection: )
//Line       MCDResponse
//                   (Line has a collection of RespParam:)
//TimeStamp                 MCDResponseParameter
//ObjectStruct       MCDResponseParameter
//                   (ObjectStruct has a collection of RespParam:)
//Triangle                  MCDResponseParameter
//ampl       MCDResponseParameter
//
//*********************************************************************

//used variables
MCDCollector     Collector
MCDResults           CollectorResult
MCDResult        Result
MCDResponse      Line
MCDResponseParameter    ObjectStruct
A_UINT64         TimeStamp
A_INT8       Triangle
A_FLOAT32            ampl
A_UINT32         count
A_UINT32         BufferId

MCDDatatypeShortname Shortname_TimeStamp    = NEW
MCDDatatypeShortname("TimeStamp")
MCDDatatypeShortname Shortname_ObjectStruct = NEW
MCDDatatypeShortname("ObjectStruct")
MCDDatatypeShortname Shortname_Triangle     = NEW
MCDDatatypeShortname("Triangle")
MCDDatatypeShortname Shortname_ampl         = NEW
MCDDatatypeShortname("ampl")

// methods to access the values of these structure
```

```
//1. preparing the direct access to the MCDResponseParameter

//get the amount of count rows in MCDResult from the MCD System
//to the client via network
CollectorResult = Collector.fetchResults(count, BufferId)

//next steps need no net transfer anymore
//evaluate the first result (examplarily)
//first take the first result from the result array
//and from this result the response
Line           = CollectorResult.getItemByIndex(0).getResponses().getItemByIndex(0)

//in the response is now direct access possible

//2. direct access to the values

// the MCDResponseParameter has its value in a MCDValue
// so it can store any value in the same way

//every result line of the collector result has a time stamp and a object struct
TimeStamp    = Line.getResponseParameters().getItemByName(Shortname_TimeStamp)
.getValue().getUint64()


//object struct : these are the values of each object in the collector in
//a struct to allow diferent data types and separate it from the time
//stamp and the status
ObjectStruct = Line.getResponseParameters().getItemByName(Shortname_ObjectStruct)

Triangle     = ObjectStruct.getResponseParameters.getItemByName(Shortname_Triangle)
.getValue().getInt8()

ampl         = ObjectStruct.getResponseParameters.getItemByName(Shortname_ampl).
               getValue().getFloat32()
```

A.2 Example job source code for a single ECU job

```
//********************************************************************
//***  Part of the ASAM MCD3 Specification                         ***
//***  File        : Java source code of a SingleECUJob            ***
//***  Version     : 2.20.00                                       ***
//***  Date        : August, 06 2008                               ***
//********************************************************************



//import all ASAM MCD standard classes and interfaces
//(datatypes, mcd and d) from package asam.d and
//asam.job
import asam.d.*;
import asam.job.SingleEcuJobTemplate;

//import all needed java classes of the allowed java
//classes for jobs
import java.lang.Object.*;

//every specific job implements the JobTemplate interface
public class SingleEcuJobExample implements SingleEcuJobTemplate
{



     /** Constructor */
     public SingleEcuJobExample()
     {
     //constructor remains empty.
     //important if anonymous classes had to be instantiated at runtime
     }


     /** the execution instructions for this job */
     public void execute (MCDRequestParameters    inputParameters,
                          MCDJobApi      jobHandler,
                          MCDDLogicalLink    link,
                          MCDSingleEcuJob    jobObject) throws MCDException
     {

     /**
     * This is the method where the handling of the job
     * is described.
     *
     * In this example three services will be executed
     * ten times and a result structure is build from
     * the results, the services deliver. The result
     * is the same as in specification example DTC and
     * it is described in a method before main.
     * Only every second time the intermediate results
     * will be delivered. Only one ECU is answering,
     * so that the result object only contains one
     * response. It is easy to extend this example to
     * more than 1 ECU. Every Response can have more
     * than one FSP_Sequence ResponseParameter. It
* depends on the evaluation of the service results.
*/

//MCDDbResponse as database template for
//runtime response (only one response in a
//SingleEcuJob available)
MCDDbResponse dbResponse = jobObject.getDbObject().getDbResponses().getItemByIndex(0);
//used services
MCDService services[]               = null;
//results of used services
MCDResult[] serviceResult           = new MCDResult[3];
//1. final response parameter (FSP_Sequence)
MCDResponseParameter finalRespParam = null;
//final response
MCDResponse finalResponse           = null;
//final result
MCDResult finalResult               = null;
//the single response parameters
MCDResponseParameter FSP_Sequence   = null,
                     FSP            = null,
                     DTC            = null,
                     DTC_State      = null,
                     Complete       = null,
                     Temperature    = null,
                     Env_Sequence   = null,
                     Env            = null,
                     Env_Temp       = null,
                     Env_Speed  = null;
//all collected and not sended intermediate results
MCDResults rtResultCollection       = jobObject.createResultCollection();
//counter of value arrays
int i = 0, j = 0, k = 0, l = 0, n = 0;

//these are the temporary variable
//with every loop they contain other values or contain
//no value after each loop the variables will be reseted
MCDValue[]   dtc           = null;
MCDValue[]   dtc_state     = null;
MCDValue[]   temperature   = null;
MCDValue[]   env_temp  = null;
MCDValue[]   env_speed = null;
int noOfFSPSeq = 0, noOfFSP = 0, muxbranch[] = null, noOfEnvSeq = 0;

//here is a example table,
//which variables in which loop are containing the result
//1. loop -> dtc[0], dtc_state[0], muxbranch[0](0), temperature[0]
//2. loop -> dtc[0], dtc_state[0], dtc[1], dtc_state[1],
//           muxbranch[0](1), muxbranch[1](0), temperature[0],
//           env_speed[0], env_temp[0]
//3. loop -> dtc[0], dtc_state[0], dtc[1], dtc_state[1],
//           muxbranch[0](0), muxbranch[1](0),
//           env_code[0], temperature [1]
//4. loop -> dtc[0], dtc_state[0],muxbranch[0](1),
//           env_speed[0], env_temp[0], env_speed[1],
//           env_temp[1], env_speed[2], env_temp[2]
// and so on
//These values will be set to the defined MCDValue arrays like in
```

```
//example for creating a correct value
//MCDValue[] dtc = new MCDValue[2];
//dtc[0] = jobHandler.createValue(MCDDataType. eA_UINT32);
//dtc[0].setUint32(100);

// after that :
//- all MCDValues are valid and with a correct type and value
//- the counts are set

   //example for servicenames
   String[] servicenames = {"service1", "service2"};

//Begin of the code for this job

//first create the services used in this job
for (i=0;i<3;i++)
{
   //casting the created MCDDiagComPrimitive to MCDService
   services[i] = (MCDService)
                link.getDiagComPrimitives.add (link.getDbObject()
                                                   .getDbLocation()
                                                   .getDbServices()
                                                   .getItemByName(servicenames[i]));
}
//build the final result:
//1. result via MCDJob
finalResult = jobObject.createResult(MCDResultType.eRESPONSE, 0, "", 0, "", MCDSeverity.eMESSAGE);

//2. response for the result (with the first Resp-Param down to the first field or //multiplexer)
//   this will be the positive result for this job
MCDDbLocation location = link.getDbObject().getDbLocation();
finalResponse          = finalResult.getResponses().add(location, true);

//3. the first and only FSP_Sequence in the final result
finalRespParam         = finalResponse.getResponseParameters().getItemByIndex(0);

//poll 10 times
//-> this means 10 intermediate results in 5 sendResults
//and one final result in a own sendResult
//will be delivered to the MCD System

for (i=1;i<=10;i++)
{
       //execute the 3 services every time you want make
       //an intermediate result
       for (n=0;n<3;n++)
       {
         serviceResult[n] = services[n].executeSync();
       }

       /*
       now evaluate and compute the service
       results to temporary variables
       in order to build a job result
       */

       //build an intermediate result
```

```
        MCDResult intermediateResult     = jobObject.createResult (MCDResultType.eRESPONSE, 0,
                                                    "", 0, "", MCDSeverity.eMESSAGE);

        //with its response (positive)
        MCDResponse intermediateResponse = intermediateResult.getResponses()
                                                    .add((link.getDbObject()
                                                    .getDbLocation()), true);

for (j=0;j<noOfFSPSeq;j++)
{
    //begin with j. ResponseParameter FSP_Sequence
   FSP_Sequence = intermediateResponse.getResponseParameters().getItemByIndex(j);

   for (k=0;k<noOfFSP;k++)
   {
    //reset branch counter
           //(important for arrays of DOPs in mux branch)
           int branch1 = 0, branch2 = 0;

           //add a FSP structure down to the
           //Multiplexer complete
           FSP         = FSP_Sequence.getParameters().addElement();
           //set value for DTC (the value of the k. element of MCDValue array is already set)
           DTC         = FSP.getParameters().setParameterByName("DTC", dtc[k]);
           //set value for DTC_State
         //(the value of the k. element of MCDValue array is already set)
           DTC_State   = FSP.getParameters().setParameterByName ("DTC_State", dtc_state[k]);
           //set branch of multiplexer
           //Return value ist MCDResponseParameter
         Complete    = FSP.getParameters().getItemByName("Complete");
         Complete.getParameters().addMuxBranchByIndex((short) muxbranch[k]);

           //depending on multiplexer branch:
           switch (muxbranch[k])
           {
           case 0 : //Env_Temperature
                    //set value for Env_Temperature
                    Temperature = Complete
                                 .getParameters()
                                 .setParameterByName("Env_Temperature", temperature[branch1]);
                     //the next time use the next array
                     //element in temporary variable
                     //Temperature
                     branch1++;
                break;
           case 1 : //Env_Sequence
                    //get DDO: Env_Sequence
                     Env_Sequence = Complete.getParameters().getItemByName("Env_Sequence");
                     //for all needed elements of this sequence
                     for (l=0;l<noOfEnvSeq;l++)
                     {
                       //create an element
                           Env = Env_Sequence.getParameters().addElement();
                           //set value Env_Temperature
                           Env.getParameters()
                                      .setParameterByName ("Env_Temperature",env_temp[branch2]);
                           //set value Env_Speed
```

```
                               Env.getParameters()
                                          .setParameterByName ("Env_Speed", env_speed[branch2]);
                          }
                          //the next time use the next array
                          //elements in temporary variable
                          //env_temp and env_speed
                          branch2++;
                   break;
               }

                    //build the necessary information to the final
                    //result the final result will have 1 FSP_Sequence
                    //and all created FSP in it adds an element to
                    //a sequence as a copy of an existing
                    //MCDResponseParameter (with the full structure)
                    //the return value is not relevant
                    MCDResponseParameter aParameter = finalRespParam.getParameters()
                                                                    .addElementWithContent(FSP);
                }
            }

            MCDResult aResult = rtResultCollection.addWithContent(intermediateResult);

            //send result as intermediate result after every second run,
            //except we are in the final run and send a final result soon anyhow.
            if (rtResultCollection.getCount() == 2 && j!=10)
            {
            //send the intermediate results
            jobHandler.sendIntermediateResults (rtResultCollection);
            }

            try
           {
               //now wait for 1 minute
               wait(60000);
            }
            catch (java.lang.InterruptedException ie)
            {
               //exception handling
            }
        }
     //send final result
     jobHandler.sendFinalResult(finalResult);

     //delete the used services
     for (i=0;i<3;i++)
     {
        link.getDiagComPrimitives.remove(services[i]);
     }

     //end of job
     return;
   }
}
```

Annex B
(normative)

COMPUCODE template

This annex gives definition of code template for a COMPUCODE. The code for a COMPUCODE shall implement this template, and can then be executed by the D-server.

COMPUCODE's code template is defined as follows:

```
 /*
 * Definition of the Java interface for COMPU_CODES.
 */
package asam.d.compucode;

 /*
 * Interface for a COMPU-METHOD of type COMPUCODE
 * @author ASAM MCD 3 Standardization Group
 * @version 00.01.00
 */
public interface I_CompuCode
{
   /*
   * Executes the compu-method. This method is
   * called by the COMPUCODE execution engine of an MCD-3 server in case
   * a compu-method has to be executed for a diagnostic value
   *
   * @param input The input parameter of the compu-method.
   * @return the return value of the compu-method.
   */
  public Object compute(Object input) throws java.lang.Exception;
}
```

Annex C
(normative)

Job Templates

C.1 General

This annex gives definitions of code templates for each kind of job. Job code shall implement these templates, and can then be executed by the D-server.

C.2 Single ECU Job

The single ECU job's code template is defined as follows.

```
/*
* Definition of the Java interface for ASAM Single ECU Jobs.
*/
package asam.job;

import asam.d.*;
/**
* Interface for ASAM MCD 3 compliant single ECU jobs.
* @author ASAM MCD 3 Standardization Group
* @version 02.02.00
*/
public interface SingleEcuJobTemplate {

/**
* Initializes and executes the single ECU job. This method is called
* by the job execution engine of an MCD-3 server in case one of
* the methods MCDSingleEcuJob.executeSync(),
* MCDSingleEcuJob.executeAsynch(), and MCDSingleEcuJob.startRepetition() is
* called by a client application.
*
* @param inputParameters The input parameters of the job.
* @param mcdJobApi The MCDJobApi object required for communication
*                   between the job and MCD-3 server.
* @param mcddLogicalLink The logical link the job has been started on.
* @param mcdSingleEcuJob The MCDSingleEcuJob a job class is referenced from.
* @throws an MCDException in case an error occurs during execution
*/
public void execute (
MCDRequestParameters inputParameters,
MCDJobApi mcdJobApi,
MCDDLogicalLink mcddLogicalLink,
MCDSingleEcuJob mcdSingleEcuJob) throws MCDException;
}

/**
* Notifies the job that it should cancel its execution. This method is called
* by the job execution engine of an MCD-3 server in case the method
* MCDDiagComPrimitive.cancel() is called by a client application or the
      * D-server.
      */
      public void cancel ();
```

C.3 Flash Job

The java interface of flash jobs (`MCDFlashJob`) extend the java interface of normal diagnostic jobs (`MCDSingleEcuJob`) by a parameter for an MCDFlashSessionDesc object, i.e. the flash job’s code template is defined as follows:

```
/*
* Definition of the Java interface for ASAM Flash Jobs.
*/
package asam.job;

import asam.d.*;
/**
* Interface for ASAM MCD 3 compliant flash jobs.
* @author ASAM MCD 3 Standardization Group
* @version 02.02.00
*/
public interface FlashJobTemplate {

/**
* Initializes and executes the flash job. This method is called
* by the job execution engine of an MCD-3 server in case one of
* the methods MCDFlashJob.executeSync() and MCDFlashJob.executeAsynch()
* is called by a client application.
*
* @param inputParameters The input parameters of the flash job.
* @param mcdJobApi The MCDJobApi object required for communication
*                  between flash job and MCD-3 server.
* @param mcddLogicalLink The logical link the job has been started on.
* @param mcdFlashJob The MCDFlashJob a job class is referenced from.
* @param mcdFlashSessionDesc The flash session to be processed by this
*                            flash job implementation.
* @throws an MCDException in case an error occurs during execution
*/
public void execute (
MCDRequestParameters inputParameters,
MCDJobApi mcdJobApi,
MCDDLogicalLink mcddLogicalLink,
MCDFlashJob mcdFlashJob,
MCDFlashSessionDesc mcdFlashSessionDesc) throws MCDException;
}

/**
* Notifies the job that it should cancel its execution. This method is called
* by the job execution engine of an MCD-3 server in case the method
* MCDDiagComPrimitive.cancel() is called by a client application or the
* D-server.
*/
public void cancel ();
```

C.4 Multiple ECU Job

The multiple ECU job’s code template is defined as follows.

```
/*
* Definition of the Java interface for ASAM Multiple ECU Jobs.
*/
package asam.job;

import asam.d.*;
/**
* Interface for ASAM MCD 3 compliant multiple ECU jobs.
* @author ASAM MCD 3 Standardization Group
* @version 02.02.00
*/
public interface MultipleEcuJobTemplate {

/**
* Initializes and executes the multiple ECU job. This method is
* called by the job execution engine of an MCD-3 server in case one
* of the methods MCDMultipleEcuJob.executeSync(),
* MCDMultipleEcuJob.executeAsynch(), and
* MCDMultipleEcuJob.startRepetition() is called by a client application.
*
* @param inputParameters The input parameters of the job.
* @param mcdJobApi The MCDJobApi object required for communication
*                  between the job and MCD-3 server.
* @param mcddLogicalLink The logical link the job has been started on.
* @param mcdMultipleEcuJob The MCDMultipleEcuJob a job class is referenced from.
* @param mcdProject The MCDProject that provides the context for the multiple ECU job.
* @throws an MCDException in case an error occurs during execution
*/
public void execute (
MCDRequestParameters inputParameters,
MCDJobApi mcdJobApi,
MCDDLogicalLink mcddLogicalLink,
MCDMultipleEcuJob mcdMultipleEcuJob,
MCDProject mcdProject) throws MCDException;
}

/**
* Notifies the job that it should cancel its execution. This method is called
* by the job execution engine of an MCD-3 server in case the method
* MCDDiagComPrimitive.cancel() is called by a client application or the
* D-server.
*/
public void cancel ();
```

C.5 Security Access Job

The security access job's code template is defined as follows.

```
/*
* Definition of the Java interface for ASAM Security Access Jobs.
*/
package asam.job;

import asam.d.*;
/**
* Interface for ASAM MCD 3 compliant security access jobs.
* @author ASAM MCD 3 Standardization Group
* @version 02.02.00
*/
public interface SecurityAccessJobTemplate {

/**
* Initializes and executes the security access job. This method is
* called by the job execution engine of an MCD-3 server in case one
* of the methods MCDSecurityAccessJob.executeSync(),
* MCDSecurityAccessJob.executeAsynch(), and
* MCDSecurityAccessJob.startRepetition() is called by a client application.
*
* @param inputParameters The input parameters of the job.
* @param mcdJobApi The MCDJobApi object required for communication
*                  between the job and MCD-3 server.
* @param mcddLogicalLink The logical link the job has been started on.
* @param mcdSecurityAccessJob The MCDSecurityAccessJob a job class is referenced from.
* @throws an MCDException in case an error occurs during execution
*/
public void execute (
MCDRequestParameters inputParameters,
MCDJobApi mcdJobApi,
MCDDLogicalLink mcddLogicalLink,
MCDSecurityAccessJob mcdSecurityAccessJob) throws MCDException;
}

/**
* Notifies the job that it should cancel its execution. This method is called
* by the job execution engine of an MCD-3 server in case the method
* MCDDiagComPrimitive.cancel() is called by a client application or the
* D-server.
*/
public void cancel ();
```

Annex D
(informative)

Gateway Handling

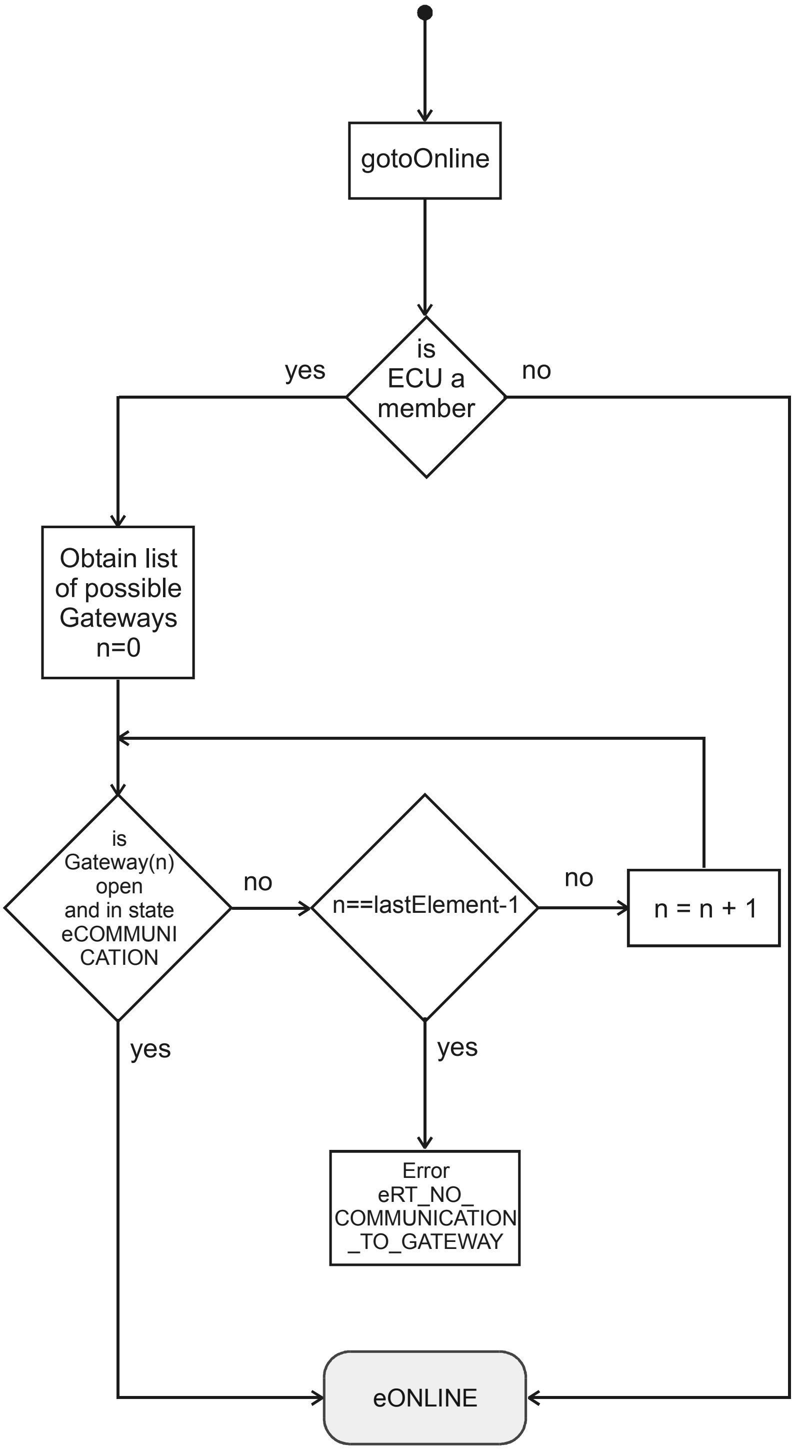


Figure D.1 — GatewayHandling

Annex E
(normative)

Value reading and setting by string

Formats of input and output strings of `MCDValue.get/setValueAsString`.

Table E.1 — MCDValue Data type conversion

MCDValue Data type	Unicode2 String
eA_ASCIISTRING	position by position, character by character
eA_BITFIELD	position by position, bit by bit (other chars than 0 and 1 are ignored)
eA_BYTEFIELD	position by position (excluding non-hex characters, including A-Fa-f)
eA_FLOAT64	normalized form: 0.0123 = 0.123 E-1 (cf. e.g. IEEE 754-2008)
eA_FLOAT32	
eA_INT8	position by position, digit by digit, with leading sign if negative
eA_INT16	
eA_INT32	
eA_INT64	
eA_UNICODE2STRING	nothing to do
eA_UINT8	position by position, digit by digit
eA_UINT16	
eA_UINT32	
eA_UINT64	

The ODX Datatype A_UTF8STRING will be presented as an character string of Unicode 2 characters (data type A_UNICODE2STRING). It shall be differed between the input- and output parameter:

Table E.2 — eA_ASCIISTRING

Input Format Set Method	Output Format Get Method
ASCII text in Unicode	ASCII text in Unicode

Table E.3 — eA_UNICODE2STRING

Input Format Set Method	Output Format Get Method
as is	as is

Table E.4 — eA_BOOLEAN

Input Format Set Method	Output Format Get Method
Accept “false” and “true” in any cases	false is "false" and true is " true"

Table E.5 — eA_BITFIELD

Input Format Set Method	Output Format Get Method
• ‚0‘s and ‚1‘s with non or one blank between • RegEx: ([01]\s?)*[01] • e.g.: „0101“	• ‚0‘s and ‚1‘s without a blank between • RegEx: [01]*[01] • e.g.: „0101“

Table E.6 — eA_BYTEFIELD

Input Format Set Method	Output Format Get Method
• Bytes as pair of hex-chars (‚A‘-‚F‘) • one blank between each pair • case of hex-chars regardless • RegEx: ([0-9a-fA-F]{2}\s)*[0-9a-fA-F]{2} • e.g.: „0a ff C3“	• As at input, but exactly one blank between a pair • case of alpha chars is big • • RegEx: ([0-9A-F]{2}\s)*[0-9A-F]{2} • e.g.: „0A FF C3“

Table E.7 — eA_FLOAT32

Input Format Set Method	Output Format Get Method
• Mantissa and exponent decimal • The floating point is the point ‘.’ • Floating point not forced • Negative mantissa or exponent marked by a minus ‘-’ in front • Exponent introduced by ‘e’ or ‘E’ • No blanks anywhere allowed • Exponent optional if = 0 • Max. count of decimal places for : - Mantissa : 7 - Exponent: 2 • e.g.: „2.0E-2“ or „-2.0“ or „3467208e2“	• Mantissa and exponent decimal • The floating point is the point ‘.’ • Negative mantissa or exponent marked by a minus ‘-’ in front • Exponent introduced by ‘E’ • No blanks • Number normalized (one digit before the floating point), exponent is adapted • Max. count of decimal places for : - Mantissa : 7 (one before, 6 after point) - Exponent: 2 • e.g.: „2.0E-2“ or „-2.0E2“ or „3.467208E8“

Table E.8 — eA_FLOAT64

Input Format Set Method	Output Format Get Method
• Mantissa and exponent decimal • The floating point is the point ‘.’ • Floating point not forced • Negative mantissa or exponent marked by a minus ‘-’ in front • Exponent introduced by ‘e’ or ‘E’ • No blanks anywhere allowed • Exponent optional if = 0 • Max. count of decimal places for : - Mantissa : 16 - Exponent: 3 • e.g.: „2.0E-2“ or „-2.0“ or „3467208e2“	• Mantissa and exponent decimal • The floating point is the point ‘.’ • Negative mantissa or exponent marked by a minus ‘-’ in front • Exponent introduced by ‘E’ • No blanks • Number normalized (one digit before the floating point) • Max. count of decimal places for : - Mantissa : 16 (one before, 15 after point) - Exponent: 3 • e.g.: „2.0E-2“ or „-2.0E2“ or „3.467208E8“

Table E.9 — eA_UINT8 /16 /32 /64

Input Format Set Method	Output Format Get Method
Number • in decimal Digits (0-9) or • case insensitive hex chars (0-9A-Fa-f) • size depending of the range of the type • e.g.: „22“, or “FFFFA”, or “3f7a”	Number • in decimal Digits (0-9) • size depending of the range of the type • e.g.: „22“

Table E.10 — eA_INT8 /16 /32 /64

Input Format Set Method	Output Format Get Method
Number • in decimal Digits (0-9) or • case insensitive hex chars (0-9A-Fa-f) • size depending of the range of the type • Minus char ‘-’ in front, if negaitve; blankbetween minus and number allowed • e.g.: „-22“, or “FFFFA”, or “3f7a”	Number • in decimal Digits (0-9) • size depending of the range of the type • Minus char ‘-’ in front, if negaitve; no blank between minus and number • • • e.g.: „22“

Annex F
(normative)

Bus types

Table F.1 — D - BUS TYPE LIST

Bus Type Name	**Description**
CAN_HighSpeed_ISO_11898_2	Dual-wire high-speed CAN in accordance with ISO 11898-2. **Bustype parameter** TerminationType • NOTERM No termination provided by VCI. • ISO 15765-4 VCI provides an AC termination (100Ohm resistor and 560pF capacitor in series) between CAN HI and GND as well as CAN LO and GND. • ISO 11898-2_R120 VCI provides 120Ohm CAN termination between CAN HI and GND. ECU setup tester is connected to has to provide the other 120 Ohm CAN termination. • ISO 11898-2_R60 VCI provides 60Ohm CAN termination between CAN HI and GND. ECU setup tester is connected to does not have to provide the any termination. Baudrate, SamplingPoint, Sync Jump Width These parameters are covered as protocol parameters.
CAN_LowSpeed_ISO_11898_3	Dual-wire low-speed CAN in accordance with ISO 11898-3. **Bustype parameter** TerminationType • NOTERM No termination provided by VCI. Baudrate, SamplingPoint, Sync Jump Width These parameters are covered as protocol parameters.
CAN_TruckTrailor_ISO_11992	Dual-wire low-speed CAN in accordance with ISO 11992-1. **Bustype parameters** *Not available* *
SAE_J2411	Single Wire CAN Question: Does this cover GMLAN too ? *
KLine_ISO_9141_2	12V based K-Line in accordance with ISO 9141-2. **Bustype parameters** *Not available* *
KLine_ISO_14230_1	12/24V based K Line in accordance with ISO 14230-1. **Bustype parameters** *Not available* *

Table F.1 (*continued*)

Bus Type Name	**Description**
SAE_J1850_VPW	GM Class 2 (variable pulse width) **Bustype parameters** *Not available **
SAE_J1850_PWM	Ford SCP (pulse width modulation) **Bustype parameters** *Not available **
SCI_SAE_J2610	Daimler Chrysler SCI **Bustype parameters** *Not available **
SAE_J1939	*Not available **
SAE_J1708	*Not available **
GMW_3089	*Not available * (maybe:* GM protocol*)*
SDL	*Not available **
XDE_5024	*Not available **
CCD	*Not available * (maybe:* Chrysler protocol *)*
SVSL	*Not available * (maybe:* Mitsubishi protocol*)*
SWS	*Not available * (maybe:* Mitsubishi protocol*)*
DMC	*Not available * (maybe:* Mitsubishi protocol*)*
LIN	*Not available **
MOST	*Not available **
FlexRay	*Not available **

* the “Not available” parts in this table should be filled by the PDU API Core Team in near future.

Further definitions:

- BUSTYPE names may only use name space of shortnames in ODX (a-z, A-Z, 0-9, _)
- Max. Length 128 Bytes

Annex G
(normative)

System parameter

G.1 Overview

The base of all the time dependent system parameter is UTC. This avoids problems with leap seconds.

Represented time informations are based on ISO 8601.

Table G.1 — System parameter

ASCIISTRING for system parameter	data type (physical)	coding	example
TIMEZONE	A_INT32	Count of minutes to UTC	+60 (Berlin)
YEAR	A_UINT32	YYYY	2004
MONTH	A_UINT32	MM	03 (march)
DAY	A_UINT32	DD	01 (first day of month)
HOUR	A_UINT32	hh	22 (10 pm)
MINUTE	A_UINT32	mm	00 (full hour)
SECOND	A_UINT32	ss	00 (full minute)
TESTERID	A_BYTEFIELD		"00F056"
USERID	A_BYTEFIELD		"043FF0"
CENTURY	A_UINT32	CC	20 (in year 2004)
WEEK	A_UINT32		04 (Fourth week of the year)

This table gives a short overview about different system parameters. The actual valid system parameters you will find inside the ODX specification[11].

G.2 Description of the system parameters

G.2.1 TIMEZONE

The timezone is coded in an A_INT32 value. The range is between –720 and +780.

Table G.2 — Examples for timezones

Minutes	UTC time offset	Related Towns
-720	UTC – 12h	Eniwetok, Kwajalein
-660	UTC – 11h	Midway islands, Samoa
-600	UTC – 10h	Hawaii
…		
-480	UTC – 8h	Los Angeles, Seattle, Vancouver
…		
-300	UTC – 5h	New York, Atlanta, Detroit, Toronto
…		
-60	UTC – 1h	Azores
0	UTC	London
+60	UTC + 1h	Berlin, Rome
+120	UTC + 2h	Athens, Istanbul
+180	UTC + 3h	Moscow, Nairobi
+210	UTC + 3,5h	Teheran
+240	UTC + 4h	Abu Dhabi
…		
+720	UTC + 12h	Aukland, Wellington
+780	UTC + 13h	Nuku'alofa

G.2.2 YEAR

The year is coded in an A_UINT32 value with four digits.

G.2.3 MONTH

The year is coded in an A_UINT32 value with two digits. It is starting with January as 01 up to December with 12.

G.2.4 DAY

The day is coded in an A_UINT32 value with two digits. It is starting with the first day of the month as 01 up to the last day of the month.

G.2.5 HOUR

The hour is coded in an A_UINT32 value with two digits in a 24 hours rhythm. It is starting with 12am as 00 (via 1am as 01) up to 11pm as 23.

G.2.6 MINUTE

The minute is coded in an A_UINT32 value with two digits. It is starting with 00 (full hour) up to 59.

G.2.7 SECOND

The second is coded in an A_UINT32 value with two digits. It is starting with 00 (full minute) up to 59.

G.2.8 TESTERID

The tester ID is coded in an A_BYTEFIELD.

G.2.9 USERID

The user ID is coded in an A_BYTEFIELD.

G.2.10 CENTURY

The century is coded in an A_UINT32 value with two digits. This value contains the first two digits of the four-digit year (unlike language usage).

G.2.11 WEEK

The century is coded in an A_UINT32 value with two digits.

The first week of a year is the first week which includes at least four days of the new year. Alternatively, the first week of a year is the week which includes the first Thursday of January and January 4. As a result, week 01 of a year can contain days of the previous year and week 53 can contain days of the following year. For example, 2004-01-01 is a Thursday. Hence, 2004-W01 does comprise the days 2003-12-29 to 2004-01-04. Furthermore, 2005-01-01 is a Saturday. As a result, 2004-W53 comprises the days 2004-12-27 to 2005-01-02. Week 2005-W01 starts on 2005-01-03.

Annex H
(normative)

Cooperation level

If an MCD object is considered a shared object, it life-cycle is bound to the MCD-server. In this case, the MCD-server decides whether a client can successfully call a method at a shared object. If an object has been passed to a client as a client-controlled object, the client can always successfully call all methods at this object – provided the access is not restricted by any rules and regulations. For client-controlled objects, a cooperation level does not have any affect.

With respect to collection derived from class MCDCollection, two different kinds of subclasses need to be distinguished – shared collections and client-controlled collections. In the first case, access to methods in the collection is restricted by the cooperation level of the object which contains the collection. In the latter case, the collection is owned by the client only. Therefore, no cooperation level restricts the access to the methods of this collection.

The table describing which methods can be called in which cooperation level is to be read and interpreted as follows: What is allowed by a second client at a object which was created by another client. For abstract classes in the inheritance hierarchy, that is, for classes which cannot be instantiated at runtime, the table only applies if a corresponding subclass is a leaf of the inheritance hierarchy and if this class is a shared object.

Table H.1 — Base

Class	Stereo Type	Name	Creation cooperation level				Comment
			No Cooperation	Read Only	Extending Modification	Full Access	
MCDCollection							
	<<M,C,D>>	getCount	Not Applicable	Allowed	Allowed	Allowed	only for shared objects derived from MCDCollection
MCDLogicalLinks		**Shared object without cooperation level**					
	<<D>>	addByAccessKeyAndInterfaceResource	Not Applicable	Not Applicable	Not Applicable	Not Applicable	
	<<D>>	addByAccessKeyAndVehicleLink	Not Applicable	Not Applicable	Not Applicable	Not Applicable	
	<<D>>	addByDbObject	Not Applicable	Not Applicable	Not Applicable	Not Applicable	
	<<D>>	addByName	Not Applicable	Not Applicable	Not Applicable	Not Applicable	
	<<M,C>>	addByNames	Not Applicable	Not Applicable	Not Applicable	Not Applicable	
	<<M,C>>	addByObjects	Not Applicable	Not Applicable	Not Applicable	Not Applicable	
	<<D>>	addByVariant	Not Applicable	Not Applicable	Not Applicable	Not Applicable	

Table H.1 (*continued*)

Class	Stereo Type	Name	Creation cooperation level				Comment
			No Cooperation	Read Only	Extending Modification	Full Access	
	<<M,C,D>>	getItemByIndex	Not Applicable	Not Applicable	Not Applicable	Not Applicable	
	<<M,C,D>>	getItemByName	Not Applicable	Not Applicable	Not Applicable	Not Applicable	
	<<D>>	remove	Not Applicable	Not Applicable	Not Applicable	Not Applicable	The call to this method is allowed for all clients, but the method will fail if the client is not the master of the logical link to be removed or has not enough rights to remove the logical link.
	<<M,C,D>>	removeAll	Not Applicable	Not Applicable	Not Applicable	Not Applicable	The call to this method is allowed for all clients, but the method will fail if the client is not the master of the/all logical link(s) or has not enough rights to remove the logical link(s).
	<<M,C,D>>	removeByIndex	Not Applicable	Not Applicable	Not Applicable	Not Applicable	The call to this method is allowed for all clients, but the method will fail if the client is not the master of the logical link to be removed or has not enough rights to remove the logical link.
	<<M,C,D>>	removeByName	Not Applicable	Not Applicable	Not Applicable	Not Applicable	The call to this method is allowed for all clients, but the method will fail if the client is not the master of the logical link to be removed or has not enough rights to remove the logical link.
MCDNamed Collection							
	<<M,C,D>>	getNames	Not Applicable	Allowed	Allowed	Allowed	only for shared objects derived from MCDCollection
MCDNamed Object							
	<<M,C,D>>	getDescription	Allowed	Allowed	Allowed	Allowed	basic information required for listing at GUI

Table H.1 (*continued*)

Class	Stereo Type	Name	Creation cooperation level				Comment
			No Cooperation	**Read Only**	**Extending Modification**	**Full Access**	
	<<M,C,D>>	getLongName	Allowed	Allowed	Allowed	Allowed	basic information required for listing at GUI
	<<M,C,D>>	getShortName	Allowed	Allowed	Allowed	Allowed	basic information required for listing at GUI
MCDObject							
	<<M,C,D>>	getObjectType	Allowed	Allowed	Allowed	Allowed	basic information required for object identification
	<<M,C>>	getParent	Not Allowed	Allowed	Allowed	Allowed	

Table H.2 — Logical Link

Class	Stereo Type	Name	Creation cooperation level				Comment
			No Cooperation	**Read Only**	**Extending Modification**	**Full Access**	
MCDAccessKeys							
	<<D>>	getItemByIndex	Not Allowed	Allowed	Allowed	Allowed	
MCDDLogicalLink							
	<<D>>	clearQueue	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<D>>	close	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<D>>	disableReducedResults	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<D>>	enableReducedResults	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<D>>	getActivityState	Allowed	Allowed	Allowed	Allowed	
	<<D>>	getConfigurationRecords	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getDefinableDynIds	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getDiagComPrimitives	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getFlashSessionDescs	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getIdentifiedVariantAccessKeys	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getInterfaceResource	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getMatchedDbEcuVariantPattern	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getMatchingInterfaceResources	Not Allowed	Allowed	Allowed	Allowed	

Table H.2 (*continued*)

Class	Stereo Type	Name	Creation cooperation level				
			No Cooperation	**Read Only**	**Extending Modification**	**Full Access**	**Comment**
	<<D>>	getQueueFillingLevel	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getQueueSize	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getSelectedVariantAccessKeys	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getUnitGroup	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	gotoOffline	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<D>>	gotoOnline	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<D>>	isIntermediateResultForFunctional AddressingEnabled	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	isUnsupportedCom ParametersAccepted	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	open	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<D>>	reset	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<D>>	resume	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<D>>	selectInterfaceResource	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<D>>	sendBreak	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<D>>	setIntermediateResultForFunctional Addressing	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<D>>	setQueueSize	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<D>>	setUnitGroupByName	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<D>>	suspend	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<D>>	unsupportedComParametersAccepted	Not Allowed	Not Allowed	Not Allowed	Allowed	
MCDLogicalLink							
	<<M,C,D>>	configureEventHandler	Not Allowed	Allowed	Allowed	Allowed	
	<<M,C,D>>	getCooperationLevel	Allowed	Allowed	Allowed	Allowed	
	<<M,C,D>>	getDbObject	Allowed	Allowed	Allowed	Allowed	
	<<M,C,D>>	getLockState	Allowed	Allowed	Allowed	Allowed	
	<<M,C,D>>	getState	Allowed	Allowed	Allowed	Allowed	
	<<M,C,D>>	getType	Not Allowed	Allowed	Allowed	Allowed	
	<<M,C,D>>	isOwned	Allowed	Allowed	Allowed	Allowed	
	<<M,C,D>>	lock	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<M,C,D>>	releaseEventHandler	Not Allowed	Allowed	Allowed	Allowed	
	<<M,C,D>>	setEventHandler	Not Allowed	Allowed	Allowed	Allowed	

Table H.2 (*continued*)

			Creation cooperation level				
Class	**Stereo Type**	**Name**	**No Cooperation**	**Read Only**	**Extending Modification**	**Full Access**	**Comment**
	<<M,C,D>>	unlock	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<M,C,D>>	unlockTree	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<M,C,D>>	isModifiedByOtherClient	Allowed	Allowed	Allowed	Allowed	Signals that the status of the MCDLogicalLink has been modified by another client (e.g. MemoryPage, QueueSize). This modification marker includes changes to collections directly contained in an MCDLogicalLink, i.e. MCDDiagComPrimitives, MCDConfigurationRecords, MCDFlashSessionDescs, MCDCollectors, MCDCharacteristics. Modifications to objects inside such a collection are not signalled by this method.
MCDMCLogical Link							
	<<M,C>>	connectToModule	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<M,C>>	disconnectFromModule	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<C>>	getCharacteristics	Not Allowed	Allowed	Allowed	Allowed	
	<<M>>	getCollectors	Not Allowed	Allowed	Allowed	Allowed	
	<<C>>	getMemoryPage	Not Allowed	Allowed	Allowed	Allowed	
	<<M,C>>	getRateInfos	Not Allowed	Allowed	Allowed	Allowed	
	<<M,C>>	GetTargetStae	Not Allowed	Allowed	Allowed	Allowed	
	<<M,C>>	getUsedBinary	Not Allowed	Allowed	Allowed	Allowed	
	<<M,C>>	saveBinaryToFile	Not Allowed	Not Allowed	Allowed	Allowed	
	<<C>>	setMemoryPage	Not Allowed	Not Allowed	Not Allowed	Allowed	
MCDRateInfo							
	<<M,C>>	getCode	Not Applicable	Allowed	Allowed	Allowed	
	<<M,C>>	getValue	Not Applicable	Allowed	Allowed	Allowed	

Table H.2 (*continued*)

Class	Stereo Type	Name	Creation cooperation level				
			No Cooperation	**Read Only**	**Extending Modification**	**Full Access**	**Comment**
MCDRateInfos							
	<<M,C>>	getItemByIndex	Not Allowed	Allowed	Allowed	Allowed	
	<<M,C>>	getItemByName	Not Allowed	Allowed	Allowed	Allowed	

Table H.3 — Characteristic

Class	Stereo Type	Name	Creation cooperation level				
			No Cooperation	**Read Only**	**Extending Modification**	**Full Access**	**Comment**
MCDAscii Characteristic							
	<<C>>	read	Not allowed	Allowed	Allowed	Allowed	
	<<C>>	write	Not allowed	Not allowed	Not allowed	Allowed	
MCD Characteristic							
	<<C>>	getCooperationLevel	Allowed	Allowed	Allowed	Allowed	
	<<C>>	getDbObject	Allowed	Allowed	Allowed	Allowed	
	<<C>>	getLockState	Allowed	Allowed	Allowed	Allowed	
	<<C>>	isOwned	Allowed	Allowed	Allowed	Allowed	
	<<C>>	lock	Not allowed	Not allowed	Not allowed	Allowed	
	<<C>>	unlock	Not allowed	Not allowed	Not allowed	Allowed	
	<<C>>	unlockTree	Not allowed	Not allowed	Not allowed	Allowed	
MCD Characteristics							
	<<C>>	add	Not Applicable	Not allowed	Allowed	Allowed	
	<<C>>	getItemByIndex	Not Allowed	Allowed	Allowed	Allowed	
	<<C>>	getItemByName	Not Allowed	Allowed	Allowed	Allowed	
	<<C>>	removeAll	Not Applicable	Not allowed	Not allowed	Allowed	
	<<C>>	removeByIndex	Not Applicable	Not allowed	Allow own	Allowed	
	<<C>>	removeByName	Not Applicable	Not allowed	Allow own	Allowed	

Table H.3 (*continued*)

			Creation cooperation level				
Class	**Stereo Type**	**Name**	**No Cooperation**	**Read Only**	**Extending Modification**	**Full Access**	**Comment**
MCDCurve Characteristic							
	<<C>>	getAxis	Not allowed	Allowed	Allowed	Allowed	
	<<C>>	getValue	Not allowed	Allowed	Allowed	Allowed	
	<<C>>	read	Not allowed	Allowed	Allowed	Allowed	
	<<C>>	write	Not allowed	Not allowed	Not allowed	Allowed	
MCDMap Characteristic							
	<<C>>	getValue	Not allowed	Allowed	Allowed	Allowed	
	<<C>>	getXAxis	Not allowed	Allowed	Allowed	Allowed	
	<<C>>	getYAxis	Not allowed	Allowed	Allowed	Allowed	
	<<C>>	read	Not allowed	Allowed	Allowed	Allowed	
	<<C>>	write	Not allowed	Not allowed	Not allowed	Allowed	
MCDMatrix Characteristic							The cooperation level entries for sub objects of characteristics is not applicable since you cannot retrieve these objects if the parent state is “No cooperation”
	<<C>>	read	Not applicable	Allowed	Allowed	Allowed	
	<<C>>	write	Not applicable	Not allowed	Not allowed	Allowed	
MCDScalar Characteristic							
	<<C>>	read	Not allowed	Allowed	Allowed	Allowed	
	<<C>>	write	Not allowed	Not allowed	Not allowed	Allowed	
MCDValueBlock Characteristic							
	<<C>>	read	Not allowed	Allowed	Allowed	Allowed	
	<<C>>	write	Not allowed	Not allowed	Not allowed	Allowed	
MCDVector Characteristic							The cooperation level entries for sub objects of characteristics is not applicable since you cannot retrieve these objects if the parent state is “No cooperation”

Table H.3 (*continued*)

Class	Stereo Type	Name	Creation cooperation level				Comment
			No Cooperation	Read Only	Extending Modification	Full Access	
	<<C>>	read	Not applicable	Allowed	Allowed	Allowed	
	<<C>>	write	Not applicable	Not allowed	Not allowed	Allowed	
	<<C>>	getSize	Not applicable	Allowed	Allowed	Allowed	

Table H.4 — Collector

Class	Stereo Type	Name	Creation cooperation level				Comment
			No Cooperation	Read Only	Extending Modification	Full Access	
MCDBuffer							
	<<M>>	getDownSampling	Not Applicable	Allowed	Allowed	Allowed	
	<<M>>	getFillingLevel	Not Applicable	Allowed	Allowed	Allowed	
	<<M>>	getRate	Not Applicable	Allowed	Allowed	Allowed	
	<<M>>	getSize	Not Applicable	Allowed	Allowed	Allowed	
	<<M>>	getTimeStamping	Not Applicable	Allowed	Allowed	Allowed	
	<<M>>	setDownSampling	Not Applicable	Not allowed	Not allowed	Allowed	
	<<M>>	setRate	Not Applicable	Not allowed	Not allowed	Allowed	
	<<M>>	setSize	Not Applicable	Not allowed	Not allowed	Allowed	
	<<M>>	setTimeStamping	Not Applicable	Not allowed	Not allowed	Allowed	
MCDCollected Object							
	<<M>>	getDbObject	Allowed	Allowed	Allowed	Allowed	
	<<M>>	getCollectorDescription	Not Applicable	Allowed	Allowed	Allowed	
MCDCollected Objects							
	<<M>>	addAscii	Not Applicable	Not allowed	Allowed	Allowed	only for own Objects

Table H.4 (*continued*)

Class	Stereo Type	Name	Creation cooperation level				Comment
			No Cooperation	Read Only	Extending Modification	Full Access	
	<<M>>	addCurve	Not Applicable	Not allowed	Allowed	Allowed	
	<<M>>	addMap	Not Applicable	Not allowed	Allowed	Allowed	
	<<M>>	addScalar	Not Applicable	Not allowed	Allowed	Allowed	
	<<M>>	addValueBlock	Not Applicable	Not allowed	Allowed	Allowed	only for own Objects
	<<M>>	removeByIndex	Not Applicable	Not Allowed	Allowed Own	Allowed	only for own Objects
	<<M>>	removeAll	Not Applicable	Not allowed	Not allowed	Allowed	
	<<M>>	removeByName	Not Applicable	Not allowed	Allow Own	Allowed	only for own Objects
MCDCollector							
	<<M>>	configureEventHandler	Not allowed	Allowed	Allowed	Allowed	
	<<M>>	createBufferIDForPolling	Not allowed	Allowed	Allowed	Allowed	
	<<M>>	fetchResults	Not allowed	Allowed	Allowed	Allowed	
	<<M>>	getBuffer	Not allowed	Allowed	Allowed	Allowed	
	<<M>>	getCollectedObjects	Not allowed	Allowed	Allowed	Allowed	
	<<M >>	getCooperation Level	Allowed	Allowed	Allowed	Allowed	
	<<M>>	getFillingLevel	Not allowed	Allowed	Allowed	Allowed	
	<<M>>	getLockState	Allowed	Allowed	Allowed	Allowed	
	<<M>>	getNoOfSamplesToFireEvent	Allowed	Allowed	Allowed	Allowed	
	<<M>>	isModifiedByOtherClient	Allowed	Allowed	Allowed	Allowed	
	<<M >>	isOwned	Allowed	Allowed	Allowed	Allowed	
	<<M>>	lock	Not allowed	Not allowed	Not allowed	Allowed	
	<<M>>	releaseEventHandler	Not allowed	Allowed	Allowed	Allowed	
	<<M>>	removeBufferIDForPolling	Not allowed	Allowed	Allowed	Allowed	
	<<M>>	setEventHandler	Not allowed	Allowed	Allowed	Allowed	
	<<M>>	setNoOfSamplesToFireEvent	Not allowed	Not allowed	Not allowed	Allowed	
	<<M>>	unlock	Not allowed	Not allowed	Not allowed	Allowed	

Table H.4 (*continued*)

Class	Stereo Type	Name	Creation cooperation level				Comment
			No Cooperation	**Read Only**	**Extending Modification**	**Full Access**	
MCDCollector Description							
	<<M>>	getRepresentationType	Not Applicable	Allowed	Allowed	Allowed	
MCDCollectors							
	<<M>>	add	Not Applicable	Not allowed	Allowed	Allowed	
	<<M>>	deactivate	Not Applicable	Not allowed	Not allowed	Allowed	
	<<M>>	getCollectorID	Not Applicable	Allowed	Allowed	Allowed	
	<<M>>	getItemByIndex	Not Allowed	Allowed	Allowed	Allowed	
	<<M>>	removeAll	Not Applicable	Not allowed	Not allowed	Allowed	
	<<M>>	removeByIndex	Not Applicable	Not allowed	Allow Own	Allowed	
MCDCurve Collector Description							
	<<M>>	getDescriptionType	Not Applicable	Allowed	Allowed	Allowed	
	<<M>>	getStart	Not Applicable	Allowed	Allowed	Allowed	
	<<M>>	getStop	Not Applicable	Allowed	Allowed	Allowed	
MCDMapCollector Description							
	<<M>>	getDescriptionType	Not Applicable	Allowed	Allowed	Allowed	
	<<M>>	getXStart	Not Applicable	Allowed	Allowed	Allowed	
	<<M>>	getXStop	Not Applicable	Allowed	Allowed	Allowed	
	<<M>>	getYStart	Not Applicable	Allowed	Allowed	Allowed	
	<<M>>	getYStop	Not Applicable	Allowed	Allowed	Allowed	
MCDTimeDelay							
	<<M >>	configure	Not Applicable	Not allowed	Not allowed	Allowed	

Table H.4 (*continued*)

Class	Stereo Type	Name	Creation cooperation level				Comment
			No Cooperation	Read Only	Extending Modification	Full Access	
	<<M >>	getResolution	Not Applicable	Allowed	Allowed	Allowed	
	<<M >>	getValue	Not Applicable	Allowed	Allowed	Allowed	
MCDScalar Collector Description							
	<<M>>	getDescriptionType	Not Applicable	Allowed	Allowed	Allowed	
MCDValueBlock Collector Description	<<M>>	getXStart	Not Applicable	Allowed	Allowed	Allowed	
	<<M>>	getXStop	Not Applicable	Allowed	Allowed	Allowed	
	<<M>>	getYStart	Not Applicable	Allowed	Allowed	Allowed	
	<<M>>	getYStop	Not Applicable	Allowed	Allowed	Allowed	

Table H.5 — Global

Class	Stereo Type	Name	Creation cooperation level				Comment
			No Cooperation	Read Only	Extending Modification	Full Access	
MCDGlobal							
	<<M>>	activate	Not allowed	Not allowed	Not allowed	Allowed	
	<<M>>	change	Not allowed	Not allowed	Not allowed	Allowed	
	<<M>>	check	Not allowed	Not allowed	Not allowed	Allowed	
	<<M>>	deactivate	Not allowed	Not allowed	Not allowed	Allowed	
	<<M>>	getStartDelay	Not allowed	Allowed	Allowed	Allowed	
	<<M>>	getStartWatcher	Not allowed	Allowed	Allowed	Allowed	
	<<M>>	getState	Allowed	Allowed	Allowed	Allowed	
	<<M >>	getStopDelay	Not allowed	Allowed	Allowed	Allowed	
	<<M>>	getStopWatcher	Not allowed	Allowed	Allowed	Allowed	
	<<M>>	setStartWatcher	Not allowed	Not allowed	Not allowed	Allowed	
	<<M>>	setStopWatcher	Not allowed	Not allowed	Not allowed	Allowed	
	<<M>>	start	Not allowed	Not allowed	Not allowed	Allowed	
	<<M>>	stop	Not allowed	Not allowed	Not allowed	Allowed	

Table H.6 — Requests

Class	Stereo Type	Name	Creation cooperation level				
			No Cooperation	**Read Only**	**Extending Modification**	**Full Access**	**Comment**
MCDParameter							
	<<M,C,D>>	getDataType	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	getDbDTC	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	getDbObject	Allowed	Allowed	Allowed	Allowed	
	<<M,C,D>>	getDbUnit	Not Applicable	Allowed	Allowed	Allowed	
	<<M,C,D>>	getDecimalPlaces	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	getParameterType	Not Applicable	Allowed	Allowed	Allowed	
	<<M,C,D>>	getRadix	Not Applicable	Allowed	Allowed	Allowed	
	<<M,C,D>>	getScaleConstraint	Not Applicable	Allowed	Allowed	Allowed	
	<<M,C,D>>	getSystemParameterName	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	getTextTableElement	Not Applicable	Allowed	Allowed	Allowed	
	<<M,C,D>>	getValue	Not Applicable	Allowed	Allowed	Allowed	
	<<M,C,D>>	getValueRangeInfo	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	isComplex	Not Applicable	Allowed	Allowed	Allowed	
MCDRequest							
	<<D>>	getDbObject	Allowed	Allowed	Allowed	Allowed	
	<<D>>	getPDU	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	getRequestParameters	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	hasPDU	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	setPDU	Not Applicable	Not allowed	Not allowed	Allowed	
MCDRequestParameter							
	<<D>>	addParameters	Not Applicable	Not allowed	Not allowed	Allowed	

Table H.6 (*continued*)

			Creation cooperation level				
Class	**Stereo Type**	**Name**	**No Cooperation**	**Read Only**	**Extending Modification**	**Full Access**	**Comment**
	<<D>>	getCodedValue	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	getLengthKey	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	getParameters	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	isVariableLength	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	setCodedValue	Not Applicable	Not allowed	Not allowed	Allowed	
	<<D>>	setValue	Not Applicable	Not allowed	Not allowed	Allowed	
	<<D>>	setValueUnchecked	Not Applicable	Not allowed	Not allowed	Allowed	
MCDRequestParameters							
	<<D>>	getItemByIndex	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getItemByName	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	setParameterByName	Not Applicable	Not allowed	Not allowed	Allowed	

Table H.7 — Primitives

			Creation cooperation level				
Class	**Stereo Type**	**Name**	**No Cooperation**	**Read Only**	**Extending Modification**	**Full Access**	**Comment**
MCDControl Primitive							
	<<D>>	getProtocolParameters	Not Applicable	Allowed	Allowed	Allowed	
MCDDiagCom Primitive							
	<<D>>	cancel	Not Applicable	Not allowed	Not allowed	Allowed	
	<<D>>	configureEventHandler	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	executeSync	Not Applicable	Not allowed	Not allowed	Allowed	
	<<D>>	getCooperationLevel	Allowed	Allowed	Allowed	Allowed	
	<<D>>	getDbObject	Allowed	Allowed	Allowed	Allowed	

Table H.7 (*continued*)

Class	Stereo Type	Name	Creation cooperation level				Comment
			No Cooperation	Read Only	Extending Modification	Full Access	
	<<D>>	getErrors	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	getLockState	Allowed	Allowed	Allowed	Allowed	
	<<D>>	getRequest	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	getState	Allowed	Allowed	Allowed	Allowed	
	<<D>>	isModifiedByOtherClient	Allowed	Allowed	Allowed	Allowed	
	<<D>>	isOwned	Allowed	Allowed	Allowed	Allowed	
	<<D>>	lock	Not allowed	Not allowed	Not allowed	Allowed	
	<<D>>	releaseEventHandler	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	resetToDefaultValues	Not Applicable	Not allowed	Not allowed	Allowed	
	<<D>>	setEventHandler	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	unlock	Not allowed	Not allowed	Not allowed	Allowed	
MCDDiagCom Primitives							
	<<D>>	addByDbObject	Not Applicable	Not Allowed	Allowed	Allowed	
	<<D>>	addByName	Not Applicable	Not Allowed	Allowed	Allowed	
	<<D>>	addBySemanticAttribute	Not Applicable	Not Allowed	Allowed	Allowed	
	<<D>>	addByType	Not Applicable	Not Allowed	Allowed	Allowed	
	<<D>>	addDiagVariableServiceBy RelationType	Not Applicable	Not Allowed	Allowed	Allowed	
	<<D>>	addDynIdComPrimitiveByT ypeAnd DefinitionMode	Not Applicable	Not Allowed	Allowed	Allowed	
	<<D>>	configureEventHandler	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getItemByIndex	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getItemByName	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	releaseEventHandler	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	remove	Not Applicable	Not Allowed	Allowed	Allowed	The call to this method is allowed for all clients, but the method will fail if the client is not the master of the DiagComPrimitive to be removed or has not enough rights to remove the DiagComPrimitive.

Table H.7 (*continued*)

Class	Stereo Type	Name	Creation cooperation level				Comment
			No Cooperation	**Read Only**	**Extending Modification**	**Full Access**	
	<<D>>	removeAll	Not Applicable	Not Allowed	Allowed	Allowed	The call to this method is allowed for all clients, but the method will fail if the client is not the master of the DiagComPrimitive(s) to be removed or has not enough rights to remove the DiagComPrimitive(s).
	<<D>>	removeByIndex	Not Applicable	Not Allowed	Allowed	Allowed	The call to this method is allowed for all clients, but the method will fail if the client is not the master of the DiagComPrimitive to be removed or has not enough rights to remove the DiagComPrimitive.
	<<D>>	removeByName	Not Applicable	Not Allowed	Allowed	Allowed	The call to this method is allowed for all clients, but the method will fail if the client is not the master of the DiagComPrimitive to be removed or has not enough rights to remove the DiagComPrimitive.
	<<D>>	setPrimitiveEventHandler	Not Allowed	Allowed	Allowed	Allowed	
MCDDiagService							
	<<D>>	getProtocolParameters	Not Applicable	Allowed	Allowed	Allowed	
MCDDataPrimitive							
	<<D>>	executeAsync	Not Applicable	Not allowed	Not allowed	Allowed	
	<<D>>	fetchResults	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	getFilter	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	getNewAsciiStrings	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	getNumberOfResults	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	getRepetitionTime	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	getResultBufferSize	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	isFilterModeSet	Not Applicable	Allowed	Allowed	Allowed	

Table H.7 (*continued*)

Class	Stereo Type	Name	Creation cooperation level				Comment
			No Cooperation	Read Only	Extending Modification	Full Access	
	<<D>>	removeFilter	Not Applicable	Not allowed	Not allowed	Allowed	
	<<D>>	setFilter	Not Applicable	Not allowed	Not allowed	Allowed	
	<<D>>	setRepetitionTime	Not Applicable	Not allowed	Not allowed	Allowed	
	<<D>>	setResultBufferSize	Not Applicable	Allowed	Allowed	Allowed	Server tracks result buffer requests for every client separately (see meeting minutes 2005-06-15, Visu-IT!, Regensburg)
	<<D>>	startRepetition	Not Applicable	Not allowed	Not allowed	Allowed	
	<<D>>	stopRepetition	Not Applicable	Not allowed	Not allowed	Allowed	
	<<D>>	updateRepetitionParameters	Not Applicable	Not allowed	Not allowed	Allowed	
MCDDynIdClearComPrimitive							
	<<D>>	getDynId	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	setDynId	Not Applicable	Not allowed	Not allowed	Allowed	
	<<D>>	getDefinitionMode	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	setDefinitionMode	Not Applicable	Not allowed	Not allowed	Allowed	
	<<D>>	getDefinitionModes	Not Applicable	Allowed	Allowed	Allowed	
MCDDynIdDefineComPrimitive							
	<<D>>	getDynId	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	setDynId	Not Applicable	Not allowed	Not allowed	Allowed	
	<<D>>	setDynIdParams	Not Applicable	Not allowed	Not allowed	Allowed	
	<<D>>	getDefinitionMode	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	setDefinitionMode	Not Applicable	Not allowed	Not allowed	Allowed	
	<<D>>	getDefinitionModes	Not Applicable	Allowed	Allowed	Allowed	

Table H.7 (*continued*)

Class	Stereo Type	Name	Creation cooperation level				
			No Cooperation	Read Only	Extending Modification	Full Access	Comment
MCDDynIdRead ComPrimitive							
	<<D>>	getDynId	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	setDynId	Not Applicable	Not allowed	Not allowed	Allowed	
	<<D>>	getDefinitionMode	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	setDefinitionMode	Not Applicable	Not allowed	Not allowed	Allowed	
	<<D>>	getDefinitionModes	Not Applicable	Allowed	Allowed	Allowed	
MCDFlashJob							
	<<D>>	setFlashSessionDesc	Not Applicable	Not Allowed	Not Allowed	Allowed	
	<<D>>	setFlashSessionDescByFlashKey	Not Applicable	Not Allowed	Not Allowed	Allowed	
	<<D>>	setFlashSessionDescByName	Not Applicable	Not Allowed	Not Allowed	Allowed	
MCDJob							
	<<D>>	getJobInfo	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	getProgress	Not Applicable	Allowed	Allowed	Allowed	
MCDProtocol ParameterSet							
	<<D>>	fetchValueFromInterface	Not Applicable	Not allowed	Not allowed	Allowed	
	<<D>>	fetchValuesFromInterface	Not Applicable	Not allowed	Not allowed	Allowed	
MCDStart Communication							
	<<D>>	hasSuppressPositiveResponseCapability	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	isSuppressPositiveResponse	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	setSuppressPositiveResponse	Not Applicable	Not allowed	Not allowed	Allowed	

Table H.7 (*continued*)

Class	Stereo Type	Name	Creation cooperation level				Comment
			No Cooperation	**Read Only**	**Extending Modification**	**Full Access**	
MCDStop Communication							
	<<D>>	hasSuppressPositiveResponseCapability	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	isSuppressPositiveResponse	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	setSuppressPositiveResponse	Not Applicable	Not allowed	Not allowed	Allowed	
MCDService							
	<<D>>	getDefaultResultBufferSize	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	isSuppressPositiveResponse	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	setRuntimeTransmissionMode	Not Applicable	Not allowed	Not allowed	Allowed	
	<<D>>	setSuppressPositiveResponse	Not Applicable	Not allowed	Not allowed	Allowed	
MCDBaseVariant Identificator							
	<<D>>	getDbEcuGroup	Not Applicable	Not allowed	Allowed	Allowed	
	<<D>>	executeBaseVariantIdentification	Not Applicable	Not allowed	Allowed	Allowed	
	<<D>>	executeBaseVariantIdentificationAnd Selection	Not Applicable	Not allowed	Allowed	Allowed	
	<<D>>	setDiagComEventHandler	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	releaseDiagComEventHandler	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	configureDiagComEventHandler	Not Applicable	Allowed	Allowed	Allowed	

Table H.8 — Watcher

			Creation cooperation level				
Class	Stereo Type	Name	No Cooperation	Read Only	Extending Modification	Full Access	Comment
MCDWatcher							
	<<M>>	activate	Not allowed	Not allowed	Not allowed	Allowed	
	<<M>>	change	Not allowed	Not allowed	Not allowed	Allowed	
	<<M>>	check	Not allowed	Not allowed	Not allowed	Allowed	
	<<M>>	configureEventhandler	Not allowed	Allowed	Allowed	Allowed	
	<<M>>	deactivate	Not allowed	Not allowed	Not allowed	Allowed	
	<<M>>	fire	Not allowed	Not allowed	Not allowed	Allowed	
	<<M>>	getActivationWatcher	Not allowed	Allowed	Allowed	Allowed	
	<<M>>	getAutoDeactivate	Not allowed	Not allowed	Not allowed	Allowed	
	<<M>>	getCooperationLevel	Allowed	Allowed	Allowed	Allowed	
	<<M>>	getLockState	Allowed	Allowed	Allowed	Allowed	
	<<M>>	getState	Allowed	Allowed	Allowed	Allowed	
	<<M>>	getTrigger	Not allowed	Allowed	Allowed	Allowed	
	<<M>>	isConnected	Allowed	Allowed	Allowed	Allowed	
	<<M>>	isModifiedByOtherClient	Allowed	Allowed	Allowed	Allowed	
	<<M>>	isOwned	Allowed	Allowed	Allowed	Allowed	
	<<M>>	lock	Not allowed	Not allowed	Not allowed	Allowed	
	<<M>>	releaseEventHandler	Not allowed	Allowed	Allowed	Allowed	
	<<M>>	setActivationWatcher	Not allowed	Not allowed	Not allowed	Allowed	
	<<M>>	setAutoDeactivate	Not allowed	Not allowed	Not allowed	Allowed	
	<<M>>	setEventHandler	Not allowed	Allowed	Allowed	Allowed	
	<<M>>	setTrigger	Not allowed	Not allowed	Not allowed	Allowed	
	<<M>>	unlock	Not allowed	Not allowed	Not allowed	Allowed	
	<<M>>	unlockTree	Not allowed	Not allowed	Not allowed	Allowed	

Table H.9 — Flash Programming

			Creation cooperation level				
Class	Stereo Type	Name	No Cooperation	Read Only	Extending Modification	Full Access	Comment
MCDFlashData Block							
	<<D>>	clearFlashSegments	Not Applicable	Not allowed	Not allowed	Allowed	
	<<D>>	getActiveFileName	Not Applicable	Allowed	Allowed	Allowed	

Table H.9 (*continued*)

Class	Stereo Type	Name	Creation cooperation level				
			No Cooperation	Read Only	Extending Modification	Full Access	Comment
	<<D>>	getDbObject	Allowed	Allowed	Allowed	Allowed	
	<<D>>	getFlashSegments	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	getMatchingFileNames	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	loadFlashSegments	Not Applicable	Not allowed	Not allowed	Allowed	
MCDFlashData Blocks							
	<<D>>	getItemByIndex	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getItemByName	Not Allowed	Allowed	Allowed	Allowed	
MCDFlash Segment							
	<<D>>	createFlashSegmentIterator	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	getBinaryData	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	getCompressedSize	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	getSourceEndAddress	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	getSourceStartAddress	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	getUncompressedSize	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	setBinaryData	Not Applicable	Not Allowed	Not Allowed	Allowed	
	<<D>>	setSourceStartAddress	Not Applicable	Not Allowed	Not Allowed	Allowed	
MCDFlash SegmentIterator							
	<<D>>	getBinaryDataChunkSize	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	getFirstBinaryDataChunk	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	getNextBinaryDataChunk	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	hasNextBinaryDataChunk	Not Applicable	Allowed	Allowed	Allowed	

Table H.9 (*continued*)

Class	Stereo Type	Name	Creation cooperation level				Comment
			No Cooperation	Read Only	Extending Modification	Full Access	
	<<D>>	setBinaryDataChunk	Not Applicable	Not Allowed	Not Allowed	Allowed	
MCDFlash Segments							
	<<D>>	add	Not Applicable	Not Allowed	Not Allowed	Allowed	
	<<D>>	getItemByIndex	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getItemByName	Not Allowed	Allowed	Allowed	Allowed	
MCDFlash Session							
	<<D>>	getDbObject	Allowed	Allowed	Allowed	Allowed	
	<<D>>	getFlashDataBlocks	Not Applicable	Allowed	Allowed	Allowed	
MCDFlash SessionDesc							
	<<D>>	configureEventHandler	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getCooperationLevel	Allowed	Allowed	Allowed	Allowed	
	<<D>>	getDbObject	Allowed	Allowed	Allowed	Allowed	
	<<D>>	getFlashSession	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getLockState	Allowed	Allowed	Allowed	Allowed	
	<<D>>	isModifiedByOtherClient	Allowed	Allowed	Allowed	Allowed	
	<<D>>	isOwned	Allowed	Allowed	Allowed	Allowed	
	<<D>>	lock	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<D>>	releaseEventHandler	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	setEventHandler	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	unlock	Not Allowed	Not Allowed	Not Allowed	Allowed	
MCDFlash SessionDescs							
	<<D>>	addByDbObject	Not Allowed	Not Allowed	Allowed	Allowed	
	<<D>>	addByFlashKey	Not Allowed	Not Allowed	Allowed	Allowed	
	<<D>>	addByNameAndDbEcuMem	Not Allowed	Not Allowed	Allowed	Allowed	
	<<D>>	configureEventHandler	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getItemByIndex	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getItemByName	Not Allowed	Allowed	Allowed	Allowed	

Table H.9 (*continued*)

Class	Stereo Type	Name	Creation cooperation level				Comment
			No Cooperation	Read Only	Extending Modification	Full Access	
	<<D>>	releaseEventHandler	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	remove	Not Applicable	Not Allowed	Allowed	Allowed	The call to this method is allowed for all clients, but the method will fail if the client is not the master of the FlashSessionDesc to be removed or has not enough rights to remove the FlashSessionDesc.
	<<D>>	removeAll	Not Applicable	Not Allowed	Allowed	Allowed	The call to this method is allowed for all clients, but the method will fail if the client is not the master of the FlashSessionDesc(s) to be removed or has not enough rights to remove the FlashSessionDesc(s).
	<<D>>	removeByIndex	Not Applicable	Not Allowed	Allowed	Allowed	The call to this method is allowed for all clients, but the method will fail if the client is not the master of the FlashSessionDesc to be removed or has not enough rights to remove the FlashSessionDesc.
	<<D>>	removeByName	Not Applicable	Not Allowed	Allowed	Allowed	The call to this method is allowed for all clients, but the method will fail if the client is not the master of the FlashSessionDesc to be removed or has not enough rights to remove the FlashSessionDesc.
	<<D>>	setFlashEventHandler	Not Allowed	Allowed	Allowed	Allowed	

Table H.10 — ECU Configuration

Class	Stereo Type	Name	Creation cooperation level				Comment
			No Cooperation	Read Only	Extending Modification	Full Access	
MCDConfiguration Item							
	<<D>>	getDbObject	Allowed	Allowed	Allowed	Allowed	
	<<D>>	getItemValue	Not Allowed	Allowed	Allowed	Allowed	

Table H.10 (*continued*)

Class	Stereo Type	Name	Creation cooperation level				Comment
			No Cooperation	Read Only	Extending Modification	Full Access	
MCDConfiguration Record							
	<<D>>	getActiveFileName	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getConfigurationIdItem	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getConfigurationRecordValue	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getCooperationLevel	Allowed	Allowed	Allowed	Allowed	
	<<D>>	getDataIdItem	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getDbObject	Allowed	Allowed	Allowed	Allowed	
	<<D>>	getLockState	Allowed	Allowed	Allowed	Allowed	
	<<D>>	getMatchingFileNames	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getOptionItems	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getReadDiagComPrimitives	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getSystemItems	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getWriteDiagComPrimitives	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	isModifiedByOtherClient	Allowed	Allowed	Allowed	Allowed	
	<<D>>	isOwned	Not Applicable	Allowed	Allowed	Allowed	
	<<D>>	loadCodingData	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<D>>	lock	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<D>>	setConfigurationRecordValue	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<D>>	setConfigurationRecordValueByDbObject	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<D>>	unlock	Not Allowed	Not Allowed	Not Allowed	Allowed	
MCDConfiguration Records							
	<<D>>	addByConfigurationID	Not Allowed	Not Allowed	Allowed	Allowed	
	<<D>>	addByDbObject	Not Allowed	Not Allowed	Allowed	Allowed	
	<<D>>	addByName	Not Allowed	Not Allowed	Allowed	Allowed	
	<<D>>	configureEventHandler	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getItemByIndex	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getItemByName	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	releaseEventHandler	Not Allowed	Allowed	Allowed	Allowed	

Table H.10 (*continued*)

Class	Stereo Type	Name	Creation cooperation level				Comment
			No Cooperation	Read Only	Extending Modification	Full Access	
	<<D>>	remove	Not Applicable	Not Allowed	Allowed	Allowed	The call to this method is allowed for all clients, but the method will fail if the client is not the master of the ConfigurationRecord to be removed or has not enough rights to remove the ConfigurationRecord.
	<<D>>	removeAll	Not Applicable	Not Allowed	Allowed	Allowed	The call to this method is allowed for all clients, but the method will fail if the client is not the master of the ConfigurationRecord(s) to be removed or has not enough rights to remove the ConfigurationRecord(s).
	<<D>>	removeByIndex	Not Applicable	Not Allowed	Allowed	Allowed	The call to this method is allowed for all clients, but the method will fail if the client is not the master of the ConfigurationRecord to be removed or has not enough rights to remove the ConfigurationRecord.
	<<D>>	removeByName	Not Applicable	Not Allowed	Allowed	Allowed	The call to this method is allowed for all clients, but the method will fail if the client is not the master of the ConfigurationRecord to be removed or has not enough rights to remove the ConfigurationRecord.
	<<D>>	setEventHandler	Not Allowed	Allowed	Allowed	Allowed	
MCDOptionItem							
	<<D>>	getMatchingDbItemValue	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	setItemValue	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<D>>	setItemValueByDbObject	Not Allowed	Not Allowed	Not Allowed	Allowed	
MCDOptionItems							
	<<D>>	getItemByIndex	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getItemByName	Not Allowed	Allowed	Allowed	Allowed	
MCDReadDiag ComPrimitives							
	<<D>>	getItemByIndex	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getItemByName	Not Allowed	Allowed	Allowed	Allowed	

Table H.10 (*continued*)

Class	Stereo Type	Name	Creation cooperation level				Comment
			No Cooperation	Read Only	Extending Modification	Full Access	
MCDSystemItems							
	<<D>>	getItemByIndex	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getItemByName	Not Allowed	Allowed	Allowed	Allowed	
MCDWriteDiagComPrimitives							
	<<D>>	getItemByIndex	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getItemByName	Not Allowed	Allowed	Allowed	Allowed	

Table H.11 — Audience

Class	Stereo Type	Name	Creation cooperation level				Comment
			No Cooperation	Read Only	Extending Modification	Full Access	
MCDAudience							
	<<D>>	isAfterMarket	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	isAfterSales	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	isDevelopment	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	isManufacturing	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	isSupplier	Not Allowed	Allowed	Allowed	Allowed	

Table H.12 — ODX DOP Infos

Class	Stereo Type	Name	Creation cooperation level				Comment
			No Cooperation	Read Only	Extending Modification	Full Access	
MCDInterval							
	<<D>>	getLowerLimit	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getLowerLimitAsCodedValue	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getLowerLimitIntervalType	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getUpperLimit	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getUpperLimitAsCodedValue	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getUpperLimitIntervalType	Not Allowed	Allowed	Allowed	Allowed	

Table H.12 (*continued*)

Class	Stereo Type	Name	Creation cooperation level				
			No Cooperation	Read Only	Extending Modification	Full Access	Comment
MCDInternalConstraint							
	<<D>>	getInterval	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getScaleConstraints	Not Allowed	Allowed	Allowed	Allowed	
MCDScaleConstraint							
	<<D>>	getDescription	Allowed	Allowed	Allowed	Allowed	basic information required for listing at GUI
	<<D>>	getDescriptionID	Allowed	Allowed	Allowed	Allowed	basic information required for listing at GUI
	<<D>>	getInterval	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getRangeInfo	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getShortLabel	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getShortLabelID	Not Allowed	Allowed	Allowed	Allowed	
MCDTextTableElement							
	<<D>>	getInterval	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getLongName	Allowed	Allowed	Allowed	Allowed	basic information required for listing at GUI
	<<D>>	getLongNameID	Allowed	Allowed	Allowed	Allowed	basic information required for listing at GUI

Table H.13 — MCDMonitoringLink

Class	Stereo Type	Name	Creation cooperation level				
			No Cooperation	Read Only	Extending Modification	Full Access	Comment
MCDMonitoringLink							
	<<D>>	getInterfaceResource	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getMessageFilters	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getMonitoringCollector	Not Allowed	Allowed	Allowed	Allowed	
	<<D>>	getCooperationLevel	Allowed	Allowed	Allowed	Allowed	
	<<D>>	isModifiedByOtherClient	Allowed	Allowed	Allowed	Allowed	
	<<D>>	getLockState	Allowed	Allowed	Allowed	Allowed	
	<<D>>	lock	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<D>>	unlock	Not Allowed	Not Allowed	Not Allowed	Allowed	

Table H.14 — MCDWriteReadRecorder

Class	Stereo Type	Name	Creation cooperation level				Comment
			No Cooperation	Read Only	Extending Modification	Full Access	
MCDWriteReadRecorder							
	<<M>>	export	Not Allowed	Allowed	Allowed	Allowed	
	<<M>>	getPauseWatcher	Not Allowed	Allowed	Allowed	Allowed	
	<<M>>	getRecorderCollectors	Not Allowed	Allowed	Allowed	Allowed	
	<<M>>	isModifiedByOtherClient	Not Allowed	Allowed	Allowed	Allowed	
	<<M>>	lock	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<M>>	pause	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<M>>	resume	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<M>>	setPauseWatcher	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<M>>	unlock	Not Allowed	Not Allowed	Not Allowed	Allowed	
	<<M>>	unlockTree	Not Allowed	Not Allowed	Not Allowed	Allowed	

Annex I
(normative)

System properties

Table I.1 — System properties

Property Name	Description	Type	Values	Default Values	Mandatory/ Optional	Validity	Precondition
classPath	If set, additional entries for the `java.class.path` property are set. It is used to determine the classes which shall be loaded when the Job Processor is started. One or more files or directories can be given and they shall be separated by the path separator character of the platform (e.g. ';' when using Windows). Directories and/or *.jar files are allowed to be given in this property. If directories are given, only *.class files are collected. Only absolute paths are allowed. Changes to this property have only effect, if a new project is selected.	A_ASCIISTRING	All texts which describe a valid path are allowed.	No default value is defined.	O		MCDSystem needs to be in state eINITIALIZED
createVIDbTemplate	Allows to switch on the creation of detailed DBtemplates for control primitive VARIANT_IDENTIFICATION and VARIANT_IDENTIFICATION_AND_SELECTION. The default value of this property is false.	A_BOOLEAN	true, false	false	M	false (default), true	MCDSystem needs to be in state eINITIALIZED

Table I.1 (*continued*)

Property Name	Description	Type	Values	Default Values	Mandatory/ Optional	Validity	Precondition
ExecuteJobsUnchecked	If set to true, the D-server will ignore missing or wrong hash-values or unsupported hash algorithms at java jobs. That is, the server will execute any java job regardless of the fact whether it is signed or not. The new value becomes active with the next call to one of the functions executeSync(), executeAsync() or startRepetition() at an instance of MCDSingleEcuJob.	A_BOOLEAN	false,true	false	O	false (default), true	Allowed until first logical link has been created (MCDSystem needs to be in state eINITIALIZED, eDBPROJECT_ CONFIGURATION, ePROJECT_SELECTED or eVIT_SELECTED)
JobCancellationTimeout	Allows to define the time in milliseconds the D-server waits after it has called a Java job's cancel method before the D-server terminates the Java job's thread. Zero (0) is interpreted as immediate termination after the cancel method has returned.	A_UINT32	0 – MAX A_UINT32	0	mandatory	0 (default) – MAX A_UINT32	none

PDU-API.[Vendor]. OptionString	If set, the given string will be passed to the D-PDU-API as argument in the call of PDUConstruct.
	The [Vendor]-part is a placeholder for the ShortName of the D-PDU API. See ISO 22900-2:2009, Annex "D-PDU API Root Description File (RDF) " for details.
	The content of the OptionString is D-PDU API vendor-depended and not checked or interpreted by the D-server. See documentation of the specific D-PDU API for details.
		A_ASCIISTRING		No default value is defined.	O	PDU-API. D_PDU_API_1. OptionString = "IP = 192.168.1.1";
						PDU-API. D_PDU_API_1. OptionString = "Debug";
							VCI access layer shall not be prepared by means of MCDSystem::prepare VciAccessLayer().

Table I.1 (*continued*)

Property Name	Description	Type	Values	Default Values	Mandatory/ Optional	Validity	Precondition
	The D-server assumes that the OptionString shall not modify the behaviour of the D-PDU API. Any OptionString passed to the D-PDU API shall be ISO 22900-2 compliment to avoid causing any harm. Changes to this property do only have any effect if `MCDSystem::PrepareVciAccessLayer()` is called afterwards.						
PumpWindowsMessage Loop	If set to true, a call to the D-server from the Client pumps the Windows Message loop. (Single Threaded Apartments). The new value becomes active with the next method call.	A_BOOLEAN	false,true	true	O	false, true(default)	MCDSystem needs to be in state eINITIALIZED

Annex J
(informative)

VCI Module selection sequence

Table J.1 — VCI module selection sequence

Methods called at MCD-server API	Methods called at D-PDU API
MCDSystem::prepareVciAccessLayer()	PDUConstruct() will be executed for **all** available VCI-AccessLibs (e.g. D-PDU API)
interface = MCDSystem::getCurrevtInterfaces()	PDUGetModuleIds(): delivers list of handles hMod1, hMod2, ... of available connected VCI modules
Decide to which interface(s)to connect ... interface1.connect() [interface2.connect() ...]	 PDUModuleConnect(hMod1) [PDUModuleConnect(hMod2) ...]
create/open/gotoOnline LogicalLinks, execute DiagComPrimitives, ... close/reset (remove) LogicalLinks	... all methods call carry the selected VCI module handle hModx as input parameter ...

interface1.disconnect()
[interface2.disconnect()...]
	PDUModuleDisconnect(hMod1)
	PDUModuleDisconnect(hMod2)

MCDSystem::unprepareVciAccessLayer()	PDUDestruct() will be executed for **all** available VCI-AccessLibs (e.g. D-PDU API)

Annex K
(informative)

Service example illustrations

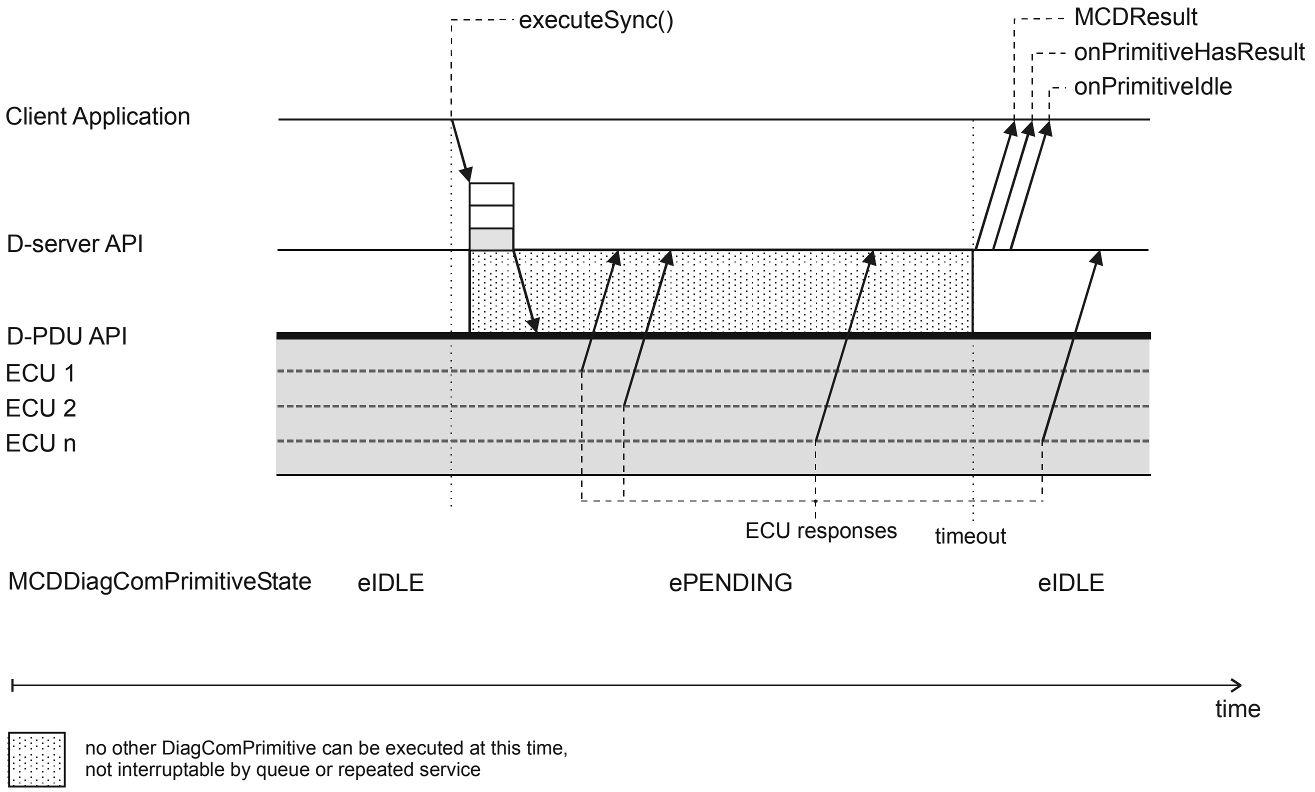
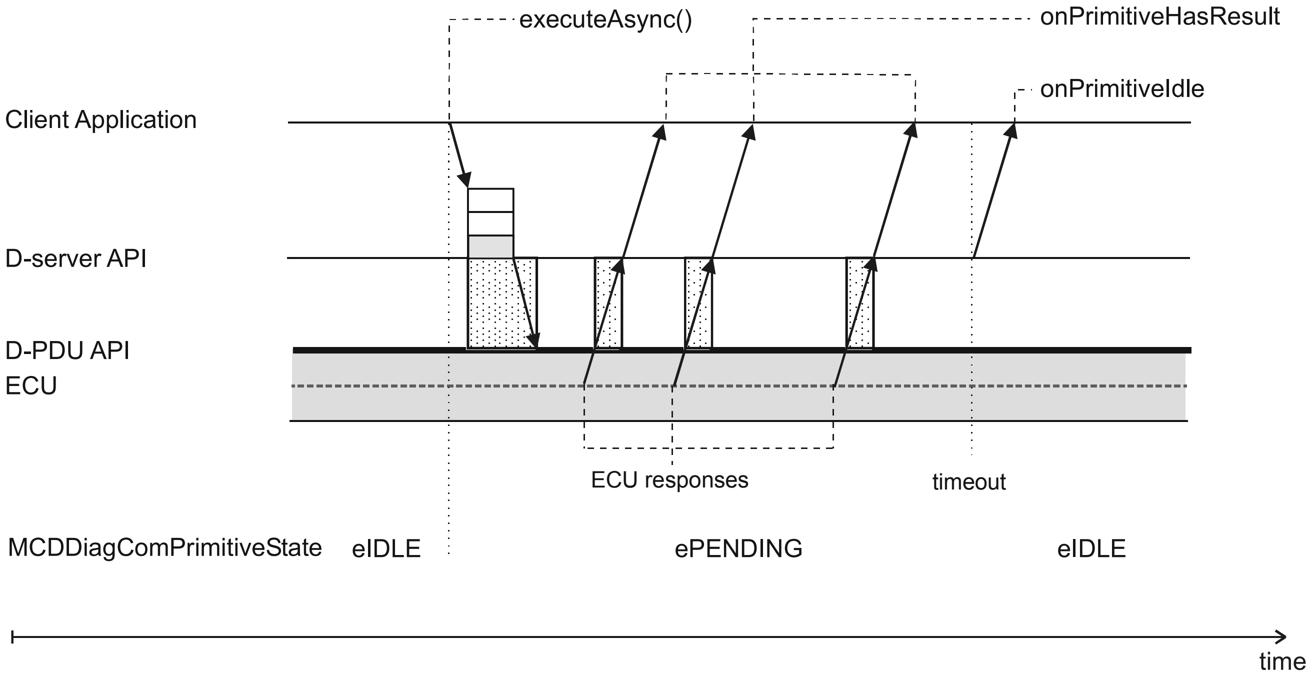
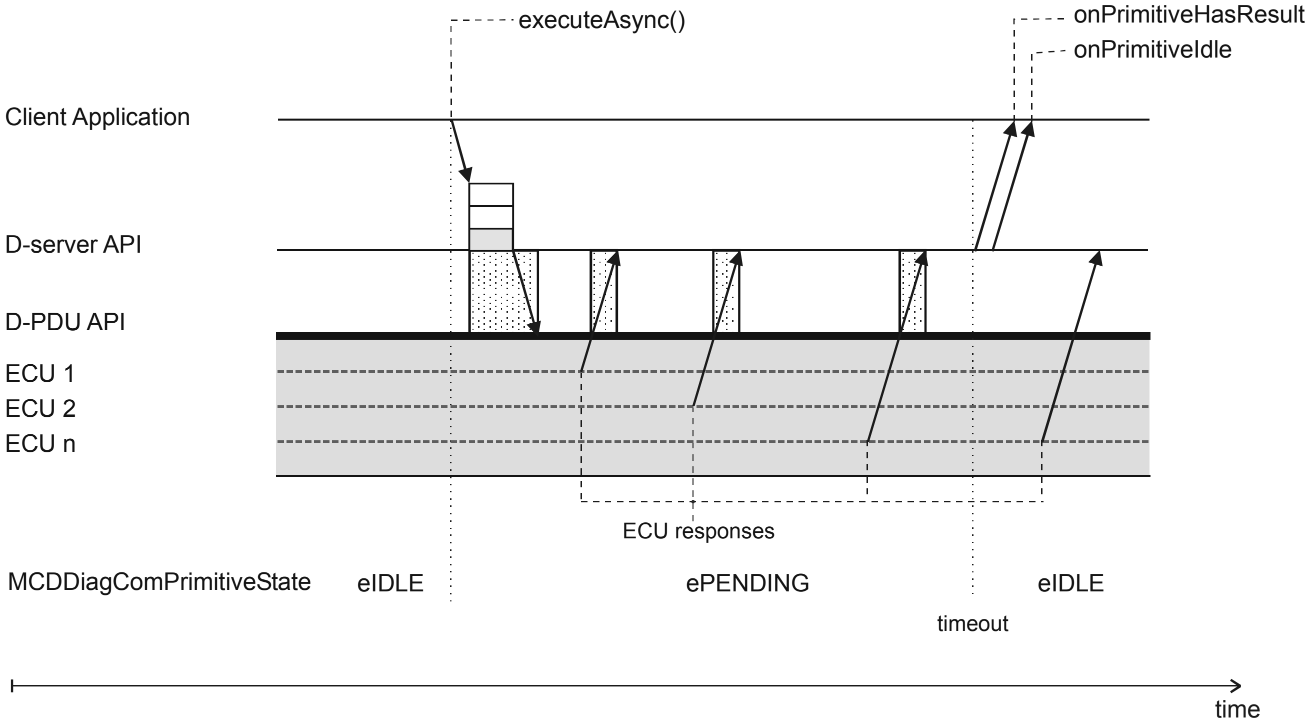


Figure K.1 — S=S; EM=S; T=RT; AM=F; IR=N; SPR



▨ no other DiagComPrimitive can be executed at this time, not interruptable by queue or repeated service

Figure K.2 — S=S; EM=A; T=RT; AM=P; IR=N; MPR



▨ no other DiagComPrimitive can be executed at this time, not interruptable by queue or repeated service

Figure K.3 — S=S; EM=A; T=RT; AM=F; IR=N; SPR

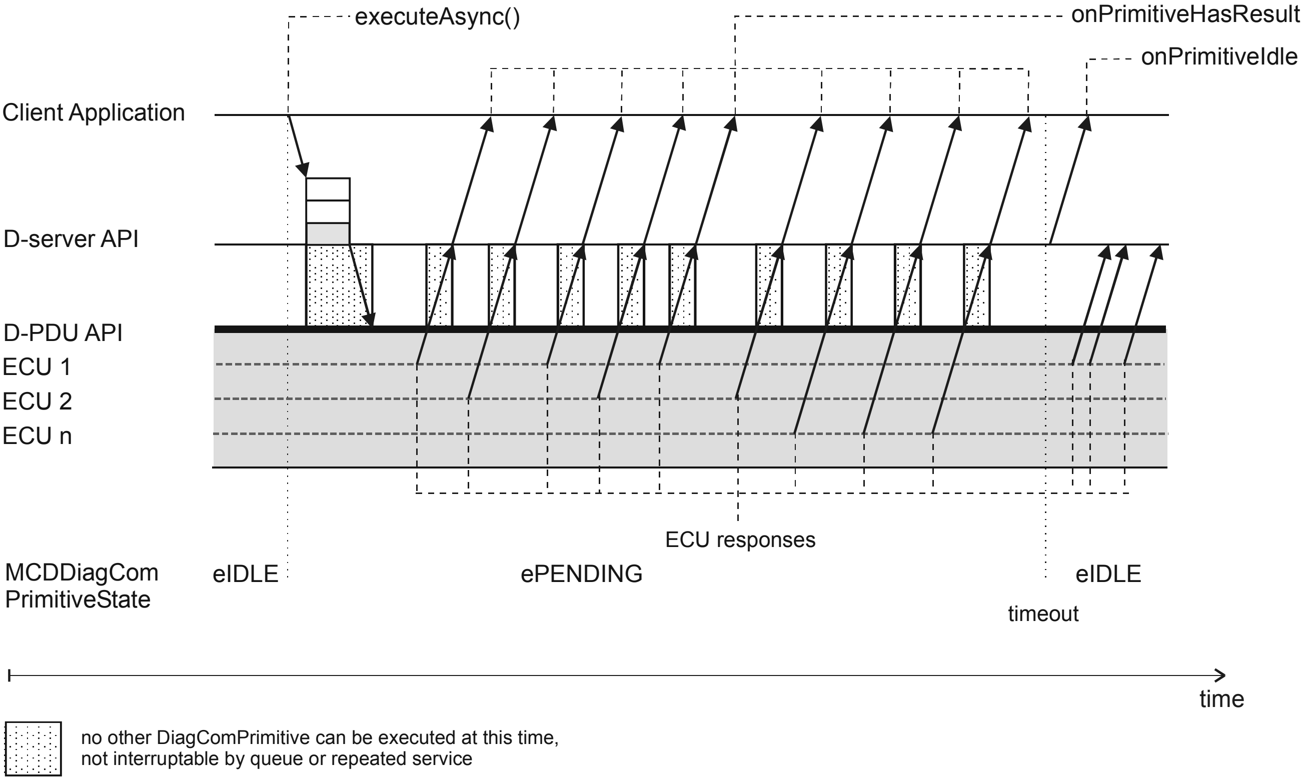


Figure K.4 — S=S; EM=A; T=RT; AM=F; IR=N; MPR

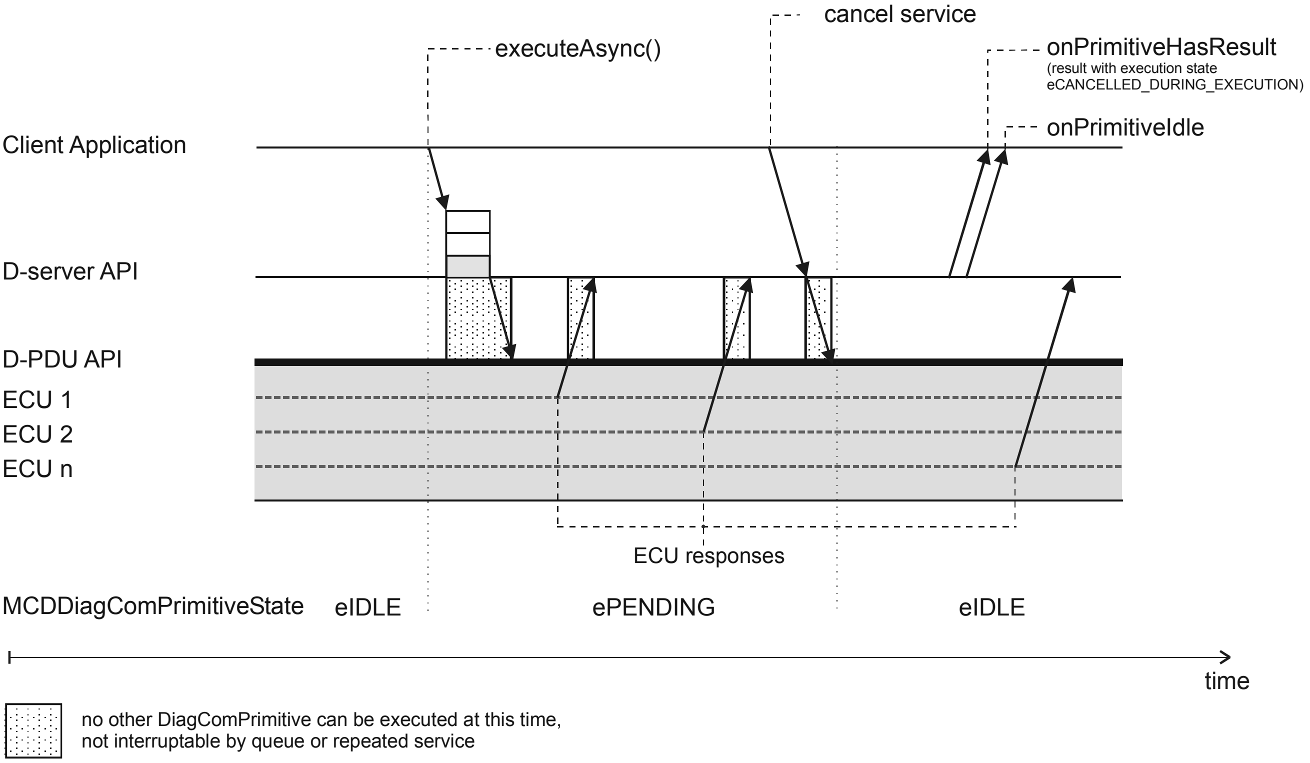


Figure K.5 — S=S; EM=A; T=C; AM=P; IR=N; SPR

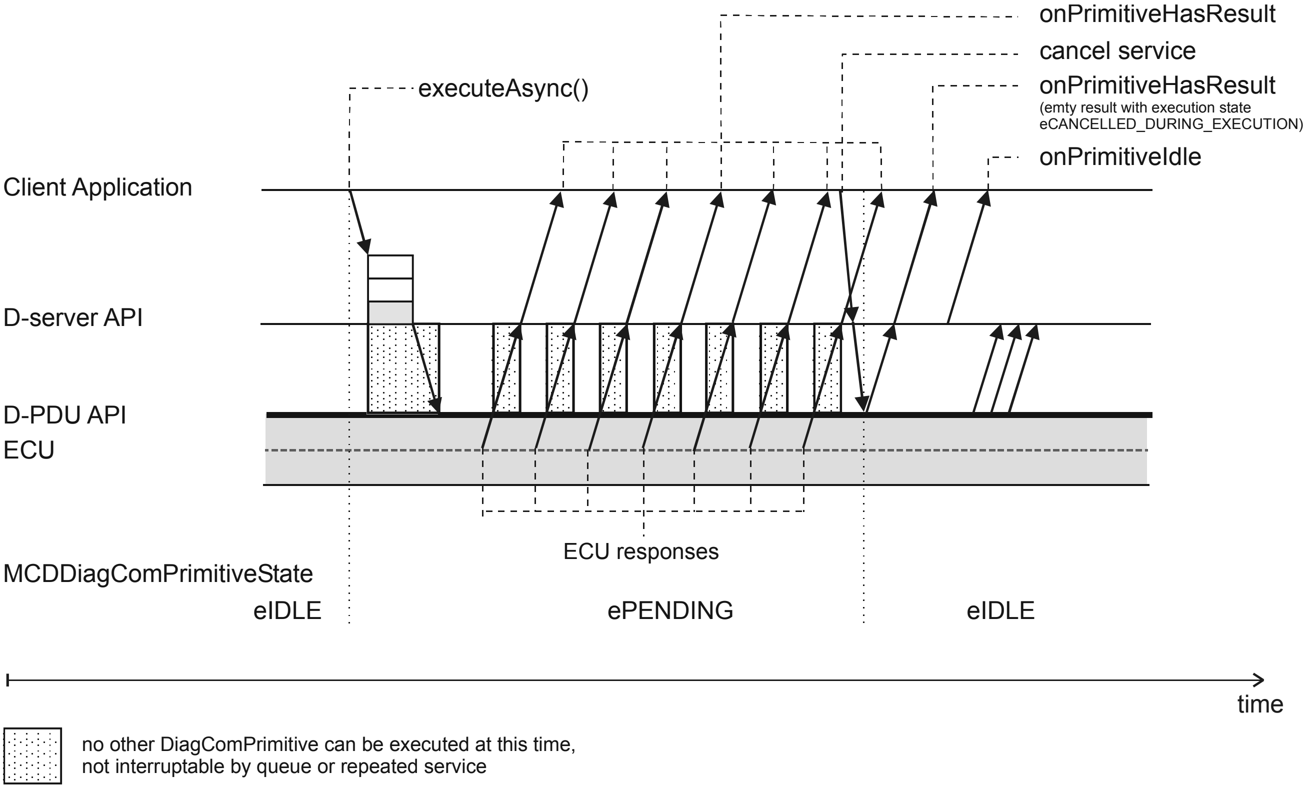


Figure K.6 — S=C; EM=A; T=C; AM=P; IR=N; MPR

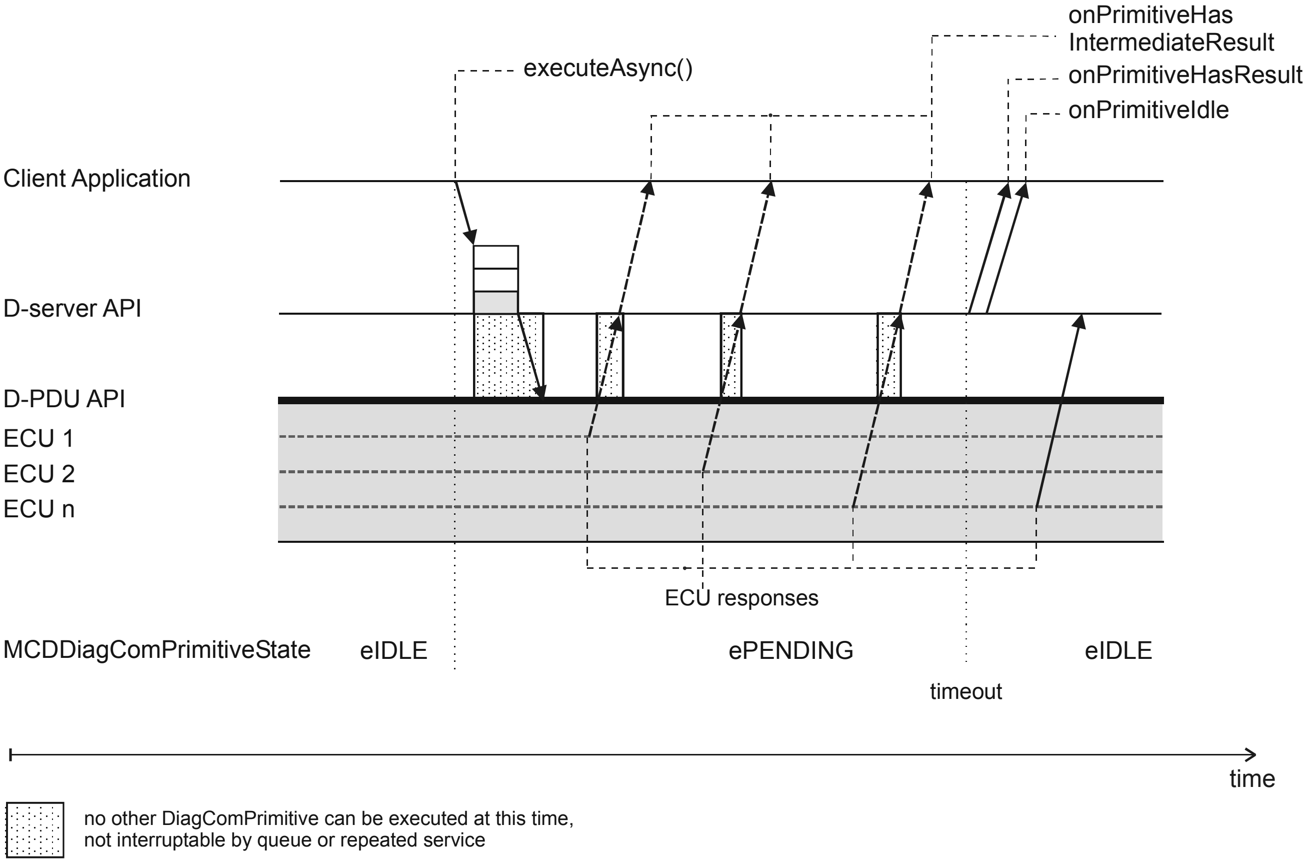


Figure K.7 — S=S; EM=A; T=RT; AM=F; IR=Y; SPR

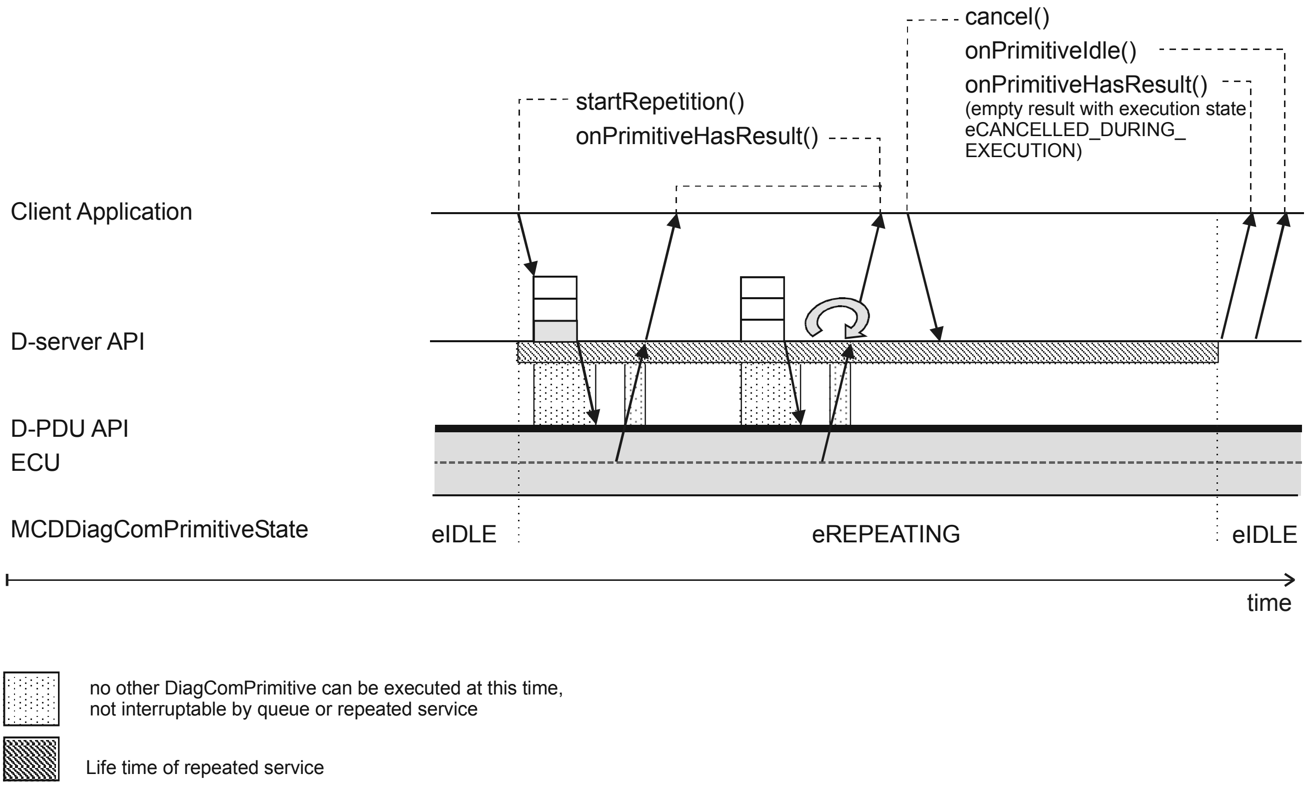

cancel()
onPrimitiveIdle()
onPrimitiveHasResult()
(empty result with execution state eCANCELLED_DURING_ EXECUTION)
startRepetition()
onPrimitiveHasResult()
Client Application
D-server API
D-PDU API
ECU
MCDDiagComPrimitiveState
eIDLE
eREPEATING
eIDLE
time
no other DiagComPrimitive can be executed at this time, not interruptable by queue or repeated service
Life time of repeated service

Figure K.8 — S=R; EM=A; T=C; AM=P; IR=M; SPR cancel between execution

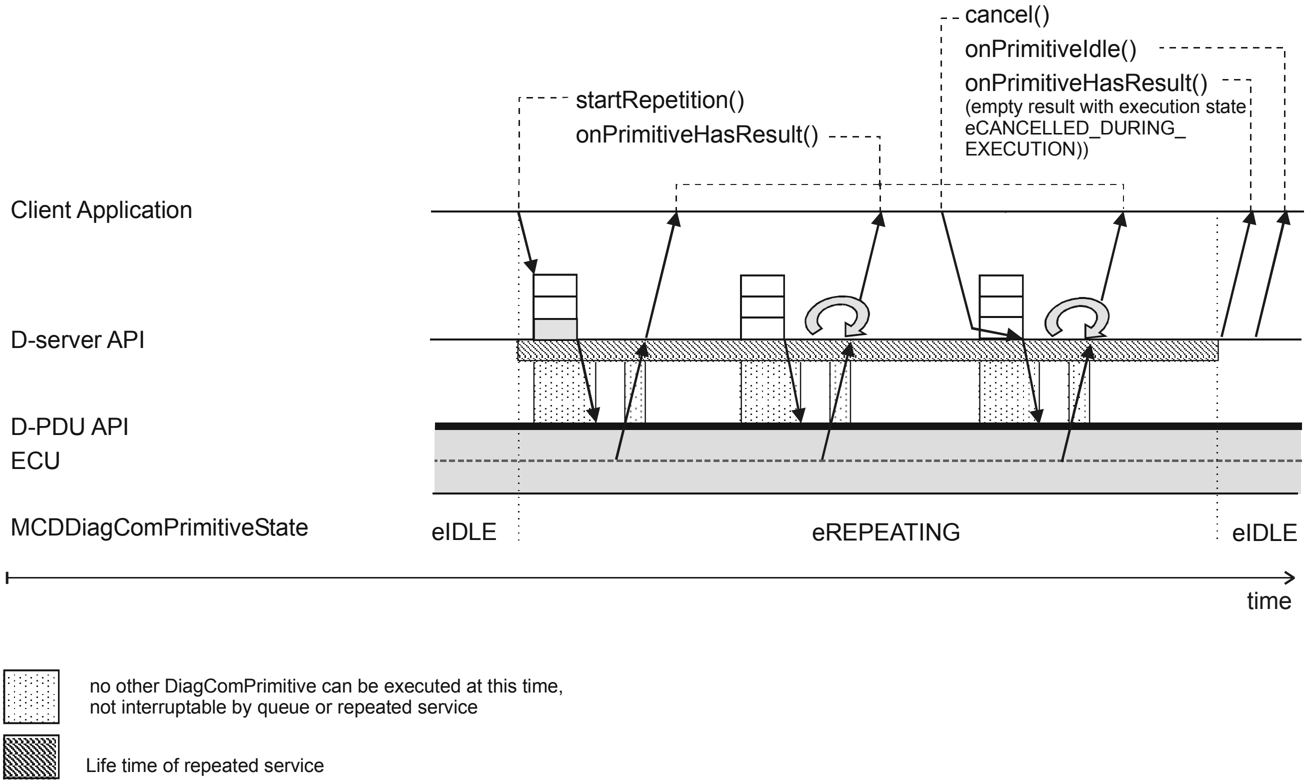

cancel()
onPrimitiveIdle()
onPrimitiveHasResult()
(empty result with execution state eCANCELLED_DURING_ EXECUTION))
startRepetition()
onPrimitiveHasResult()
Client Application
D-server API
D-PDU API
ECU
MCDDiagComPrimitiveState
eIDLE
eREPEATING
eIDLE
time
no other DiagComPrimitive can be executed at this time, not interruptable by queue or repeated service
Life time of repeated service

Figure K.9 — S=R; EM=A; T=C; AM=P; IR=M; SPR cancel execution while queueing

Annex L
(informative)

Examplary monitoring message format

An exemplary format for monitored messages as returned by the method `MCDCollector::fetchMonitoringFrames()` is defined in the following.

NOTE This definition applies especially to monitoring of CAN bus messages; other protocol types might require different/additional message format definitions. Also, the definitions in this chapter should be taken as an implementation guideline – depending e.g. on specifics of the protocol driver implementation layer of an actual system, message type flags or other implementation details can differ from the definitions in this chapter.

Table L.1 — Format of monitored message

Type	Timestamp	Address	Length	Data
RX	00000000000011700068	0000012D	8	0F 0F 00 00 30 CD 85 AC
RX	00000000000011746938	000003F6	5	02 00 00 00 00

The message type definitions are dependent on the actual protocol driver implementation used by the system. For example, when using a standard D-PDU API driver and the standard link types available to such a driver (e.g. an ISO_11898_RAW protocol link), it might not even be possible to discern between RX (receive) and TX (transmit) messages, as the monitoring link is in a receiver-only role and doesn't transmit any messages of its own. In case the actual implementation lends itself to a more detailed discrimination of messages types, the following type definitions should be used.

Message type:

— RX (Receive)

— TX (Transmit)

Examples for CAN-specific message types:

— ES (ERROR_STUFF) ➔ Bit stuffing error, more than 5 consecutive bits of equal polarity

— EF (ERROR_FORM) ➔ Form error, e.g. violation of end of frame (EOF) format

— EA (ERROR_ACK) ➔ ACK error, transmitting node receives no dominant acknowledgement bit

— EC (ERROR_CRC) ➔ CRC error, received CRC code does not match calculated code

Timestamp:
20 digit decimal (microsec), e.g. 00000001234560089768

The timestamp is only useful for putting messages into chronological relation to each other. As multiple controllers are potentially involved in the timestamp generation for logged messages (CAN-controller, VCI hardware controller, the controller that is running the D-server, the controller that is running the client application, etc.), it is not feasible to provide an exhaustive definition in the scope of this part of ISO 22900. A D-server uses the timestamp from the message delivered by the D-PDU API.

Communication node address:
8 digits hexadecimal, padded with leading zeros if necessary. In case there is no address information available, the address column should consist of 8 dots ('……..').

Data length:
Decimal value between 0 and N ([0,N])

Datastream:
Two-digit hexadecimal, e.g. 3F FF FF FF FF 3F 37 B5 (no separation by blanks)

General rules:

- Data blocks (message type, timestamp, address etc.) within a message are separated by blanks (‘ ‘).

In case an error (e.g. a buffer overrun) occurred in the communications hardware, the following message is to be inserted into the monitoring data: "### <Error description>".

Annex M
(normative)

Overview of the methods for isModifiedByOtherClient flag

This annex shows overview of the methods, which leads to set the isModifiedByOtherClient Flag to true. First the operations in the class itself are shown and second the methods in owned subobjects.

Table M.1 — Methods which lead to a modification at MCDCollector

Operations	
`setNoOfSamplesToFireEvent`	
`setStartWatcher`	
`setStopWatcher`	
Operations in owned subobjects	
Class	**Operation**
MCDBuffer	`setDownSampling`
MCDBuffer	`setRate`
MCDBuffer	`setSize`
MCDBuffer	`setTimeStamping`
MCDCollectedObjects	`addAscii`
MCDCollectedObjects	`addCurve`
MCDCollectedObjects	`addMap`
MCDCollectedObjects	`addScalar`
MCDCollectedObjects	`addValueBlock`
MCDCollectedObjects	`removeAll`
MCDCollectedObjects	`removeByIndex`
MCDCollectedObjects	`removeByName`
MCDTimeDelay	`configure`

Table M.2 — Methods which lead to a modification at MCDConfigurationRecord

Operations
`loadCodingData`
`setConfigurationRecordValue`
`setConfigurationRecordValueByDbObject`

Table M.2 (*continued*)

Operations in owned subobjects	
Class	**Operation**
MCDOptionItem	`setItemValue`
MCDOptionItem	`setItemValueByDbObject`

Table M.3 — Methods which lead to a modification at MCDDataPrimitive

Operations
`removeFilter`
`setFilter`
`setRepetitionTime`
`setResultBufferSize`
`updateRepetitionParameters`

Table M.4 — Methods which lead to a modification at MCDDiagComPrimitive

Operations	
`resetToDefaultValues`	
Operations in owned subobjects	
Class	**Operation**
MCDRequest	`setPDU`
MCDRequest:: MCDRequestParameters	`setParameterByName`
MCDRequest:: MCDRequestParameters::MCDRequestParameter	`addParameters`
MCDRequest:: MCDRequestParameters::MCDRequestParameter	`setCodedValue`
MCDRequest:: MCDRequestParameters::MCDRequestParameter	`setValue`
MCDRequest:: MCDRequestParameters::MCDRequestParameter	`SetValueUnchecked`

Table M.5 — Methods which lead to a modification at MCDDiagService

Operations in owned subobjects	
Class	**Operation**
MCDProtocolParameters	`addByDbObject`
MCDProtocolParameters	`addByName`
MCDProtocolParameters	`remove`
MCDProtocolParameters	`removeAll`
MCDProtocolParameters	`removeByIndex`
MCDProtocolParameters	`removeByName`
MCDProtocolParameter	`addParameters`

Table M.6 — Methods which lead to a modification at MCDDynIdClearComPrimitive

Operations
setDynId
setDefinitionMode

Table M.7 — Methods which lead to a modification at MCDDynIdDefineComPrimitive

Operations
setDynId
setDynIdParams
setDefinitionMode

Table M.8 — Methods which lead to a modification at MCDDynIdReadComPrimitive

Operations
setDynId
setDefinitionMode

Table M.9 — Methods which lead to a modification at MCDDLogicalLink

Operations	
clearQueue	
selectInterfaceResource	
setIntermediateResultForFunctionalAddressing	
setQueueSize	
setUnitGroupByName	
unsupportedComParametersAccepted	
Operations in owned subobjects	
Class	**Operation**
MCDConfigurationRecords	addByConfigurationIDAndDbConfigurationData
MCDConfigurationRecords	addByDbObject
MCDConfigurationRecords	addByNameAndDbConfigurationData
MCDConfigurationRecords	remove
MCDConfigurationRecords	removeAll
MCDConfigurationRecords	removeByIndex
MCDConfigurationRecords	removeByName
MCDDiagComPrimitives	addByDbObject

Table M.9 (*continued*)

MCDDiagComPrimitives	addByName
MCDDiagComPrimitives	addBySemanticAttribute
MCDDiagComPrimitives	addByType
MCDDiagComPrimitives	addDiagVariableServiceByRelationType
MCDDiagComPrimitives	addDynIdComPrimitiveByTypeAndDefinitionMode
MCDDiagComPrimitives	remove
MCDDiagComPrimitives	removeAll
MCDDiagComPrimitives	removeByIndex
MCDDiagComPrimitives	removeByName
MCDFlashSessionDescs	addByDbObject
MCDFlashSessionDescs	addByFlashKey
MCDFlashSessionDescs	addByNameAndDbEcuMem
MCDFlashSessionDescs	remove
MCDFlashSessionDescs	removeAll
MCDFlashSessionDescs	removeByIndex
MCDFlashSessionDescs	removeByName
MCDConfigurationRecords	addByConfigurationIDAndDbConfigurationData
MCDConfigurationRecords	addByDbObject

Table M.10 — Methods which lead to a modification at MCDFlashJob

Operations
setFlashSessionDesc
setFlashSessionDescByFlashKey
setFlashSessionDescByName

Table M.11 — Methods which lead to a modification at MCDFlashSessionDesc

Operations in owned subobjects	
Class	**Operation**
MCDFlashSession:: MCDFlashDataBlocks:: MCDFlashDataBlock	clearFlashSegments
MCDFlashSession:: MCDFlashDataBlocks:: MCDFlashDataBlock	loadFlashSegments
MCDFlashSession:: MCDFlashDataBlocks:: MCDFlashDataBlock:: MCDFlashSegments	add
MCDFlashSession:: MCDFlashDataBlocks:: MCDFlashDataBlock:: MCDFlashSegments:: MCDFlashSegment	setBinaryData
MCDFlashSession:: MCDFlashDataBlocks:: MCDFlashDataBlock:: MCDFlashSegments:: MCDFlashSegment	setSourceStartAddress

Table M.11 (*continued*)

MCDFlashSession:: MCDFlashDataBlocks:: MCDFlashDataBlock:: MCDFlashSegments:: MCDFlashSegment	`setBinaryData`
MCDFlashSession:: MCDFlashDataBlocks:: MCDFlashDataBlock:: MCDFlashSegments:: MCDFlashSegment :: MCDFlashSegmentIterator	`setBinaryDataChunk`

Table M.12 — Methods which lead to a modification at MCDMCLogicalLink

Operations	
`setMemoryPage`	
Operations in owned subobjects	
Class	**Operation**
MCDCharacteristics	`add`
MCDCharacteristics	`removeAll`
MCDCharacteristics	`removeByIndex`
MCDCharacteristics	`removeByName`
MCDCollectors	`add`
MCDCollectors	`removeAll`
MCDCollectors	`removeByIndex`
MCDCollectors	`removeByName`

Table M.13 — Methods which lead to a modification at MCDMonitoringLink

Operations in owned subobjects	
Class	**Operation**
MCDMessageFilters	`addByFilterType`
MCDMessageFilters	`remove`
MCDMessageFilters	`removeAll`
MCDMessageFilters	`RemoveByIndex`
MCDMessageFilter	`EnableMessageFilter`
MCDMessageFilter	`SetFilterType`
MCDMessageFilter	`SetFilterCompareSize`
MCDMessageFilterValues	`Add`
MCDMessageFilterValues	`Remove`
MCDMessageFilterValues	`removeAll`
MCDMessageFilterValues	`RemoveByIndex`

Table M.14 — Methods which lead to a modification at MCDProtocolParameterSet

Operations
`fetchValueFromInterface`
`fetchValuesFromInterface`

Table M.15 — Methods which lead to a modification at MCDService

Operations
`setRuntimeTransmissionMode`
`setSuppressPositiveResponse`

Table M.16 — Methods which lead to a modification at MCDStartCommunication

Operations
`setSuppressPositiveResponse`

Table M.17 — Methods which lead to a modification at MCDStopCommunication

Operations
`setSuppressPositiveResponse`

Table M.18 — Methods which lead to a modification at MCDWatcher

Operations
`setActivationWatcher`
`setAutoDeactivate`
`setTrigger`

Table M.19 — Methods which lead to a modification at MCDWriteReadRecorder

<table>
<tr><th colspan="2">Operations</th></tr>
<tr><td colspan="2"><code>setPauseWatcher</code></td></tr>
<tr><td colspan="2"><code>setStartWatcher</code></td></tr>
<tr><td colspan="2"><code>setStopWatcher</code></td></tr>
<tr><td colspan="2"></td></tr>
<tr><th colspan="2">Operations in owned subobjects</th></tr>
<tr><th>Class</th><th>Operation</th></tr>
<tr><td>MCDTimeDelay</td><td><code>configure</code></td></tr>
<tr><td>MCDWriteReadRecorderCollectors</td><td><code>add</code></td></tr>
</table>

Table M.19 (*continued*)

MCDWriteReadRecorderCollectors	removeAll
MCDWriteReadRecorderCollectors	removeByIndex
MCDWriteReadRecorderCollectors	removeByName
MCDWriteReadRecorderCollector	setRate
MCDWriteReadRecorderCollector	setDownSampling
MCDWriteReadRecorderCollector:: MCDCollectedObjects	addAscii
MCDWriteReadRecorderCollector:: MCDCollectedObjects	addCurve
MCDWriteReadRecorderCollector:: MCDCollectedObjects	addMap
MCDWriteReadRecorderCollector:: MCDCollectedObjects	addScalar
MCDWriteReadRecorderCollector:: MCDCollectedObjects	addValueBlock
MCDWriteReadRecorderCollector:: MCDCollectedObjects	removeAll
MCDWriteReadRecorderCollector:: MCDCollectedObjects	removeByIndex
MCDWriteReadRecorderCollector:: MCDCollectedObjects	removeByName

Table M.20 — Methods which lead to a modification at MCDSystem

Operations	
ResetProperty	
SetPhysFormat	
SetProperty	
Operations in owned subobjects	
Class	**Operation**
MCDMonitoringLinks	addByInterfaceResource
MCDMonitoringLinks	Remove
MCDMonitoringLinks	removeAll
MCDMonitoringLinks	removeByIndex
MCDMonitoringLinks	removeByName
MCDLogicalLinks	addByAccessKeyAndInterfaceResource
MCDLogicalLinks	addByAccessKeyAndVehicleLink
MCDLogicalLinks	addByDbObject
MCDLogicalLinks	AddByName
MCDLogicalLinks	AddByNames
MCDLogicalLinks	AddByObjects
MCDLogicalLinks	AddByVariant
MCDLogicalLinks	Remove
MCDLogicalLinks	removeAll
MCDLogicalLinks	RemoveByIndex

Table M.20 (*continued*)

MCDLogicalLinks	RemoveByName
MCDRecorders	addRead
MCDRecorders	addWriteRead
MCDRecorders	remove
MCDRecorders	removeAll
MCDRecorders	RemoveByIndex
MCDRecorders	RemoveByName
MCDWatchers	add
MCDWatchers	remove
MCDWatchers	removeAll
MCDWatchers	RemoveByIndex
MCDWatchers	removeByName

Bibliography

[1] ISO 7637-2, *Road vehicles — Electrical disturbances from conduction and coupling — Part 2: Electrical transient conduction along supply lines only*

[2] ISO 8601, *Data elements and interchange formats — Information interchange — Representation of dates and times*

[3] ISO/IEC 8859-1, *Information technology — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1*

[4] ISO/IEC 8859-2, *Information technology — 8-bit single-byte coded graphic character sets — Part 2: Latin alphabet No. 2*

[5] ISO 15031-3, *Road vehicles — Communication between vehicle and external equipment for emissions-related diagnostics — Part 3: Diagnostic connector and related electrical circuits, specification and use*

[6] ISO 15031-4, *Road vehicles — Communication between vehicle and external equipment for emissions-related diagnostics — Part 4: External test equipment*

[7] ISO 15031-5, *Road vehicles — Communication between vehicle and external equipment for emissions-related diagnostics — Part 5: Emissions-related diagnostic services*

[8] ISO 16750-2, *Road vehicles — Environmental conditions and testing for electrical and electronic equipment — Part 2: Electrical loads*

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

[10] ISO 27145 (all parts)[5], *Road vehicles — Implementation of WWH-OBD communication requirements*

[11] ASAM MCD 2 D ODX, *Diagnostic Data Model Specification V 2.1*

[12] ASAM MCD 2 MC, *Measurement and Calibration Data Specification V1.5*

[13] ASAM MCD 3, *Application Programming Interface Specification*

[14] ASAM MCD 3 RefGuide, *Programmers Reference Guide*, part: MCD, MD and D; part: MCD, MC, MD, M and C

[15] IEEE 754-2008, *IEEE Standard for Floating-Point Arithmetic*

[16] IEEE 1003.1-2001, *IEEE Information Technology — Portable Operating System Interface (POSIX.)*

[17] Interface Definition COM-IDL, *MCD 3 UML Object Model to COM Automation Mapping Rules*

[18] Interface Definition C++, *MCD 3 UML Object Model to C++ Mapping Rules*

[19] Interface Definition Java, *MCD 3 UML Object Model to Java Automation Mapping Rules*

[20] RFC 3305, *Report from the Joint W3C/IETF URI Planning Interest Group: Uniform Resource Identifiers (URIs), URLs, and Uniform Resource Names (URNs): Clarifications and Recommendations*

[5] Under preparation. [Revision of ISO/PAS 27145 (all parts).]

[21] SAE J1708, *Serial data communications between microcomputer systems in heavy-duty vehicle applications*

[22] SAE J1850, *Class B data communications network interface*

[23] SAE J1939, *Recommended practice for a serial control and communications vehicle network*

[24] SAE J2411, *Single wire CAN network for vehicle applications*

[25] SAE J2610, *Serial data communication interface*

ICS 43.040.15

Price based on 379 pages